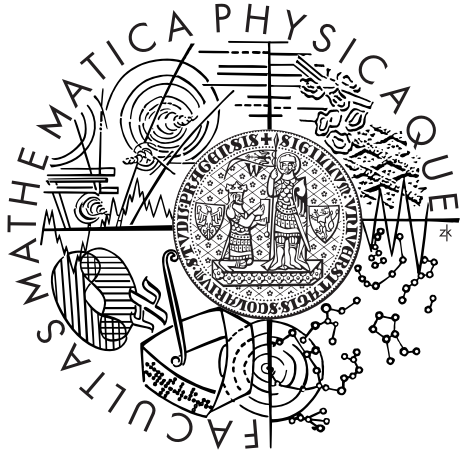
Charles University in Prague
Faculty of Mathematics and Physics

DOCTORAL THESIS



Lukáš Marek

Instrumentation and Evaluation for Dynamic Program Analysis

Department of Distributed and Dependable Systems

Advisor: Doc. Ing. Petr Tůma, Dr.
Study programme: Computer Science
Specialization: Software Systems

Prague 2014

Acknowledgments

I would like to thank my advisor Petr Tůma for his encouragement while deciding what to do next after finishing my master degree and his support and guidance through my PhD study. I would like to thank all my colleagues from the D3S department and especially my colleagues from the room 205 for making the study pleasant and fun experience. I would like to also thank my SCIEX-NMSch advisor Walter Binder and his colleagues for hosting me for one beautiful year at University of Lugano in Switzerland.

I would like to thank my family and my girlfriend Lucie for their endless support.

I hereby declare that I have authored this doctoral thesis on my own[1], using only the cited literature and other technical sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, on ....

[1]The papers included in Part II have been written in cooperation with their respective co-authors.

Annotations

Title

Instrumentation and Evaluation for Dynamic Program Analysis

Author

Lukáš Marek
e-mail: *lukas.marek@d3s.mff.cuni.cz*, phone: +420 221 914 190

Department

Department of Distributed and Dependable Systems
Faculty of Mathematics and Physics
Charles University in Prague, Czech Republic

Advisor

Doc. Ing. Petr Tůma, Dr.
Department of Distributed and Dependable Systems
e-mail: *petr.tuma@d3s.mff.cuni.cz*, phone: +420 221 914 267

Abstract

A dynamic program analysis provides essential information during later phases of an application development. It helps with debugging, profiling, performance optimizations or vulnerability detection. Despite that, support for creating custom dynamic analysis tools, especially in the domain of managed languages, is rather limited.

In this thesis, we present two systems to help improve application observability on the Java platform. DiSL is a language accompanied with a framework allowing simple and flexible instrumentation for the dynamic program analysis. DiSL provides high level abstractions to enable quick prototyping even for programmers not possessing a knowledge of Java internals. A skilled analysis developer gains full control over the instrumentation process, thus does not have to worry about unwanted allocations or hidden execution overhead.

ShadowVM is a platform that provides isolation between the observed application and the analysis environment. To reduce the amount of possible interactions between the analysis and the application, ShadowVM offloads analysis events out of the context of the application. Even though the isolation is the primary focus of the platform, ShadowVM introduces a number of techniques to stay performance comparable and provide a similar programming model as existing dynamic analysis frameworks.

Keywords

Bytecode instrumentation; dynamic program analysis; aspect-oriented programming; JVM

Anotace

Název práce

Instrumentace a vyhodnocení pro dynamickou analýzu aplikací

Autor

Lukáš Marek
e-mail: *lukas.marek@d3s.mff.cuni.cz*, phone: +420 221 914 190

Katedra

Katedra distribuovaných a spolehlivých systémů
Matematicko-fyzikální fakulta
Univerzita Karlova v Praze

Školitel

Doc. Ing. Petr Tůma, Dr.
Katedra distribuovaných a spolehlivých systémů
e-mail: *petr.tuma@d3s.mff.cuni.cz*, tel.: +420 221 914 267

Abstrakt:

Dynamická analýza aplikací zprostředkovává důležité informace během pozdějších fází vývoje. Napomáhá při laděni, profilování, výkonnostní optimalizaci nebo při detekci bezpečnostních chyb. Nicméně, podpora pro vytváření vlastních nástrojů pro dynamickou analýzu, speciálně v oblasti řízených jazyků, je poměrně omezená.

Tato práce prezentuje dva systémy, které pomáhají zlepšit sledování aplikací na platformě Java. DiSL je jazyk a framework, který umožňuje jednoduchou a flexibilní instrumentaci zaměřenou na dynamickou analýzu. DiSL poskytuje abstrakce vyšší úrovně pro rychlé prototypování i pro vývojáře, kteří nemají znalosti interních systémů v Javě. Kvalifikovaný vývojář získává plnou kontrol na instrumentačním procesem, tudíž se nemusí bát nevyžádaných alokací nebo skryté běhové režie.

ShadowVM je platforma poskytující separaci mezi sledovanou aplikací a prostředím pro analýzu. Pro zmírnění interakcí mezi analýzou a aplikací, ShadowVM transportuje události analýzy mimo kontext aplikace. I když je primárním cílem platformy izolace, ShadowVM zavádí několik technik tak, aby zůstala rychlostně srovnatelná a vytvářela obobné vývojové podmínky jako existující frameworky pro dynamickou analýzu.

Klíčová slova

Bajtkódová instrumentace; dynamická analýza programů; aspektově orientované programování; JVM

Contents

Part I

Introduction and Contribution Overview

Chapter 1
Introduction

Dynamic program analysis plays an important role in software development. It is used to uncover implementation defects or obtain various application characteristics. There are many examples of sophisticated dynamic analysis such as:

Taint analysis The taint analysis observes propagation of values from an initial set of variables through the application. The initial set typically contains variables holding the application input received over a network, through GUI or from a database. As a malicious input can compromise a run of the application, the application uses specialized methods to verify/sanitize the received input. The goal of the taint analysis is to track whether all the received data went through the sanitization process. If the data manages, through a series of computations and assignments, to escape the sanitization and is used as an input in a potentially unsafe operation, the taint analysis reports the problematic data flow.

Execution time profiling Another often used dynamic analysis is the execution time profiling. The analysis measures the execution time of a method by acquiring the timing information at the entry and at the exit of the method. The observation coverage of execution time profiling is fully adjustable, starting from a single method to the whole application code. As the observation often slows down the application by orders of magnitude, limiting the coverage to only a small part of the application brings noticeable performance gains. On the contrary, while observing a more complex method, it may be useful to profile the method with a granularity of basic blocks to obtain more detailed information about its behaviour.

Data race detection Yet another example of dynamic analysis is data race detection in multi-threaded applications. At runtime, the data race analysis monitors all field accesses and lock operations. When a non-volatile field is accessed by multiple threads without relevant lock acquisition, the analysis reports a possible data race.

Object lifetime analysis In long running applications, memory leaks may deplete all available application memory. Even though the problem is typical for languages with dedicated allocation and deallocation routines, memory leaks may happen even in languages using garbage collector. An object is considered a memory leak if the application is holding a reference to the object but does not intend to use it in the future. Such object cannot be garbage-collected and occupies memory. The object lifetime analysis tracks all manipulations with references (pointers) and is able to determine whether an object can be deallocated, i.e., which references on objects are held longer than necessary.

To perform such analyses, the developer often uses well known dynamic analysis tools like OProfile [32], VTune [12, 75], gprof [10, 57], JDB [18] or Visu-

alVM [42]. However, such tools are often crafted to collect only one or a set of similar metrics. When a more complex analysis is required, the developer is forced to create a custom solution.

Dynamic analysis tools are carefully designed to not interfere with the observed application. Any change in the state or control flow of the application could potentially lead to its invalid behaviour and distorted observation results. It is therefore highly recommended to use one of the analysis frameworks such as dtrace [48] (SystemTap [38, 74]), Valgrind [40, 70] or pin [34, 64] as they provide safe environment for writing custom dynamic analyses. While such frameworks exist for native applications, managed languages still lack a widely adopted solution.

One of the key aspects of a framework for building custom dynamic analyses is the ability to inform about various types of events happening in the application and offer rich context information while analysing these events. The analysis should be performed in a safe environment where the developer does not have to worry about undesired side effects on the observed application. As performance is one of the biggest problems of dynamic analysis frameworks, induced overhead should be minimal and fully under control of the developer. We believe that one reason for lack of wide adoption of existing Java analysis frameworks is their inability to deliver a satisfactory solution in all the mentioned aspects. Therefore, the primary goal of the thesis is to create a platform that would ease development of dynamic program analyses.

1.1 Thesis structure

The structure of the thesis is as follows. The next section gives a brief overview of application observability in general, followed by a summary of observation alternatives in Java. The second part of this chapter summarizes our experience with dynamic analysis in managed runtimes and concludes with the revisited goals of the thesis.

The second chapter summarizes our contribution starting with a high level instrumentation language for Java called DiSL. The rest of the second chapter introduces an environment for analysis evaluation called ShadowVM.

Chapters three, four and five list three published papers containing the details of our work. Chapter six describes the related work. Chapter seven concludes.

1.2 Dynamic analysis

The purpose of the dynamic analysis is to observe and evaluate application behaviour. Figure 1.1 shows a simple architecture of a dynamic analysis platform. During execution (1), a probe triggers an event in the observed application. The triggered event is dispatched (2) to the analysis logic, where it is evaluated (3).

As illustrated at the bottom of Figure 1.1, we see the dynamic analysis as composed from two parts. The *observation part*, which gathers contextual information and triggers the evaluation events, and the *evaluation part*, which performs the analysis evaluation based on the triggered events.

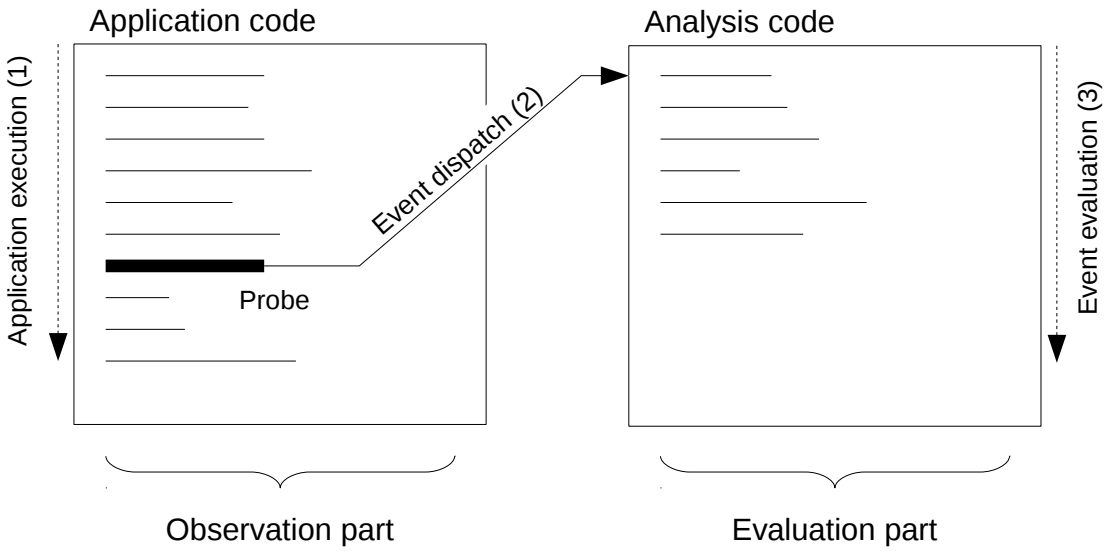


Figure 1.1: Dynamic analysis architecture.

1.3 Application observability

As an application is executed, each action like object allocation or method invocation may be of interest to the analysis. Depending on the type of observed actions, the analysis may choose from a variety of techniques to analyse the application behaviour.

1.3.1 Observation using execution callbacks

Execution environments often provide an interface for various execution callbacks. For example, managed languages may expose callbacks connected to class loading, JIT compilation, garbage collection or virtual machine life cycle. Interpreted languages may provide even more execution events related to object allocation, exception handling, field access, method execution or synchronization.

Another kind of callbacks may be provided by hardware. Special registers inside the CPU allow to monitor different types of low-level performance events like cache accesses, memory accesses, instruction execution or branch predictions. A callback is triggered every time a number of events reaches a predefined threshold.

As the callbacks are built into a particular platform, an extension is often problematic. Another limitation is the amount of additional information (context information) they are able to provide. For example, an analysis may require detailed information about an allocated object, like place of the allocation and the size of the object. If the callback does not provide such information, the analysis developer is forced to modify the execution environment (if possible).

A similar problem is with defining new events. As the callbacks are provided by some of the execution environment subsystems like the interpreter or the garbage collector, the particular subsystem has to be modified to obtain a new type of events. Such modifications require deep knowledge of the platform and the analysis created for the modified platform loses portability. Therefore, if the platform does not support the required events or does not supply the necessary context information, it is often better to choose a different observation technique.

1.3.2 Instrumentation

The observation technique allowing to easily define new types of events is instrumentation. Instrumentation allows to observe any sequence of code executed in the observed application. During the instrumentation process (weaving), instrumentation code is inserted before or after the application code. When invoked, the instrumentation code triggers an event which is passed to the analysis for evaluation. If processing of the event is very short, like an increment of a counter, the evaluation logic may be included in the instrumentation code. When a more sophisticated analysis is required, the instrumentation code invokes a separate evaluation method.

Instrumentation weaving is possible at different stages of the application deployment. Each stage has its own advantages and disadvantages. Weaving of the application source code allows the instrumentation to easily detect high-level language constructs and insert the instrumentation code as a text string written in the application language. The weaving is often performed offline or during compilation, hence access to the sources of the observed application is necessary.

Instrumentation of the application is also possible on the machine code level. In comparison to the source code, machine code instruction set provides only a thin level of abstraction over the hardware it operates. Detection of language constructs like objects, classes or fields (member variables) is therefore much more difficult.

Managed languages often support compilation into an intermediate representation (bytecode). As the bytecode is an intermediate step between the source code and the native code waving of the bytecode has several advantages. Proprietary applications often does not provide source code, but they are compiled and distributed in the bytecode form. In comparison with the native code, the intermediate representation is often high-level enough to easily recognize constructs of the original language. During execution, bytecode is optimized by an interpreter or a JIT compiler, hence the overhead of the instrumentation code is often noticeably reduced.

1.3.3 Sampling

A different type of monitoring is sampling. During sampling, event processing is not triggered based on the application behaviour. Instead, an observation method is triggered after a given time period. Sampling is therefore useful only for certain type of analyses like performance monitoring or collection of aggregated results.

1.4 Dynamic analysis evaluation

During execution, the application triggers various events that are of interest to the analysis. The triggered event together with additional context information is supplied to the main analysis logic for evaluation. During evaluation, the analysis may perform various computations and build non-trivial data structures to be able to track the application behaviour.

We recognize two types of analysis evaluation, based on the context in which the evaluation is performed. An evaluation performed in the context of the anal-

ysed application is called *in-process analysis* and the analysis offloaded out of the context of the analysed application is called *out-of-process analysis*.

1.4.1 In-process analysis

Events triggered during in-process analysis have the form of simple method callbacks. It is vital that the execution environment supports some form of isolation between the application and the analysis. Otherwise, the evaluation and the analysed application share many resources like memory, execution threads or the standard input and output. As the observed application is tightly coupled with the evaluation logic, in-process analysis may introduce undesired perturbations. Depending on the type of the perturbation, it may invalidate the analysis results or break the application execution.

1.4.2 Out-of-process analysis

To prevent perturbation, the analysis may be offloaded out of the application context. The main goal of offloading is to substantially reduce the amount of code executed in the context of the observed application.

Some execution environments already provide an observation mechanism capable pf offloading the analysis events. An example of such a mechanism is the debugging interface. If the execution environment does not provide a suitable observation mechanism, the analysis may uses some type of inter-process communication (IPC) to offload the events. The commonly used IPC techniques include pipe, network socket, shared memory or remote procedure call.

As our implementation targets Java platform, the next section briefly overviews the available monitoring alternatives under Java.

1.5 Application observability in Java

The more complex a system is, the more important it is that the system provides means to monitor itself. As applications in Java are executed in a virtual machine with a JIT compiler, garbage collector and several other service threads, it is vital to provide an environment where the application and the VM could be safely observed.

Historically, Java contained two low level interfaces for application and VM monitoring. ***JVMPI*** [28] was a native interface intended for profiling and allowed to observe memory allocations, execution times, locking or thread events. The second, called ***JVMDI*** [27], was a debugging interface and allowed to set breakpoints and watches or access class, object, method and field information. JVMPI never reached a stable state and was always marked as experimental. It was probably because of its many limitations, and so it was replaced [73] in Java 1.5 by a more mature ***JVM Tool Interface (JVMTI)*** [29]. During the transition from JVMPI to JVMTI, JVMDI was also merged into the JVMTI and deprecated in Java 6.

As a merger of the two, JVMTI became a rich API useful for debugging, profiling and other kinds of application analysis using different types of monitoring

techniques like sampling, instrumentation or predefined callbacks. JVMTI also allows to monitor various JVM resources and events connected with garbage collection, class loading, compilation or JVM lifecycle.

JVMTI is a part of a collection of interfaces called ***Java Platform Debugger Architecture (JPDA)*** [25, 26]. Apart from JVMTI, JPDA also contains high-level a Java language interface for remote debugging called ***Java Debugger Interface (JDI)*** [19] and a protocol describing communication between a debugger and a debugged JVM called ***Java Debug Wire Protocol (JDWP)*** [20]. Compared to JVMTI, JDI provides Java interface and its communication model assumes that events are processed in different JVM, thus it is easier to use and potentially safer. Nevertheless, JDI is not as versatile as JVMTI. As the name suggests, JDI is mainly targeted on real-time debugging and is not designed for application wide monitoring.

JVMTI is very powerful, however working in a native code may be seen as too complicated for simple dynamic analysis. For that reason, Java also provides a ***java.lang.instrument*** [1] package for instrumentation directly from Java space. The *instrument* interface does not provide any predefined callbacks or sampling interface but can be easily used for instrumentation based analysis.

A bit different type of monitoring offers technology called ***Java Management Extensions (JMX)*** [23]. JMX is a framework, which allows to monitor and control the application using objects called Managed Beans (MBean in short). MBeans do not use instrumentation to trigger events instead, they are developed directly as a part of an application. The MBean object controls a component (resource) of the application and allows to read the component state, set and get its configuration and register event notification listeners.

During application execution, MBeans are registered in JMX framework waiting for a remote connection. As MBean is required to implement one of the predefined JMX interfaces, it can be accessed through Java VisualVM [42] monitoring tool. Alternatively, custom monitoring tools can be used.

1.5.1 Instrumentation in Java

As mentioned in the thesis goals, the monitoring interface should provide flexibility in definition of new observation events. The predefined callbacks in JVMTI and JDI provide only a fixed set of events, with no options for extensibility. JMX provides custom monitoring but only if the MBean objects are designed during the application development. Creating MBeans for an existing application means to modify its source code or bytecode, which effectively means to instrument it. As the instrumentation provides the highest flexibility when defining new events, we will discuss its applicability in Java in more detail.

Instrumentation allows to intercept application behaviour based on execution of an arbitrary bytecode pattern. It is therefore possible to intercept object allocations, method executions, exception handling of field access. Instrumentation is able to capture execution patterns with various granularity. It is not a problem to instrument method entry and method exit. It is also possible to instrument a sin-

[1] `http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html`

gle instruction like an evaluation of a conditional statement or a data conversion of a numeric value.

Instrumentation of an application written in Java can be done either offline, online using a Java level *java.lang.instrument* interface, or online using native JVMTI. Each solution has its advantages and disadvantages. We discuss them in turn.

Offline instrumentation

Offline instrumentation does not have any direct support in Java. Every offline instrumentation framework needs to load whole application and library code from a persistent storage, traverse the code and apply the instrumentation. The instrumentation can be done either on source code or pre-compiled bytecode.

Although the offline instrumentation does not have any direct support in Java, it has the highest flexibility of all. During the instrumentation, a class can be renamed, split into several parts and a class hierarchy can be modified. Classes can be freely traversed and re-instrumented several times, which is useful in cases where some sort of static analysis of the whole application should be performed before the main instrumentation. Offline instrumentation does not delay a start of the application.

The disadvantage of offline instrumentation is in coverage. There is no possibility to instrument dynamically generated classes and also classes that are not specified directly on the class path, for example classes loaded by a custom classloader over a network. As the instrumentation is done on classes stored on a filesystem, it can be unintentionally applied more than once.

java.lang.instrument interface

A convenient online instrumentation is accessible through a *java.lang.instrument* instrumentation API. Java Virtual Machine automatically intercepts loaded classes and passes them through the instrumentation interface to a custom Java agent in a form of bytecode. Java does not provide any support for parsing and changing the bytecode, but the agent is able to use some widely adopted bytecode manipulation library like ASM [2, 47], BCEL [4, 53] or Javassist [17, 49], thus source code of the resulting agent is usually compact.

As the Java agent is running in the same context as the observed application, the instrumentation process may introduce several undesirable side-effects.

The instrumentation agent is written in pure Java and it requires an already initialized JVM to be able to run. During the initialization (JVM bootstrap phase), JVM loads an initial set of core classes before loading the instrumentation agent. The core classes escape instrumentation and the only option how to instrument them is a process called retransformation [14]. During retransformation, JVM replaces code of a loaded class with an altered code. As the class is already known to JVM and can be already instantiated, retransformation imposes strict rules on introduced changes. During retransformation, fields and methods must not be added, modified or renamed. It is also forbidden to change method signatures or change class inheritance. As a consequence, the Java agent approach allows only a limited set of modifications to the classes loaded during the JVM bootstrap phase.

Another problem is hidden in the instrumentation process itself. During the instrumentation, the Java agent is allowed to use any class from the Java Class Library. When requested, the JVM loads the class, bypassing the Java agent instrumentation callback. Instrumentation callback has to be bypassed, otherwise the agent could request the same class and trigger another class loading introducing infinite recursion. The loaded class is however immediately visible also to the application. The agent is able to retransform the class when it finishes the ongoing instrumentation process, however the un-instrumented class may be already used by the application.

JVMTI interface

Another option for performing online instrumentation is a JVM Tool Interface (JVMTI). JVMTI is a native interface and allows to instrument every class loaded by JVM. There is no need to bypass the instrumentation process in specific scenarios like in the case of the *java.lang.instrument* interface and there is (almost) no unintended interaction between the instrumentation code and the observed application.

The instrumentation process may however influence initialization during a JVM bootstrap phase. Instrumentation from native space allows to intercept and instrument all loaded classes, even classes loaded during the JVM bootstrap phase. JVM does not permit arbitrary modification to the classes loaded during bootstrap phase and so, instrumentation code may crash the JVM. The rules on permitted instrumentation are not documented and differs from version to version of JVM. It is therefore hard to judge, if the problem is a JVM bug or a limitation of JVMTI.

Applying instrumentation directly in native code can be challenging. To our knowledge, there is no widely adopted C/C++ library for bytecode manipulation. Resulting code responsible for performing the instrumentation is therefore many times bigger (in the terms of lines of code) than a similar solution done using the *java.lang.instrument* interface.

1.6 Dynamic analysis evaluation in Java

We consider two types of the dynamic analysis evaluation, the in-process analysis and the out-of-process analysis.

Java does not provide any isolation specifically to support the in-process analysis. During the in-process analysis, the analysis and the observed application share many resources like service threads, the heap, garbage collector, or the Java Class Library. As a consequence, the evaluation may create a lot of perturbation in the observed application. Various problems that may arise are discussed in more detail in the following section.

Contrary to in-process analysis, Java provides several interfaces to support out-of-process analysis. All main observation interfaces offload the events out of the application context. JDI and JMX offload the events into another Java process, whereas JVMTI offloads the events into native code.

Another option for offloading events is the Java Native Interface (JNI) [24].

JNI allows the Java code to invoke native methods. Therefore, JNI can be used to transport events to native code.

1.7 Dynamic analysis pitfalls

Following subsections summarize variety of perturbations we have encountered while developing dynamic analysis tools. Even though the problems are tackled by almost every dynamic analysis, they are often not mentioned in the literature. Hence, there is only limited knowledge among the developers about possible dangers connected to the dynamic analysis development.

Our list is not meant to be complete as it is in general hard to predict what parts of the application will be influenced by the observation. The perturbations are often discovered later on as an inconsistency in the observed results or as a failed execution of the observation application.

Execution time

A presence of additional code in the observed application may already cause perturbation. An execution time is probably the most visible one, where a high amount of instrumented sites may extend the time of execution by orders of magnitude.

The size of inserted code may additionally influence code optimisations made by a JIT compiler. The JIT compiler inlines small methods to eliminate a cost of a method call and to apply more advanced optimizations. If the method size grows over a certain threshold, the JIT compiler does not perform inlining as the resulting code would be too large.

Object allocation strategy

Another useful JIT compiler optimization is called scalar replacement. The scalar replacement optimization uses escape analysis to decide whether it can allocate objects directly on the execution stack. Such optimization can be done only if an object reference does not escape out of the scope of the method where it was allocated. However, if the instrumentation code exports the reference outside of the method, the object has to be allocated on the application heap and later on collected by a garbage collector.

Available memory

During in-process analysis, the application and the analysis evaluation share a single memory heap. Frequent allocations by the evaluation influence the overall memory consumption and subsequently increase the frequency of garbage collector runs. If the evaluation stores a substantial amount of data, it may deplete all available memory and crash the observed application.

Triggering of events

The in-process analysis and the observed application not only share the memory heap or the execution threads but also the set of loaded classes. Most of the load-

ed classes are used only by the application or by the analysis but some classes like third party libraries or classes from the Java Class Library may be shared. Methods of such shared libraries should behave differently depending on the context from which they are invoked. If a method is invoked from the application code, it should generate the analysis events. On the contrary, if invoked from the analysis code, the instrumentation code should be skipped.

A solution for the third party libraries is to load the classes by different class loaders where the application class loader loads an instrumented version of the class and the analysis class loader loads an uninstrumented version. Unfortunately, such solution is not applicable on classes from the Java Class Library, as they can be loaded only by the system class loader. Therefore, code in the Java Class Library requires a different solution for switchable event triggering. One of such mechanisms is called dynamic bypass [67].

From a technical side, the dynamic bypass is a simple flag indicating whether the current execution originates from the application or from the analysis. Every inserted instrumentation code is wrapped by a branching condition checking the state of the dynamic bypass flag. If the dynamic bypass flag is set (indicating the code is called from the analysis code), it skips the instrumentation code, i.e. it does not generate additional events.

Shared state corruption

The dynamic bypass mechanism described above only works for reentrant code. If the code is not reentrant, the analysis may damage the state of the shared class.

We illustrate the problem on the printing method. An application uses the printing method which protects data of each stream with a lock. The application starts printing into a stream and successfully acquires a lock. Let us assume that the stream is the standard output stream, so it is easily accessible from the application and also from the analysis. During the printing, the instrumentation code is invoked and execution is transferred to the analysis to evaluate a triggered event. After evaluation, the analysis starts printing the results to the standard output. As the execution thread already holds the output lock acquired by the application code, it is allowed to enter into a protected part of the printing method. The printing method invoked from the application left the stream in an inconsistent state and the subsequent call damages the state of the stream.

Method size limit and class loading

Methods in Java are limited in size to 64 KiB of bytecode. The limit constrains the amount of instrumentation code that can be directly inlined into a method body. As the analysis developer cannot generally predict the size of the instrumented method, the instrumentation code should contain only essential code to trigger an event. All other code connected to the event evaluation should be extracted into a separate class.

Classes containing the evaluation logic are not application classes, therefore they have to be loaded separately. Loading extra classes can be problematic in a case of certain class loaders, used for example in OSGI [33], where the class loader supports loading of classes from predefined packages only. In OSGI, the standard

delegation of the class loading process to a parent class loader is allowed only for classes from the Java Class Library. This behaviour provides better isolation between the application components, but poses severe problems for application monitoring. When the instrumentation code invokes a method in an evaluation class, the application class loader fails to load the class and generates an exception. In the worst case, the exception is handled by the observed application, not producing any error but corrupting the application behaviour.

Static initializers

While the Java code is being executed, it gradually loads additional application and library classes. After a class is loaded, a static initializer is invoked to initialize the class.

The dynamic bypass (described above) prevents triggering of events during execution of the analysis code. This mechanism is also active when the analysis code triggers class loading. Such behaviour is correct only if the loaded class is exclusively used by the analysis. If the application uses the loaded class, the static initializer is not invoked and the analysis misses events that would otherwise be part of the application execution.

Analysing events triggered by analysis

To ensure correct results of the analysis, an instrumented run of the application should produce exactly the same events as the uninstrumented one. Below, we describe a situation, where usage of specific Java classes may result in observation of additional events; events not triggered by the observed application.

The application and the analysis can both take advantage of a *WeakReference*[2] class in conjunction with a *ReferenceQueue*[3] container. Weak references are useful for object caching and analysis often uses weak references for object lifetime tracking. The dynamic bypass mechanism can prevent triggering of events while the *WeakReference* object is manipulated directly, but operations on *ReferenceQueue* are handled by a special service thread created by the JVM. The analysis should monitor events created by the service thread as the work of the thread is triggered by the application. Nevertheless, the service thread is also processing weak references created by the analysis, thus triggering additional events not related to the application behavior.

Introduction of deadlock

While analysing multi-threaded applications, the analysis should never change synchronization behaviour in the observed application. In other words, the analysis should never acquire a lock that could be potentially held by the application as it could lead to a deadlock. An obvious problem is an invocation of methods on a shared object (like printing into standard output), where the analysis and the application compete for one shared lock. However, the analysis may introduce a deadlock even if it does not invoke such a shared object directly.

[2] `http://docs.oracle.com/javase/7/docs/api/java/lang/ref/WeakReference.html`
[3] `http://docs.oracle.com/javase/7/docs/api/java/lang/ref/ReferenceQueue.html`

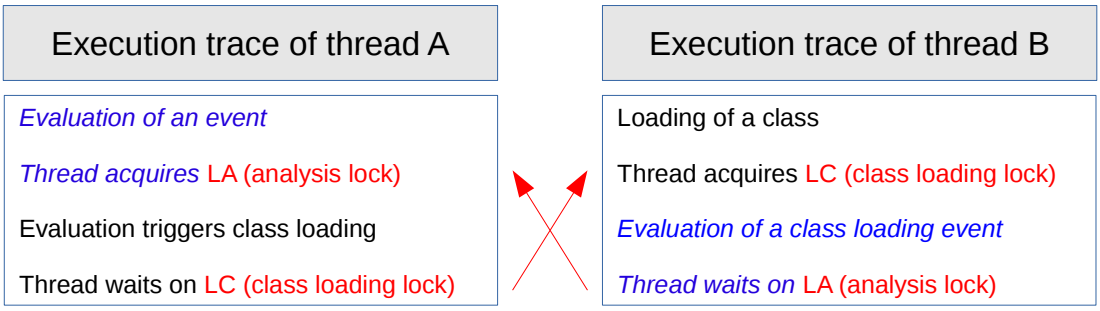


Figure 1.2: An application deadlock caused by analysis evaluation. Text written in black denotes application execution, blue italic text denotes analysis execution, and bold red text denotes lock acquisition.

Figure 1.2 illustrates an execution trace leading to a deadlock in a multi-threaded application caused by the analysis evaluation. The analysis is observing the multi-threaded application and uses a lock to protect consistency of its data. The application is a standard multi-threaded application with two threads, A and B. Thread A is currently evaluating a triggered analysis event. Thread B is loading a new application class. Thread A is holding a lock LA as it is currently modifying the analysis data. Thread B is holding a lock LC as the standard Java class loader implementation is synchronized.

Among other events, the analysis is observing the class loading behaviour. Therefore, thread B triggers an event and starts waiting on the analysis evaluation lock LA already held by thread A. Meanwhile, analysis code executed by thread A triggers class loading. The class loading process invokes the class loader to load a new class. Thread A starts waiting on the class loading lock LC and creates deadlock.

1.8 Goals revisited

The primary goal of the thesis is to define a platform that would ease the development of dynamic program analyses. The platform is essentially composed of two parts, the *event observation* and the *event evaluation*. The event observation should be flexible enough to support implementation of a wide range of dynamic analyses. The event evaluation should provide an isolated environment allowing the analysis developer to write the evaluation logic without worrying about possible perturbation of the observed application.

