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,

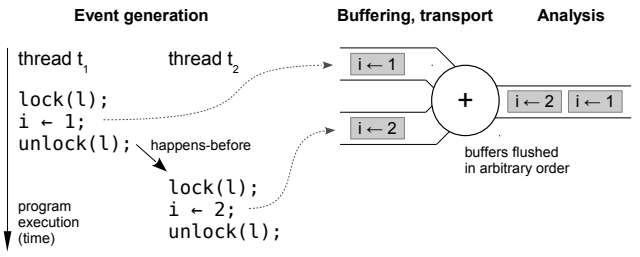

Figure 2. Multiple buffering can cause reordering of observations. Here, two assignments ordered by synchronization in the program are nevertheless reordered in the event stream fed to the analysis.

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.

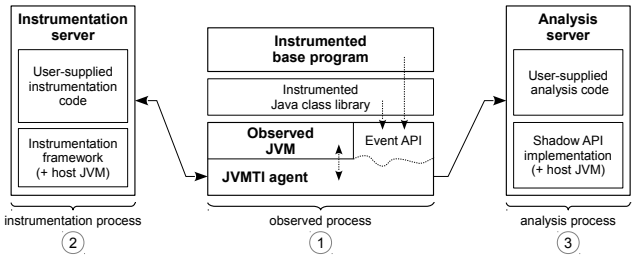

Figure 3. ShadowVM architecture at a high level.

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.

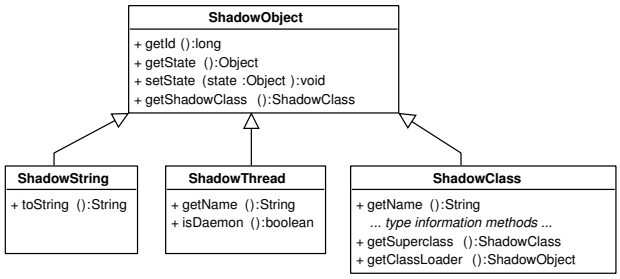

Figure 5. The Shadow API provides an analysis-friendly view of objects and threads in the observed program. Shadow objects are in bijection with the subset of objects in the observed program that have been passed to the analysis. Each object allows association of shadow state, which can be used to store 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 `WeakReferences`. 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 `WeakReferences`. Excluding the reference handling code avoids this problem, but loses coverage of reference handling on behalf of the observed program. Our design avoids using `WeakReferences` 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

	In-process	ShadowVM			
		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.

	In-process	ShadowVM concurrent tagging			
		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.