Looking closer at the target platform of our initial implementation, the instrumentation provides the most flexible and portable interface. Java exposes two distinct interfaces for bytecode instrumentation, JVMTI and Java level instrumentation. JVMTI provides instrumentation on the native code level, hence the developer is required to have a deep knowledge about the whole managed infrastructure (virtual machine). The java level agent can instrument directly in the managed code, but the instrumentation process may disturb the observed application. Offline instrumentation is also possible, nevertheless the instrumentation process may leave libraries and dynamically loaded classes un-instrumented.

The validity of the performed analysis is our primary focus, therefore neither Java level agent nor offline instrumentation is suitable. The remaining option is

instrumentation using JVMTI. As JVMTI is a native interface, the goal will be to provide an event specification method that hides the complexity of the native level instrumentation.

Java itself does not provide any support for isolation inside one process. To prevent undesired perturbation, the event evaluation needs to be offloaded out of the Java context.

In 1.6, we mentioned three (JDI, JMX, JVMTI) interfaces for offloading events out of the application context. JMX and JDI do not provide capabilities we require. Especially they are not designed to transfer bigger amounts (tens of megabytes per second) of data.

JVMTI provides the interface to observe only a predefined set of JVM events in the native space. However, the instrumentation produced using JVMTI is able to trigger custom events in the observed application, therefore we require a solution allowing to offload custom events out of the Java space. JNI allows to call an arbitrary native method from Java, thus it can offload any event into the native space. As we aim to ease the development of the dynamic analyses, our goal will be to pass the event out of the native space into a more convenient environment for the event evaluation.

After summarizing the monitoring capabilities of the target platform, we are ready to expand the initial goals as follows.

The primary goal of the thesis is to create a platform that would ease the development of dynamic program analyses. Such a platform should support simple but flexible method for capturing events originating in the observed application. The events of interest should be specified using an instrumentation language with the following attributes:

- High-level language constructs enable a developer not familiar with instrumentation internals to create a custom dynamic analysis.
- A skilled developer should be allowed to define new constructs to capture arbitrary application behaviour.
- The language should provide rich access to static and dynamic context information.
- As the dynamic analysis often requires processing of large amount of events, incurred overhead of the instrumentation code should be minimal and fully in the hands of the analysis developer.

The platform should provide an isolated environment, where the evaluation of the captured events does not cause perturbation in the observed application. The environment for event evaluation should have the following properties:

- Isolation of the observed application limits perturbation caused by the dynamic analysis.
- The isolation is achieved by processing analysis events out of the context of the observed application.

- A programming model of the evaluation environment should be close to the programming model of the hosting language.
- The isolation poses a reasonable overhead so that the implemented dynamic analysis stays performance competitive.
- As we aim for a solution that will be applicable in practice, the implementation should be usable in production JVMs.

Chapter 2
Overview of Contribution

This chapter summarizes the work we have done to ease development and reduce the incurred perturbation while observing a runtime behaviour of an application.

The presented observation platform is composed from two parts. The first called DiSL, is a language for instrumentation specifically designed for dynamic program analysis. The second called ShadowVM, is a system for offloading dynamic analysis out of the context of an observed application.

Combined together, DiSL and ShadowVM provide a feature complete infrastructure for creating custom dynamic analyses of Java applications while improving the development efficiency, minimizing induced perturbation and providing competitive performance compared to similar frameworks. Although most of the problems solved by DiSL and ShadowVM are generic to all managed runtimes, both systems are tightly coupled with the Java environment.

2.1 DiSL, domain specific language for Java bytecode instrumentation

DiSL is a language for rapid development of instrumentation targeted on the domain of dynamic program analysis. The key concepts beside the development efficiency are small runtime overhead, simple extensibility, and observation coverage of the whole Java Class Library.

The DiSL language is hosted in Java and uses Java annotations to guide the instrumentation process. The language is inspired by the Aspect Oriented Programing [61] and adopts AOP's three basic concepts: shadow, join-point, and advice. The DiSL instrumentation framework is implemented on top of ASM [2, 47], a widely adopted Java instrumentation library.

The Instrumentation in DiSL is written as a standard Java class where methods are annotated by a custom DiSL annotation. A method contains instrumentation code that will be inlined into the application. The annotation specifies whether the instrumentation code will be inserted before or after a defined block of code and includes additional parameters for the instrumentation process like scoping.

Markers

In DiSL, instrumentation code is inserted before or after a block of code marked by a construct called *Marker*. DiSL provides a library of predefined *Markers* to easily insert instrumentation code before or after a method body, a method invocation or a single bytecode instruction. Custom *Marker* allows to define arbitrary block that may be of interest to the instrumentation developer.

Context information

Providing access to static and dynamic contextual information is an essential feature of every instrumentation framework. DiSL exposes two interfaces called *StaticContext* and *DynamicContext* for accessing information about context in which is an event triggered. The *DynamicContext* is a predefined interface providing access to local variables, variables on the Java stack, *this* object and method arguments. The *StaticContext* is fully customizable and allows the developer to pre-compute custom static information during weave time and access the pre-computed information in the instrumentation code. Access to basic information about classes and methods under instrumentation is in DiSL exposed through a predefined library of StaticContext classes.

Instrumentation scoping

A scoping language implemented in DiSL allows to restrict the instrumentation to particular classes or methods. A scoping pattern is expressed as a string matching a class name, a method name and a method signature of a method to be instrumented. The scoping pattern may additionally use wildcards to substitute a part of the pattern value.

When more control over the scoping is needed, a *Guard* construct enables to evaluate more comprehensive conditions using Java code. The *Guard* indicates for each instrumented location whether the location should be instrumented or not. The *Guard* is a standard Java class with one annotated method evaluating the scoping condition. The *Guard* method has access to all context information available using *StaticContext* and *DynamicContext* interfaces.

Data passing

To perform more complex analysis, the instrumentation may require to share information across several instrumented locations. DiSL provides two distinct mechanisms for easy and efficient data passing. A standard Java field defined in an instrumentation class and annotated by a *ThreadLocal* annotation acts as a thread local variable. The behaviour is the same as the Java thread local variable, however DiSL translates all operations on the variable as a direct access to the Java *Thread* class, thus making it more efficient.

For efficient data passing between instrumentation code inserted into the same method body, DiSL introduces a construct called synthetic local variable. It is again a field defined in an instrumentation class and annotated by a *SyntheticLocal* annotation. In each application method with inserted instrumentation code, DiSL creates a new local variable and translates all operations on the synthetic local field as operations on the local variable.

Exception handling

The instrumentation written in DiSL is free to use arbitrary Java code. As the instrumentation is meant for observation only, it is not desirable to throw any exception out of the scope of the instrumentation. Such an exception would otherwise propagate through the application and change its control flow. All the inserted code is therefore automatically wrapped by a try-catch block handling

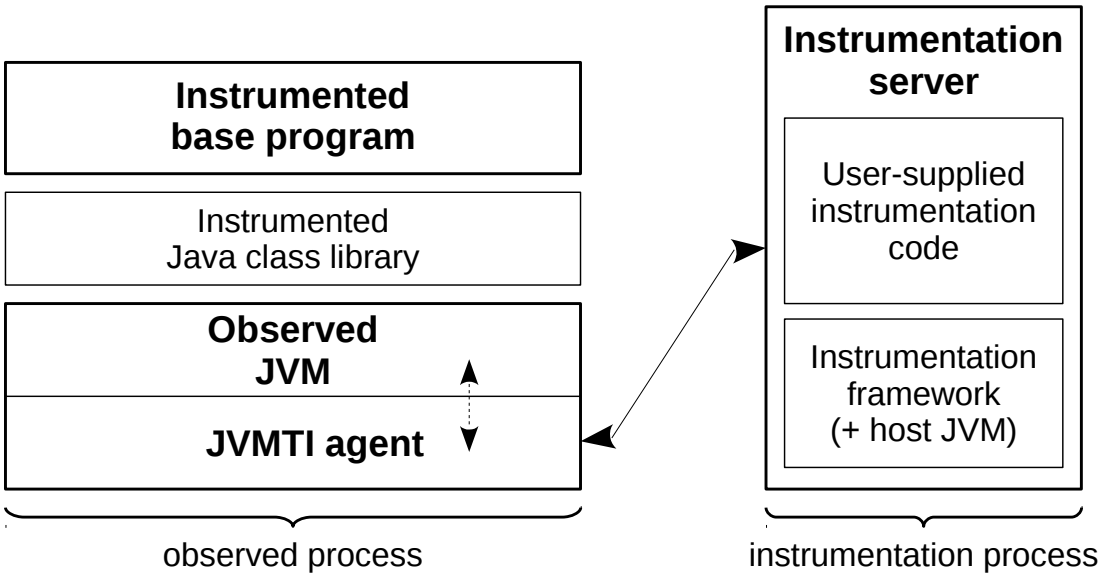


Figure 2.1: An architecture of the DiSL instrumentation framework.

all exceptions introduced in the instrumentation. The wrapping can be disabled when the instrumentation reaches production quality.

Framework architecture

DiSL contains a simple wrapper library for offline instrumentation. The limitation of offline instrumentation is in the use of thread local variables, where special instrumentation of *java.lang.Thread* is required. Besides offline instrumentation, DiSL ships a framework for online application instrumentation. Even though the majority of DiSL features are designed to support hotswapping constraints[1] and could potentially instrument an already running application, the framework is currently limited to instrument classes while being loaded by the JVM.

As shown in Figure 2.1, DiSL uses two JVMs to separate the observed application and the instrumentation. The separation reduces perturbation and allows to instrument the whole Java Class Library without complex instrumentation process. The instrumentation process is as follows. A native agent in the observed VM is using the JVMTI interface to intercept newly loaded application classes. The intercepted classes are sent to the second VM for instrumentation. As the instrumentation process is separated from the observed VM, it can safely run in Java space. When instrumented, classes are sent back and loaded by the observed VM.

In summary, DiSL provides the language and the framework for easy and efficient instrumentation programming in Java [76]. The instrumentation developer has the ability to intercept any block of code either with a predefined library of *Markers* or through the extensible marking interface. Customizable *StaticContext* allows to pre-compute arbitrary static information during weave time and efficiently access the computed information during runtime. The *DynamicContext* interface exposes dynamic context information without any hidden memory

[1]Thread local variables in current implementation do not work without instrumentation done in a bootstrap phase.

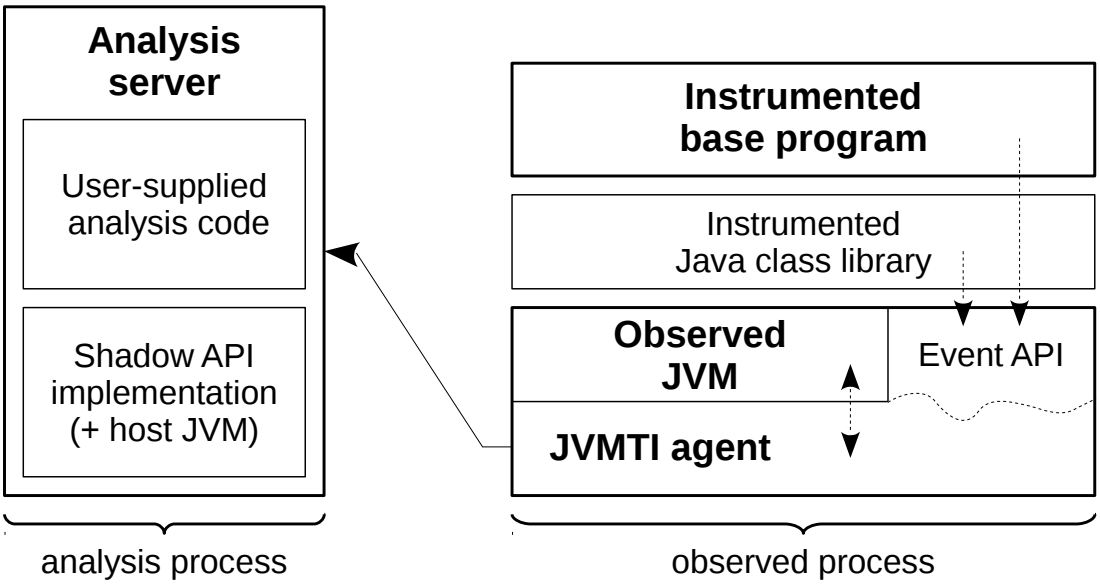


Figure 2.2: An architecture of the ShadowVM dynamic analysis evaluation framework.

allocation. Weave-time scope restriction is enabled using the *Scope* and *Guard* constructs while dynamic condition evaluation can be inserted directly in instrumentation code.

The DiSL framework itself does not provide an environment for dynamic analysis evaluation. It provides the simple dynamic bypass mechanism to prevent triggering of dynamic analysis events from the instrumentation. Even though such mechanism offers basic protection, it does not comprehensively solve all problems connected to in-process analysis evaluation.

The purpose of this section was to provide only a brief overview of DiSL. A more detailed description together with an evaluation can be found in the included paper called *DiSL: A Domain-Specific Language for Bytecode Instrumentation*.

2.2 ShadowVM, framework for remote dynamic analysis evaluation

ShadowVM is a framework for offloading analysis evaluation out of the context of the observed application. The motivation for designing ShadowVM was to resolve all problems connected to in-process analysis evaluation [59]. The Holy Grail would be to have a system performing evaluation outside of the context of the observed application but still maintaining performance and context availability as with in-process analysis. Because this is not possible without heavy modifications to production JVMs, ShadowVM introduces several compromises to provide a convenient evaluation environment while maintaining reasonable performance.

Figure 2.2 illustrates an architecture of ShadowVM. Similarly to DiSL, ShadowVM uses two virtual machines to prevent perturbation of the observed application. One JVM (the observed VM) is running the native agent responsible for marshaling events from the observed application, while a second JVM (ShadowVM) is performing the evaluation.

Event propagation

In the in-process analysis, the event interface between the instrumentation and the evaluation part is often implemented as a simple method call with the context information passed as method arguments. The evaluation has a full access to the context of the observed application including threads, heap and class hierarchy. In the case of the out-of-process analysis, all context information required for the evaluation has to be transferred from the observed application to the evaluation. In theory, the out-of-process analysis interface could be fully transparent but it would require either to rebuild the whole state of the application in the evaluation VM or stop the JVM so the context can be queried on request.

To significantly reduce the event offloading overhead, ShadowVM communicates with the observed VM asynchronously. When an event is triggered in the observed VM, the event data is marshaled and buffered and an application thread is able to proceed. As the observed application starts changing its state before the event is evaluated, the event should contain all the context information required for its processing. In other words, the analysis is not able to query any additional[2] context information during evaluation.

An interface providing event offloading on the observed VM looks like a standard method invocation. The method is just a stub for a set of java-to-native method calls responsible for marshaling, buffering and sending of triggered events. On ShadowVM, the event is un-marshaled and scheduled for the evaluation.

ShadowVM poses no restrictions on the transferred types, however only Java basic types (byte, int, long, double, ...) are transferred to ShadowVM in the same form as they were sent. ShadowVM does not replicate field values of an object instead, only a unique object identifier and class information are transferred.

Shadow API

On the ShadowVM side, objects and classes are accessible through Shadow API. Upon arrival, each object is recreated as a *ShadowObject*. The *ShadowObject* does not provide any access to the original object's methods or fields. The only accessible information is the identity and the class hierarchy information of the corresponding object. If some other information like a field value is required, it needs to be transferred separately.

The Shadow API provides special handling for references of type *java.lang.String*[3], *java.lang.Thread*[4] and *java.lang.Class*[5]. Each reference is recreated as the corresponding *ShadowObject*, providing access to additional information. The *ShadowString* object provides access to the original *String* value, the *ShadowThread* object allows to access part of the corresponding thread information and the *ShadowClass* mirrors the original *java.lang.Class* interface for accessing class information.

[2]The exception is class information, which is accessible during evaluation through the Shadow API.

[3]`http://docs.oracle.com/javase/7/docs/api/java/lang/String.html`

[4]`http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html`

[5]`http://docs.oracle.com/javase/7/docs/api/java/lang/Class.html`

Event oredering

A threading model of the in-process analysis is usually same as of the observed application. As the interface between the instrumentation and the evaluation is a simple method call, application threads executing the instrumentation are also used to perform the evaluation. In ShadowVM, the use of asynchronous communication opens a possibility for different event ordering models during evaluation.

ShadowVM allows to specify ordering model determining how an event is buffered, transferred and processed. The default (per-thread) ordering guarantees that events produced by the same thread on the observed VM are processed in the same order in ShadowVM. This ordering is similar to in-process analysis with an exception that no guarantees are made in respect to other events produced by different threads, even if the threads are synchronized. When a more fine grained control over the ordering is needed, the ShadowVM allows to specify an ordering group for each event. The event ordering group ensures that events sharing the same event group are processed in the same order in which they are produced. The default per-thread ordering may be seen as a special case of group ordering where the events produced by the same thread share the same event group. Using event groups, the analysis developer may easily create a global ordering among all the produced events by letting them share one global event group.

On ShadowVM, events from the same event group are processed by the same thread. For convenience, ShadowVM maintains one-to-one mapping between event groups and event processing threads, therefore it allows to use thread features like thread-local variables.

Even though the ordering specification may slightly complicate the event offloading process, it brings benefits during event evaluation. Locking required during event offloading is already done by the ShadowVM framework. In the scenario, where events are buffered to only one event group (simulating total ordering), ShadowVM takes care about all the necessary locking. It is guaranteed, that events are processed by a single thread no matter what threading model is used in the observed application. In contrast, in-process analysis cannot do any assumption about the threading model used by the observed application and locking has to be adapted to the worst-case scenario.

Life-cycle events

During the evaluation, analysis process thousands of events resulting in thousands of *ShadowObject*s created on the ShadowVM side. *ShadowObject*s may be referenced from an analysis data structure and the structure grows infinitely as the analysis processes more events. In-process analysis usually tracks analysed objects using weak references and deallocates structures connected to the analysed objects at the time the object is garbage collected. It also registers shutdown hooks to do final processing during observed VM shutdown.

To provide a similar cleanup mechanism, the Shadow API introduces two kinds of life-cycle events. An object lifetime event is triggered every time an object referenced by a *ShadowObject* is reclaimed. It is guaranteed that the object lifetime event is triggered as the last event referencing such an object and may be used as a cleanup event. A VM death life-cycle event is triggered at the

end of the analysis when the observed VM is shutting down. The VM death life-cycle event is meant for final cleanup and no other events arrive after VM death.

In the ShadowVM programming model, event context propagation can be seen as limiting. Especially the propagation of object references, where the analysis needs to propagate all requested fields of an object separately. The propagation of object state could be automated in the framework, however it would hit another limit where each reference stored in a field would be again only the *ShadowObject* without any data. A solution would be either propagate the whole reachable state at the time an event is generated, or propagate every observable state change of the application. We believe that the analysis developer should have full control over the information being transferred and an automated solution should be used only when truly required.

To provide better isolation, ShadowVM offloads the dynamic analysis out of the context of the observed application. The Shadow API provides access to objects and class information and introduces life-cycle events to support long-running analyses. ShadowVM introduces several mechanisms to improve the performance and increase the usability of the platform. The analysis developer is granted full control over the data being transmitted to eliminate unnecessary data transfers. The events are transmitted and evaluated asynchronously to decrease the execution time required for the application threads to dispatch the events. To reduce lock contention while buffering the events, ShadowVM introduces several event ordering models.

The purpose of this section was to provide only a brief overview of ShadowVM. A more detailed description together with an evaluation can be found in the included paper called *ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform*.

Part II

Collection of Papers

Preface

Foundations of this thesis were published on several research conferences. In the following chapters, we include full version of the selected publications.

The paper called *DiSL: A Domain-Specific Language for Bytecode Instrumentation* describes the DiSL instrumentation language and its implementation in the DiSL framework. The evaluation part of the paper compares DiSL to ASM and AspectJ, often used instrumentation tools. The author of this thesis largely contributed to the design of the DiSL language. He was also the lead developer and one of the two main authors of the DiSL framework and substantially contributed to the text of the paper.

The second paper, *ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform*, describes the framework for offloading dynamic analyses out of the context of the observed application and assess its performance compared to the in-process analysis. The author of this thesis designed and implemented most of the framework and substantially contributed to the text of the paper.

The third paper, called *Introduction to Dynamic Program Analysis with DiSL*, is a complete presentation of DiSL. It introduces most features of DiSL based on examples and demonstrates the benefits of DiSL on several case studies. The author of this thesis prepared the demonstration of DiSL and its description. The case studies and their evaluation, together with the implementation of the examples, were prepared by the co-authors.

A complete list of all author's publications can be found at the end of the thesis.

Chapter 3
DiSL: A Domain-Specific Language for Bytecode Instrumentation

Lukáš Marek,
Alex Villazón,
Yudi Zheng,
Danilo Ansaloni,
Walter Binder,
Zhengwei Qi

Contributed paper at the **11th Annual International Conference on Aspect-oriented Software Development (AOSD 2012)**.

In conference proceedings,
published by ACM,
pages 239-250,
ISBN 978-1-4503-1092-5,
March 2012.

The original version is available electronically from the publisher's site at `http://dx.doi.org/10.1145/2162049.2162077`.

DiSL: A Domain-Specific Language for Bytecode Instrumentation

Lukáš Marek
Charles University, Czech Republic
lukas.marek@d3s.mff.cuni.cz

Alex Villazón
Universidad Privada Boliviana, Bolivia
avillazon@upb.edu

Yudi Zheng
Shanghai Jiao Tong University, China
zheng.yudi@sjtu.edu.cn

Danilo Ansaloni Walter Binder
University of Lugano, Switzerland
{danilo.ansaloni, walter.binder}@usi.ch

Zhengwei Qi
Shanghai Jiao Tong University, China
qizhwei@sjtu.edu.cn

Abstract

Many dynamic analysis tools for programs written in managed languages such as Java rely on bytecode instrumentation. Tool development is often tedious because of the use of low-level bytecode manipulation libraries. While aspect-oriented programming (AOP) offers high-level abstractions to concisely express certain dynamic analyses, the join point model of mainstream AOP languages such as AspectJ is not well suited for many analysis tasks and the code generated by weavers in support of certain language features incurs high overhead. In this paper we introduce DiSL (domain-specific language for instrumentation), a new language especially designed for dynamic program analysis. DiSL offers an open join point model where any region of bytecodes can be a shadow, synthetic local variables for efficient data passing, efficient access to comprehensive static and dynamic context information, and weave-time execution of user-defined static analysis code. We demonstrate the benefits of DiSL with a case study, recasting an existing dynamic analysis tool originally implemented in AspectJ. We show that the DiSL version offers better code coverage, incurs significantly less overhead, and eases the integration of new analysis features that could not be expressed in AspectJ.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

General Terms Languages, Measurement, Performance

Keywords Bytecode instrumentation, dynamic program analysis, aspect-oriented programming, JVM

AOSD'12, March 25–30, 2012, Potsdam, Germany.

1. Introduction

Dynamic program analysis tools support numerous software engineering tasks, including profiling, debugging, testing, program comprehension, and reverse engineering. Despite of the importance of dynamic analysis, prevailing techniques for building dynamic analysis tools are based on low-level abstractions that make tool development, maintenance, and customization tedious, error-prone, and hence expensive. For example, many dynamic analysis tools for the Java Virtual Machine (JVM) rely on bytecode instrumentation, supported by a variety of bytecode engineering libraries that offer low-level APIs resulting in verbose implementation code.

In an attempt to simplify the development of dynamic analysis tools, researchers have explored the use of aspect-oriented programming (AOP) languages, such as AspectJ [16]. Examples of aspect-based dynamic analysis tools are the DJProf profilers [20], the RacerAJ data-race detector [10], and the Senseo Eclipse plugin for augmenting static source code views with dynamic information [21]. However, as neither mainstream AOP languages nor the corresponding weavers have been designed to meet the requirements of dynamic program analysis, the success of using AOP for dynamic analysis remains limited. For example, in AspectJ, join points that are important for dynamic program analysis (e.g., the execution of bytecodes or basic blocks) are missing, access to reflective dynamic join point information is expensive, data passing between woven advice in local variables is not supported, and the mixing of low-level bytecode instrumentation and high-level AOP code is not foreseen.

In this paper, we introduce DiSL, a new domain-specific language for bytecode instrumentation. DiSL relies on AOP principles for concisely expressing efficient dynamic analysis tools. The language provides an open join point model defined by an extensible set of bytecode markers, efficient access to static and dynamic context information, optimized processing of method[1] arguments, and synthetic local vari-

[1] In this paper, "method" stands for "method or constructor".

ables for efficient data passing. While DiSL significantly raises the abstraction level when compared to prevailing bytecode manipulation libraries, it also exposes a low-level API to implement new bytecode markers. The DiSL weaver guarantees complete bytecode coverage to ensure that analysis results represent overall program execution. DiSL follows similar design principles as @J [8], an AOP language for dynamic analysis, which however lacks an open join point model and efficient access to method arguments.

Compared to high-level dynamic analysis frameworks such as RoadRunner [13] or jchord[2] that restrict the locations that can be instrumented, DiSL offers the developer fine-grained control over the inserted bytecode; that is, DiSL is not tailored for any specify dynamic analysis task, but provides constructs for concisely expressing any bytecode instrumentation. Instrumentation sites can be specified with a combination of bytecode markers, scoping expressions, and guards; guards represent static analyses executed at weave-time. Instrumentation code is provided in the form of snippets, that is, code templates that are instantiated for each selected instrumentation site and inlined. Snippets may access synthetic local variables to pass data from one instrumentation site to another. Snippets may access any static or dynamic context information; they may also process an arbitrary number of method arguments in a custom way.

The scientific contributions of this paper are twofold:

1. We present our design goals, the DiSL language constructs, and the implementation of the DiSL weaver.
2. We present a case study to illustrate the benefits of DiSL. We recast Senseo [21, 22] in DiSL; Senseo is an AspectJ-based profiling tool that supports various software maintenance tasks. In contrast to the former AspectJ implementation, the DiSL version of the tool features complete bytecode coverage, introduces significantly less overhead, and can be easily extended to collect additional dynamic metrics on the intra-procedural control flow.

This paper is structured as follows: Section 2 describes the design goals underlying DiSL. Section 3 gives a detailed overview of the DiSL language constructs. The software architecture of the DiSL weaver and its implementation are discussed in Section 4. Our case study is introduced in Section 5 and evaluated in Section 6. Section 7 discusses related work, Section 8 summarizes the strengths and limitations of DiSL, and Section 9 concludes.

2. Design of DiSL

Designing a good language for instrumentation-based dynamic program analysis is challenging, because we need to reconcile three conflicting design goals: (1) high expressiveness of the language, (2) a convenient, high-level programming model, and (3) high efficiency of the developed analysis tools. On the one hand, existing bytecode manipulation libraries meet the first and the third goal, but provide only low-level abstractions that make tool development cumbersome. On the other hand, mainstream AOP languages achieve the second goal, but lack expressiveness (e.g., lack of join points that would allow tracing the intra-procedural control flow) and suffer from inefficiencies (e.g., access to dynamic reflective join point information may require the allocation of unnecessary objects). The design of DiSL aims at bridging the gap between low-level bytecode manipulation frameworks and high-level AOP. Below, we motivate the main design choices underlying DiSL.

Open join point model. DiSL allows any region of bytecodes to be used as a join point, thus following an open join point model. That is, the set of supported join point *shadows* [15] is not hard-coded. To enable the definition of new join points, DiSL provides an extensible mechanism for marking user-defined bytecode regions (i.e., shadows).

Compatibility with Java and the JVM. DiSL is a domain-specific embedded language which has Java as its host language. DiSL instrumentations are implemented in Java, and annotations are used to express where programs are to be instrumented. Dynamic analysis tools written in DiSL can be compiled with any Java compiler and executed on any JVM.

Advice inlining and data passing in synthetic local variables. Advice in DiSL are expressed in the form of code *snippets* that are inlined, giving the developer fine-grained control over the inserted code. DiSL *instrumentations* (corresponding to aspects in AOP) describe where snippets are to be inserted into the base program. Thanks to inlining, snippets woven into the same method are able to efficiently communicate data through *synthetic local variables* [6].

Efficient access to complete static and dynamic context information. In DiSL, all static context information is exposed to the developer. This feature is similar to AspectJ's static reflective join point information (offering class and method properties), but exposes additional information at the basic block and bytecode level. DiSL also supports user-defined static analysis to compute further static context information at weave-time. In addition, DiSL provides a simple, yet powerful reflective API to gather dynamic context information which gives access to local variables and to the operand stack, supporting also efficient access to an arbitrary number of method arguments.

No support for around advice. Mainstream AOP languages support advice execution *before*, *after*, and *around* join points. Three common use cases of around advice are (1) passing data around a join point, (2) skipping a join point, and (3) executing a join point multiple times. As we assume that instrumentations do not alter the control flow in the base program, only the first use is relevant for us. However, for the first use case, the same behavior can be achieved with before and after advice using synthetic local variables [6].

[2] `http://code.google.com/p/jchord/`

Hence, DiSL only supports before and after advice, which helps keep the weaver simple.

Complete bytecode coverage. DiSL is designed for weaving with complete bytecode coverage. That is, the DiSL weaver ensures that all methods that have a bytecode representation can be woven, including methods in the standard Java class library. To this end, the DiSL weaver relies on implementation techniques developed in previous work [19].

3. Language Features

In this section we give an overview of the language features of DiSL. In Section 3.1 we introduce DiSL instrumentations specified in the form of snippets; markers determine where snippets are woven in the bytecode. The mechanism to control the inlining order of snippets is explained in Section 3.2. Synthetic local variables for efficiently passing data between woven snippets are presented in Section 3.3, and efficient access to thread-local variables is discussed in Section 3.4. In Section 3.5 we introduce static context to provide static reflective information, and we present the reflective API for obtaining dynamic context information in Section 3.6. In Section 3.7 we explain DiSL's support for method arguments processing. In Section 3.8 we introduce guards that enable the evaluation of conditionals at weave-time to decide whether a join point is to be captured, as well as a scoping construct to restrict weaving.

3.1 Instrumentations, Snippets, and Markers

DiSL *instrumentations* are Java classes. An instrumentation can only have *snippets* that are static methods annotated with `@Before`, `@After`, `@AfterReturning`, or `@AfterThrowing`. Snippets are defined as static methods, because their body is used as a template that is instantiated and inlined at the matching join points in the base program. Snippets do not return any value and must not throw any exception (that is not caught by a handler in the snippet).

Because of DiSL's open join point model, pointcuts are not hardcoded in the language but defined by an extensible library of *markers*. Markers are standard Java classes implementing a special interface for join point selection. DiSL provides a rich library of markers including those for method body, basic block, individual bytecode, and exception handler. In addition, the developer may extend existing markers or implement new markers from scratch.

The marker class is specified in the `marker` attribute in the snippet annotation. The weaver takes care of instantiating the selected marker, matching the corresponding join points, and weaving the snippets.

In addition to the predefined markers, DiSL offers join point extensibility by exposing the internal representation of method bodies to the developer, who has to implement code to *mark* the bytecode regions defining the shadows for the new join points.

3.2 Control of Snippet Order

It is common that several shadows coincide in the starting instruction, that is, several snippets may apply to the same join point. Similar to AspectJ's *advice precedence* resolution, DiSL provides a simple mechanism to control snippet ordering through the `order` attribute in the snippet annotation. The order is specified as a non-negative integer value. For `@Before`, snippets with higher order are inlined before snippets with lower order. For `@After`, `@AfterReturning`, and `@AfterThrowing`, snippets with lower order are inlined before snippets with higher order. Thus, the order indicates "how close" to the shadow the snippet shall be inlined.

3.3 Synthetic Local Variables

DiSL provides an efficient communication mechanism to pass arbitrary data between snippets. The mechanism relies on inlining so as to store the data in a local variable, which is therefore visible in the scope of the woven method body. DiSL provides the `@SyntheticLocal` annotation to specify the holder variable. Synthetic local variables must be declared as static fields and can be used in any snippet. The weaver takes care of translating the static field declared in the instrumentation into a local variable in each instrumented method, and of replacing the bytecodes that access the static field with bytecodes that access the introduced local variable. For details, we refer to [6].

3.4 Thread-local Variables

DiSL supports thread-local variables with the `@ThreadLocal` annotation. This mechanism extends `java.lang.Thread` by inserting the annotated field. While the inserted fields are instance fields, thread-local variables must be declared as static fields in the instrumentation class, similar to synthetic local variables. These fields must be initialized to the default value of the field's type.[3] The DiSL weaver translates all access to thread-local variables in snippets into bytecodes that access the corresponding field of the currently executing thread. An `inheritable` flag can be set in the `@ThreadLocal` annotation such that new threads "inherit" the value of a thread-local variable from the creating thread. Note that the standard Java class library offers classes with similar semantics (`java.lang.ThreadLocal` and `java.lang.InheritableThreadLocal`). However, accessing fields directly inserted into `java.lang.Thread` results in more efficient code.

3.5 Static Context Information

Accessing static context information is essential for dynamic analyses, for example, gathering information about

[3] During JVM bootstrapping, in general, inserted code cannot be executed because it may introduce class dependencies that can violate JVM assumptions concerning the class initialization order. Hence, threads created during bootstrapping could not initialize inserted thread-local fields in the beginning.

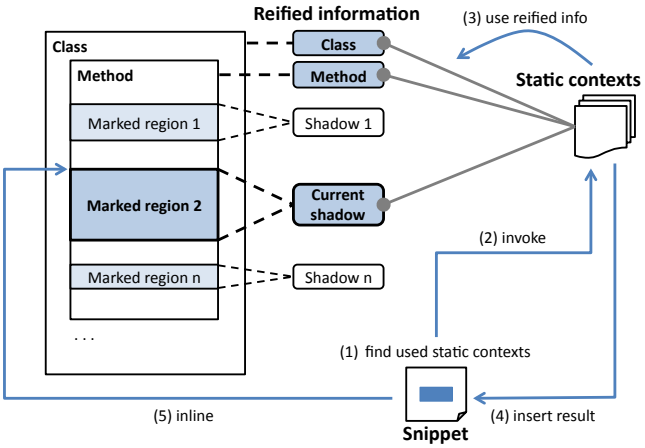


Figure 1. Gathering static context information at weave-time

```
public class CallingContextBBAnalysis {
  @ThreadLocal
  static CCTNode currentNode;

  @SyntheticLocal
  static CCTNode callerNode;

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

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

  @Before(marker = BasicBlockMarker.class, order = 0)
  static void onBasicBlock(BasicBlockStaticContext bbsc) {
    currentNode.profileBB(bbsc.getBBIndex());
  }
}
```

Figure 2. Sample instrumentation for calling context-aware basic block profiling (class CCTNode is not shown)

the method, basic block, or bytecode instruction that is executed. Because of the open join point model of DiSL, there is no bound static part of a join point as in AspectJ. In DiSL, the programmer can gather reflective static information at weave-time by using various *static contexts*. DiSL provides a library of commonly used static contexts such as `MethodStaticContext`, `BasicBlockStaticContext`, and `BytecodeStaticContext`. The developer may also implement custom static context classes.

For every snippet, the programmer can specify any number of static contexts as argument. Each static context class implements the `StaticContext` interface and provides methods without argument that must return a value of a Java primitive type or a string. The reason for this restriction is that DiSL stores the results of static context methods directly in the constant pool of the woven class. Static contexts receive read-only access to the shadow containing the following reflective information: the class and method under instrumentation, the snippet, and the beginning and ending positions of the current shadow.

Figure 1 depicts the reflective approach for gathering static context information. After shadow marking according to the selected marker, the snippet is parsed to locate invocations to static context methods (step 1). Static contexts are then instantiated by the weaver and the corresponding methods are invoked for every shadow (step 2). Static context methods access the exposed reflective data to compute the static information to be returned (step 3). The weaver replaces the invocation of the static context methods in the snippet with bytecodes to access the computed static information (step 4). The snippet code is inlined before or after the matching shadows (step 5).

Figure 2 shows how static contexts are used in an instrumentation for calling context-aware basic block analysis. The goal is to help developers find hotspots in their programs taking both the inter- and intra-procedural control flow into account. The presented instrumentation collects statistics on basic block execution for each calling context.

For storing inter-procedural calling context information, a Calling Context Tree (CCT) [3] is used. For each thread, the current CCT node is kept in the thread-local variable `currentNode` that is updated upon method entry and completion (`onMethodEntry(...)` and `onMethodCompletion()` snippets using the `BodyMarker`). The synthetic local variable `callerNode` is used to store the CCT node corresponding to the caller. The `CCTNode.getRoot()` method returns the root node of the CCT. The method `profileCall(...)` takes a method identifier as argument and returns the corresponding callee node in the CCT. The method identifier is obtained from the `MethodStaticContext`; it is inserted as a string in the constant pool of the woven class.[4]

The `onBasicBlock()` snippet captures all basic block join points using the `BasicBlockMarker`. The idea is to count how many times each basic block is executed, so as to detect hot basic blocks. To this end, the snippet uses the `BasicBlockStaticContext` for gathering the index of the captured basic block. This value is used to increment the corresponding counter in the CCT node (not shown). Note that the `order` of the `@Before` snippets ensures that the initialization of the synthetic local variable `callerNode` and the update of the thread-local variable `currentNode` are done at the very beginning of the method body, before they are accessed in the first basic block.

3.6 Dynamic Context Information

Access to dynamic join point information (e.g., `getThis()`, `getTarget()`, and `getArgs()` in AspectJ) requires gathering data from *local variables* and from the *operand*

[4] This is similar to the use of `JoinPoint.StaticPart` in AspectJ. While AspectJ inserts static fields in the woven class to hold reflective static join point information, DiSL avoids structural modifications of the woven class.

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

Figure 3. DynamicContext interface

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

Figure 4. Profiling array access

stack [15]. DiSL provides an API to explicitly access this information. Figure 3 shows the `DynamicContext` API which provides reflective information through the `getLocalVariableValue`(...) to access a local variable, `getStackValue`(...) to access a stack value, and `getThis`() returning this object or null in the case of a static method. Similar to static contexts, the `DynamicContext` can be passed to snippets as an argument. The programmer must provide the index and the type of the data to access. Note that the use of `DynamicContext` is not restricted to any particular marker. The developer must know how to access the correct data from local variables or from the operand stack. The weaver takes care of translating calls to the API methods into bytecode sequences to retrieve the desired values.

An example of the use of `DynamicContext` is access to the return value of a method, which is on top of the stack upon normal method completion. The programmer may implement an `@AfterReturning` snippet with the `BytecodeMarker` (for different `return` bytecodes) and use `getStackValue`(0,...) to retrieve the return value. The index zero indicates the top of the stack.

The combination of `DynamicContext` with the `BytecodeMarker` provides a powerful mechanism to gather join point information for implementing dynamic analysis tools, such as memory profilers. For example, Figure 4 shows how to capture array accesses, which is not possible in AspectJ. The `beforeArrayStore`(...) snippet captures all objects being stored in arrays, where the element type is a reference type. The profiler can keep track which object has been stored at which position of an array. Before every `aastore` bytecode, the snippet gets the `array`, the `index`[5] where the element will be stored, and the object to be `stored` from the operand stack (at positions 2, 1, and 0, respectively). The `process`(...) method processes the collected information (not shown).

[5] The use of Java generics in the API results in autoboxing of primitive values (e.g., `index`) in the compiled snippet. The DiSL weaver removes the unnecessary boxing code before inlining.

```
public interface ArgumentProcessorContext {
  Object getReceiver(ArgumentProcessorMode mode);
  Object[] getArgs(ArgumentProcessorMode mode);
  void apply(Class<?> argumentProcessor,
             ArgumentProcessorMode mode);
}

public enum ArgumentProcessorMode {
  METHOD_ARGS, CALLSITE_ARGS
}
```

Figure 5. Argument processor API

3.7 Argument Processors

Method arguments are retrieved from local variables or, in the case of call sites, from the operand stack. DiSL's `DynamicContext` can be used to access these values when the argument index and type are known, which is not always the case. DiSL also provides a reflective mechanism, called *argument processor*, to process all arguments by their types.

The `ArgumentProcessorContext` interface (see Figure 5) can be used within snippets to access method arguments; it is to be passed to snippets as an argument, similar to static contexts or `DynamicContext`. Two modes can be specified, to process either arguments of the method where the snippet is inlined (`METHOD_ARGS`), or arguments of a method invocation (`CALLSITE_ARGS`). The `getReceiver`(...) method returns the receiver, or `null` for static methods. The `getArgs`(...) method returns all arguments in an object array, similar to `JoinPoint.getArgs`() in AspectJ for `execution` respectively `call` pointcuts. However, if the programmer needs to selectively access arguments, or does not want them to be wrapped in an object array (e.g., for performance reasons or to preserve the original type for arguments of primitive types), the API provides the `apply`(...) method, where the programmer can specify an argument processor class that handles the generation of code to access the arguments.

Argument processors are classes annotated with `@ArgumentProcessor`. At weave-time, DiSL checks which argument processor is selected in the snippet, and for each matching join point, generates the code to process the arguments according to their types.

Argument processors must implement static `void` methods, where the first parameter is required and additional (optional) parameters may be passed. The type of the first parameter selects the type of argument to be captured. The first parameter's type can only be `java.lang.Object` or a primitive type. For each argument of the woven method, the weaver checks whether the selected argument processor has a method where the first parameter type matches the current method argument type. In this case, the weaver generates the code to access the corresponding argument, which is eventually inlined within the snippet. As additional parameters, the argument processor method can take any static context, `DynamicContext`, or `ArgumentContext`. `ArgumentContext` is an interface to access argument type, argument position, and the total number of arguments.

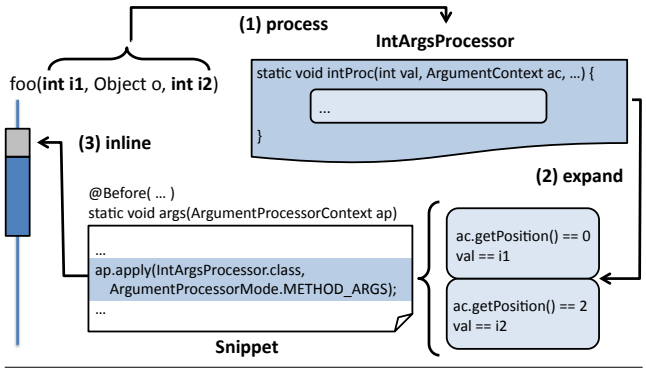


Figure 6. Processing of integer arguments

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

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

Figure 7. Snippet guard restricting weaving to methods that return objects

Figure 6 illustrates the weaving of a snippet before a join point in method `foo`(...). In this example, the developer only wants to process arguments of type `int`. For method `foo`(...), only two of the arguments will match the `intProc`(...) processor method (`i1` and `i2`). First, the weaver finds out which argument processor and mode should be applied to the snippet (step 1). Then, the invocation to `apply`(...) in the snippet is replaced with the expanded method bodies of the processor for each matching argument (step 2). In the example, the generated code will give access to the two integer arguments, i.e., the snippet will contain expanded processor code to access the values `i1` and `i2`. Finally, the expanded snippet is inlined (step 3). For `METHOD_ARGS`, the generated code retrieves the arguments from local variables; for `CALLSITE_ARGS`, the arguments are taken from the operand stack. The use of `CALLSITE_ARGS` throws a weave-time error if the snippet is not woven before a method invocation bytecode.

There are several advantages of using argument processors compared to, for example, `JoinPoint.getArgs`() in AspectJ. Firstly, there is no need for creating objects that hold dynamic join point information. DiSL efficiently takes the correct values directly from local variables or from the stack. Secondly, argument types are preserved. The values of primitive types are not boxed as in AspectJ. Finally, it is straightforward to apply argument processors to a subset of arguments, without requiring complex pointcuts to be written. We will illustrate these advantages in more detail with our case study and evaluation in Sections 5 and 6.

3.8 Guards and Scope

DiSL provides two complementary mechanisms for restricting the application of snippets. The first one, *guard*, is based on weave-time evaluation of conditionals. The second one, *scope*, is based on method signature matching.

Guards allow us to evaluate complex weave-time restrictions for individual join points. A guard has to implement a static method annotated with `@GuardMethod`. The guard method may take any number of static contexts as arguments. The guard method returns a boolean value indicating whether the current joint point is to be instrumented. Static contexts can be used to expose reflective weave-time information to the guard. The guard has to be specified with the `guard` attribute of the snippet annotation.

In contrast to AspectJ's `if` pointcut, the evaluation of guards is done for each join point at weave-time. This avoids runtime overhead due to the evaluation of statically known conditionals. To illustrate this point, let's consider the example shown in Figure 7. The programmer wants to restrict weaving only to methods returning objects; methods returning values of primitive types (or `void`, which we consider a primitive type here) shall not be woven. The `evalGuard`(...) method of the `MethodReturnsRef` guard uses `ReturnTypeStaticContext` to determine whether the return type of the instrumented method is primitive. Because this evaluation is performed at weave-time, the `onMethodEntry`(...) snippet will be inlined only in methods that return objects.

Another interesting example of weave-time conditional evaluation is the use of data flow analysis within guards. This feature helps avoid inlining snippets that would otherwise access uninitialized objects (passing an uninitialized object to another method as argument would be illegal and cause a verification failure). For example, the programmer may capture all `putfield` bytecodes in constructors, where the target is a properly initialized object. Consequently, `putfield` bytecodes that write to the object under initialization before invocation of the superclass constructor will not be captured.

Even though guards are expressive, in many common cases, a more concise scoping expression is sufficient. In DiSL, scope is a simplified signature pattern matching pointcut designator. The `scope` attribute of the snippet annotation specifies which methods shall be instrumented. Scope expressions specify method, class, or package names and may contain wildcards (e.g., `scope` = "$*$ `java.io.`$*$(..)"). Typically, scope evaluation is faster than guard evaluation, as it is done only once for each method. In contrary, a guard has to be invoked (using reflection) for each join point in the method. The best combination is the usage of scope expressions for fast method filtering and of guards for fine-grained joint point selection.

4. Implementation

DiSL is implemented in Java using the ASM[6] bytecode manipulation library in about 100 classes and 8000 lines of code. The DiSL weaver[7] runs on top of jBORAT[8], a lightweight toolkit providing support for instrumentation with complete bytecode coverage [19]. jBORAT uses two JVMs: an *instrumentation JVM* where bytecode instrumentation is performed and an *application JVM* that executes the instrumented application. This separation of the instrumentation logic from the instrumented application reduces perturbations in the application JVM (e.g., class loading and initialization triggered by jBORAT or by the DiSL weaver do not happen within the application JVM). DiSL simplifies deployment with scripts, hiding the complex JVM setup from the user.

Figure 8 gives an overview of the DiSL weaver running on top of jBORAT. During initialization, DiSL parses all instrumentation classes (step 1). Then it creates an internal representation for snippets and initializes the used markers, guards, static contexts, and argument processors. When DiSL receives a class from jBORAT (step 2), the weaving process starts with the snippet selection. The selection is done in two phases, starting with scope matching (step 3) and followed by shadow creation and selection. Shadows are created using the markers associated with the snippets selected in the previous scope matching phase. Shadows are evaluated by guards and only snippets with at least one valid shadow are selected (step 4). At this point, all snippets that will be used for weaving are known. Static contexts are used to compute the static information required by snippets (step 5). Argument processors are evaluated for snippets, and argument processor methods that match method arguments are selected (step 6). All the collected information is finally used for weaving (step 7). Argument processors are applied, and calls to static contexts are replaced with the computed static information. The weaver also generates the bytecodes to access dynamic context information. Finally, the woven class is emitted and passed back to jBORAT (step 8).

5. Case Study: Senseo

In this section, we illustrate the benefits of DiSL by recasting *Senseo* [21], a dynamic analysis tool for code comprehension and profiling. Senseo uses an aspect written in AspectJ for collecting calling context-sensitive dynamic information for each invoked method, including statistics on the runtime types of method arguments and return values, the number of method invocations, and the number of allocated objects. These metrics are visualized by an Eclipse plugin[9] that enriches the static source code views with the collected dynamic information. Senseo helps developers understand the dynamic behavior of applications and locate performance problems.

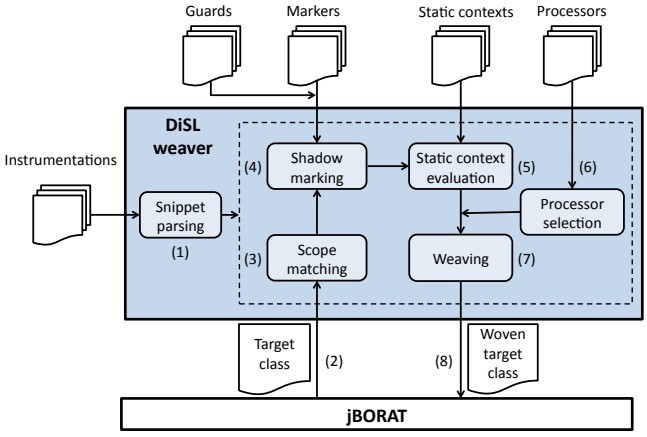


Figure 8. Overview of DiSL weaving process

The original version of Senseo has two main limitations: (1) lack of intra-procedural profiling and (2) high overhead for metrics collection. Both limitations stem from the use of AspectJ to express the instrumentation. Because of the absence of join points at the level of basic blocks, dynamic metrics on the intra-procedural control flow are missing, making it difficult for the developer to locate hot methods with complex intra-procedural control flow that are not invoked frequently. Moreover, access to dynamic join point information is inefficient due to the boxing of primitive values and because of the allocation of object arrays, notably for processing method arguments. For example, although only the first argument of method `paint(Object o, int x, int y)` could receive objects of different runtime types, the AspectJ implementation of Senseo collects the runtime types of all three arguments upon each invocation, because `JoinPoint.getArgs()` returns all arguments in a newly created object array, boxing values of primitive types.

Figure 9 shows the (simplified) DiSL instrumentation `Senseo2` that overcomes the limitations of the previous AspectJ implementation. To collect dynamic metrics for each calling context, the `onMethodEntry(...)` and `onMethodCompletion()` snippets reify the calling context in a similar way as explained in Section 3.5 (Figure 2). Each CCT node stores the dynamic information collected within the corresponding calling context, as explained below.

Number of method executions. The counting of method executions is subsumed in the `onMethodEntry(...)` snippet and performed in the `profileCall(...)` method by incrementing a counter. This information is used to compute the number of method calls for each calling context.

Number of allocated objects and arrays. To count the number of allocated objects and arrays, the `onAllocation()`

[6] `http://asm.ow2.org/`

[7] `http://disl.origo.ethz.ch/`

[8] jBORAT stands for Java Bytecode Overall Rewriting and Analysis Toolkit.

[9] `http://scg.unibe.ch/research/senseo`

```
public class Senseo2 {
  @ThreadLocal
  static CCTNode currentNode;

  @SyntheticLocal
  static CCTNode callerNode;

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

    proc.apply(ReferenceProcessor.class,
               ProcessorMode.METHOD_ARGS);
  }

  @After(marker = BodyMarker.class, order = 2)
  static void onMethodCompletion() {
    currentNode = callerNode;
  }

  @AfterReturning(marker = BodyMarker.class, order = 1,
                  guard = MethodReturnsRef.class)
  static void onReturnRef(DynamicContext dc) {
    Object obj = dc.getStackValue(0, Object.class);
    currentNode.profileReturn(obj);
  }

  @AfterReturning(marker=BytecodeMarker.class, order=0,
        args = "new,newarray,anewarray,multianewarray")
  static void onAllocation() {
    currentNode.profileAllocation();
  }

  @Before(marker = BasicBlockMarker.class, order = 0)
  static void onBasicBlock(BasicBlockStaticContext bbsc){
    currentNode.profileBB(bbsc.getBBIndex());
  }
}

@ArgumentProcessor
public class ReferenceProcessor {
  static void objProc(Object obj, ArgumentContext ac) {
    Senseo2.currentNode.profileArgument(ac.getPosition(),
                                        obj);
  }
}
```

Figure 9. DiSL instrumentation for collecting runtime information for Senseo

snippet uses the `BytecodeMarker` to capture allocation bytecodes for both objects (`new`) and arrays (`newarray`, `anewarray`, and `multianewarray`). The `profileAllocation()` method updates an allocation counter in the current CCT node.

Runtime argument and return types. To collect runtime type information only for arguments of reference types, the `onMethodEntry(...)` snippet uses the argument processor `ReferenceProcessor`. Since this argument processor only defines the `objProc(...)` method to process arguments of reference types, all arguments with primitive types are automatically skipped. The `objProc(...)` method invokes the `profileArgument(...)` method of the current CCT node, passing the position of the argument and the reference.

For collecting runtime return types, the `onReturnRef(...)` snippet uses the `MethodReturnsRef` guard (see Figure 7 in Section 3.8) to ensure that the return type of a woven method is a reference type. Because the returned object reference is on top of the operand stack upon method completion, it is accessed with the `DynamicContext` API.

	DiSL	AspectJ	ASM
Physical lines-of-code	74	44	489
Logical lines-of-code	44	19	338

Table 1. Lines-of-code for three implementations of Senseo

Basic-block metrics. As the execution of basic blocks cannot be captured with AspectJ, the following information is collected only by the DiSL version of Senseo. The `onBasicBlock(...)` snippet captures every basic block using the `BasicBlockMarker`; the `BasicBlockStaticContext` provides the index of the captured basic block (`getBBIndex()`). This allows us to keep track how many times a basic block is executed in each calling context.

Comparing different Senseo implementations. For a comparison of DiSL with low-level bytecode manipulation libraries and with AOP, it is interesting to consider the lines-of-code (LOC) used in the different implementations of the same tool. Hence, we implemented a third version of Senseo with the ASM bytecode manipulation library and compared the source code of the DiSL, AspectJ, and ASM versions. In contrast to the DiSL and ASM versions, the AspectJ version lacks basic block profiling, that is, it offers less functionality.

Table 1 summarizes the physical and logical LOC metrics of the three implementations, considering only the code related to the actual instrumentation logic (and disregarding the Java code for analysis at runtime, which is common to all three implementations). Compared to ASM, the DiSL and AspectJ versions are significantly smaller, as the direct manipulation of bytecodes requires much more development effort than relying on the high-level pointcut/advice mechanism of AspectJ and DiSL. The higher LOC number of the DiSL implementation compared to the AspectJ version is mainly due to the separation of the code that is evaluated at weave-time (guards) from the instrumentation code (snippets). However, weave-time evaluation brings significant performance gains as we will show in Section 6.

In summary, our case study illustrates how DiSL enables the concise implementation of a practical dynamic analysis tool, thanks to DiSL's open join point model, efficient access to both static and dynamic context information, weave-time evaluation of conditionals, and argument processors. Dynamic analysis tools written in DiSL are much more concise than equivalent tools developed with bytecode manipulation libraries.

6. Performance Evaluation

In this section, we evaluate the runtime performance of the DiSL instrumentation presented in the Senseo case study.

	Reference	SenseoAJ application only		SenseoDiSL application only		SenseoDiSL full coverage		Senseo2 application only		Senseo2 full coverage	
	[s]	[s]	ovh.	[s]	ovh.	[s]	ovh.	[s]	ovh.	[s]	ovh.
avrora	5.11	30.96	6.06	12.61	2.47	12.41	2.43	13.66	2.67	14.62	2.86
batik	1.28	2.70	2.11	1.78	1.39	2.47	1.93	2.14	1.67	3.09	2.41
eclipse	16.16	152.92	9.46	70.73	4.38	81.52	5.04	152.41	9.43	163.36	10.11
fop	0.35	3.36	9.60	1.68	4.80	3.09	8.83	2.07	5.91	3.93	11.23
h2	5.84	63.25	10.83	25.27	4.33	31.78	5.44	29.55	5.06	41.81	7.16
jython	2.67	5.70	2.13	3.89	1.46	28.28	10.59	4.29	1.61	34.21	12.81
luindex	0.90	7.06	7.84	2.71	3.01	3.31	3.68	3.45	3.83	4.30	4.78
lusearch	1.98	13.09	6.61	5.49	2.77	6.57	3.32	6.19	3.13	8.85	4.47
pmd	2.05	10.09	4.92	5.10	2.49	7.60	3.71	6.54	3.19	10.31	5.03
sunflow	3.45	57.24	16.59	21.44	6.21	20.49	5.94	24.57	7.12	25.37	7.35
tomcat	1.97	4.46	2.26	3.16	1.60	6.70	3.40	3.87	1.96	9.32	4.73
tradebeans	5.56	71.48	12.86	30.76	5.53	76.43	13.75	42.90	7.72	117.40	21.12
tradesoap	6.77	25.40	3.75	12.80	1.89	53.60	7.92	17.30	2.56	76.12	11.24
xalan	1.11	20.39	18.37	8.15	7.34	11.38	10.25	10.08	9.08	17.33	15.61
geo. mean			6.47		3.09		5.26		3.91		7.19

Table 2. Execution times and overhead factors for SenseoAJ, SenseoDiSL, and Senseo2

First, we compare the previous AspectJ implementation with an equivalent DiSL instrumentation (i.e., without basic block metrics). In addition, we evaluate our DiSL instrumentation with full bytecode coverage, collecting also basic block metrics. Second, we explore the different sources of the measured overhead. Third, we investigate the differences in the collected profiles, considering the number of intercepted join points, when weaving only application code, respectively when weaving with full bytecode coverage. Fourth, we study weaving time and overall class loading latency due to jBORAT and DiSL.

For our measurements, both the DiSL weaver and the AspectJ weaver run on top of jBORAT. This ensures exactly the same weaving coverage for application code (otherwise, the AspectJ load-time weaver would exclude some application classes from weaving). Both the instrumentation JVM and the application JVM run on the same host. We use the benchmarks in the DaCapo suite (dacapo-9.12-bach)[10] as base programs in our evaluation. All measurements correspond to the median of 15 benchmark runs within the same application JVM. The measurement machine is an Intel Core2 Quad Q9650 (3.0 GHz, 8 GB RAM) that runs Ubuntu GNU/Linux 10.04 64-bit. We use AspectJ 1.6.11[11], DiSL pre-release version 0.9, and Oracle's JDK 1.6.0_27 Hotspot Server VM (64-bit) with 7 GB maximum heap size.

Table 2 reports the runtime overhead for the original AspectJ version of Senseo (SenseoAJ), for the equivalent instrumentation in DiSL, that is, without basic block metrics (SenseoDiSL), and for the DiSL instrumentation including basic block metrics (Senseo2). On average (geometric mean for DaCapo), the overhead factor introduced by SenseoAJ is 6.47, while for SenseoDiSL, with the same code coverage, the overhead is only a factor of 3.09. With full bytecode coverage, the average overhead of SenseoDiSL is a factor of 5.26; surprisingly, the overhead is still lower than for SenseoAJ covering only application code. Finally, the average overhead introduced by Senseo2 is a factor of 7.19.

[10] http://www.dacapobench.org/

[11] http://eclipse.org/aspectj/

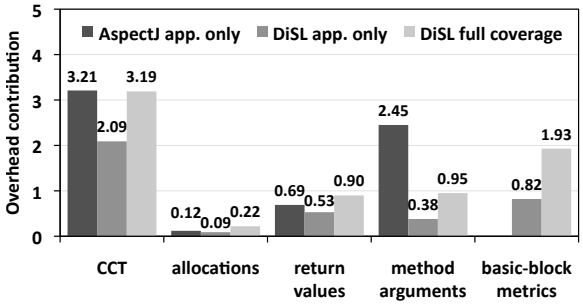


Figure 10. Contributions to the average overhead factor for different versions of Senseo

	application only	full coverage	increase [%]
Method bodies	5.60E+09	8.84E+09	57.75
Methods returning a ref.	1.76E+08	3.44E+08	95.28
Methods with ref. arg.	1.78E+09	2.57E+09	44.56
Object and array alloc.	1.14E+09	2.00E+09	76.11
Basic blocks	2.21E+10	3.34E+10	51.26

Table 3. Total number of intercepted join points for a single iteration of the whole DaCapo suite

Figure 10 quantifies the different overhead contributions. For CCT reification, the DiSL implementation benefits from efficient access to static context information, from data passing in synthetic local variables, and from the use of an @ThreadLocal variable (compared to a java.lang.ThreadLocal variable in SenseoAJ). The overheads for capturing allocations and runtime types of return values are relatively small for both implementations. The biggest difference between the two implementations is observed for the processing of method arguments; the DiSL instrumentation leverages an argument processor, whereas the AspectJ implementation relies on JoinPoint.getArgs(). As shown in Figure 10, argument processing in the AspectJ version introduces more than 6 times the overhead of the equivalent DiSL instrumentation.

Table 3 summarizes the number of intercepted join points for a single iteration of each considered benchmark, weaving only application code, respectively weaving with full byte-

	SenseoAJ	SenseoDiSL		Senseo2	
	app. only	app. only	full cov.	app. only	full cov.
Weaving [s]	54.97	43.28	134.17	65.61	174.73
Latency [s]	66.42	53.86	155.25	75.06	213.71

Table 4. Total weaving time and latency for a single iteration of the whole DaCapo suite

code coverage. For all kinds of join points, full bytecode coverage results in an increase of 45–95% in the number of intercepted join points. These results confirm that supporting weaving with full bytecode coverage is essential in the context of dynamic program analysis.

Finally, we compare the total time required to weave the complete benchmark suite. Table 4 reports (a) the total weaving time measured in the instrumentation JVM, and (b) the total weaving latency observed by the application JVM. This allows us to know the latency introduced by jBORAT. Overall, for application only, `SenseoAJ` is woven in 54.97s, whereas `SenseoDiSL` requires only 43.28s. The DiSL weaver outperforms the AspectJ weaver by a factor 1.27. With full coverage, `SenseoDiSL` requires 134.17s, and adding basic block metrics with `Senseo2` increases the weaving time to 65.61s for application code, and to 174.73s with full coverage. The latency contribution of jBORAT is between 14% and 24%, due to client-server communication.

Our evaluation confirms that DiSL enables the development of efficient dynamic analysis tools, which often cannot be achieved with general-purpose AOP languages. For our case study, the DiSL instrumentation reduces the overhead by more than factor 2 in comparison with the previous AspectJ version. Even with full bytecode coverage, the DiSL instrumentation still outperforms the AspectJ version.

7. Related Work

In previous work, we presented @J [8], a Java annotation-based AOP language for simplifying dynamic analysis. Similar to DiSL, @J uses snippet inlining and provides constructs for basic block analysis. However, @J lacks the open join point model of DiSL (i.e., @J does not support custom join point definitions), reflective access to weave-time information, and support for efficient access to reflective dynamic join point information (i.e., @J lacks argument processors). @J supports staged advice where weave-time evaluation of advice yields runtime residues that are woven. While this feature can be used to emulate guards in DiSL, it requires the use of additional synthetic local variables and more complex composition of snippets.

In [7] we discussed some early ideas on a high-level declarative domain-specific aspect language (DSAL) for dynamic analysis. DiSL provides all necessary language constructs to express the dynamic analyses that could be specified in the DSAL. That is, in the future, DiSL can serve as an intermediate language to which the higher-level DSAL programs are compiled.

High-level dynamic analysis frameworks such as RoadRunner [13] or jchord[12] ease composition of a set of common dynamic analyses. In contrast, DiSL is not tailored for any specify dynamic analysis task and offers the developer fine-grained control over the inserted bytecode.

The use of AOP for dynamic analysis [10, 20–22] has revealed some limitations in general-purpose AOP languages for that particular domain. In [1], a meta-aspect protocol (MAP) for dynamic analysis is proposed to overcome these limitations. Similar to our approach, the authors propose a flexible join point model where shadows are accessible in advice. Code snippets are used to inject callbacks to advice. MAP uses a meta object to reify context at runtime. While MAP allows fast prototyping of dynamic analyses, it does not focus on high efficiency of the developed analysis tools. In contrast, DiSL avoids any indirections to efficiently access static and dynamic context information.

The AspectBench Compiler (*abc*) [5] eases the implementation of AspectJ extensions. As intermediate representation, *abc* uses Jimple to define shadows. Jimple has no information where blocks, statements and control structures start and end, thus requiring extensions to support new pointcuts for dynamic analysis. In contrast, DiSL provides an extensible library of markers without requiring extensions of the intermediate representation.

Prevailing AspectJ weavers lack support for embedding custom static analysis in the weaving process. In [18] compile-time statically executable advice is proposed, which is similar to static context in DiSL. SCoPE [4] is an AspectJ extension that allows analysis-based conditional pointcuts. However, advice code together with the evaluated conditional is always inserted, relying on the just-in-time compiler to remove dead code. DiSL's guards together with static context allows weave-time conditional evaluation and can prevent the insertion of dead code.

In [2], the notion of region pointcut is introduced. Because a region pointcut potentially refers to several combined but spread join points, an external object shared between the join points holds the values to be passed between them. DiSL's markers provide a similar mechanism, and synthetic local variables help avoid passing data through an external object. In addition, region pointcuts are implicitly bound to the block structure of the program. In contrast, DiSL allows arbitrary regions to be marked.

Javassist [11] is a load-time bytecode manipulation library allowing definition of classes at runtime. The API allows two different levels of abstraction: source-level and bytecode-level. In particular, the source-level abstraction does not require any knowledge of the Java bytecode structure and allows insertion of code fragments given as source text. Compared to DiSL, Javassist does not follow a pointcut/advice model and does not provide built-in support for synthetic local variables.

[12] `http://code.google.com/p/jchord/`

Josh [12] is an AspectJ-like language that allows developers to define domain-specific extensions to the pointcut language. Similar to guards, Josh provides static pointcut designators that can access reflective static information at weave-time. However, the join point model of Josh does not include arbitrary bytecodes and basic blocks as in DiSL.

The approach described in [17] enables customized pointcuts that are partially evaluated at weave-time. It uses a declarative language to synthesize shadows. Because only a subset of bytecodes is converted to the declarative language, it is not possible to define basic block pointcuts as in DiSL.

Steamloom [9, 14] provides AOP support at the JVM level and improves performance of advice execution by optimizing dynamic pointcut evaluation. In DiSL, performance gains stem from static contexts combined with efficient access to dynamic context information. No JVM support is needed.

8. Discussion

In this section we discuss the strengths and limitations of DiSL for implementing dynamic analysis tools, comparing DiSL with the mainstream AOP language AspectJ [16] and with the low-level bytecode manipulation library ASM.

Expressiveness. AspectJ lacks certain join points that are important for some dynamic analysis tasks (e.g., bytecode-level and basic block-level join points). Thus, it is not possible to implement analysis tools that trace the intra-procedural control flow. In DiSL, any bytecode region can be a shadow, thanks to the support for custom markers. Likewise, with ASM, any bytecode location can be instrumented.

In AspectJ, the programmer has no control over the inserted bytecode. The AspectJ weaver inserts invocations to advice methods; inlining of advice is not foreseen. In contrast, the DiSL programmer writes snippets that are always inlined. If desired, it is trivial to mimic the behavior of the AspectJ weaver by writing snippet code that invokes "advice" methods. Still, if DiSL code is written in Java and compiled with a Java compiler, the snippets cannot contain arbitrary bytecode sequences. For example, it is not possible to write a snippet in Java that yields a single `dup` bytecode when inlined. Using ASM, there are no restrictions concerning the inserted bytecode.

Level of abstraction. In comparison with AspectJ, DiSL offers a lower abstraction level. The DiSL programmer needs to be aware of bytecode semantics, whereas AspectJ does not expose any bytecode-level details to the programmer. Nonetheless, DiSL relieves the developer from dealing with low-level bytecode manipulations such as producing specific bytecode sequences, introducing local variables, copying data from the operand stack, etc. Using ASM, the programmer also needs to deal with such low-level details, resulting in verbose tool implementations.

Compliance of the generated bytecode with the JVM specification. Weaving any aspect written in AspectJ results in valid bytecode that passes verification. In contrast, woven DiSL code may fail bytecode verification; it is up to the programmer to ensure that the inserted code is valid. For instance, synthetic local variables must be initialized before they are read, and the stack locations and local variables accessed through `DynamicContext` must be valid. Similarly, bytecode instrumented with tools written in ASM may fail verification.

While it is usually desirable that woven code passes verification, violating certain constraints on bytecode sometimes simplifies analysis tasks. For example, if the analysis needs to keep track of objects that are currently being initialized by a thread, the programmer may want to store uninitialized objects in a data structure on the heap, although the resulting bytecode would be illegal. Nonetheless, the analysis can be successfully executed by explicitly disabling bytecode verification. With AspectJ, such tricks are not possible.

Interference of inserted code with the base program. With ASM, local variables or data on the operand stack belonging to the base program may be unintentionally altered by inserted code. In contrast, AspectJ and DiSL guarantee that instrumentations cannot modify local variables or stack locations of the base program.

Bytecode coverage. For many analysis tasks, it is essential that the overall execution of the base program can be analyzed. However, prevailing AspectJ weavers do not support weaving the Java class library. In contrast, DiSL has been designed for weaving with complete bytecode coverage, which does not introduce any extra effort for the developer. With ASM, it is possible to develop tools that support complete bytecode coverage. However, the ASM programmer has to manually deal with the intricacies of bootstrapping the JVM with a modified class library and preventing infinite regression when inserted bytecode calls methods in the instrumented class library.

9. Conclusion

In this paper we presented DiSL, a new domain-specific language for bytecode instrumentation. The language is embedded in Java and makes use of annotations. DiSL allows the programmer to express a wide range of dynamic program analysis tasks in a concise manner. DiSL has been inspired by the pointcut/advice mechanism of mainstream AOP languages such as AspectJ. On the one hand, DiSL omits certain AOP language features that are not needed for expressing instrumentations (e.g., around advice and explicit structural modifications of classes). On the other hand, DiSL offers an open join point model, synthetic local variables, comprehensive and efficient access to static and dynamic context information, and support for weave-time execution of static analyses. These language features allow expressing bytecode transformations in the form of code snippets that are in-

lined before or after bytecode shadows as indicated by (custom) markers, if user-defined constraints specified as guards are satisfied. As case study, we recasted the dynamic analysis tool Senseo in DiSL and compared it with the previous implementation in AspectJ. In contrast to the AspectJ version, the DiSL implementation ensures complete bytecode coverage, reduces overhead, and allows us to gather additional intra-procedural execution statistics.

In an ongoing research project, we are working on advanced static checkers for DiSL instrumentations to help detect errors before weaving, on partial evaluation of instantiated snippets before inlining, and on general techniques to split overlong methods that exceed the maximum method size imposed by the JVM. In addition, we are exploring the use of higher-level, declarative domain-specific languages for dynamic program analysis. We plan to compile such higher-level languages to DiSL, which will serve us as a convenient intermediate language.

Acknowledgments

The research presented here was conducted while L. Marek, A. Villazón, and Y. Zheng were with the University of Lugano. It was supported by the Scientific Exchange Programme NMS–CH (project code 10.165), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Exchange Grant (project no. EG26–032010) and Institutional Partnership (project no. IP04–092010), by the Swiss National Science Foundation (project CRSII2_136225), and by the Czech Science Foundation (project GACR P202/10/J042). The authors thank Aibek Sarimbekov and Achille Peternier for their help with jBORAT, and Andreas Sewe for testing DiSL and providing detailed feedback.

References

[1] M. Achenbach and K. Ostermann. A meta-aspect protocol for developing dynamic analyses. In *Proceedings of the First International Conference on Runtime Verification*, RV'10, pages 153–167. Springer-Verlag, 2010.

[2] S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for AspectJ. In *ACP4IS '09: Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, pages 43–48. ACM, 2009.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96. ACM, 1997.

[4] T. Aotani and H. Masuhara. SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 161–172. ACM, 2007.

[5] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *AOSD '05: Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, pages 87–98. ACM, 2005.

[6] W. Binder, D. Ansaloni, A. Villazón, and P. Moret. Flexible and efficient profiling with aspect-oriented programming. *Concurrency and Computation: Practice and Experience*, 23(15):1749–1773, 2011.

[7] W. Binder, P. Moret, D. Ansaloni, A. Sarimbekov, A. Yokokawa, and E. Tanter. Towards a domain-specific aspect language for dynamic program analysis: position paper. In *Proceedings of the sixth annual workshop on Domain-specific aspect languages*, DSAL '11, pages 9–11. ACM, 2011.

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

[9] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92. ACM, 2004.

[10] E. Bodden and K. Havelund. Aspect-oriented Race Detection in Java. *IEEE Transactions on Software Engineering*, 36(4):509–527, 2010.

[11] S. Chiba. Load-time structural reflection in Java. In *Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000)*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Verlag, Cannes, France, June 2000.

[12] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 102–111. ACM, 2004.

[13] 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, pages 1–8. ACM, 2010.

[14] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg, and M. Krebs. An execution layer for aspect-oriented programming languages. In *VEE '05: Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*, pages 142–152. ACM, 2005.

[15] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD '04: Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 26–35. ACM, 2004.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353. Springer-Verlag, 2001.

[17] K. Klose, K. Ostermann, and M. Leuschel. Partial evaluation of pointcuts. In *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 320–334. Springer-Verlag, 2007.

[18] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the law of Demeter with AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*, AOSD '03, pages 40–49. ACM, 2003.

[19] P. Moret, W. Binder, and É. Tanter. Polymorphic bytecode instrumentation. In *AOSD '11: Proceedings of the 10th International Conference on Aspect-Oriented Software Development*, pages 129–140. ACM, Mar. 2011.

[20] D. J. Pearce, M. Webster, R. Berry, and P. H. J. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, June 2007.

[21] D. Röthlisberger, M. Härry, W. Binder, P. Moret, D. Ansaloni, A. Villazón, and O. Nierstrasz. Exploiting dynamic information in IDEs improves speed and correctness of software maintenance tasks. *IEEE Transactions on Software Engineering*, PrePrint, 2011.

[22] D. Röthlisberger, M. Härry, A. Villazón, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in IDEs with dynamic metrics. In *ICSM '09: Proceedings of the 25th IEEE International Conference on Software Maintenance*, pages 253–262, Edmonton, Alberta, Canada, 2009. IEEE Computer Society.

Chapter 4

ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform

Lukáš Marek,
Stephen Kell,
Yudi Zheng,
Lubomír Bulej,
Walter Binder,
Petr Tůma,
Danilo Ansaloni,
Aibek Sarimbekov,
Andreas Sewe

Contributed paper at the **12th International Conference on Generative Programming: Concepts & Experiences (GPCE 2013)**.

In conference proceedings,
published by ACM,
pages 105-114,
ISBN 978-1-4503-2373-4,
October 2013.

The original version is available electronically from the publisher's site at `http://dx.doi.org/10.1145/2517208.2517219`.

ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform

Lukáš Marek
Faculty of Mathematics and Physics
Charles University, Czech Republic
lukas.marek@d3s.mff.cuni.cz

Stephen Kell Yudi Zheng
Faculty of Informatics
University of Lugano, Switzerland
firstname.lastname@usi.ch

Lubomír Bulej Walter Binder
Faculty of Informatics
University of Lugano, Switzerland
firstname.lastname@usi.ch

Petr Tůma
Faculty of Mathematics and Physics
Charles University, Czech Republic
petr.tuma@d3s.mff.cuni.cz

Danilo Ansaloni Aibek Sarimbekov
Faculty of Informatics
University of Lugano, Switzerland
firstname.lastname@usi.ch

Andreas Sewe
Software Technology Group
TU Darmstadt, Germany
andreas.sewe@cs.tu-darmstadt.de

Abstract

Dynamic analysis tools are often implemented using instrumentation, particularly on managed runtimes including the Java Virtual Machine (JVM). Performing instrumentation robustly is especially complex on such runtimes: existing frameworks offer limited coverage and poor isolation, while previous work has shown that apparently innocuous instrumentation can cause deadlocks or crashes in the observed application. This paper describes ShadowVM, a system for instrumentation-based dynamic analyses on the JVM which combines a number of techniques to greatly improve both isolation and coverage. These centre on the offload of analysis to a separate process; we believe our design is the first system to enable genuinely full bytecode coverage on the JVM. We describe a working implementation, and use a case study to demonstrate its improved coverage and to evaluate its runtime overhead.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Frameworks

Keywords Dynamic analysis; JVM; instrumentation

1. Introduction

To gain insight about how to optimise, debug, extend and refactor large systems, programmers depend on analysis tools. One popular class of tools is *dynamic program analysis* tools, which observe a program in execution and report additional data about that execution. Many popular bug-finding and profiling tools are of this form, including the Valgrind suite [19], DTrace [2], and GProf [12]. Meanwhile, research continues to devise more complex and specialised tools, for race detection [10], white-box testing [25], security policy enforcement [28] and more.

GPCE'13, October 27 - 28 2013, Indianapolis, IN, USA.

http://dx.doi.org/10.1145/10.1145/2517208.2517219

Developing dynamic analyses is difficult. One approach is to invasively modify the host runtime system, but this is an expert task yielding a non-portable solution. Alternatively, instrumentation frameworks including Pin [6] and DynamoRIO [5] (exporting a roughly compiler-style intermediate representation), and also Javassist [7], Soot [21] and DiSL [16] (targeting Java bytecode), are highly general. However, using them can be challenging, since they require deep understanding of both the intermediate representation and the host runtime environment. More constrained frameworks [2, 11] provide stronger properties with less user effort, but each caters to a smaller set of use cases. Outwith these use cases, developing a *high-quality* dynamic analysis remains a Herculean task, plagued by the recurrence of three mutually antagonistic requirements: *isolation*, meaning roughly that observing the program does not cause it to deviate from the path it would ordinarily take; *coverage*, meaning the ability to observe all relevant events during execution, including both user code and system code; and *performance*, meaning the minimisation of slowdown caused by the analysis.

In this paper we present ShadowVM[1], a system for dynamic analysis of programs running within the Java Virtual Machine (JVM) which advances on prior work by simultaneously combining strong isolation and high coverage. Analyses execute asynchronously with respect to the observed program, allowing parallelism to mitigate isolation-induced slowdowns. To our knowledge, ours is the first complete dynamic analysis framework offering asynchronous execution without effectively serializing heavily instrumented workloads. It does so by exploiting heterogeneity among dynamic analyses, which typically only need to preserve the order of observed events for particular *subsets* of events. In summary, this paper presents the following contributions:

- We describe an architecture and programming model for dynamic analyses of Java bytecode which enforces isolation by performing all analysis computation in a separate process. This enables *asynchronous* remote evaluation while permitting a familiar programming model similar to that of existing instrumentation frameworks.
- We summarise the state of the art regarding *coverage* on the JVM, identifying challenges which so far limit the coverage available under existing systems, and explaining how our implementation

[1] Sources available at http://disl.ow2.org

circumvents these challenges. We believe our system to be the first offering truly complete bytecode coverage on the JVM.

- We evaluate the isolation and coverage of our implementation compared to classic in-process analysis and provide experimental evidence of reduced perturbation and improved coverage. To quantify the cost of the improved isolation in terms of performance, we evaluate the runtime overhead and scalability of our solution with parallel workloads.

We begin by motivating our approach in greater depth.

2. Motivation

A popular mechanism used by dynamic analysis tools to observe applications on the Java platform is *bytecode instrumentation*. The analysis tool inserts "hooks", in the form of bytecode snippets, into locations of interest in the application code. When the application execution reaches a particular location, the corresponding hook is executed as a part of the application. Compared to alternative observation mechanisms, such as debugging interfaces or virtual machine modifications, bytecode instrumentation is often more portable, less complex, and offers higher performance. However, observation through bytecode instrumentation also exhibits two significant problems: one concerning the safety of *high coverage* analyses, and another concerning the semantics of *asynchronously executing* analysis tools. We discuss these in turn.

2.1 Coverage versus isolation

Observation through bytecode instrumentation necessarily mixes the application code with (at least some of) the analysis code. This can lead to problems achieving high coverage in analyses, i.e. to observe program activity in all code, including sensitive bytecode regions such as system-level libraries. Java analysis tools usually cannot avoid calling these libraries from within the analysis code, because the libraries offer the standard or even the only means of performing many essential operations—including input and output (e.g. for exporting the analysis results), reflective acquisition of metadata (e.g. for inspecting the class and field information pertaining to the instrumented event), and keeping references to program objects (e.g. through the weak reference mechanism). When the libraries offering these functions are themselves instrumented, library-internal resources become shared between the application and the analysis in an uncoordinated way. Consequently, even very basic instrumentation scenarios can suffer from subtle problems including state corruption (from introduced reentrancy), deadlocks (from lock order violations), and memory exhaustion (from sharing the weak reference queue handler) [22].

A cheap way to avoid this interference, i.e. to improve the *isolation* between analysis and the application, is to exclude common library code from instrumentation. This exclusion technique is commonly used in various dynamic analysis frameworks; Figure 1 shows three examples from well-known frameworks. Exclusion limits the observation power of the analysis, since it can no longer analyse library operations—resource usage by library code is invisible, data flow through library code cannot be tracked, and so on. Managed runtimes, notably the Java platform, suffer particularly because core functionality, including class loading and some aspects of memory management, is implemented in bytecode and cannot be cleanly and effectively replaced or virtualized to isolate the base program from the analysis.

Instead of sacrificing coverage by using exclusion to achieve isolation, we prefer to perform the analysis "outside" the observed program. Doing so in native code appears feasible (given appropriate care to inadvertent sharing of state through native method implementations). Unlike bytecode, native code can safely perform input and output operations through the operating system interfaces,

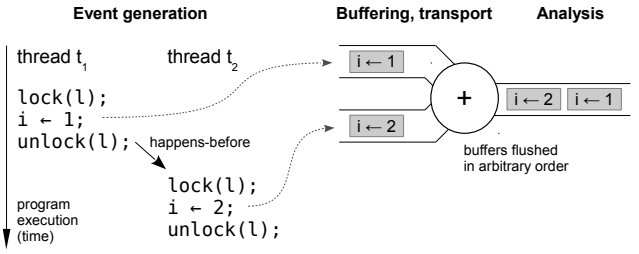


Figure 2. Multiple buffering can cause reordering of observations. Here, two assignments ordered by synchronization in the program are nevertheless reordered in the event stream fed to the analysis.

access object references through the virtual machine API, and implement out-of-band reflection. We pursue this approach; details in the context of our tool are in §4.2 and §6.

Writing analyses in native code requires knowledge of C or C++ and the associated virtual machine and system API. This can become a practical obstacle for Java developers. We therefore seek a programming environment where analyses are written at the same level of abstraction as when using plain bytecode instrumentation, but with improved coverage and isolation. For example, it should be possible to safely perform Java-style reflection, to keep references to objects in the observed program, and to freely use existing library code when implementing the analysis. This environment should also provide specialised mechanisms for common analysis tasks, such as associating analysis state with application objects. We describe such an environment in §4.

2.2 Resource lifecycle events

Even with the best coverage possible, bytecode instrumentation can only observe bytecode execution events. In some cases this is incomplete—not only because the application can execute native code, but also because the events of interest can occur inside the code of the virtual machine itself, rather than in the application. Some events are not associated with particular bytecode execution (such as virtual machine startup and shutdown) yet are highly relevant to analyses (e.g. for purposes of state management). Frameworks focusing solely on bytecode instrumentation neglect such events. This class of events can be viewed as events in the *lifecycles* of the basic system resources: program objects, threads, and the virtual machine itself. They contrast with the usual state transitions *within* a given system resource, such as within objects (field updates) and within threads (calls, returns, computations on the operand stack), which are cleanly captured by bytecode. Although hooks for several lifecycle events are available through either the Java API or the virtual machine API, their use from within the analysis is complicated by isolation and synchronization issues. For example, with the standard JVM shutdown notification API (in `java.lang.Runtime`), the shutdown hooks run concurrently with other hooks and with daemon threads, which execute application code. An analysis therefore cannot rely on the virtual machine shutdown event being the last event observed. Another example is the JVM reference handling mechanism, used for notification of object death. This mechanism cannot be safely used by analysis that also observes the application reference handling behavior [22]. In general, the problem is that these hooks are neither isolated from the application nor ordered relative to other observed events. §4.4 explains how our programming model avoids these problems by introducing lifecycle event ordering guarantees.

```
# RoadRunner's default exclusion list
java..*
javax..*
com.sun..*
org.objectweb.asm..*
sun..*
```

```
// Chord's implicit exclusion logic:
public boolean isImplicitlyExcluded(String cName) {
  return cName.equals("java.lang.J9VMInternals") ||
    cName.startsWith("sun.reflect.Generated") ||
    cName.startsWith("java.lang.ref.");
}
```

```
// BTrace excludes "sensitive" classes
private static boolean isSensitiveClass(String name) {
  return name.equals("java/lang/Object") ||
    name.startsWith("java/lang/ThreadLocal") ||
    name.startsWith("sun/reflect") ||
    name.equals("sun/misc/Unsafe") ||
    name.startsWith("sun/security/") ||
    name.equals("java/lang/VerifyError");
}
```

Figure 1. Exclusion lists from the RoadRunner [11], Chord [18] and BTrace (http://kenai.com/projects/btrace) frameworks. Such exclusions are found in prevailing bytecode-level dynamic analysis frameworks, limiting the coverage available to tools built with them.

2.3 Asynchronous analysis

To exploit modern multiprocessor hardware, designs which relax synchronisation between application and analysis are increasingly desirable. Several existing systems and techniques, such as Shadow Profiling [26], SuperPin [29], and CAB [14], support offloading the analysis to separate cores for parallel processing. So far, however, little attention has been paid to the impact of asynchronous analysis design on the ability to observe application event ordering.

With a synchronous design, the hooks inserted through bytecode instrumentation execute the analysis code as a part of the application, synchronously (with respect to the thread running the inserted bytecode). The virtual machine applies the semantic rules governing program execution to both the analysis and the application together—in particular, the analysis actions are ordered with the program actions using the intra-thread semantics of the Java language and the happens-before relation of the Java Memory Model.

In contrast, an asynchronous analysis design separates the hooks from the analysis code. The hooks still execute as a part of the application and are therefore still ordered with the program actions. However, instead of executing the analysis code directly, the hooks notify the analysis code through asynchronous communication. The analysis code executes in a separate thread or even a separate process, and the communication involved may easily change the order in which the individual actions are ultimately observed by the analysis. Figure 2 shows an example of this reordering, where the instrumentation uses multiple thread-local buffers to avoid contention. Since these are flushed to an output stream in a non-deterministic order (e.g. when the buffer is full), the original program ordering is lost. Dynamic program analyses differ in their sensitivity to these changes: count-based analyses tend to work with any ordering; thread-local analyses may require ordering guarantees from the thread perspective; other analyses are yet more demanding. Because additional ordering guarantees bring additional costs, an efficient instrumentation framework should exploit the heterogeneity of the analyses and provide only the ordering that is required. We consider this further in §4.3.

3. ShadowVM design goals

ShadowVM addresses some of the issues that make the development of high-quality dynamic analyses difficult. It has several goals, each corresponding to one or more features in the design.

Isolation. We wish to avoid sharing state with the observed program to the greatest extent possible. This is necessary both generally to reduce perturbation and specifically to avoid various known classes of bugs which less well-isolated approaches inherently risk introducing [22]. Our design's **hook–analysis separation** achieves this by factoring analyses into a remotely executed part and short local "hooks" inserted by bytecode instrumentation, which trap immediately to native code. Although this pattern has been advocated, e.g. in the JVMTI documentation, we know of few dynamic analyses which actually follow it. This undoubtedly owes to a lack of supporting infrastructure—a lack which our work addresses.

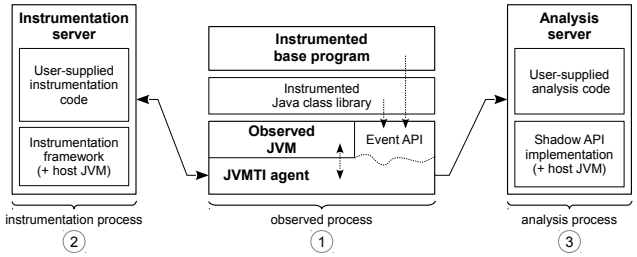


Figure 3. ShadowVM architecture at a high level.

High coverage. We wish to allow instrumentation of both user-level application code and system-level core libraries. Previous approaches have provided only partial solutions. There is a fundamental tension with isolation, since achieving coverage deep in the system-level libraries risks perturbing core JVM behaviour. We explore these difficulties in §5. Our approach combines several implementation techniques, notably **out-of-process analysis** and the aforementioned "straight to native" hooks. These are able to cover all bytecode execution. To our knowledge, ours is the first system offering genuinely complete bytecode coverage on the JVM.

Performance. We require dynamic analyses to perform well in spite of the additional level of isolation provided by our system. To this end, our **asynchronous analysis** design exploits the availability of spare CPU cores. Meanwhile, our **flexible ordering models** help extract latent parallelism while preserving the event ordering relationships on which the analysis' functional correctness depends.

Productivity. We wish to allow instrumentation and analysis to be free of unnecessary constraints on how they may be programmed. In particular, it must be possible to implement them in Java code, rather than only in native code. We also require that they may be expressed in terms of a well-defined and convenient API. We define a "shadow API" for this purpose. Two notable features are its convenient **associative shadow state** abstraction and the **ordering guarantees** it offers, which reflect the selected ordering model.

4. Writing analyses using ShadowVM

Writing a dynamic analysis using ShadowVM is in many ways similar to the use of a bytecode-level instrumentation system such as DiSL, BTrace or (the dynamic analysis part of) Chord. However, our design differs to improve the robustness of the resulting tool. The most significant difference is a "hook–analysis separation": since analysis code does not run in the same process as the observed program, instrumentation is strongly separated from analysis by a generated stub layer which notifies the remote analysis of events of interest. Figure 3 shows the high-level architecture of the system.

```
1 // ——— runs in the observed VM
2 public class AllocCounterStub {
3   // instrument: snippet inserted after each "new" bytecode
4   @AfterReturning(marker=BytecodeMarker.class, args="new")
5   public static void allocSnippet(
6     DynamicContext dc, AllocationSiteStaticContext sc) {
7     // transmit event to analysis
8     AllocCounterRE.onAlloc(
9       dc.getStackValue(0, Object.class), // object allocated
10      sc.getAllocationSite() );          // alloc site
11  }
12 }
13 // ——— runs in the analysis VM
14 public class AllocCounter implements AllocAnalysis {
15   AtomicLong counter = new AtomicLong();
16   public void onAlloc(
17     ShadowObject o, ShadowString allocSite) {
18     counter.incrementAndGet();
19   }
20 }
```

Figure 4. This simple analysis counts object allocations by allocation site. For simplicity, this code only instruments the `new` bytecode. Other bytecodes allocating objects would require similar treatment.

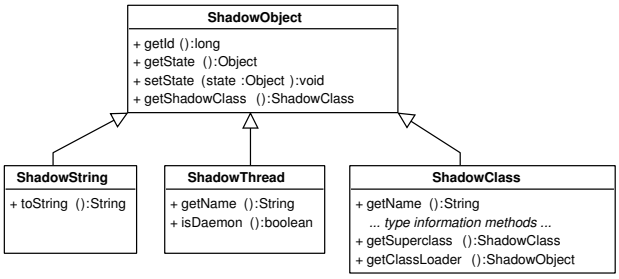


Figure 5. The Shadow API provides an analysis-friendly view of objects and threads in the observed program. Shadow objects are in bijection with the subset of objects in the observed program that have been passed to the analysis. Each object allows association of shadow state, which can be used to store arbitrary data.

Whereas the instrumented base program is executed by the JVM within the *observed process*, a second process performs all bytecode instrumentation. This process separation is essentially hidden from the user. A third process performs the analysis itself; this separation is much more apparent. Finally, we note that since in our implementation, both analysis and instrumentation are implemented in Java, each process runs its own JVM.[2]

Three other distinctions of our programming model are: its flexible approach to analysis-visible object state (in which the user controls how objects in the observed program are represented for analysis); notification ordering (in which more relaxed orderings can be requested, offering improved performance); and resource lifecycle events, which allow notifications not directly available through Java bytecode instrumentation. We discuss each of these, beginning with an example.

4.1 Introductory example

Figure 4 shows a simple example analysis implemented using ShadowVM. It consists of an instrumentation part and an analysis part. The instrumentation part, lines 2–12, uses a pre-existing annotation-based instrumentation language, DiSL [16], to define a "hook" as a code snippet woven into the program on the events of interest (here execution of the `new` bytecode). This hook simply extracts the information from the instrumentation context (here the object allocated, retrieved from the top of the stack using `getStackValue(0, Object.class)`) and calls into an `onAlloc` method of the `AllocCounterRE` class. The definition of this method is not shown because it is a stub routine generated from the `AllocAnalysis` interface exposed by the analysis part. The stub simply notifies the analysis VM of the event. The analysis part, lines 14–20, defines the analysis computation, its interface being a single `onAlloc` method.

The hook runs in the observed VM, whereas the analysis runs in the analysis VM. Unlike other bytecode instrumentation systems, under our design the hook only invokes native notification calls, which marshal their arguments into a wire representation that is sent over a socket to the analysis VM. The analysis VM runs an event loop which receives the notifications and dispatches them to the appropriate analysis method. (The dispatch logic is also responsible for creating analysis threads; we describe this in §6.4.)

4.2 Shadow API and object representation

In the analysis VM, analyses are clients of the *Shadow API*, shown in Figure 5. This API provides methods for reflecting on the class metadata in the observed program, and for associating analysis state with objects in the base program. Each object is reified as a *shadow object* which provides an associative API but (by default) does not replicate the fields of the original object—only object identity and class information are available by default. This design reflects the fact that object contents are not required by many analyses (e.g. most profilers). Meanwhile, many that *are* sensitive to object contents (e.g. a shape analysis over heap structure) may benefit from a customised representation of the fields (e.g. only recording distinct pointer fields, rather than every field). To replicate object contents, the analysis must receive field write events from the observed VM, and associate these field values with the shadow objects.

The use of distinct "shadow" object, thread and string classes necessitates some translation in the mapping from analysis APIs to notification (stub) APIs. Whereas the analysis API's method signatures must be in terms of `ShadowObject`, `ShadowString` etc. (and primitive types), in the observed VM these will appear as the usual `Object`, `String`, etc. Instances of `java.lang.Class` are also shadowed specially: every class loaded in the observed VM has corresponding "shadow class" metadata available in the analysis VM. This is essential because many analyses generate output in terms of the structure of the program (profiles per class or per method, backtraces on events of interest, etc.).

4.3 Threading and ordering

Analysis code runs according to a particular threading model, where different models are suitable for different analyses. The analysis VM creates threads for processing incoming notifications. Different analyses have different requirements concerning in what order they must process notifications. These requirements reflect the dependency structure of the analysis computation. For example, just as profilers rarely require object contents, many profilers are insensitive to reordering of events of the same kind, because they effectively perform counting (counter increments are commutative).

In general, the developer needs to be aware of the ordering requirements of a particular analysis, and choose an appropriate implementation strategy for the analysis code. ShadowVM's most conservative ordering yields similar behaviour to the existing dynamic analysis frameworks such as Chord, where the observed program is effectively serialized for analysis.[3] However, ShadowVM also provides higher-performance (but more relaxed) ordering configurations, for use when appropriate.

[2] We use "instrumentation VM" and "instrumentation process" interchangeably. Similarly, the "observed" and "analysis" processes are also "VMs".

[3] Although strictly speaking, nothing in Chord's design serializes the program, ordering is handled by contending for a lock on a unique shared buffer. Frequent contention for this lock in all threads, as generated by any moderate or heavy instrumentation, effectively serializes the program.

We describe ordering using the following terminology. *Program actions* are state transitions in the observed process. A subset of these are of interest to the analysis, so are hooked. This generates an *event*, which is a message encapsulating the values gathered by the hook code, and which is transmitted to the analysis VM. *Notification* is the receipt of an event by the analysis VM from the observed VM. We say that the hooked program action in the observed VM *triggers* a notification in the analysis VM. The ordering of notifications is not, in general, the same as the ordering of the program actions that triggered them. The different ordering models we now describe cause different subsets of the ordering of program actions to be preserved in the ordering of notifications. (We note that the "ground truth" ordering of program actions is determined by the behaviour of the host system. In our case, since the host system is a JVM, this behaviour is circumscribed by the Java standard.)

Per-thread configuration. In this configuration, notifications are ordered by the (per-thread) program order in the observed program. Events from each thread are stored in a dedicated FIFO buffer pool by the agent, the pools are flushed in arbitrary order with respect to each other. Notifications are dispatched to multiple analysis threads corresponding to the threads that trigger the events in the observed program.

Per-group configuration. This is the most flexible configuration. The developer specifies a group identifier to be used with each hook. Each group has its own FIFO buffer pool for notifications. It can be seen as a generalisation of per-thread ordering where pools need not map to threads. For example, a group could map to a set of threads, a single object, code in a particular set of classes, and so on. Similar to the per-thread configuration, the analysis server dispatches notifications to the corresponding analysis methods in multiple threads, one thread per group.

Global-ordering configuration. This is the most conservative configuration. Conceptually, it can be thought of as per-group configuration with a single group identifier. A single buffer pool is used, and the analysis server dispatches all notifications to the corresponding analysis methods in a single thread.

4.4 Resource lifecycle events

ShadowVM analyses can request notifications for special events which do not correspond to execution of bytecodes. Rather, they relate to some unit of *resource* in the program, where these resources can be threads, objects or the VM itself. These special events mark the end of resource *lifetimes*. For example, the user can request notification of object death (which occurs in the garbage collector, so has no corresponding bytecode). Our attention to ordering guarantees extends to these events. Specifically, we guarantee that following a notification of the death of a thread, object or the VM, no further notifications referencing that entity will occur. In the case of the VM, a "VM death" notification is the last one of the execution.

This contrasts with existing APIs which might be used for such purposes, such as finalizers, or the Java library's `Runtime.addShutdownHook` method. These APIs offer few or no guarantees about the scheduling of hook code, making them difficult to employ from analyses without risking loss of coverage. For example, although an analysis could register its own shutdown hook, there is no guarantee that some user-supplied hook would not run after it. A similar lack of guarantee applies to object finalizers (which, in any case, need not mark the end of an object's lifetime, owing to resurrection [4]). Meanwhile, there is no portable way to identify the bytecode representing the precise end of a thread's execution.

As with other notifications, these ordering guarantees are enforced in the buffer management code. For all ordering configurations, ShadowVM ensures that the lifecycle events are delivered in proper order related to the notifications produced by hooks.

5. Coverage challenges

In any instrumentation-based design, isolating the analysed program from the analysis is inherently in tension with coverage, because the inserted code necessarily shares an execution context with the analysed program code. By default, therefore, it is not isolated from it. Isolation can only be provided by adopting a discipline which restricts what is done from the inserted code, yet still provides analyses with essential functionality such as allocating memory, keeping references to objects in the observed program, and performing I/O. In this section we summarise the specific difficulties of achieving this on the Java platform, and the extent to which existing solutions have (and have not) overcome them.

Exclusion list. A simple way to avoid isolation difficulties is to sacrifice coverage, by omitting instrumentation of core classes. This avoids bootstrapping problems, interference between program and analysis (through shared library state), and infinite regress (if these libraries are used from instrumentation). We saw some exclusion lists in Figure 1.

Load–instrument gaps. High coverage relies on intercepting the loading of a high proportion of the base program code, so that an instrumented version can be substituted. Many naive instrumentation implementations on the Java platform miss some coverage by missing load events (therefore never instrumenting the loaded code), or by allowing execution of uninstrumented versions of the code for some time. The Java instrumentation API in `java.lang.instrument` suffers from this problem because it does not allow applying the instrumentation during JVM start. Moreover, since the Java code performing the instrumentation may itself trigger additional classes to be loaded (which cannot be transformed at that point), it leaves the untransformed version available for use by other threads. We avoid this by performing instrumentation outside the observed VM (in a separate process) using JVMTI's `ClassFileLoad` hook (which does not lead to concurrent use of the uninstrumented code).

Missed initializers. Possibly the most obvious problem with instrumenting core libraries is infinite regress when those libraries are invoked from the inserted code. It is easily avoided using a per-thread "bypass" flag [17]. However, a side-effect of bypass is that initializers for classes used by the analysis are run while the bypass is active and so are not analysed; if the same classes are used later by the program, their initializers will not be re-run, and so will not be covered. ShadowVM does not suffer from this problem because the only classes referenced by its implementation are `Object` and `String` which are preloaded by the JVM bootstrap long before the first bytecode is executed.

Avoiding bootstrap bypass. Special-case handling is inevitably required for instrumentation affecting the very earliest bytecode that the JVM executes, a.k.a. the "bootstrap phase". The hook code snippets as well as our generated stub classes are carefully restricted to a safe subset of bytecode operations. For example, it is not safe to allocate objects in inserted code that might be invoked from `Object.` $<$ `clinit` $>$ (causing a stack overflow). Since our stubs need only call a static native method, this careful construction is possible—the fact that calls to our native stub code can be called so early owes to the fact that an initial set of classes, including `Object` and `Class`, is necessarily special-cased by the JVM.[4]

[4] The JVM's definition of a class being "loaded" implies that a `Class` object exists for it [13, §12.2]—yet, to instantiate the `Class` object for class `Object` under these rules, both `Class` and its superclass `Object` would (circularly)

Reference handling. The standard way for an analysis to maintain references to objects in the observed program is to use `WeakReference`s. Usually, one shared reference handler thread (or a garbage collection thread) processes cleared reference objects (`WeakReference`, `SoftReference`, `PhantomReference`) on behalf of all other threads. If this thread's code is instrumented, it may create a self-sustaining allocation cycle, because the inserted code within the reference handler may allocate more `WeakReference`s. Excluding the reference handling code avoids this problem, but loses coverage of reference handling on behalf of the observed program. Our design avoids using `WeakReference`s and thus avoids this problem.

6. ShadowVM Architecture

We have summarized the high-level, multi-process architecture of ShadowVM earlier in §4. Here, we review the key architectural elements in greater detail. In general, the architecture is driven by the design goals elaborated in §3, and the ShadowVM responsibilities are split between three processes, as shown in Figure 3.

Firstly, the observed VM (augmented with a JVMTI agent) contains the instrumented base program and class library. The inserted hook code is responsible for producing base program events that are of interest to the analysis. The agent has two key responsibilities: installing instrumented base program code in the observed VM, and forwarding events produced by the hooks in the base program to the analysis.

A second VM contains the instrumentation server, itself written in Java. The instrumentation server performs all bytecode instrumentation, communicating Java bytecode with the observed VM's agent via a socket.

The third VM contains the analysis server, which hosts the analysis written against the Shadow API. The analysis server is responsible for dispatching event notifications received via socket from the observed VM's agent to the analysis code, while respecting the selected ordering configuration.

We now review the various responsibilities in turn.

6.1 Load-time instrumentation

To ensure load-time instrumentation of the base program, the agent intercepts all class loading events in the observed VM and requests instrumented versions from the instrumentation VM. The use of a separate VM to perform instrumentation avoids the substantial perturbation which would be caused if instrumentation were performed within the observed VM. For example, doing so would bring forth a significant amount of class loading and initialization activity, which would then not be analysed at the proper point in the observed program's execution. Besides reducing perturbation, this separation is also essential to enable high-coverage instrumentation encompassing the Java Class Library (JCL).

6.2 Base-program event generation

The user-defined hooks in the base program are responsible for generating the events of interest for a particular analysis. The hooks are expressed as DiSL [16] snippets. However, unlike conventional analyses based on bytecode instrumentation (including ordinary uses of DiSL), the hook code is always of the same restricted form: invoking a native helper method (event API) provided by the observed VM's agent, passing as arguments values capturing the program state that is relevant to the instrumented event. Beyond this point, the reified event is the responsibility of the agent, and in this paper we do not concern ourselves further with how the instrumentation itself is expressed or performed. We simply assume that the events of interest at bytecode level can be intercepted and handled appropriately, and to simplify hook development, we provide a library of snippets for various event types.

6.3 Event forwarding

The agent natively implements an *event API*, into which hooks call during the execution of the instrumented base program, producing base program events. The methods of this API marshal their arguments into buffers and the agent delivers the event notifications in an asynchronous manner to the analysis server executing on the analysis VM. This separation is crucial to achieve high coverage and isolation, because it allows instrumenting the base program without any bypass mechanisms. It also allows using extra computing power for analysis without perturbing the base-program execution.

The communication between observed and analysis processes requires carefully designed buffering and threading strategies in order to yield high-performance asynchronous analyses while respecting ordering constraints (introduced in §4.3). Events produced by base program threads are stored and marshaled into buffers in the context of the event API invocations. Object references in the buffers are processed by a separate thread that ensures, with the help of object tagging, that objects have unique identity and that it is preserved on the analysis server. Another thread then sends the completed buffers to the analysis VM.

6.4 Notification delivery

Recall that the analysis code runs in a separate JVM (*analysis VM*). Base program event notifications are sent via socket to the analysis server. Dispatch logic in the analysis server consumes from this socket, performs appropriate unmarshaling, and invokes methods of the analysis.

Apart from the threading model described in §4.3, which is exposed to the analysis, the analysis server has to cooperate with the agent to maintain notification ordering mandated by the selected ordering configuration. The internal threading model of the analysis server was designed to properly order resource lifecycle notifications with respect to base program event notifications. In addition to the threads dispatching base program notifications, the analysis server also creates a dedicated thread to deliver resource lifecycle event notifications to the analysis.

7. Evaluation

We consider the high degree of isolation and full bytecode coverage to be the key benefits of ShadowVM. We therefore aim at evaluating the difference in perturbation and analysis coverage when a base program is subjected to a heavy-weight dynamic analysis—once implemented in the classic in-process manner, and once implemented using ShadowVM.

With respect to performance, the distributed nature of the ShadowVM approach comes with an inherent overhead due to reification and forwarding of events to the analysis VM. However, the ShadowVM approach also has an inherent scaling potential, which hinges on the ability of a particular analysis to execute in multiple threads mirroring the base-program threads. We therefore aim at quantifying the overhead of a ShadowVM-based analysis compared to classic in-process analysis and to assess the scalability of ShadowVM with parallel workloads.

As case study for our evaluation, we chose the field immutability analysis (FIA) by Sewe et al. [1][5]. In summary, FIA tracks all object allocations and field accesses and maintains a per-field "state-machine" that describes the mutability of that field. If a field is written outside the dynamic extent of an object's constructor, it is

already need to be loaded and initialized. All JVMs therefore employ some kind of special-casing to avoid this circularity.

[5] The sources are available at http://www.disl.scalabench.org/modules/immutability-disl-analysis/.

Benchmark	Uninstrumented	In-process FIA	ShadowVM FIA
avrora	1020	1221	1022
batik	2042	2248	2044
fop	1868	2129	1870
h2	919	1120	921
jython	2651	2828	2653
luindex	783	984	785
lusearch	680	886	682
pmd	1194	1387	1196
sunflow	938	1104	940
xalan	1168	1389	1170

Table 1. Comparison of class loading perturbation. The table presents the number of classes loaded by the observed VM.

marked mutable. Explicit field initialization during construction and reliance on implicit zeroing of fields by the VM are taken into account. Overall, the analysis is relatively heavy-weight and would be a typical candidate for offloading to a separate VM.

We recast the original in-process FIA to ShadowVM and evaluate the differences in perturbation, coverage, and performance. To assess scalability of FIA under ShadowVM, we run it with both per-thread (which suffices for FIA) and global ordering configurations. The base programs for our evaluation come from the DaCapo suite [23] (release 9.12). Of the fourteen benchmarks in the suite, we excluded tomcat, tradebeans, and tradesoap due to well known issues unrelated to ShadowVM.[6] We also excluded eclipse, which exhibits too non-deterministic behaviour under instrumentation and thus prevents fair comparison.

All experiments were run on a 64-bit multi-core platform with Oracle Hotspot Server VM[7], and with all non-essential system services disabled.

7.1 Perturbation

With respect to perturbation, the ShadowVM approach should improve on classic in-process analysis thanks to the isolation from the observed VM. Consequently, a ShadowVM-based analysis should exhibit minimal (if any) influence on class loading or garbage collections triggered by the base-program.

7.1.1 Class loading perturbation

We first evaluate the class loading perturbation caused by a dynamic analysis. To this end, we simply capture the sequence of classes loaded by the observed VM in response to base-program execution. The data collected when running the uninstrumented base program serve as a reference for comparison with the data collected when running with either the in-process or ShadowVM-based FIA implementation.

Table 1 lists the numbers of classes loaded by the observed VM when running base programs from the DaCapo suite. We note that in the case of the ShadowVM-based FIA implementation, the observed VM loads exactly two more classes than the uninstrumented version. These two classes wrap the native methods designated for reifying the base-program events in the observed VM's agent. In contrast, the in-process FIA implementation loads significantly more classes, because it is implemented using those classes.

7.1.2 Garbage collection perturbation

Next, we evaluate the perturbation in garbage collection behavior. Ideally, an analysis should not influence the memory allocation patterns imposed on the JVM by the observed base program. The experimental setup is similar to that of the previous evaluation, except we collect information on garbage collections performed by the JVM during the execution of the base program. The maximum heap size is limited to two gigabytes and the actual heap size never reaches the limit. Apart from the maximum heap size, the JVM is in default configuration. Again, the data collected when running the uninstrumented base program serve as a reference for comparison.

Table 2 lists the numbers of garbage collections in the young and old generation spaces, the amount of allocated (garbage collected) memory, and the final heap size including the sizes of the young and old generation spaces.

We note that regarding memory consumption and garbage collection, the ShadowVM FIA implementation exhibits very similar behavior compared to that of the uninstrumented base program. There is a slight increase in the total amount of allocated memory, which can be attributed to the FIA tracking each allocated object and passing its reference to the native space. This slightly increases the lifetime of the base program's objects and, more importantly, effectively disables the JIT compiler optimization that converts certain heap allocations to stack allocations, resulting in increased heap consumption. In contrast, the optimization can be still used in the uninstrumented base program.

The in-process FIA implementation reveals a significantly higher memory consumption, because the analysis keeps its state on the heap shared with the base program. Consequently, the allocation rate increases, resulting in a higher number of garbage collections.

7.2 Coverage

With respect to coverage, a ShadowVM-based analysis should improve on classic in-process analysis, because there is no need for a "bypass" mechanism, which enables complete instrumentation of the base program, including the JCL, and including the JVM bootstrap phase. To evaluate the difference in coverage between the two FIA implementations, we compare the total number of object allocations observed by the respective implementation, along with a breakdown of allocations observed by one and not the other implementation. Since the original in-process FIA implementation uses DiSL for base-program instrumentation, it already has a near-complete coverage, with only a small exclusion list. We therefore expect the difference to be small, but still in favor of the ShadowVM-based FIA implementation.

Even though the designers of the DaCapo suite took great care to avoid non-determinism in the benchmarks [23], the allocation profiles vary slightly between benchmark runs, regardless of the FIA implementation used to analyze them. To assess the variability, we have configured the benchmarks for small workload and executed each benchmark ten times with both FIA implementations, collecting the allocation profiles observed during the first iteration in each of the ten runs.

Table 3 shows the number of object allocations observed by both FIA implementations for each of the benchmarks. The variation in the allocation volume is under 0.5% in all benchmarks except h2, where it fits under 0.7%. With the exception of jython, the ShadowVM-based FIA implementation observes slightly more object allocations than the original in-process implementation.

However, in all cases, there are several thousands of objects that are observed by one FIA implementation and not the other. This effect is visible in Table 4 and there are several reasons for the difference, each contributing to the result.

First, there is a slight variability in the allocation profiles between benchmark runs, indicating that the benchmarks do not always allocate the same objects.

Second, the in-process analysis starts tracking object allocations only after the JVM has been initialized, does not track allocations

[6] See bug ID 2955469 and 2934521 in the DaCapo bug tracker at http://sourceforge.net/tracker/?group_id=172498&atid=861957.

[7] 2x Intel Xeon X5650 2.67GHz with 24 cores, 48 GB of RAM, OpenJDK 1.7.0_09-icedtea 64-Bit Server VM (build 23.2-b09) running on Fedora 18

Benchmark	Uninstrumented			In-process FIA			ShadowVM FIA		
	GC young/old	Allocated memory	Final heap size	GC young/old	Allocated memory	Final heap size	GC young/old	Allocated memory	Final heap size
avrora	1/1	65 906	740 480	218/1	26 675 365	639 456	1/1	66 548	740 480
batik	1/1	122 865	740 480	10/1	2 230 259	997 792	1/1	127 293	740 480
fop	1/1	62 600	740 480	4/1	887 367	856 371	1/1	68 568	740 480
h2	3/1	956 686	933 632	92/1	49 247 450	1 124 621	5/1	1 076 014	740 480
jython	1/1	172 068	740 480	73/1	12 023 437	1 075 930	1/1	177 161	740 480
luindex	1/1	32 029	740 480	2/1	450 167	740 480	1/1	39 170	740 480
lusearch	4/1	729 793	933 632	14/1	6 835 034	1 168 493	4/1	721 823	740 480
pmd	1/1	35 912	740 480	2/1	271 791	740 480	1/1	38 070	740 480
sunflow	2/1	306 649	740 480	22/1	11 953 069	1 174 035	2/1	218 245	740 480
xalan	1/1	190 011	740 480	11/1	5 083 672	1 163 386	1/1	192 811	740 480

Table 2. Memory characteristics presented as mean over ten runs. Final heap size and allocated memory shows the size in kilobytes.

Benchmark	In-process FIA		ShadowVM FIA	
avrora	830 972 ±	0.32 %	849 675 ±	0.42 %
batik	376 728 ±	0.29 %	383 638 ±	0.27 %
fop	352 346 ±	0.00 %	359 032 ±	0.00 %
h2	15 999 644 ±	0.66 %	16 028 646 ±	0.57 %
jython	2 449 022 ±	0.00 %	2 443 509 ±	0.00 %
luindex	38 528 ±	0.01 %	42 317 ±	0.01 %
lusearch	840 635 ±	0.00 %	843 682 ±	0.00 %
pmd	69 697 ±	0.01 %	75 985 ±	0.01 %
sunflow	2 303 802 ±	0.00 %	2 307 116 ±	0.00 %
xalan	694 117 ±	0.02 %	699 041 ±	0.03 %

Table 3. Average number of allocations observed (± sample mean standard deviation)

Benchmark	Objects observed only by			
	In-process FIA		ShadowVM FIA	
avrora	506	0.06 %	19 209	2.26 %
batik	676	0.18 %	7586	1.98 %
fop	872	0.25 %	7559	2.11 %
h2	163 690	1.02 %	192 692	1.20 %
jython	9483	0.39 %	3971	0.16 %
luindex	350	0.91 %	4139	9.78 %
lusearch	386	0.05 %	3434	0.41 %
pmd	603	0.86 %	6891	9.07 %
sunflow	376	0.02 %	3690	0.16 %
xalan	3616	0.52 %	8540	1.22 %

Table 4. Average number of objects observed only by one implementation of the field-immutability analysis but not the other (percentages relative to number of objects observed by the respective implementation)

originating in daemon threads to avoid triggering undefined behavior when manipulating weak references, and bypasses the instrumentation when using JCL classes. The ShadowVM implementation, on the other hand, tracks allocations during the whole run of the benchmark, including JVM initialization. Therefore, even if the same objects are observed later, the in-process analysis cannot determine their allocation site and they appear distinct in the comparison.

Third, the in-process analysis may perturb the benchmark state through sharing JVM resources with the base program, resulting in allocations unique for that analysis.

And finally, the two analyses do not have a common point at which they stop tracking object allocations. The ShadowVM-based implementation stops upon receiving the "VM Death" event, while the in-process implementation ends when the JVM executes a pre-registered shutdown hook. Unfortunately, there is no documented relation between the two events—we observe the JVM to still execute some bytecode after emitting the JVMTI "VM Death" event.

In our experiments, the input data of Table 4 for avrora and h2 exhibit high variability, suggesting the reported mean value for those benchmarks is not informative. Still, the huge difference in observed events between the in-process and the ShadowVM FIA implementation for avrora reflects the fact that the number of events observed by ShadowVM is orders of magnitude higher than by in-process analysis.

For h2, the situation is more complicated. In two thirds of the runs, the ShadowVM FIA observes more events than the in-process version. However, for some runs, the number of events observed by the in-process FIA can be up to 5 times higher than in the ShadowVM version. This might indicate some kind of state perturbation in the in-process version, causing more objects to be allocated.

The behavior of jython is also unexpected. It is the only benchmark, where the number of observed allocations is higher with the in-process FIA. The instrumentation coverage of the in-process version is lower compared to the ShadowVM version. We were unable to find the reason for five thousand unique allocations among two and half million, and again we suspect that the in-process FIA may cause some shared state perturbation.

In summary, the ShadowVM FIA implementation is able to capture class loading events and daemon thread events missed by the in-process version. The behavior of some of the benchmarks, when observed using the in-process FIA leads us to believe that our goal of reducing perturbation in the observed system makes sense.

7.3 Performance

In this section, we evaluate the steady-state performance of the in-process and ShadowVM FIA implementations with the DaCapo benchmarks. As mentioned earlier, the ShadowVM FIA implementation is used with both per-thread and global ordering to evaluate the two main ordering configurations.

The experimental setup is identical to the previous evaluations. To obtain mean execution time, we execute each benchmark 5 times in a new process. To obtain steady-state results, we collect the execution time of the fifth iteration of the benchmark during each execution. Measuring execution time after reaching the steady-state provides time for the JIT compiler to optimize the base program code. The measured overhead can be then attributed only to the execution of the inserted hook code and event forwarding.

Table 5 shows the runtime overhead of the steady-state scenario, with the in-process FIA as the baseline. The steady state performance of the ShadowVM FIA is typically about two times worse than the in-process analysis, the worst observed slowdown being a factor

		ShadowVM			
	In-process	per-thread ordering		global ordering	
Benchmark	[ms]	[ms]	overhead	[ms]	overhead
avrora	141 307	851 782	6.03	849 792	6.01
batik	9563	19 796	2.07	26 734	2.80
fop	5072	6240	1.23	8619	1.70
h2	82 831	157 781	1.90	233 792	2.82
jython	14 473	27 681	1.91	34 989	2.42
luindex	1491	3922	2.63	5219	3.50
lusearch	23 693	360 220	15.20	250 892	10.59
pmd	1430	1774	1.24	2359	1.65
sunflow	57 466	133 843	2.33	158 307	2.75
xalan	18 631	276 160	14.82	232 416	12.47

Table 5. Average steady-state execution time of the in-process FIA and the ShadowVM FIA using per-thread and global ordering configurations. The overhead of the ShadowVM FIA uses the execution time of the in-process FIA as a reference.

		ShadowVM concurrent tagging			
	In-process	4 bench. threads		8 bench. threads	
Benchmark	[ms]	[ms]	overhead	[ms]	overhead
avrora	141 307	606 356	4.29	600 930	4.25
lusearch	23 693	47 843	2.02	27 130	1.15
xalan	18 631	37 322	2.00	18 951	1.02

Table 6. Average steady-state execution time of the in-process FIA and an experimental (concurrent tagging) ShadowVM FIA using per-thread ordering. The ShadowVM overhead is calculated with the in-process FIA as a reference.

of fifteen. Besides the overhead of marshaling inherent to the ShadowVM design, the main sources of overhead are related to object tagging and creation of global references in native code. Both facilities are provided by the JVM, but their implementation represents a major bottleneck for the ShadowVM use case.

A small but systematic difference is visible when comparing per-thread and global-ordering configurations. In most cases, the relaxed synchronization of the per-thread configuration is beneficial, however, for a few benchmarks the per-thread configuration performs worse than global-ordering. After further investigation, we believe this effect is caused by excessively fine-grained synchronization between the benchmark threads inside the native code executed as a part of the inserted analysis hooks.

To separate the synchronization effects due to Hotspot JVM from the performance of ShadowVM, we have modified the Hotspot JVM to support concurrent object tagging (the tags are normally kept in a globally locked hash map). The essence of the change was replacing the hash map with a concurrent one. Table 6 shows the performance of a ShadowVM prototype adjusted to run with concurrent tagging for the three benchmarks that exhibited the most pronounced synchronization effects. The adjustment reduces the analysis overhead significantly, unfortunately, requiring proprietary virtual machine adjustments goes against many benefits of analyses based on bytecode instrumentation. Still, we believe the illustrated benefits would justify introducing similar adjustment into the standard Hotspot VM.

8. Related Work

Binary translation systems including Pin [6], Valgrind [19], and DynamoRIO face similar issues of isolating analysis code from the observed program. By performing instrumentation directly at the machine code level, they avoid our complications in escaping from the Java world. Conversely, they are a poor fit for observing managed runtimes, since the abstractions of the VM (such as objects, references to objects, reflective information, and VM threads if not implemented natively) are not easily visible from instrumentation code. Use of private dynamic compilation infrastructure means that baseline slowdown is high (around 2x–5x)—especially when instrumenting a JVM, where two levels of dynamic compilation are now operating.

Shadow profiling [26] and SuperPin [29] support running analysis code asynchronously in overlapping slices, which, given enough cores, can together analyse all events produced by the observed program. However, they work well only if there are no data dependencies between the work done by distinct slices. In practice, since the places where slices begin and end are dictated by rates of production and consumption, handling slice boundaries is problematic and can introduce divergence or loss of coverage [14]. The use of `fork()` to create slices also limits these systems to analysis of single-threaded applications.

Several instrumentation frameworks for Java bytecode may be used to create dynamic analyses, including Javassist [7], Soot [21], ASM[8] and DiSL [16]. These vary in details and expressiveness, but crucially, none assists in isolating the analysis from the observed VM, nor supports asynchronous processing.

BTrace[9] conservatively disallows all potentially dangerous instrumentation in its default configuration—providing a form of isolation, but also limiting its expressiveness to simple applications (e.g. its inability to perform reflection makes it unable to model object fields).

A notable exception is Chord [18], which supports piping a trace of events to a separate process for analysis. This isolates analysis from the program and allows full coverage (§2.1). However, since each instrumentation snippet contends for a shared buffer in this mode, heavy instrumentation effectively serializes the program, in contrast to our flexible approach (§4.3). In addition, Chord's "multi-JVM mode" offers less straightforward support API relative to the unisolated default mode. In particular, program metadata such as class and method names is only available by accessing files dumped from the instrumented JVM, making it more difficult to use.

The RoadRunner dynamic analysis framework [11] caters to data race detectors and closely related dynamic analyses. A key innovation is its compositional pipe-and-filter design. However, unlike Unix pipes, processing along the pipeline is still done synchronously. This makes sense since race detection is highly order-sensitive. However, as a consequence, it cannot introduce parallelism, making it unsuitable (unlike our system) for analyses with weaker ordering requirements. Moreover, the analysis developer is offered no assistance in ensuring isolation of program from analysis.

Aftersight [8] offers a platform for "decoupled" dynamic program analyses, based on the record-replay infrastructure of the VMware virtual machine monitor. Programs are observed under record, generating a log, which is analysed using a special CPU emulator (based on QEmu [3]) which replays the observed program. Observed workloads can be run "behind" the analysis for real-time monitoring, at a cost of slowdown, or else analysis can be run offline with only modest recording overhead. The main contrast with our work is that multiprocessor workloads are not supported: if a multithreaded program is observed, it is implicitly serialized.

Pipa [20] is an extension of dynamic binary translators which provides an efficient representation of profiling data suitable for fast handoff to an asynchronous (pipelined) processing stage, together with carefully optimised dynamic instrumentation code at the binary level. Meanwhile, CAB [14] provides a cache-friendly buffering

[8] http://asm.ow2.org/

[9] http://kenai.com/projects/btrace

design which offers further performance improvement. Since our current implementation lacks cache-aware buffering, uses fairly naive data encoding, and relies on the host JVMs for dynamic compilation, we believe CAB and Pipa to be complementary to our work, in that these techniques could be used to further increase the performance of our approach.

Problems related to full-coverage bytecode instrumentation are mentioned in the literature. The "Twin Class Hierarchy" (TCH) [9] claims to support user-defined instrumentation of the standard JCL by replicating the full hierarchy of the instrumented JCL in a separate package. This has drawbacks in that applications need to be instrumented to explicitly refer to the desired version of the JCL (original or instrumented), but more importantly, that in the presence of native code, call-backs from native code into bytecode will not reach the instrumented code [27]. TCH is therefore not suited for comprehensive instrumentation, as it fails to transparently instrument the JCL. Saff et al. [24] deem the dynamic instrumentation of the JCL to be impossible.

9. Conclusions and future work

ShadowVM allows developers to write dynamic analyses using convenient high-level languages and APIs, retaining the feel of a bytecode instrumentation system but achieving higher levels of isolation and coverage than previous systems. Its contributions include the disciplined use of native code to ensure isolation, the provision of distinct ordering models to allow efficient asynchronous analysis, and the avoidance of numerous coverage gaps that afflict previous systems. We believe it is the first system offering genuinely full bytecode coverage for the JVM. Despite the addition of a process separation, its performance is acceptable for many use cases.

Considerable future work stands to further improve ShadowVM. Coverage could be improved by allowing instrumentation of JNI interactions with the VM, and of VM-internal events currently exposed only through JVMTI callbacks. For analysing some program behaviours, particularly memory usage, a deeper understanding of VM-internal activity has previously been shown to be helpful [15]. A different transport strategy, perhaps based on shared memory instead of socket communication, could potentially also improve performance, although careful coordination with the garbage collector will be required to make shared memory work reliably. We believe that careful extensions to existing JVM implementations could significantly improve the performance of object tagging and global references, which have proven to be bottlenecks in the current implementation. More generally, the optimal observation mechanism will likely require invasive modifications to existing VM implementations and, indeed, their architectures. Meanwhile, ShadowVM constitutes (to our knowledge) the most comprehensive portable solution.

Acknowledgments

This work was supported by the Swiss National Science Foundation (project CRSII2_136225), by a Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04–092010), by the European Commission (Seventh Framework Programme grant 287746), by the Grant Agency of the Czech Republic project GACR P202/10/J042), by the EU project ASCENS 257414, and by Charles University institutional funding SVV-2013-267312.

References

[1] A. Sewe, et al. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In *Proc. ISMM '12*, pages 97–108. ACM, 2012.

[2] B. Cantrill, et al. Dynamic instrumentation of production systems. In *Proc. ATEC '04*, pages 15–28. USENIX Association, 2004.

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proc. ATEC '05*, pages 41–41. USENIX Association, 2005.

[4] Hans-J. Boehm. Destructors, finalizers, and synchronization. In *Proc. POPL '03*, pages 262–272. ACM, 2003.

[5] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, MIT, 2004. AAI0807735.

[6] C. Luk, et al. Pin: building customized program analysis tools with dynamic instrumentation. In *Proc. PLDI '05*, pages 190–200. ACM, 2005.

[7] S. Chiba. Load-time structural reflection in Java. In *Proc. ECOOP'00*, pages 313–336. Springer-Verlag, 2000.

[8] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *Proc. ATC'08*, pages 1–14. USENIX Association, 2008.

[9] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *Proc. OOPSLA '04*, pages 288–300. ACM, 2004.

[10] C. Flanagan and S. N. Freund. FastTrack: efficient and precise dynamic race detection. In *Proc. PLDI '09*, pages 121–133. ACM, 2009.

[11] C. Flanagan and S. N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *Proc. PASTE '10*, pages 1–8. ACM, 2010.

[12] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A call graph execution profiler. In *Proc. SIGPLAN '82*, pages 120–126. ACM, 1982.

[13] J. Gosling, et al. *Java(TM) Language Specification, The (Java SE 7 Edition, 4th Edition)*. Addison-Wesley Professional, 2013.

[14] J. Ha, et al. A concurrent dynamic analysis framework for multicore hardware. In *Proc. OOPSLA '09*, pages 155–174. ACM, 2009.

[15] K. Ogata, et al. A study of Java's non-Java memory. In *Proc. OOPSLA '10*, pages 191–204. ACM, 2010.

[16] L. Marek, et al. DiSL: a domain-specific language for bytecode instrumentation. In *Proc. AOSD '12*, pages 239–250. ACM, 2012.

[17] P. Moret, W. Binder, and É. Tanter. Polymorphic bytecode instrumentation. In *Proc. AOSD '11*, pages 129–140. ACM, 2011.

[18] Mayur Naik. Chord user guide, March 2011. URL http://pag-www.gtisc.gatech.edu/chord/user_guide/. Retrieved on 2013/3/28.

[19] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, 2007.

[20] Q. Zhao, et al. Pipa: pipelined profiling and analysis on multi-core systems. In *Proc. CGO '08*, pages 185–194. ACM, 2008.

[21] R. Vallée-Rai, et al. Optimizing Java bytecode using the Soot framework: Is it feasible? In *Proc. CC '00*, pages 18–34. Springer-Verlag, 2000.

[22] S. Kell, et al. The JVM is not observable enough (and what to do about it). In *Proc. VMIL '12*, pages 33–38. ACM, 2012.

[23] S. M. Blackburn, et al. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. OOPSLA '06*, pages 169–190. ACM, 2006.

[24] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. ASE '05*, pages 114–123. ACM, 2005.

[25] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proc. ESEC/FSE-13*, pages 263–272. ACM, 2005.

[26] T. Moseley, et al. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proc. CGO '07*, pages 198–208. IEEE Computer Society, 2007.

[27] E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *Proc. GPCE '06*, pages 89–94. ACM, 2006.

[28] W. Enck, et al. TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. OSDI'10*, pages 1–6. USENIX Association, 2010.

[29] S. Wallace and K. Hazelwood. Superpin: Parallelizing dynamic instrumentation for real-time performance. In *Proc. CGO '07*, pages 209–220. IEEE Computer Society, 2007.

Chapter 5
Introduction to Dynamic Program Analysis with DiSL

Lukáš Marek,
Yudi Zheng,
Danilo Ansaloni,
Lubomír Bulej,
Aibek Sarimbekov,
Walter Binder,
Petr Tůma

Accepted for publication in **5th Special Issue on Experimental Software and Toolkits**

In Science of Computer Programming,
to be published by Elsevier,
In Press.

The original version is available electronically from the publisher's site at `http://dx.doi.org/10.1016/j.scico.2014.01.003`.

Introduction to Dynamic Program Analysis with DiSL

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

[a] *Charles University, Czech Republic*
[b] *University of Lugano, Switzerland*

Abstract

Dynamic program analysis (DPA) tools assist in many software engineering and development tasks, including profiling, program comprehension, and performance model construction and calibration. On the Java platform, many DPA tools are implemented either using aspect-oriented programming (AOP), or rely on bytecode instrumentation to modify the base program code. The pointcut/advice model found in AOP enables rapid tool development, but does not allow expressing certain instrumentations due to limitations of mainstream AOP languages—developers thus use bytecode manipulation to gain more expressiveness and performance. However, while the existing bytecode manipulation libraries handle some low-level details, they still make tool development tedious and error-prone. Targeting this issue, we provide the first complete presentation of DiSL, an open-source instrumentation framework that reconciles the conciseness of the AOP pointcut/advice model and the expressiveness and performance achievable with bytecode manipulation libraries. Specifically, we extend our previous work to provide an overview of the DiSL architecture, advanced features, and the programming model. We also include case studies illustrating successful deployment of DiSL-based DPA tools.

Keywords:
dynamic program analysis, bytecode instrumentation, aspect-oriented programming, domain-specific languages, Java Virtual Machine

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

1. Introduction

With the growing complexity of software systems, there is an increased need for tools that allow developers and software engineers to gain insight into the dynamics and runtime behavior of those systems during execution. Such insight is difficult to obtain from static analysis of the source code, because the runtime behavior depends on many other factors, including program inputs, concurrency, scheduling decisions, and availability of resources.

Software developers therefore use various *dynamic program analysis* (DPA) tools, that observe a system in execution and distill additional information from its runtime behavior. The existing DPA tools can aid in a variety of tasks, including profiling [1, 2], debugging [3, 4, 5, 6], and program comprehension [7, 8], with increasingly sophisticated tools being introduced by the research community.

In the context of model-driven engineering (MDE), and specifically the area of model-based performance prediction and engineering, dynamic analyses can aid in automating construction and calibration of performance models. While there is a long-lasting trend towards deriving software performance models from other models created during development [9], a well-designed dynamic analysis can aid in construction of performance models for existing software and runtime platforms. Viewed from the perspective of aspect-oriented modeling approaches [10], dynamic analyses related to runtime performance monitoring can be considered crosscutting concerns, and represented as model aspects [11, 12] intended for composition with primary models.

The construction of DPA tools is difficult, in part because of the need to rewrite the base program code to capture occurrences of important events in the base program execution. On the Java platform, *bytecode instrumentation* is the prevailing technique used by existing DPA tools to modify the base program code. Libraries such as ASM [13] and BCEL [14] are often used to manipulate the bytecode, raising the level of abstraction to the level of classes, methods, and sequences of bytecode instructions, and relieving developers of the lowest-level details, such as handling the Java class files.

However, even with the bytecode manipulation libraries, implementing the instrumentation for a DPA tool is error-prone, requires advanced developer expertise, and results in code that is verbose, complex, and difficult to

maintain. While frameworks such as Soot [15], Shrike [16], or Javassist [17], raise the level of abstraction further, they often target more general code transformation or optimization tasks in their design, which does not necessarily help in the instrumentation development.

Some researchers and authors of various DPA tools have thus turned to aspect-oriented programming (AOP) [18], which offers a convenient, 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). This pointcut/advice model allows expressing certain instrumentations in a very concise manner, thus greatly simplifying the instrumentation development. Tools like the DJProf profiler [19], the RacerAJ data race detector [20], or the Senseo Eclipse plugin for augmenting static source views with dynamic metrics [8], all implemented using AspectJ [18], are examples of a successful application of this approach.

Despite the convenience of the AOP-based programming model, mainstream AOP languages such as AspectJ only provide a limited selection of join point types. The resulting lack of flexibility and expressiveness then makes many relevant instrumentations impossible to implement, forcing researchers back to using low-level bytecode manipulation libraries. Moreover, with AOP being primarily designed for purposes other than instrumentation, the high-level programming model and language features provided by AOP are often expensive in terms of runtime overhead [21].

To reconcile the convenience of the pointcut/advice model found in AOP and the expressiveness and performance attainable by using the low-level bytecode manipulation libraries, we have previously presented DiSL [22, 23, 24], an open-source framework that enables rapid development of efficient instrumentations for Java-based DPA tools. To raise the level of abstraction, DiSL adopts the AOP-based pointcut/advice model, which allows one to express instrumentations in a very concise manner, similar to AOP aspects. To retain the flexibility of low-level bytecode manipulation libraries, DiSL features an *open join-point model*, which allows any region of bytecodes to represent a join point. To achieve the performance attainable with low-level libraries, DiSL provides specialized features that provide constant-time access to static information related to an instrumentation site, which is computed at weave time, and features that allow caching and passing data between inserted code in different places. These features enable implementation of

efficient instrumentations, without incurring the overhead caused by having to resort to very high-level, but costly, AOP features.

In addition, DiSL supports *complete bytecode coverage* [25] and mostly avoids[1] structural modifications of classes (i.e., adding methods and fields) that would be visible through the Java reflection API and could break the base program.

The general contribution of this paper is in providing the first complete presentation of the DiSL framework that serves as introduction to dynamic program analysis with DiSL. In previous work, we presented the design and the basic features of the DiSL framework [22, 23, 24]. Here we extend our previous work in the following directions:

- we provide an overview of the DiSL framework architecture;
- we present the advanced features that contribute to the flexibility and performance of the DiSL framework;
- we present case studies based on reimplementing existing tools using DiSL;
- we provide a tutorial-style introduction to DiSL programming, to help developers of DPA tools to get started with DiSL;
- and finally, we include step-by-step instruction on how to obtain, compile, and run DiSL-based analyses on base programs.

The rest of the paper is structured as follows: Section 2 stems from [24] and introduces the DiSL framework using a simple execution time profiler as a running example, pointing out the advantages of using DiSL to implement the instrumentation. Section 3 provides an overview of the advanced features of DiSL, which also serve as extension points of the DiSL framework. Section 4 presents the high-level architecture of DiSL and provides overview of the instrumentation process. In Section 5 we provide an overview of selected case studies performed during the development of DiSL, and we discuss related work in Section 6. Section 7 concludes the paper.

[1] For performance reasons, DiSL modifies the `java.lang.Thread` class. DiSL also allows an arbitrary user-defined class transformation to become part of the instrumentation process. Both exceptions are discussed in Section 3.5.

2. DiSL by Example

A common example of a dynamic program analysis tool is a method execution time profiler, which usually instruments the method entry and exit join points and introduces storage for timestamps. We describe the main features of DiSL by gradually developing the instrumentation for such a profiler. The same instrumentation is also available on the DiSL home page[2] among the examples. For step-by-step instructions on how to run the examples, please refer to Appendix A.

Note that we intentionally use simple code to illustrate DiSL concepts, oblivious to the overhead it may cause. A developer writing a real instrumentation-based profiler would have to be much more conscious about the overhead introduced by the inserted code. To manage overhead, the developer would have to avoid expensive operations such as memory allocations, string concatenations, or repeated queries for information, as much as possible. To help with that, DiSL provides features that allow caching and passing data between snippets (synthetic local variables and thread local variables), and that allow moving computation of static information to weave time (custom static context). We introduce these features shortly.

2.1. Method Execution Time Profiler

In the first version of our execution time profiler, we simply print the entry and exit times for each method execution as it happens. For that, we need to insert instrumentation at the method entry and method exit join points.

Each DiSL instrumentation is defined through methods declared in standard Java classes. Each method—called *snippet* in DiSL terminology—is annotated so as to specify the join points where the code of the snippet shall be inlined.[3] The profiler instrumentation code on Figure 1 uses two such snippets, the first one prints the entry time, the second one the exit time.

The code uses two annotations to direct inlining. The `@Before` annotation requests the snippet to be inlined before each marked bytecode region (representing a join point); the use of the `@After` annotation places the second snippet after (both normal and abnormal) exit of each marked region. The regions themselves are specified with the `marker` parameter of the annotation. In our example, `BodyMarker` marks the whole method (or constructor) body.

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

[3] The method name can be arbitrarily chosen by the programmer.

```
public class SimpleProfiler {

  @Before(marker=BodyMarker.class)
  static void onMethodEntry() {
    System.out.println("Method entry " + System.nanoTime());
  }

  @After(marker=BodyMarker.class)
  static void onMethodExit() {
    System.out.println("Method exit " + System.nanoTime());
  }
}
```

Figure 1: Instrumenting method entry and exit

The resulting instrumentation thus prints a timestamp upon method entry and exit.

Instead of printing the entry and exit times, we may want to print the elapsed wall-clock time from the method entry to the method exit. The elapsed time can be computed in the after snippet, but to perform the computation, the timestamp of method entry has to be passed from the before snippet to the after snippet.

In traditional AOP languages, which do not support efficient data exchange between advices, this situation would be handled using a local variable within the around advice. In contrast, an instrumentation framework such as DiSL has no need for the usual form of the around advice, which lets the advice code decide whether to skip or proceed with the method invocation [22]. DiSL therefore only supports inlining snippets before and after a particular join point, together with a way for the snippets inlined into the same method to exchange data using *synthetic local variables* [26], as illustrated on Figure 2.

Synthetic local variables are static fields annotated as @SyntheticLocal. The variables have the scope of a method invocation and can be accessed by all snippets that are inlined in the method; that is, they become local variables. Synthetic local variables are initialized to the default value of their declared type (e.g., 0, false, null).

Next, we extend the output of our profiler to include the name of each profiled method. In DiSL, the information about the instrumented class,

```
public class SimpleProfiler {

  @SyntheticLocal
  static long entryTime;

  @Before(marker=BodyMarker.class)
  static void onMethodEntry() {
    entryTime = System.nanoTime();
  }

  @After(marker=BodyMarker.class)
  static void onMethodExit() {
    System.out.println("Method duration " + (System.nanoTime() - entryTime));
  }
}
```

Figure 2: Passing data between snippets using a synthetic local variable

method, and bytecode region can be obtained through dedicated *static context interfaces*. In this case, we are interested in the MethodStaticContext interface, which provides the method name, signature, modifiers and other static data about the intercepted method and its enclosing class. Figure 3 refines the after snippet of Figure 2 to access the fully qualified name of the instrumented method.

```
@After(marker=BodyMarker.class)
static void onMethodExit(MethodStaticContext msc) {
  System.out.println(msc.thisMethodFullName() + " duration "
    + (System.nanoTime() - entryTime));
}
```

Figure 3: Accessing the method name through static context

Static context interfaces provide information that is already available at the instrumentation time. When inlining the snippets, DiSL therefore replaces the calls to these interfaces with the corresponding static context information, thus improving the efficiency of the resulting tools.

DiSL provides a set of static context interfaces, which can be declared as arguments to the snippets in any order. The default set of available interfaces was mainly designed to make DiSL immediately useful to instrumentation developers using AspectJ and ASM. The method static context provides

information available in AOP languages such as AspectJ, which we consider to be the minimum. In addition, the set includes interfaces that provide static context information for join points that do not exist in AspectJ, but that we found to be often used in ASM-based DPA tools. This includes basic block static context, which is generally needed in profiling and code coverage analyses with fine-grained resolution, and field access and method invocation static contexts, which are generally needed for shadowing and tracking base program values.

The DiSL programmer may also define custom static context interfaces to perform additional static analysis at instrumentation time or to access information not directly provided by DiSL, but available in the underlying ASM-based bytecode representation.

2.2. Adding Stack Trace

Sometimes knowing the name of the profiled method is not enough. We may also want to know the context in which the method was called. Such context is provided by the stack trace of the profiled method.

There are several ways to obtain the stack trace information in Java, such as calling the getStackTrace() method from java.lang.Thread, but frequent calls to this method may be expensive. Our example therefore obtains the stack trace using instrumentation. Figure 4 shows two additional snippets that maintain the call stack information in a shadow call stack. Upon method entry, the method name is pushed onto the shadow call stack. Upon method exit, the method name is popped off the shadow call stack.

Each thread maintains a separate shadow call stack, referenced by the thread-local variable callStack.[4] In our example, callStack is initialized for each thread in the before snippet. The thread-local shadow call stack can be accessed from all snippets through the callStack variable; for example, it could be included in the profiler output.

To make sure all snippets observe the shadow call stack in a consistent state, the two snippets that maintain the shadow call stack have to be inserted in a correct order relative to the other snippets. DiSL allows the programmer to specify the order in which snippets matching the same join point should be inlined using the order integer parameter in the snippet annotation.

[4]DiSL offers a particularly efficient implementation of thread-local variables with the @ThreadLocal annotation.

```
@ThreadLocal
static Stack<String> callStack;

@Before(marker=BodyMarker.class, order=1000)
static void pushOnMethodEntry(MethodStaticContext msc) {
  if (callStack == null) { callStack = new Stack<String>(); }
  callStack.push(msc.thisMethodFullName());
}

@After(marker=BodyMarker.class, order=1000)
static void popOnMethodExit() {
  callStack.pop();
}
```

Figure 4: Reifying a thread-specific call stack using dedicated snippets

The smaller this number, the closer to the join point the snippet is inlined. In our profiler, the time measurement snippets and the shadow call stack snippets match the same join points (method entry, resp. method exit). We assign a higher order value (1000) to the call stack reification snippets and keep the lower default order value (100) of the snippets for time measurement.[5] Consequently, the callee name is pushed onto the shadow call stack before the entry time is measured, and the exit time is measured before the callee name is popped off the stack.

2.3. Profiling Object Instances

Our next extension addresses situations where the dependency of the method execution time on the identity of the called object instance is of interest. Figure 5 refines the after snippet of Figure 2 by computing the identity hash code of the object instance on which the intercepted method has been called.

The snippet uses the DynamicContext *dynamic context interface* to get a reference to the current object instance. Similar to the static context interfaces, the dynamic context interfaces are also exposed to the snippets as method arguments. Unlike the static context information, which is resolved at instrumentation time, calls to the dynamic context interface are replaced

[5] If snippet ordering is used, it is recommended to override the value in all snippets for improved readability.

```
@After(marker=BodyMarker.class)
static void onMethodExit(MethodStaticContext msc, DynamicContext dc) {
  int identityHC = System.identityHashCode(dc.getThis());
  ...
}
```

Figure 5: Accessing dynamic context information in a snippet

with code that obtains the required dynamic information at runtime. Besides the object reference used in the example, DiSL provides access to other dynamic context information including the local variables, the method arguments, and the values on the operand stack.

2.4. Selecting Profiled Methods

Often, it is useful to restrict the instrumentation to certain methods. For example, we may want to profile only the execution of methods that contain loops, because such methods are likely to contribute more to the overall execution time.

DiSL allows programmers to restrict the instrumentation scope using the *guard* construct. A guard is a user-defined class whose one method carries the @GuardMethod annotation. This method determines whether a snippet matching a particular join point is inlined. Figure 6 shows the signature of a guard restricting the instrumentation only to methods containing loops. The body of the methodContainsLoop() guard method, not shown here, would implement the detection of a loop in a method. A loop detector based on control flow analysis is included as part of DiSL.

```
public class MethodsContainingLoop {

  @GuardMethod
  public static boolean methodContainsLoop() {
    ... // Loop detection based on control flow analysis
  }
}
```

Figure 6: Skeleton of a guard for selecting only methods containing a loop

The loop guard is associated with a snippet using the guard annotation parameter, as illustrated in Figure 7. Note that the loop guard is not used

in the shadow call stack snippets. We want to maintain complete stack trace information without omitting the methods that do not contain loops.

```
@Before(marker=BodyMarker.class, guard=MethodsContainingLoop.class)
static void onMethodEntry() { ... }

@After(marker=BodyMarker.class, guard=MethodsContainingLoop.class)
static void onMethodExit(...) { ... }
```

Figure 7: Applying time measurement snippets only in methods containing a loop

3. Advanced DiSL Features

The features presented so far cover basic DiSL usage. We continue with examples illustrating the more advanced features of DiSL. These can be roughly split into two categories.

The first category includes method argument processing, using other markers from the DiSL marker library, and creating custom static context implementations. These features mostly require the developer to be familiar with DiSL, and some may need a basic knowledge of Java bytecode. The requirements for using a custom static context depend on the actual usage. For example, generating custom names for code locations using the information already available in other static contexts just requires the developer to use the existing API, and to adhere to the requirements for static context implementations.

The second category again includes custom static context, as well as custom custom markers and bytecode transformers. Compared to the previous case, here we consider using custom static context for more complex tasks, such as performing a custom static analysis. Custom markers allow definition of new join points, and custom transformer are just hooks into the instrumentation process, where anything is allowed. Using these features basically means extending DiSL, which will require the developer to have a solid knowledge of DiSL, ASM, and Java bytecode.

The need for features from the first category will come gradually, as a result of using DiSL for more sophisticated instrumentations. The features in the second category are really meant for extending DiSL, and we anticipate that most DiSL users will never use them. We now review each of the features in turn, in the order of increasing developer requirements.

3.1. Analyzing Method Arguments

DiSL provides two different mechanisms for analyzing method arguments. The first approach provides the method arguments to the snippet in an object array. The entire array is constructed dynamically at runtime, with arguments of primitive types boxed. Conceptually simple, the approach requires object allocation and always processes all arguments.

The second approach aims at situations where the overhead of using object arrays is not acceptable. The approach uses code fragments called *argument processors*. Each argument processor analyzes only one type of method arguments. The code of the argument processor is inlined into the snippet where it is applied. With argument processors, it is possible to access method arguments without object allocation.

Technically, the argument processor is an annotated Java class containing argument processing methods. The first argument of each argument processor method is of the type being processed, that is, any basic Java type (int, byte, double ...), String, or an object reference. As additional arguments, the methods can receive dynamic or static contexts, including *argument context*, which is a special kind of static context available only within the argument processor. The ArgumentContext interface exposes information about the currently processed argument and can be used to limit argument processing only to arguments at a particular position or with a particular type. The argument processor methods can also use thread-local or synthetic local variables.

An example of an argument processor that processes int arguments is given in Figure 8.

```
@ArgumentProcessor
public class IntArgumentPrinter {
  public static void printIntegerArgument (
    int val, ArgumentContext ac, MethodStaticContext msc) {

    System.out.printf(
      "Int argument value in method %s at position %d of %d is %d\n",
      msc.thisMethodFullName(), ac.getPosition(), ac.getTotalCount(), val
    );
  }
}
```

Figure 8: A simple argument processor for printing the values of integer arguments

The argument processor is used by applying it in an argument processor context within a snippet. The argument processor context can apply an argument processor in two modes. All snippets can apply the processor on the arguments of the current method. Snippets inserted just before a method invocation can also apply the processor on the invocation arguments. Figure 9 shows a snippet that uses the IntArgumentPrinter argument processor from Figure 8 to print out the values of the integer arguments of the currently executed method.

```
@Before(marker = BodyMarker.class)
public static void onMethodEntry(ArgumentProcessorContext apc) {
  apc.apply(IntArgumentPrinter.class, ArgumentProcessorMode.METHOD_ARGS);
}
```

Figure 9: Using an argument processor within a snippet

3.2. Join Point Marker Library

In all the examples presented earlier, profiles were collected with method granularity. Such profiles may be insufficient when profiling long methods with loops and nested invocations. In these cases, a more fine grained measurement can help identify the problematic parts of the long methods.

In the profiler example, a more fine grained measurement can be achieved using a different marker with the profiling snippets. DiSL provides a library of markers (e.g., BasicBlockMarker, BytecodeMarker) for intercepting many common bytecode patterns; Figure 10 illustrates the use of BasicBlockMarker for basic block profiling.

As presented, the change only impacts the choice of the marker class. Although the resulting instrumentation is valid, the resulting profile is of limited use because it lacks the identification of the basic blocks being profiled. We add this identification next.

3.3. Custom Static Context

There are multiple options for identifying a basic block in the profiler example. We can use the ordinal number of the basic block as made available by the BasicBlockStaticContext; however, such identification is only useful if the information about the correspondence between the basic block numbers and the profiled code is available when interpreting the results. The

```
@Before(marker=BasicBlockMarker.class)
static void onBasicBlockEntry() { ... }

@After(marker=BasicBlockMarker.class)
static void onBasicBlockExit(...) { ... }
```

Figure 10: Writing snippets to profile entry and exit from basic blocks

source code line number is a valuable alternative when working at the source code level, however, the identification is not necessarily unique and the need for additional information when interpreting the results also persists. To provide an example of custom static context, we illustrate a third option, namely identifying the basic block by the ordinal number of its first instruction and its length, counted in the number of instructions (numbers are valid for uninstrumented code). Implementing the other two approaches in DiSL is of similar complexity.

The identification of a basic block is conceptually a part of the static context of each snippet, and it would ideally be available through one of the existing static context interfaces. In this particular case, DiSL actually provides such an interface. However, this may not be true in general. While we aim to equip DiSL with a rich library of static context interfaces offering all the information that may be required by an analysis tool, we can only guess at what information will other analyses—especially new ones—require.

In the two years of development and evaluation, we found the set of static context interfaces provided by DiSL to be sufficient for almost all DPA tools we recasted or developed. A notable exception was an analysis that required static context information for loops, which required performing dominator analysis on basic blocks at weave time. Other than that, most analyses used tiny custom static context implementations that were difficult to generalize—either to obtain information related to a particular type of bytecode instructions (which is easily available from the underlying ASM-based code representation), or to precompute trace messages with static information at weave time (to avoid string concatenation at runtime).

Consequently, DiSL contains mostly static context implementations providing information that was generally useful in the tools that we have recasted so far, including a few that were used rarely, but had non-trivial implementation, such as the loop static context. Since we cannot anticipate what static information will be required by all analyses, we allow DiSL users to define

a custom static context. However, based on our experience, we expect most users to use a custom static context mainly for generating names at weave time, which allows embedding these values in the snippet code as constants.

Figure 11 illustrates a custom static context that serves as the basic block ID calculator. A custom static context is a standard Java class that extends the AbstractStaticContext class or implements the StaticContext interface directly. The methods of the custom static context class have no arguments and return a basic type or String. The BasicBlockID class from Figure 11 contains one such method, getID(), which computes the ID of a basic block.

The computation queries the first and the last instruction of the region identified by the basic block marker. After that, it iterates over the code of the entire method, first incrementing the block index until the basic block start is reached, then incrementing the block length until the basic block end is found. The method returns the ID as String whose first part is the index and second part the length.

Custom static context methods can access the current static context information through a protected field called staticContextData. The available information describes the marked region, snippet, method, and class where the custom static context is used. The region description includes one starting instruction and one or more ending instructions depending on the marker. The snippet structure holds all the information connected to the snippet where the static context is used. The method and class data are represented by ASM objects MethodNode and ClassNode.

3.4. Custom Bytecode Marker

It is not always possible to profile a method by instrumenting its body. For example, the method can be implemented in native code or can execute remotely. To profile such methods, the instrumentation has to be placed around the method invocation.

In DiSL, method invocation can be easily captured by the BytecodeMarker with adequate parameters. To illustrate the extensibility of DiSL, we instead implement a new custom marker that captures method invocations, displayed in Figure 12.

The role of a marker is to select the bytecode regions for instrumentation. A custom bytecode marker in DiSL must implement the Marker interface. Typically, the marker would not implement this interface directly, but instead inherit from the AbstractDWRMarker abstract class, which also takes care of correctly placing the weaving points. In our example, the

```
public class BasicBlockID extends AbstractStaticContext {
  public String getID() {
    // validate that the basic block has only one end
    ...

    // get starting and ending instruction from marker
    AbstractInsnNode startInsn = staticContextData.getRegionStart();
    AbstractInsnNode endInsn = staticContextData.getRegionEnds().get(0);

    // traverse entire method code and calculate instruction index
    int bbStart = -1;
    int bbLength = 0;
    boolean startFound = false;
    boolean endFound = false;
    InsnList code = staticContextData.getMethodNode().instructions;
    for(AbstractInsnNode insn = code.getFirst();
        insn != null; insn = insn.getNext()) {

      // increase block start index until start instruction found
      if(!startFound) {
        if(insn.getOpcode() != -1) ++bbStart;
        startFound = (insn == startInsn);
      }

      if(startFound) {
        // count instructions and exit when end instruction found
        if(insn.getOpcode() != -1) ++bbLength;
        if(insn == endInsn) {
          endFound = true;
          break;
        }
      }
    }

    // validate that both start and end were found
    ...

    // construct and return the basic block ID
    return bbStart + "(" + bbLength + ")";
  }
}
```

Figure 11: Custom static context computing a basic block ID

```
public class MethodInvocationMarker extends AbstractDWRMarker {
  public List<MarkedRegion> markWithDefaultWeavingReg(MethodNode method) {

    List<MarkedRegion> regions = new LinkedList<MarkedRegion>();

    // traverse all instructions
    InsnList instructions = method.instructions;
    for (AbstractInsnNode instruction : instructions.toArray()) {

      // check for method invocation instructions
      if (instruction instanceof MethodInsnNode) {

        // add region containing one instruction (method invocation)
        regions.add(new MarkedRegion(instruction, instruction));
      }
    }

    return regions;
  }
}
```

Figure 12: Custom marker implementing a method invocation join point

MethodInvocationMarker class traverses all instructions using ASM and creates a single-instruction region for each method invocation encountered; the abstract marker class is used to compute all the weaving information automatically.

Note that the example marker captures all method invocations. To reduce the instrumentation scope, the developer should use either a guard or a runtime check.

3.5. Custom Bytecode Transformer

DiSL is designed for writing tools that observe the application without modifying its behavior. It will refuse to insert snippets that would change the application control flow, modify fields, or insert methods or fields to classes. The only exception is the modification of the java.lang.Thread class performed by DiSL to provide very efficient implementation of thread-local variables.[6]

[6]DiSL adds a field to the java.lang.Thread class for every thread-local variable, which significantly outperforms an approach based on the java.lang.ThreadLocal class. Here, the

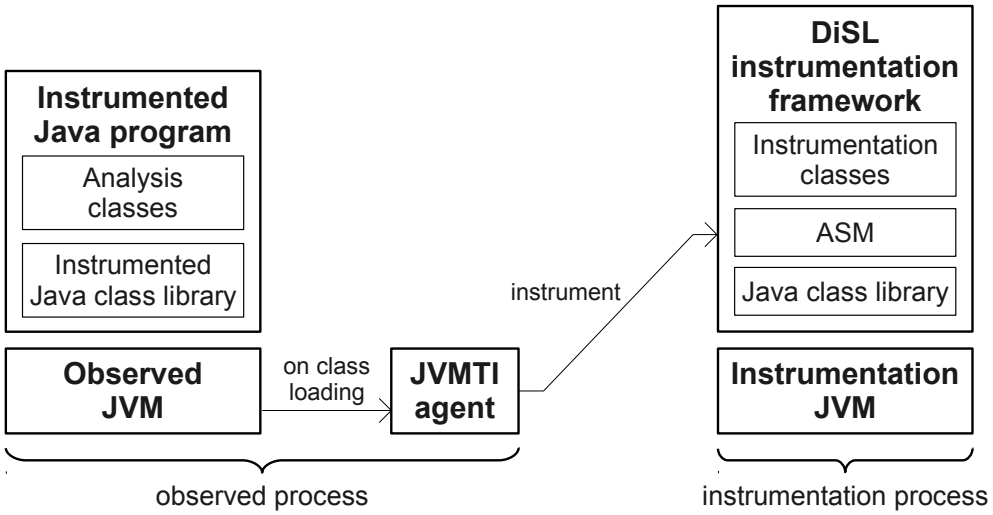


Figure 13: Architecture of DiSL.

However, in special cases, a tool implementation may require application modifications beyond what DiSL allows. This may result if a tool needs to perform structural modifications to the application code, or when the method-level scope provided by DiSL is too narrow. In these cases, DiSL can invoke a custom transformer to modify the class just before it is instrumented.

Custom transformers have to implement the ch.usi.dag.disl.Transformer interface to receive raw class data from DiSL, and to return the modified data back to DiSL. The class data is passed around as an array of bytes, and apart from the Transformer interface, DiSL neither provides any API for class transformation, nor mandates the use of any particular bytecode manipulation framework. The developer is free to modify the class data in any way, typically with the help of a bytecode manipulation framework (e.g., ASM), that is able to parse class from an array of bytes and return the result in the same form.

4. DiSL Architecture and Instrumentation Process

To minimize perturbation in the observed program, DiSL performs bytecode instrumentation within a separate Java Virtual Machine (JVM) process,

performance benefits far outweigh the chance of breaking application behavior, because the modifications are limited to a single system class.

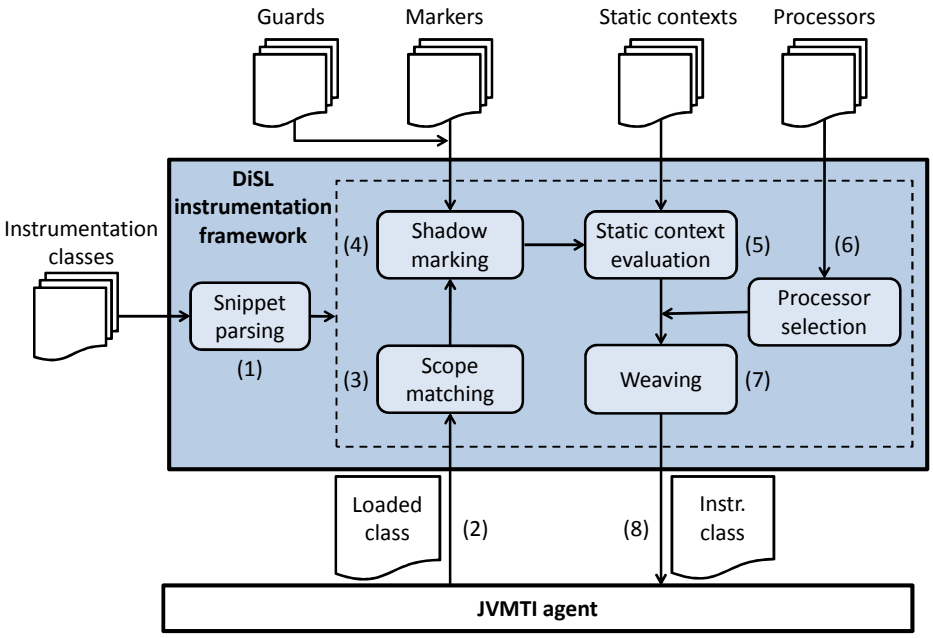


Figure 14: Overview of the DiSL instrumentation process.

that is, the *instrumentation process*. In this way, class loading and initialization triggered by the instrumentation framework do not happen within the *observed process*.

As illustrated in Figure 13, a native JVMTI agent[7] captures all class loading events (starting with java.lang.Object) in the observed JVM and sends every class as a byte array to the DiSL instrumentation framework through a socket. Here, DiSL uses ASM for instrumentation and relies on *polymorphic bytecode instrumentation* [25] to ensure complete bytecode coverage. All classes are instrumented only once, whenever they are loaded by the JVM. While dynamic instrumentation and reinstrumentation at runtime is a work in progress, it is currently not supported by DiSL.

Figure 14 gives an overview of the DiSL instrumentation process. During initialization, DiSL parses all instrumentation classes (step 1). Then it creates an internal representation for snippets and initializes the used markers, guards, static contexts, and argument processors. When DiSL receives a class from the JVMTI agent (step 2), the instrumentation process starts

[7] http://download.oracle.com/javase/6/docs/platform/jvmti/jvmti.html

with the snippet selection. The selection is done in three phases, starting with scope matching (step 3). Then, bytecode regions are marked using the markers associated with the snippets selected in the previous phase. Finally, marked bytecode regions are evaluated by guards and only snippets with at least one valid marked region are selected (step 4).

At this point, all snippets that will be used for instrumentation are known. Static contexts are used to compute the static information required by snippets (step 5). As described in Section 3, custom static contexts allow programmers to declare expressions to be evaluated at instrumentation time. However, if such expressions do not require custom context information, they can simply be embedded in the snippet code. In fact, DiSL can be configured to perform partial evaluation of inlined snippets [23]. This optimization can simplify certain snippet constructs, for example it can make conditional branching on static information in the snippet as efficient as using a guard.

Argument processors are evaluated for snippets, and argument processor methods that match method arguments are selected (step 6). All the collected information is finally used for instrumentation (step 7). Argument processors are applied, and calls to static contexts are replaced with the computed static information. The framework also generates the bytecodes to access dynamic context information. To prevent the instrumentation code from throwing exceptions that could modify the control flow in the observed program, DiSL automatically inserts code intercepting all exceptions originating from the snippets, reporting an error if an exception is thrown (and not handled) by the instrumentation. Finally, the instrumented class is returned to the observed JVM (step 8) where it is linked. For a more detailed description of the process, we refer the interested reader to [22].

5. Case Studies

To demonstrate the benefits of DiSL when developing instrumentation for DPA tools, we present several case studies involving existing DPA tools. The instrumentation parts of those tools were originally implemented either using AspectJ, or the ASM bytecode manipulation library. For evaluation purposes, we reimplemented the instrumentation part of each tool using DiSL and compared both the conciseness and the performance of the DiSL-based instrumentation with the original implementation. We first give a short overview of the recasted tools, and then proceed with the evaluation.

5.1. Tool Descriptions

JP2 [27] is a calling-context profiler for languages targeting the JVM. JP2 uses ASM-based instrumentation to collect various static (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). For each method, the metrics are associated with a corresponding node in a calling-context tree (CCT) [28], grouped by the position of the method call-site in the caller. The collected information can be then used for both inter- and intra-procedural analysis of the application.

JCarder[8] is a tool for finding potential deadlocks in multi-threaded Java applications. JCarder uses ASM-based instrumentation to construct a dependency graph for threads and locks at runtime, and if the graph contains a cycle, JCarder reports a potential deadlock.

Senseo [8] 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 AspectJ-based instrumentation to count method invocations, object allocations, and to collect statistics on method arguments and return types.

RacerAJ [20] is a tool for finding potential data races in multi-threaded Java applications. RacerAJ uses AOP-based instrumentation to monitor 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.

Besides recasting existing tools for evaluation purposes, we successfully used DiSL to develop new field immutability and field sharing analyses, which have been used to compare Java and Scala workloads [29]. These analyses are now part of a comprehensive toolchain for workload characterization for Java and other languages targeting the JVM [30].

5.2. Evaluating Instrumentation Conciseness

In our experience with developing DPA tools with AspectJ and ASM, we consistently found instrumentations developed using the AOP pointcut/advice model very clear and concise—provided that an instrumentation could

[8] http://www.jcarder.org/
[9] http://scg.unibe.ch/research/senseo

	JP2		JCarder		Senseo		RacerAJ	
	DiSL	ASM	DiSL	ASM	DiSL	AspectJ	DiSL	AspectJ
Physical LOC	96	477	89	650	74	44	136	33
Logical LOC	64	375	64	399	44	19	78	24

Table 1: The amount of code (in source lines of code) comprising the instrumentation parts of the analysis tools, written using DiSL, ASM, and AspectJ.

be expressed using the available join points. On the other hand, using ASM allowed us to perform basically any kind of instrumentation with significantly better performance, at the cost of very low-level and verbose instrumentation code, which was rather fragile and thus difficult to maintain. With DiSL, we strive for the simplicity and conciseness of the AOP-based instrumentations, without sacrificing expressiveness and performance provided by ASM.

To evaluate how DiSL compares to AspectJ and ASM in terms of instrumentation conciseness, we consider the amount of code—measured in source lines of code (SLOC)—that needs to be written in AspectJ, ASM, and DiSL to implement an equivalent instrumentation for the evaluated tools. While there is no proof that "less code" automatically translates to "better code", we believe that in this particular context, an implementation that is significantly shorter (i.e., more concise) due to use of better fitting abstractions can be considered easier to write, understand, and maintain. This view is also supported by the results of a controlled user study [31], where DiSL was shown to reduce instrumentation development time and improve instrumentation correctness compared to ASM.

Table 1 summarizes the SLOC counts[10] of both the DiSL-based and the original implementations of instrumentation for each tool. The number of physical SLOC illustrates the overall size of the implementation, while the number of logical SLOC captures the amount of code essential for the implementation. In our comparison, we use the logical SLOC count as an indicator of conciseness for implementations of equivalent instrumentations.

We observe that for the evaluated DPA tools, the DiSL-based implementations of their instrumentation parts require less code than their ASM-based equivalents. This is because because bytecode manipulation, even when using

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

Benchmark	Description
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.
eclipse	Executes some of the (non-gui) JDT performance tests for the Eclipse IDE.
fop	Takes an XSL-FO file, parses it, formats it, and generates a PDF file.
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application.
jython	Inteprets a the pybench Python benchmark.
luindex	Uses lucene to index a set of documents comprising the works of Shakespeare and the King James Bible.
lusearch	Uses lucene to do a keyword search over a corpus of data comprising the works of Shakespeare and the King James Bible.
pmd	Analyzes a set of Java classes for a range of source code problems.
sunflow	Renders a set of images using ray tracing.
xalan	Transforms XML documents into HTML.
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting web pages.
tradebeans	Runs the daytrader benchmark via Java Beans to a GERONIMO back-end with an in-memory h2 as the underlying database.
tradesoap	Runs the daytrader benchmark via SOAP to a GERONIMO back-end with an in-memory h2 as the underlying database.

Table 2: Overview of benchmarks from the DaCapo suite. Benchmarks at the bottom were excluded from evaluation due to hard-coded timeouts and well-known problems.

ASM, results in very verbose code. On the other hand, 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 a separate language. Moreover, DiSL instrumentations also include code that is evaluated at instrumentation time, which increases the source code size, but provides significant performance benefits at runtime [22]. However, in the context of instrumentations for DPA, DiSL is more flexible and expressive than AOP without impairing the performance of the resulting tools, as shown in the following section.

5.3. Evaluating Instrumentation Performance

To evaluate the instrumentation performance, we compare the execution time of the original and the recasted tools on benchmarks from the DaCapo [32] suite (release 9.12). Table 2 provides an overview of the DaCapo

	JP2		JCarder		Senseo		RacerAJ	
	ASM	DiSL	ASM	DiSL	AspectJ	DiSL	AspectJ	DiSL
avrora	9.73	10	1.31	1.83	4.48	2.68	110.29	32.58
batik	4.77	5.5	1.05	1.05	2.18	1.43	31.23	6.64
eclipse	3.08	3.46	1.01	1.53	8.28	5.97	11.94	14.82
fop	6.17	7.94	1.1	1.32	9.9	4.54	49.06	12.56
h2	12.21	13.8	2.11	3.79	10.64	4.14	157.7	79.07
jython	26.28	30.36	7.92	4.54	2.19	1.52	236.8	104.4
luindex	2.63	2.88	1.25	1.26	9.34	3.06	60.31	16.25
lusearch	24.18	31.82	7.78	8.09	7.88	3.7	147.71	49.06
pmd	4.8	7.4	1.07	1.07	3.52	2.09	53.8	24
sunflow	10.53	9.85	1.05	1.03	39.28	12.52	398.18	211.22
xalan	27.94	26.22	26.08	28.19	29.25	5.05	68.89	29.33
geomean	8.83	10.09	2.24	2.47	7.69	3.5	81.01	32.26

Table 3: Overhead factors for original and recasted tools when run on the benchmarks from the DaCapo suite.

benchmarks, as presented on the DaCapo suite web site.[11] Of the fourteen benchmarks present in the suite, we excluded the tradeswap, tradebeans, and tomcat due to well known issues[12] unrelated to DiSL. All experiments were run on a multicore platform[13] with all non-essential system services disabled.

We report results for steady-state performance in Table 3. For each benchmark, we report a steady-state overhead factor, determined from a single run with 10 iterations of each benchmark, with the first 5 iterations excluded 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. As a summary metric, we also report the geometric mean of overhead factors from all benchmarks.

The results in Table 3 indicate that the recasted tools are typically roughly as fast as their ASM-based counterparts (JP2, JCarder), but never much

[11] http://www.dacapobench.org/benchmarks.html

[12] 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.

[13] 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.

slower. Significant performance improvements can be observed in the case of AOP-based tools (Senseo, RacerAJ), which 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. 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.

6. Related Work

In previous work, we presented @J [21], a Java annotation-based AOP language for simplifying dynamic analysis. Compared to DiSL, @J does not provide an open join point model and efficient access to dynamic context information. DiSL guards can be emulated by means of staged advice, where weave-time evaluation of advice yields runtime residues that are woven. However, this requires the use of synthetic local variables and a more complex composition of snippets.

In [33] we discussed some early ideas on a high-level declarative domain-specific aspect language (DSAL) for dynamic analysis. DiSL provides all necessary language constructs to express the dynamic analyses that can be specified in the DSAL. That is, in the future, DiSL can serve as an intermediate language to which the higher-level DSAL programs are compiled.

High-level dynamic analysis frameworks such as RoadRunner [34] or Chord[14] ease composition of a set of dynamic analyses. However, such approaches do not support an open join point model and the set of context information that can be accessed at intercepted code regions is not extensible.

In [35], a meta-aspect protocol (MAP) is proposed to separate the host language from analysis-specific aspect languages. MAP supports an open join point model and advanced deployment methods (i.e., global, per object, and per block). While MAP allows fast prototyping of custom analysis languages, it does not focus on high efficiency of the developed analysis tools.

Josh [36] is an AspectJ-like language that allows developers to define domain-specific extensions to the pointcut language. Similar to guards, Josh

[14] http://pag.gatech.edu/chord/

provides static pointcut designators that can access reflective static information at weave-time. However, the join point model of Josh does not include arbitrary bytecodes and basic blocks that are readily available in DiSL.

The approach described in [37] enables customized pointcuts that are partially evaluated at weave-time, using a declarative language to define the bytecode regions to be marked. Because only a subset of bytecodes is converted to the declarative language, it is not possible to define basic block pointcuts as in DiSL.

Prevailing AspectJ weavers lack support for embedding custom static analysis in the weaving process. In [38] compile-time statically executable advice is proposed, which is similar to static context in DiSL. SCoPE [39] is an AspectJ extension that allows analysis-based conditional pointcuts. However, advice code together with the evaluated conditional is always inserted, relying on the just-in-time compiler to remove dead code. DiSL's guards, together with static context, allow weave-time conditional evaluation and prevent the insertion of dead code.

The AspectBench Compiler (*abc*) [40] eases the implementation of AspectJ extensions. As intermediate representation, *abc* uses Jimple to mark bytecode regions. Jimple has no information on where blocks, statements and control structures start and end, thus requiring extensions to support new pointcuts for dynamic analysis. In contrast, DiSL provides an extensible library of markers without requiring extensions of the intermediate representation.

Javassist [17] is a load-time bytecode manipulation library that allows load-time structural reflection and definition of new classes at runtime. The API provides two different levels of abstraction: source-level and bytecode-level. In particular, the source-level abstraction does not require any knowledge of the Java bytecode structure and allows insertion of code fragments given as source text.

Shrike [16] is a bytecode instrumentation library that is part of the T.J. Watson Libraries for Analysis (WALA) [41] and provides interesting features to increase efficiency. For example, parsing is limited to the parts of the class to be modified, bytecode instructions are represented by immutable objects, and many constant instructions can be represented with a single object shared between methods. Moreover, Shrike has a patch-based API that atomically applies all modifications to a given method body and automatically splits up methods larger than the limit imposed by the Java class file format. Dila [42]

is another library of WALA that relies on Shrike for load-time bytecode modifications.

Compared to DiSL, Javassist and Shrike do not follow a pointcut/advice model and do not provide built-in support for basic-block analysis and synthetic local variables.

7. Conclusion

This paper is the first complete presentation of DiSL, a domain-specific language and framework designed specifically for instrumentation-based dynamic program analysis. DiSL occupies a unique position among the existing instrumentation frameworks:

- DiSL instrumentations are concise—rather than relying on low-level bytecode manipulation constructs, DiSL adopts a high-level pointcut/advice model inspired by AOP, which leads to compact and readable code.
- DiSL instrumentations are flexible—the DiSL framework provides an open join-point model that allows instrumenting any bytecode sequence, coupled with techniques that extend the instrumentation coverage to any method with bytecode representation.
- DiSL instrumentations are efficient—static context information is precomputed and embedded in the inserted code as constants; dynamic context information can be accessed efficiently through special methods.

DiSL is built on top of ASM [13], the well-known bytecode manipulation library, and is itself an open-source project.[15]

Various isolated aspects of the DiSL framework were presented in previous work [22, 23, 24]. This paper extends our previous presentation of DiSL—we provide an overview of the DiSL architecture, introduce the DiSL programming model and illustrate the basic concepts with a running example. For advanced developers, we demonstrate the use of two extension points of the DiSL framework—custom static contexts and custom bytecode markers.

[15] The project is hosted by the OW2 consortium at http://disl.ow2.org

These extension points allow the developers to introduce new types of static information that can be used within snippets and guards, and to extend the set of join points recognized by DiSL. Finally, we present several case studies that demonstrate successful deployment of DiSL-based DPA tools.

We believe that the unique combination of the high-level programming model with the flexibility and detailed control of low-level bytecode instrumentation makes DiSL a valuable tool that can reduce the effort needed for developing new dynamic analysis tools and similar software applications running in the JVM. Since the implementation of DiSL is specific to JVM, it is mostly the programming language and software engineering communities targeting the JVM—both in academia and in industry—who can benefit most from DiSL. However, none of the concepts employed in DiSL are JVM specific, which makes it possible, given resources, to implement a similar tool for other managed platforms, such as the Common Language Runtime.

Acknowledgments

The research presented in this paper has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the European Commission (Seventh Framework Programme projects 287746 and 257414), by the Czech Science Foundation (project GACR P202/10/J042), and by the Sino-Swiss Science and Technology Cooperation (SSSTC) Institutional Partnership (project IP04–092010).

References

[1] M. Jovic, A. Adamoli, 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, A. Villazón, Platform-independent profiling in a virtual execution environment, Software: Practice and Experience 39 (2009) 47–79.

[3] M. Eaddy, A. Aho, W. Hu, P. McDonald, J. Burger, Debugging aspect-enabled programs, in: Proceedings of the 6th international conference on Software Composition (SC'07), Springer, 2007, pp. 200–215.

[4] S. Artzi, S. Kim, M. D. Ernst, ReCrash: Making software failures reproducible by preserving object states, in: Proceedings of the 22th European Conference on Object-Oriented Programming (ECOOP'08), volume 5142 of *Lecture Notes in Computer Science*, Springer, 2008, pp. 542–565.

[5] G. Xu, 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, 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/, 2012.

[8] 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.

[9] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni, Model-based performance prediction in software development: a survey, IEEE Transactions on Software Engineering 30 (2004) 295–310.

[10] A. Schauerhuber, W. Schwinger, E. Kapsammer, W. Retschitzegger, M. Wimmer, G. Kappel, A Survey on Aspect-Oriented Modeling Approaches, Technical Report, Vienna University of Technology, Johannes Kepler University Linz, 2007.

[11] R. France, I. Ray, G. Georg, S. Ghosh, Aspect-oriented approach to early design modelling, IEE Proceedings – Software 151 (2004) 173–185.

[12] D. Simmonds, A. Solberg, R. Reddy, R. France, S. Ghosh, An aspect oriented model driven framework, in: Enterprise Computing Conference (EDOC), pp. 119–130.

[13] OW2 Consortium, ASM – A Java bytecode engineering library, Web pages at http://asm.ow2.org/, 2012.

[14] T. A. J. Project, The byte code engineering library (bcel), Web pages at http://jakarta.apache.org/bcel/, 2012.

[15] 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 of the 9th international conference on Compiler Construction (CC'00), volume 1781 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 18–34.

[16] IBM, Shrike Bytecode Instrumentation Library, Web pages at http://wala.sourceforge.net/wiki/index.php/Shrike_technical_overview, 2012.

[17] S. Chiba, Load-time structural reflection in Java, in: Proceedings of the 14th European Conference on Object-Oriented Programming (ECOOP'2000), volume 1850 of *Lecture Notes in Computer Science*, Springer, 2000, pp. 313–336.

[18] 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.

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

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

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

[22] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, 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.

[23] 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: Objects, Models, Components, Patterns, volume 7304 of *Lecture Notes in Computer Science*, pp. 353–368.

[24] L. Marek, Y. Zheng, D. Ansaloni, A. Sarimbekov, W. Binder, P. Tůma, Z. Qi, Java bytecode instrumentation made easy: The DiSL framework for dynamic program analysis, in: Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS'12), volume 7705 of *Lecture Notes in Computer Science*, Springer, 2012, pp. 256–263.

[25] P. Moret, W. Binder, É. Tanter, Polymorphic bytecode instrumentation, in: Proceedings of the 10th international conference on Aspect-Oriented Software Development (AOSD'11), ACM, 2011, pp. 129–140.

[26] W. Binder, D. Ansaloni, A. Villazón, P. Moret, Flexible and efficient profiling with aspect-oriented programming, Concurrency and Computation: Practice and Experience 23 (2011) 1749–1773.

[27] 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.

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

[29] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, S. Z. Guyer, new Scala() instanceof Java: A comparison of the memory behaviour of Java and Scala programs, in: Proceedings of the International Symposium on Memory Management (ISMM'12), ACM, 2012, pp. 97–108.

[30] A. Sarimbekov, A. Sewe, S. Kell, Y. Zheng, W. Binder, L. Bulej, D. Ansaloni, A comprehensive toolchain for workload characterization across JVM languages, in: Proceedings of the 11th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE'13), ACM, 2013, pp. 9–16.

[31] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tůma, Z. Qi, Productive development of dynamic program analysis tools with DiSL, in: Proceedings of the 22nd Australasian Software Engineering Conference (ASWEC'13), IEEE, 2013, pp. 11–19.

[32] 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 of the 21st ACM intenational conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA'06), ACM, 2006, pp. 169–190.

[33] W. Binder, P. Moret, D. Ansaloni, A. Sarimbekov, A. Yokokawa, E. Tanter, Towards a domain-specific aspect language for dynamic program analysis: position paper, in: Proceedings of the 6th workshop on Domain-specific Aspect Languages (DSAL'11), ACM, 2011, pp. 9–11.

[34] C. Flanagan, 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.

[35] M. Achenbach, K. Ostermann, A meta-aspect protocol for developing dynamic analyses, in: Proceedings of the 1st international conference on Runtime Verification (RV'10), volume 6418 of *Lecture Notes in Computer Science*, Springer, 2010, pp. 153–167.

[36] S. Chiba, 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.

[37] K. Klose, K. Ostermann, M. Leuschel, Partial evaluation of pointcuts, in: Practical Aspects of Declarative Languages, volume 4354 of *Lecture Notes in Computer Science*, Springer-Verlag, 2007, pp. 320–334.

[38] K. Lieberherr, D. H. Lorenz, P. Wu, A case for statically executable advice: Checking the law of Demeter with AspectJ, in: Proceedings of the

2nd international conference on Aspect-Oriented Software Development (AOSD'03), ACM, 2003, pp. 40–49.

[39] T. Aotani, H. Masuhara, SCoPE: an AspectJ compiler for supporting user-defined analysis-based pointcuts, in: Proceedings of the 6th international conference on Aspect-oriented Software Development (AOSD'07), ACM, 2007, pp. 161–172.

[40] P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, J. Tibble, abc: An extensible AspectJ compiler, in: Proceedings of the 4th international conference on Aspect-Oriented Software Development (AOSD'05), ACM, 2005, pp. 87–98.

[41] IBM, Watson Libraries for Analysis (WALA), Web pages at http://wala.sourceforge.net/wiki/index.php/Main_Page, 2012.

[42] IBM, Dynamic Load-time Instrumentation Library for Java (Dila), Web pages at http://wala.sourceforge.net/wiki/index.php/GettingStarted:wala.dila, 2012.

Appendix A. Running DiSL

This appendix provides a step-by-step guide to download, compile, and run the DiSL framework. The current release of DiSL is tested with Java 6 and Java 7 on Linux, for which we provide scripts to compile and run the framework. Compatibility with other platforms will be added in future releases. To build DiSL and the examples, Java 6 or Java 7 JDK must be installed on the system, including rudimentary tools such as ant, gcc, make, and python.

The source code of DiSL can be downloaded from the DiSL home page[16], hosted by the OW2 Consortium. In particular, DiSL releases are available at http://forge.ow2.org/projects/disl/files/. After downloading and extracting the latest release of DiSL (i.e., disl-src-2.0.1.tar.bz2 at the time of writing, please consult the README file in the main directory, which provides guidelines for compiling the framework and links to additional documentation.[17]

A very simple example of a DiSL instrumentation can be found in the example/smoke directory. In this example the observed program (i.e., example/smoke/app/src/Main.java) prints a hello-world message, while the instrumentation (i.e., example/smoke/instr/src/DiSLClass.java) inlines the code to print a message at the beginning and at the end of the main method body.

Listing 1 shows the sequence of commands needed to compile the DiSL framework and to run the example. In line 1, we compile the DiSL framework. Line 3 runs the example program.

Listing 1: Compiling the framework and running the included DiSL example.

```
1 [disl]$ ant
2 [disl]$ cd example/app/smoke
3 [disl/example/app/smoke]$ ant run
```

Listing 2 shows the expected output of the instrumented program. Line 2 is the message printed by the observed program, while lines 1 and 3 are the messages printed by the instrumentation.

Listing 2: Output of the included DiSL example.

```
1 Instrumentation: Before method main
```

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

[17] http://disl.projects.ow2.org/xwiki/bin/view/Main/Doc/

```
2 Application: Inside method main
3 Instrumentation: After method main
```

It is possible to use the disl.py script to invoke DiSL with user-defined instrumentations, provided the following rules are adhered to:

- All the instrumentation and the analysis classes must be packed into a single jar file, including any external libraries used by the analysis. Such libraries can be added to the jar file using, for example, the jarjar[18] tool.
- The MANIFEST.MF file in the META-INF directory of the jar file must list all the DiSL classes used for the instrumentation; Listing 3 shows the manifest file of the included example. In this case, the instrumentation consists of a single class (i.e., DiSLClass) that can be found in the default package.

Run disl.py -h for information on how to use the script and to list all available parameters.

Listing 3: Manifest file of the included DiSL example.

```
Manifest-Version: 1.0
DiSL-Classes: DiSLClass
```

An archive of all examples and tools presented in this paper (i.e., disl-examples-1.2.tar.bz2) can be downloaded from the DiSL release page. This archive must be extracted within the main directory of DiSL and includes an additional README file that describes how to run the examples. Listing 4 shows how the runExample.sh script can be used to run the example shown in Figure 1 of this paper.

Listing 4: Running the profiler example presented in Figure 1.

```
[disl/examples]$ ./runExample.sh \
2.1-Method_Execution_Time_Profiler-Figure_1
```

[18] http://code.google.com/p/jarjar/

Part III

Related Work and Conclusion

Chapter 6
Related Work

This chapter provides an overview of the related work. The overview does not refer to every related publication, as they are already covered in the included papers. Instead, it provides a deeper comparison with related dynamic analysis frameworks, since such comparison is normally not possible in the published papers due to space constrains.

DiSL and ShadowVM are essentially a part of one framework, however they were designed to be used separately. We believe that each of the parts deserves a separate comparison. Therefore, the related work is divided into two sections. The first one describes instrumentation (comparison with DiSL) and the second one describes the dynamic analysis evaluation (comparison with ShadowVM). Some mentioned frameworks solve both problems, thus span over both sections. Even though the framework itself may be only a single tool, we compare its instrumentation and analysis evaluation capabilities separately.

6.1 Instrumentation frameworks

This section provides an overview of the instrumentation frameworks and discusses their difference with DiSL. Initially, we will cover well-known machine instruction level frameworks, following with a discussion of the instrumentation frameworks targeting the Java environment.

6.1.1 Instrumentation in machine code

Although the frameworks operating on the machine instruction level are not primarily designed for instrumentation of managed languages, they provide high flexibility, high control over the instrumentation process and good isolation from the managed environment, hence they may serve as a foundation for a more complex observation platform.

Pin

Pin [34, 64] is a C/C++ framework providing a rich API for application instrumentation. Pin operates at machine instruction level, however the API provides an abstraction over the machine architecture so the analysis written using Pin can remain architecture-independent as much as possible. Pin instruments an application during execution, similarly to the Just-In-Time (JIT) compiler from the managed runtimes. It governs the execution of all new blocks of code and gives the developer an opportunity to insert event callbacks before the code is executed.

DynamicRIO

DynamicRIO [9, 46] is an instrumentation framework similar to Pin. Compared to Pin, DynamicRIO is better suited for generic code transformations for the price of having more complicated API.

Valgrind

Another framework for instrumentation of native programs is Valgrind [69, 70, 40]. Compared to the previous two frameworks, Valgrind provides instrumentation capabilities for heavy weight analysis that are not supported under Pin or DynamicRIO. It is suitable for analysis intercepting many events and operating on large amount of analysis data. The distinguishing feature is its disassemble-and-resynthesise (D&R) technique. During D&R, Valgrind reads the machine code of the target application, translates it to an intermediate representation, applies the requested instrumentation and translates the application back to machine code. D&R improves an ability to control the effects of instrumentation code on the observed application code, thus providing a safer environment for the analysis.

SystemTap

SystemTap[1] [38, 74] is an instrumentation framework for GNU/Linux systems. SystemTap collects events from user-space applications and is also capable of monitoring a whole operating system by inserting instrumentation code directly into the Linux kernel.

SystemTap defines its own language for describing the instrumentation and handling the events. The event handling routines are not meant to perform the whole analysis, especially in cases where it would require a lot of computation. Instead, the event routine should only store the required information and quickly return. The processing of collected data is managed by other tools.

As event handlers written using SystemTap often observe sensitive data (in relation to stability of the system), the code of the event handlers is analyzed to meet certain safety properties. During script compilation, SystemTap controls properties like the length of execution, memory allocations or unsafe operations like manipulation with the observed structures.

SystemTap provides two different options how to instrument user-space applications. One way is to compile in the instrumentation probes, i.e., include the probes directly into the source code of the application. The other option is to use the debug information created by compiler to determine the place where the probes should be inserted.

Comparison with DiSL

As all of the mentioned frameworks were not primarily designed to observe managed runtimes, their applicability remains limited. They have no notion of classes, instances, synchronization primitives or a garbage collector. Machine instruction code produced by a JIT compiler is optimized to perform well on the designated

[1] SystemTap is a similar solution to Dtrace, a framework for instrumentation under the Solaris operating system.

CPU. Therefore, detecting high-level language patterns like field assignments or object creations may be complicated.

Instrumentation of managed applications on the machine instruction level assumes that the managed runtime is using a JIT compiler to translate code to machine instructions. A virtual machine may, however use an interpreter to execute the bytecode. An interpreted application does not have the form of the coherent machine instruction code and cannot be easily instrumented.

Compared to them, DiSL works on Java bytecode level, which is an intermediate representation between the Java source code and the machine code. Thankfully, modern Java compilers perform almost no optimization when compiling the Java source code to the bytecode. Without optimizations and using the specific instruction set that is much closer to object oriented language, detection of high-level language patterns in Java bytecode is relatively easy. To raise the level of abstraction, DiSL provides a predefined library of markers to instrument common execution patterns.

In summary, instrumentation on the machine instruction level requires direct support of the target managed runtime. Without the detection of high-level language patterns, the machine code instrumentation frameworks have no possibility to introduce high-level constructs to simplify the instrumentation of managed languages.

6.1.2 Instrumentation in Java

In this section, we summarize tools for instrumentation under Java. The first subsection describes generic bytecode transformation libraries. Following subsections concentrate on tools providing more convenient instrumentation interfaces.

6.1.3 Bytecode manipulation libraries

Bytecode transformation libraries like ASM, Soot or BCEL provide API for writing very efficient and fully customizable instrumentation. The API works directly with a stream of bytecodes and only a small abstraction layer is provided.

ASM

ASM [2, 47] defines two distinct APIs for bytecode manipulation. The Core API allows to inspect and modify classes using a visitor design pattern and the Tree API represents methods, fields and bytecode in a form of an object tree (similarly to the Java reflection API).The Core API is very fast and efficient especially in cases where the transformation is done during a single pass through the transformed class. The information about the class and method under transformation is provided through visitor method arguments. Visitor methods are invoked for each method, annotation and even bytecode instruction.

The Tree API is often used for more complicated transformation or bytecode analysis. The information about classes, methods, fields and instructions are stored in separate objects. These objects are grouped in lists and form a tree structure. A root of the tree is a class with references to fields and methods. The method contains list of bytecode instructions.

BCEL

BCEL [4, 53] is a bytecode transformation library similar to ASM. Compared to ASM, BCEL is slower and cumbersome while working with local variables but may require writing less code.

Soot

Soot [37, 77] is a framework for static analysis and optimization of Java bytecode. Soot provides four intermediate representations of Java bytecode. The first representation, called Baf, provides only a thin abstraction over the Java bytecode. Same as a Java bytecode, it uses stack-based bytecode representation, but its instructions are fully typed to simplify transformation and analysis. Another representation is a 3-address code representation called Jimple. Jimple is a preferred form for code analysis and optimization. The third representation is called Shimple. Shimple is a variation of Jimple using static single assignment form. The last representation, called Grimp, allows unification of several bytecode instructions into a tree and creating a structure similar to Java source code suitable for human-readable outputs.

Comparison with DiSL

The bytecode engineering libraries provide a rich API for class transformations but such an API is unnecessarily complex for dynamic analysis instrumentation. As dynamic analysis does not require to modify the behaviour of the observed application, the instrumentation API can be simpler and provide a higher level of abstraction over the instrumented bytecode. DiSL provides a predefined library of bytecode pattern markers to easily instrument for example object allocations, basic blocks or exception handlers. To provide a similar flexibility to the low-level transformation libraries, a DiSL developer may take an advantage of the underlying ASM library and design a custom marker to instrument an arbitrary execution pattern.

The bytecode engineering libraries specify the instrumentation as a series of bytecode instructions inserted in between the bytecodes of the original method. A developer using these libraries has to be either very skilled in bytecode programming or to use additional tools to convert Java code to bytecode. In contrast, DiSL allows to specify the instrumentation as Java code directly, abstracting the developer from the bytecode internals and allowing to concentrate mainly on the analysis logic.

6.1.4 Java instrumentation frameworks

The following subsections overviews instrumentation frameworks in Java. Compared to the bytecode transformation libraries, the instrumentation frameworks do not allow arbitrary class transformation, but provide high-level abstractions to simplify the specification of the instrumentation.

AspectJ

One of the most commonly used tools for dynamic analysis instrumentation in Java is AspectJ [3, 60, 61]. AspectJ is a language based on the aspect-oriented programming paradigm, built on top of Java by extending the language with additional keywords. Thanks to such high-level language integration, AspectJ provides a convenient way for writing instrumentation. Compared to low-level bytecode manipulation libraries, the programmer does not have to deal with a stream of instructions. Instead, he writes the instrumentation code directly in Java and uses AspectJ's join-point model to navigate its insertion. The join-point model in AspectJ is designed to intercept various application behaviour like field operations or execution of method bodies and exception handlers.

The access to the static and dynamic context information is provided through an objects created at runtime. For each join-point, AspectJ creates a corresponding object to expose context information like method arguments, method call target or accessed field value.

AspectJ supports several types of weaving. Convenient method for applying instrumentation is to compile the application sources using the AspectJ compiler. Another option is to use the Java instrumentation agent to perform weaving at runtime.

AspectBench Compiler

The AspectBench Compiler (abc) [1, 45] is an AspectJ compiler designed to be easily extensible. abc uses the Polyglot [35, 72] compiler as a front-end to easily extend the Java language syntax, and the Soot framework (described earlier) as the instrumentation backend.

Javassist

Javassist [17, 49] is a bytecode transformation library. Beside the low-level transformation API, Javassist allows to reflect classes similarly to Java reflection API and add fields or methods by specifying their Java code.

Comparison with DiSL

Compared to the bytecode manipulation libraries, AspectJ offers only a predefined set of join-points. It does not provide any convenient extension mechanism to instrument an arbitrary execution pattern, i.e., does not provide a mechanism to introduce new types of events. The AspectBench Compiler (abc) was designed to overcome this limitation. Even though abc allows extensibility of the AspectJ language, it is still necessary to define rules for semantic checks and implement several new core classes when creating a custom join-point or a new advice. The amount of work depends on the type of the extension, however the AspectJ language was never designed to be easily extensible. In contrast, DiSL was designed with extensibility in mind and standard extensions require implementation of only one new class.

The DiSL API allows to access the context information through pre-defined or custom made objects. During weave time, DiSL pre-computes requested values and translates all calls to context objects into series of bytecodes direct-

ly fetching the required information as constant values. AspectJ also provides object-oriented API to access context information. The objects are however not translated into simple bytecode instructions, instead the whole object tree is accessible at runtime. When the instrumentation requests the context information, AspectJ has to allocate and fill the objects with data. Even though the objects may be reused, such solution still requires additional allocations and creates runtime overhead.

Overall, AspectJ does not provide any control over the internal allocations nor helper code executed during runtime. The core principle in DiSL is to give the instrumentation developer full control over the instrumentation process. All constructs in DiSL are evaluated at weave-time and replaced by constants or bytecodes. DiSL by itself never allocates new objects and introduces only small runtime overhead[2].

None of the instrumentation methods in AspectJ is able to provide full bytecode coverage, especially instrumentation of the Java Class Library. In comparison to AspectJ, DiSL uses the Java native agent, allowing to instrument all loaded bytecode classes including all classes from the Java Class Library.

Similar to AspectJ, Javassist is limited to only a predefined set of join-points while using the high-level instrumentation API. If more fine grained instrumentation is required, a byte-code level API (similar to the low-level bytecode manipulation libraries API) may be used. DiSL does not allow to modify the observed code as it was primarily designed for program observation, but it outperforms Javassist in flexibility of the high-level instrumentation API.

6.1.5 Frameworks with predefined probes

Following frameworks does not allow to insert arbitrary instrumentation but use a set of predefined probes to observe the application.

Sofya

Sofya [36, 63, 62] is an instrumentation and analysis evaluation framework build on a publish-subscribe messaging pattern. Instrumentation process in Sofya is guided using description language called EDL. EDL supports only a selection of basic patterns and filtering based on class or method names.

Sofya does not allow to insert arbitrary code into the observed application, instead it provides a set of predefined probes. It is safer to not allow the developer to modify the monitoring probes, however such restriction greatly reduces the framework flexibility. If modifications are necessary, the developer may access an underlying BCEL bytecode manipulation library.

Sofya provides access to static and dynamic context information, but specification of what data can be requested by the evaluation is very limited. The specification is often reduced to a decision whether to transfer the whole context information or not.

[2]A top of the code specified by a programmer, the DiSL framework inserts code to support the dynamic bypass mechanism and check for unexpected instrumentation code exceptions. As none of these mechanisms is mandatory for DiSL to operate, the insertion of such code can be disabled by the programmer.

Chord

Chord [7, 68] is a platform for static and dynamic program analysis. It provides load-time instrumentation and allows to instrument classes from the Java Class Library. Chord uses a Java instrumentation agent, therefore it may potentially miss some classes and leave them un-instrumented. The bytecode manipulation in Chord is performed by the Javasist library. The analysis evaluation can be performed in the context or out of the context of the observed application (to be discussed in 6.2.2).

Same as Sofya, Chord uses a predefined set of probes to trigger various events including method entry and exit, synchronization operations and field accesses. For each of the probes, Chord provides a predefined context information.

Comparison with DiSL

Both frameworks are significantly limited when adding new events or extending context information in the existing ones. As both frameworks use an underlying bytecode manipulation library, extensions are possible. In both cases, the probes are implemented in several layers of the framework, hence an extension would require non-trivial modifications to several core framework classes. Furthermore, the resulting analysis would be bound to the modified version of the framework.

In DiSL, extensions use a predefined API. Each extension is bundled together with the instrumentation and is loaded automatically during instrumentation.

6.2 Frameworks for dynamic analysis evaluation in Java

Frameworks for analysis evaluation are usually built on top of an instrumentation framework and provide a supporting infrastructure to evaluate custom dynamic analysis in a safe environment. In this work, we distinguish two types of analysis frameworks. The first, called in-process analysis frameworks, perform analysis in the context of the observed application and may potentially influence its execution. The second type of frameworks offloads an analysis out of the context of an observed application to minimize possible perturbation; these we call out-of-process analysis frameworks.

6.2.1 In-process analysis frameworks

A great advantage of in-process analysis over out-of-process analysis is the ability to query any application context information with small overhead. Well-crafted in-process analysis may outperform out-of-process analysis by orders of magnitude depending on the scenario. As the analysis may influence the observed application, it has to be carefully designed and thoroughly tested.

AspectJ

AspectJ [3, 60, 61] is an instrumentation framework (described in 6.1.4), but is also provides basic support for in-process analysis. In AspectJ, aspect classes

contain definitions of the intercepted events together with the code being invoked when such an event is triggered. In addition, the classes may also include helper methods and fields to separate the event logic from the main analysis logic and to store the analysis data structures between the event invocations.

BTrace

BTrace [5] is a dynamic tracing tool for Java. It uses Java annotations to guide the instrumentation process and Java code to express the analysis logic. As BTrace is designed to be safe, it restricts the analysis code to only harmless operations which cannot influence the observed application. The analysis cannot, among other restrictions, allocate new objects, throw or catch exceptions, assign values to fields, use class literals or contain loops. The analysis is allowed to invoke methods only from the BTrace utility package and the helper analysis methods. All other method calls are forbidden. BTrace allows to intercept only a predefined set of events like method executions, field accesses or object allocations, with no possibility of extensions.

RoadRunner

RoadRunner [39, 56] is a dynamic analysis framework for concurrent Java programs. RoadRunner concentrates on rapid prototyping where dynamic analysis is described as a filter over a stream of predefined events. The framework allows to compose several simple dynamic analyses into chains to create a more complex analysis. RoadRunner intercepts events like lock acquisition, lock release, thread start, thread interruption, thread join or thread sleep.

Comparison with ShadowVM

The mentioned frameworks prevent perturbation in the observed application using two distinct approaches. AspectJ and RoadRunner exclude classes from instrumentation. Even though it would be possible to exclude only classes shared between the analysis and the observed application, the frameworks exclude large part of the Java Class Library.

BTrace prevents the perturbation by restricting the evaluation to only safe operations. Nevertheless, it is not always harmful to use a loop or call a pure method and restrictions imposed by BTrace limits versatility of the analysis framework.

The ShadowVM prevents the perturbation by offloading the analysis out of the context of the observed application and does not enforce restrictions that would limit the versatility or the coverage of the framework.

6.2.2 Out-of-process analysis frameworks

The out-of-process analysis frameworks offload the analysis out of the managed Java space. The two most typical approaches are either a use of Java level marshaling and transportation mechanisms like JDI [19], Java beans [23] or simple I/O API, or a use of JNI [24] with processing and transportation of events performed in native space.

Chord

The events in Chord [7, 68] (the instrumentation part described in 6.1.5) are serialized into a byte buffer. The events can be deserialized and evaluated in the context of the observed application, redirected to a file on a hard drive or streamed to another VM.

The serialization process allows to preserve only a unique identification for an object. This is not a problem during the in-process analysis, where the analysis can access a translation map and get a reference to the real object. If the analysis is performed in another VM, Chord does not provide any mechanism to query additional object data.

Sofya

Sofya [36, 63, 62] is an instrumentation (described in 6.1.5) and analysis framework. Although we split the description of the instrumentation and the analysis into two parts, they are tightly coupled. The whole Sofya's infrastructure is composed of six layers. The bottom two layers are responsible for bytecode analysis and instrumentation. The third layer is responsible for communication between the observed application and the evaluation part. The communication is mediated by JDI [19] or a custom transfer protocol. The fourth layer dispatches the gathered events to the analysis and the fifth layer provides various components for event filtering. The fifth layer also dispatches events to multiple consumers, allowing to process a single event by multiple analyses. The sixth layer contains the main analysis logic.

Using JDI, Sofya can combine the instrumentation events together with the Java debugging events. The main reason for such a solution is the ability to precisely capture a correct timing of the synchronization events, which poses an issue if captured only by the instrumentation.

Caffeine

Caffeine [6, 58] is a tool for dynamic analysis. Caffeine intercepts Java debugging events and supplies them to the JIProlog [22] engine for analysis evaluation using the JDI protocol. Caffeine restricts the type of available events to only basic ones like method invocation or field access. The analysis is evaluated in the separate JVM and the JDI provides Caffeine with good isolation between the analyzed application and the performed analysis.

*J

*J [13, 55] is a dynamic analysis framework using a native JVMPI [28] agent for event triggering with a separate Java analyzer for the event processing. The analyzer runs during the measurement and parses the stream of events from a socket. The analyzer contains a predefined set of operations such as various transformations, metrics, triggers and printers to help process the stream of events. Definition of new operation and its combinations with predefined ones is also possible.

*J does not use instrumentation and relays entirely on the (already deprecated) JVMPI interface. Context information provided for each event is fixed

with no possibility of extensions. Even though JVMPI allows to intercept instructions processed by an interpreter, capturing more sophisticated bytecode patterns would make the analysis incredibly slow.

Javana

Javana [16, 65] is a framework written in C, targeted on building analysis tools for Java. Javana uses the DIOTA [8, 66] binary instrumentation framework to capture various application behaviour on a machine instruction level. To be able to analyse the Java events properly on the machine instruction level, Javana requires several hooks to the selected JVM operations like object allocation, reallocaion and reclamation. The hooks have the form of JNI method calls invoked every time the desired JVM operation occurs. As the standard JVMs do not support such functionality, Javana uses a modified Jikes RVM [21, 44].

Javana provides only basic support for event filtering, with no possibility to restrict monitoring to only a part of the application. Any extension of the available events and their context information would mean to not only extend the framework but also the accompanying JVM. As the provided events mainly observe the lifecycle of Java objects, and the low-level application behaviour, analyses written in Javana are mainly targeted on memory access profiling.

Comparison with ShadowVM

Sofya and Chord pre-process and dispatch events directly from the context of the observed application. As the frameworks and the observed application share classes necessary for buffering, network communication or file management, the event dispatching may potentially influence the application behaviour, especially if the shared classes are also observed.

Caffeine, *J and Javana prevent the perturbation by using Java observation interfaces. Such a solution provide good isolation, but limits the dynamic analysis to only predefined events.

ShadowVM prevents the perturbation by offloading the analysis out of the context of the observed application through JNI calls. Compared to Javana, ShadowVM is using the JNI native calls to offload arbitrary events. Javana is using JNI to hook only memory-related JVM events.

Caffeine, Javana and Sofya[3] stop the application while evaluating an event. As all the frameworks are offloading events out of the application context, the communication adds additional delay to the evaluation. Even a very small analysis capturing only small amount of events may substantially slow down the observed application.

*J and Chord improve the performance by buffering the events and evaluating them asynchronously. However, each Java object is represented only as a unique object identification without any additional information.

Similar to *J and Chord, ShadowVM evaluates the events asynchronously. In addition, ShadowVM allows to access whole class hierarchy of an object and provides all class information associated with a class. To furthermore improve

[3]When the communication is mediated through JDI.

the performance, ShadowVM introduces several event ordering models to reduce lock contention while buffering the events.

6.3 Instrumentation and evaluation frameworks without sources

To be able to provide a satisfying comparison with the related work, having access to the source code of the described framework is a mandatory requirement as it is often the only source of the accurate documentation. In this subsection, we provide a list of tools for which we were unable to find the references to their sources.

- Aftersight [52] decouples the observed application from the evaluation by running the observed application in a dedicated virtual machine, logging all non-deterministic inputs and replaying them on an isolated analysis platform.
- Josh [50] is an open compiler solution for making aspect-oriented languages easily extensible.
- MAP [43] is a meta-aspect protocol for rapid prototyping of dynamic analysis.
- ConTest [71] Listeners is an instrumentation and runtime engine for multi-threaded systems.
- DejaVu [51] allows a deterministic replay of multi-threaded Java programs.

Chapter 7
Conclusion

Byte-code level program instrumentation is a difficult and error-prone task. Current instrumentation tools try to provide a higher level of abstraction to simplify the specification of the instrumented locations and the code being inserted. However, the tools still fail to provide a simple-but-versatile instrumentation API. On one hand, there are bytecode transformation libraries able to perform arbitrary instrumentation but not providing a high-level API which even a non-skilled[1] developer could use. On the other hand, the tools providing the high-level API to easily specify and guide instrumentation lack the flexibility and do not allow experienced users to control the instrumentation process.

In this thesis we presented DiSL, an instrumentation framework that is trying to bridge both worlds. DiSL provides high-level abstractions that a non-skilled programmer can use to assemble the desired instrumentation. An experienced developer can design custom *Markers*, *Guards* and *StaticContexts* to select an arbitrary location for instrumentation and access custom context information. Given such modularity, DiSL has the flexibility of Java bytecode transformation libraries but also provides the high-level API where only a very limited knowledge of bytecode is required [76].

In DiSL, the developer has a full control over the instrumentation process. No hidden allocation or additional code is inserted without the developer's knowledge. The efficiency and the transparency during the weaving allows to better predict possible perturbation and more easily manage the incurred overhead.

Processing of analysis events triggered by the instrumentation can have undesired effects on the observed application. Observation problems such as missing events or shared state corruption are hard to detect even for a skilled analysis developer. ShadowVM addresses such problems by offloading the analysis out of the context of the observed application. By running the evaluation in a separate VM, the analysis is free to allocate memory, use all classes from the Java Class Library or introduce arbitrary locking. The developer is thus freed from thinking about the possible interference and is able to fully concentrate on the analysis logic.

On the ShadowVM, the analysis accesses the object and class information using the Shadow API. For each transferred object, the analysis may explore the class hierarchy and retrieve arbitrary class information using the reflection-like API. The reclamation of an object and the termination of the observed application is announced through the Shadow API life-cycle events.

ShadowVM introduces several techniques to speed up the offloading process. ShadowVM separates the observation and the evaluation so they can run in-parallel. The introduced event ordering models help to decrease lock contention while buffering the events and increase parallelism while evaluating the events.

ShadowVM uses standard JNI and JVMTI native interfaces for event offloading, hence it is deployable in production JVMs.

[1] A developer not familiar with the Java bytecode.

DiSL is used in several research projects at the author's home department, University of Lugano and University of Darmstadt, with other universities showing interest in using DiSL. After a thorough review, DiSL was accepted to the SPEC RG tool repository[2]. DiSL (together with ShadowVM[3]) is available as open-source under the Apache license in the OW2 software repository at `http://disl.ow2.org/`.

7.1 Future work

The following section covers various ideas and problems connected to dynamic analysis still not solved by DiSL and ShadowVM.

DiSL

The instrumentation using DiSL is still not perfect. The analysis developer often knows which type of the application actions he wants to instrument. However, JVM may perform some of the actions in the native space, hence they cannot be easily instrumented using Java bytecode instrumentation.

An example of such instrumentation is object allocation. In Java bytecode, objects are allocated using the **new**[4] instruction. Unfortunately, JVM may allocate new objects in the native code, for example while loading new classes.

The analysis developer may decide to instrument the JNI calls responsible for object allocation, but this is not a perfect solution. Not all objects are necessary allocated using a JNI call and such a solution would be probably heavily dependent on the particular JVM implementation.

The JVM provides an interface informing a developer about new object allocations in the native code. Therefore, DiSL could combine the instrumentation with callbacks. We believe this is not a great solution either, as the callbacks do not provide such flexibility and context information as the instrumentation. The adaptation of DiSL for supporting callbacks would also require to bend the language to cooperate with specific events produced by various environments.

A solution to the problem may be in systems that currently serve only as research platforms, namely Maxine VM [30, 78] and JikesRVM [21, 44]. The VMs are fully implemented in Java and use only a very limited amount of native code for bootstrap process, thus they do not hide object allocations or field assignments in native code. Use of bytecode instrumentation to observe such a system is therefore more thorough [41] compared to production JVMs, which are largely implemented in native code. Fortunately, we may observe a trend [11, 54] to move from native code to Java code even in production JVMs and hopefully, we will see more observable VMs in the future.

Another instrumentation problem not solved by DiSL is the timing of synchronization events. This is particularly problematic when observing access to volatile variables. As the access to a volatile variable is based on atomic reads

[2] `http://research.spec.org/tools/`

[3] DiSL and ShadowVM are currently distributed in one software bundle.

[4] The new instruction has various alternatives to allocate arrays.

and writes, there is no simple instrumentation[5] that could observe the order of the accesses to the variable without introduction of additional locking.

A possible solution is to use the Java Debugging Interface and intercept synchronization events using data breakpoints. Another approach we are investigating is an encapsulation of the volatile variable that would allow to monitor the synchronization order.

DiSL language would benefit from many small improvements. The instrumentation written in DiSL could be validated using advanced static checking for properties like modification of the state of the observed application. The instrumentation scoping language and the Guard construct could be replaced with single mechanism. The synthetic local variables and thread local variables could support better initialization. DiSL could provide custom dynamic contexts that would provide dynamic information based on the used Marker.

Various JVM enhancements would improve applicability of DiSL. For example, the JVM should be more tolerant (stable) while instrumenting core classes of the Java Class Library. The limits on the size of one method and the mandatory Stack Map Frames [15] complicate the instrumentation process.

A feature currently missing in DiSL is instrumentation of already running applications. DiSL was designed to support on-the-fly instrumentation, however the implementation of some essential parts is still missing and a few technical issues need to be resolved.

ShadowVM

A problem not entirely solved by ShadowVM is instrumentation. ShadowVM provides a safe environment for evaluation of the analysis, but the event and its contextual data still need to be gathered in the context of the observed application. The instrumentation is still potentially vulnerable to all the problems connected to in-process analysis evaluation. The advantage compared to the traditional frameworks is a substantial reduction of the potentially harmful operations. The only responsibility of the instrumentation is to obtain the data required for the analysis, which normally means accessing variables on the (Java bytecode) stack or request the field values. All other work, including event buffering and dispatching, is performed in the native code.

One solution is to move the context gathering to the native code. Such a change would demand creating a framework similar to DiSL in the native space and thoroughly explore whether the JVMTI and JNI interfaces do not interfere with the observed application and provide enough information so the solution has the same strength as in the Java code.

Another solution would require support from the JVM. The Multitenant JVM [31] allows to run multiple applications inside one JVM host by separating shared resources like heap, file and socket I/O. We believe this could be a step in the right direction also for the analysis evaluation.

Still, the features provided by Multitenant JVM are for now insufficient for the dynamic analysis. The analysis evaluation would be run in the Multitenant

[5]By simple instrumentation we mean instrumentation inserted before or after the variable access.

JVM as a separate process. First, the Multitenant JVM would need to support separation between the (un)instrumented Java Class Libraries. Also, the analysis evaluation would need to access and reflect any resource in the application but only under a restricted mode where it cannot perform field writes or invoke methods (that would change the state of the observed application).

In addition to the better isolation, the runtime could support copy-on-write snapshots of the analysed application. Using the snapshot, the analysis would obtain access to the state in which the event was triggered for the whole time of the event evaluation. As the snapshot mechanism would be probably costly, it would be used only in cases where the analysis requires to traverse a large part of the heap during each event. Currently, such analyses reconstruct the whole heap by monitoring all modifications to the heap and traverse it offline, which is incredibly costly. Alternatively, the analysis may also traverse the heap online when an event occurs, but the heap may be modified by other threads. Compared to those techniques, the snapshot mechanism may provide better performance or heap immutability while traversing a large part of the heap during evaluation.

References

[1] abc: The AspectBench Compiler for AspectJ. http://www.sable.mcgill.ca/abc/.

[2] ASM. http://asm.ow2.org/.

[3] AspectJ - crosscutting objects for better modularity. http://eclipse.org/aspectj/.

[4] BCEL - The Byte Code Engineering Library. http://commons.apache.org/proper/commons-bcel/.

[5] BTrace. https://kenai.com/projects/btrace.

[6] Caffeine. http://www.yann-gael.gueheneuc.net/Work/Research/Caffeine/Download/.

[7] Chord: A Program Analysis Platform for Java. http://pag.gatech.edu/chord.

[8] DIOTA - Dynamic Instrumentation, Optimisation and Transformation of Applications. http://www.elis.ugent.be/diota/.

[9] DynamoRIO - Dynamic Instrumentation Tool Platform. http://www.dynamorio.org/.

[10] GNU gprof. http://sourceware.org/binutils/docs/gprof/.

[11] Graal Project. http://openjdk.java.net/projects/graal/.

[12] Intel VTune Amplifier XE 2013. http://software.intel.com/en-us/intel-vtune-amplifier-xe.

[13] *J: A Tool for Dynamic Analysis of Java Programs. http://www.sable.mcgill.ca/starj/.

[14] Java instrumentation interface. http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html.

[15] Java Stack Map Frame. http://docs.oracle.com/javase/specs/jvms/se7/html/jvms-4.html#jvms-4.10.1.4.

[16] Javana. http://www.elis.ugent.be/javana/.

[17] Javassist - Java Programming Assistant. http://www.csg.ci.i.u-tokyo.ac.jp/~chiba/javassist/.

[18] JDB: Java Debugger. http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html.

[19] JDI - Java Debug Interface. http://docs.oracle.com/javase/7/docs/jdk/api/jpda/jdi/index.html.

[20] JDWP - Java Debug Wire Protocol. http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html.

[21] Jikes RVM (Research Virtual Machine). http://jikesrvm.org/.

[22] JIProlog - Java Internet Prolog. http://www.jiprolog.com/.

[23] JMX - Java Management Extensions. http://docs.oracle.com/javase/7/docs/technotes/guides/management/.

[24] JNI - Java Native Interface. http://docs.oracle.com/javase/1.5.0/docs/guide/jni/.

[25] JPDA - Java Platform Debugger Architecture. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/jpda.html.

[26] JPDA - Structure Overview. http://docs.oracle.com/javase/7/docs/technotes/guides/jpda/architecture.html.

[27] JVMDI - Java Virtual Machine Debug Interface Reference. http://www.oracle.com/technetwork/java/javase/jvmdi-spec-135507.html.

[28] JVMPI - Java Virtual Machine Profiler Interface. http://docs.oracle.com/javase/1.5.0/docs/guide/jvmpi/jvmpi.html.

[29] JVMTI - JVM Tool Interface. http://docs.oracle.com/javase/1.5.0/docs/guide/jvmti/jvmti.html.

[30] Maxine VM. https://wikis.oracle.com/display/MaxineVM/Home.

[31] Multitenant JVM. http://www.ibm.com/developerworks/java/library/j-multitenant-java/index.html?ca=drs-.

[32] OProfile. http://oprofile.sourceforge.net/.

[33] OSGi - The Dynamic Module System for Java. http://www.osgi.org.

[34] Pin - A Dynamic Binary Instrumentation Tool. http://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[35] Polyglot - A compiler front end framework for building Java language extensions. http://www.cs.cornell.edu/Projects/polyglot/.

[36] Sofya: A Java Bytecode Analysis Tool. http://sofya.unl.edu/.

[37] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/.

[38] SystemTap. http://sourceware.org/systemtap/.

[39] The RoadRunner Dynamic Analysis. http://dept.cs.williams.edu/~freund/rr/.

[40] Valgrind. http://valgrind.org/.

[41] Virtual Machine Level Analysis. `https://wikis.oracle.com/display/MaxineVM/Virtual+Machine+Level+Analysis`.

[42] VisualVM: All-In-One Java Troubleshooting Tool. `http://visualvm.java.net/`.

[43] M. Achenbach and K. Ostermann. A meta-aspect protocol for developing dynamic analyses. In *Runtime Verification*, pages 153–167. Springer, 2010.

[44] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[45] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An extensible AspectJ compiler. In *Transactions on Aspect-Oriented Software Development I*, pages 293–334. Springer, 2006.

[46] D. L. Bruening. *Efficient, transparent, and comprehensive runtime code manipulation*. PhD thesis, Massachusetts Institute of Technology, 2004.

[47] E. Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[48] B. Cantrill, M. W. Shapiro, A. H. Leventhal, et al. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[49] S. Chiba. Load-Time Structural Reflection in Java. In E. Bertino, editor, *ECOOP 2000 — Object-Oriented Programming*, volume 1850 of *Lecture Notes in Computer Science*, pages 313–336. Springer Berlin Heidelberg, 2000.

[50] 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, pages 102–111, New York, NY, USA, 2004. ACM.

[51] J.-D. Choi, B. Alpern, T. Ngo, M. Sridharan, and J. Vlissides. A perturbation-free replay platform for cross-optimized multithreaded applications. In *Parallel and Distributed Processing Symposium., Proceedings 15th International*, pages 10–pp. IEEE, 2001.

[52] J. Chow, T. Garfinkel, and P. M. Chen. Decoupling dynamic program analysis from execution in virtual environments. In *USENIX 2008 Annual Technical Conference*, pages 1–14, 2008.

[53] M. Dahm. Byte code engineering. In *JIT'99*, pages 267–277. Springer, 1999.

[54] G. Duboscq, L. Stadler, T. Würthinger, D. Simon, C. Wimmer, and H. Mössenböck. Graal ir: An extensible declarative intermediate representation. In *Proceedings of the Asia-Pacific Programming Languages and Compilers Workshop*, 2013.

[55] B. Dufour, L. Hendren, and C. Verbrugge. *J: A Tool for Dynamic Analysis of Java Programs. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 306–307, New York, NY, USA, 2003. ACM.

[56] 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, pages 1–8, New York, NY, USA, 2010. ACM.

[57] S. L. Graham, P. B. Kessler, and M. K. Mckusick. Gprof: A Call Graph Execution Profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '82, pages 120–126, New York, NY, USA, 1982. ACM.

[58] Y.-G. Guéhéneuc, R. Douence, and N. Jussien. No Java without caffeine: A tool for dynamic analysis of Java programs. In *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference on*, pages 117–126. IEEE, 2002.

[59] S. Kell, D. Ansaloni, W. Binder, and L. Marek. The JVM is Not Observable Enough (and What to Do About It). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 33–38, New York, NY, USA, 2012. ACM.

[60] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming*, pages 327–354. Springer, 2001.

[61] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'97 — Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer Berlin Heidelberg, 1997.

[62] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analyses for Java software. *CSE Technical reports*, page 20, 2006.

[63] A. Kinneer, M. B. Dwyer, and G. Rothermel. Sofya: Supporting Rapid Development of Dynamic Program Analyses for Java. In *Companion to the Proceedings of the 29th International Conference on Software Engineering*, ICSE COMPANION '07, pages 51–52, Washington, DC, USA, 2007. IEEE Computer Society.

[64] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[65] J. Maebe, D. Buytaert, L. Eeckhout, and K. De Bosschere. Javana: A System for Building Customized Java Program Analysis Tools. *SIGPLAN Not.*, 41(10):153–168, Oct. 2006.

[66] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Compendium of Workshops and Tutorials held in conjunction with PACT'02*, 2002.

[67] P. Moret, W. Binder, and E. Tanter. Polymorphic Bytecode Instrumentation. In *Proceedings of the Tenth International Conference on Aspect-oriented Software Development*, AOSD '11, pages 129–140, New York, NY, USA, 2011. ACM.

[68] M. Naik, A. Aiken, and J. Whaley. Effective Static Race Detection for Java. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM.

[69] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[70] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

[71] Y. Nir-Buchbinder and S. Ur. ConTest Listeners: A Concurrency-oriented Infrastructure for Java Test and Heal Tools. In *Fourth International Workshop on Software Quality Assurance: In Conjunction with the 6th ESEC/FSE Joint Meeting*, SOQUA '07, pages 9–16, New York, NY, USA, 2007. ACM.

[72] N. Nystrom, M. R. Clarkson, and A. C. Myers. Polyglot: An extensible compiler framework for java. In *Compiler Construction*, pages 138–152. Springer, 2003.

[73] K. O'Hair. The JVMPI Transition to JVMTI. `http://www.oracle.com/technetwork/articles/javase/jvmpitransition-138768.html`, Jul 2004.

[74] V. Prasad, W. Cohen, F. C. Eigler, M. Hunt, J. Keniston, and B. Chen. Locating System Problems Using Dynamic Instrumentation. In *Proceedings 2005 Ottawa Linux Symposium (OLS)*, Jul 2005.

[75] J. Reinders. *VTune performance analyzer essentials*. Intel Press, 2005.

[76] A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tuma, and Z. Qi. Productive Development of Dynamic Program Analysis Tools with DiSL. In *Software Engineering Conference (ASWEC), 2013 22nd Australian*, pages 11–19, June 2013.

[77] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java Bytecode Optimization Framework. In *Proceedings of the 1999*

Conference of the Centre for Advanced Studies on Collaborative Research, CASCON '99, pages 13–. IBM Press, 1999.

[78] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Trans. Archit. Code Optim.*, 9(4):30:1–30:24, Jan. 2013.

List of Publications

Refereed Conference Publications

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 Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250, New York, NY, USA, 2012. ACM.

L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences*, GPCE '13, pages 105–114, New York, NY, USA, 2013. ACM.

L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder, and P. Tůma. Introduction to dynamic program analysis with DiSL. *Science of Computer Programming, 5th Special Issue on Experimental Software and Toolkits*, In Press.

L. Marek, Y. Zheng, D. Ansaloni, A. Sarimbekov, W. Binder, P. Tůma, and Z. Qi. Java Bytecode Instrumentation Made Easy: The DiSL Framework for Dynamic Program Analysis (Demo paper). In *Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 256–263. Springer Berlin Heidelberg, 2012.

L. Marek, Y. Zheng, D. Ansaloni, L. Bulej, A. Sarimbekov, W. Binder, and Z. Qi. Introduction to Dynamic Program Analysis with DiSL (Demo paper). In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 429–430, New York, NY, USA, 2013. ACM.

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 *Objects, Models, Components, Patterns*, volume 7304 of *Lecture Notes in Computer Science*, pages 353–368. Springer Berlin Heidelberg, 2012.

A. Sarimbekov, Y. Zheng, D. Ansaloni, L. Bulej, L. Marek, W. Binder, P. Tuma, and Z. Qi. Productive Development of Dynamic Program Analysis Tools with DiSL. In *Software Engineering Conference (ASWEC), 2013 22nd Australian*, pages 11–19, June 2013.

S. Kell, D. Ansaloni, W. Binder, and L. Marek. The JVM is Not Observable Enough (and What to Do About It). In *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '12, pages 33–38, New York, NY, USA, 2012. ACM.

D. Ansaloni, W. Binder, C. Bockisch, E. Bodden, K. Hatun, L. Marek, Z. Qi, A. Sarimbekov, A. Sewe, P. Tůma, and Y. Zheng. Challenges for Refinement

and Composition of Instrumentations: Position Paper. In *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 86–96. Springer Berlin Heidelberg, 2012.

V. Babka, L. Marek, and P. Tůma. When Misses Differ: Investigating Impact of Cache Misses on Observed Performance. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS 2009)*, pages 112–119. IEEE, December 2009.

T. Martinec, L. Marek, A. Steinhauser, P. Tůma, Q. Noorshams, A. Rentschler, and R. Reussner. Constructing Performance Model of JMS Middleware Platform. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*, ICPE '14, pages 123–134, New York, NY, USA, 2014. ACM.

Technical Reports

V. Babka, L. Bulej, M. Děcký, J. Kraft, P. Libič, L. Marek, C. Seceleanu, and P. Tůma. Resource usage modeling, Q-ImPrESS project deliverable D3.3. `http://www.q-impress.eu`, February 2009.

V. Babka, L. Bulej, P. Libič, L. Marek, T. Martinec, A. Podzimek, and P. Tůma. Resource impact analysis, Q-ImPrESS project deliverable D3.4. `http://www.q-impress.eu`, January 2011.

V. Babka, L. Bulej, A. Ciancone, A. Filieri, M. Hauck, P. Libic, L. Marek, J. Stammel, and P. Tuma. Prediction Validation, Q-ImPrESS Project Deliverable D4.2. `http://www.q-impress.eu/`, 2010.

J. Keznikl, M. Malohlava, L. Marek, and P. Tůma. Ferdinand Project Middleware List, Tech. Report No. 2011/2. Dep. of SW Engineering, Charles University in Prague, `http://d3s.mff.cuni.cz/publications/download/KeznikIMalohlavaMarekTuma-MiddlewareList.pdf`, 2011.

L. Bulej, L. Marek, and P. Tůma. Object Instance Profiling, Tech. Report No. 2009/7. Dep. of SW Engineering, Charles University in Prague, `http://d3s.mff.cuni.cz/publications/download/BulejMarekTuma-Technical2009-07.pdf`, 2009.

Other Publications

L. Marek, Y. Zheng, D. Ansaloni, W. Binder, Z. Qi, and P. Tuma. DiSL: An Extensible Language for Efficient and Comprehensive Dynamic Program Analysis. In *Proceedings of the Seventh Workshop on Domain-Specific Aspect Languages*, DSAL '12, pages 27–28, New York, NY, USA, 2012. ACM. Invited talk at DSAL '12.