The JVM is Not Observable Enough (and What To Do About It)

Stephen Kell Danilo Ansaloni Walter Binder
University of Lugano
firstname.lastname@usi.ch

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

Abstract

Bytecode instrumentation is a preferred technique for building profiling, debugging and monitoring tools targeting the Java Virtual Machine (JVM), yet is fundamentally dangerous. We illustrate its dangers with several examples gathered while building the DiSL instrumentation framework. We argue that no Java platform mechanism provides simultaneously adequate performance, reliability and expressiveness, but that this weakness is fixable. To elaborate, we contrast *internal* with *external* observation, and sketch some approaches and requirements for a hybrid mechanism.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—run-time environments

General Terms Measurement, Reliability, Performance

Keywords bytecode, instrumentation, DiSL, dynamic analysis, JPDA, JVMTI, profilers, debuggers

1. Introduction

Developers working with a virtual machine (VM) depend critically on its *observability*, meaning the ability to monitor and analyse the guest program's execution. Debuggers, profilers and other dynamic analysis tools are the programmer's interface to observability. In turn, the authors of these tools rely on VM-level mechanisms to build these tools; essentially every virtual machine provides some such facilities.

The Java Virtual Machine (JVM) is the target of many tools, developed by product engineers and researchers alike. It provides two basic observation facilities: the Java Platform Debug Architecture, a set of interfaces for interrogating a debug server running inside the VM; and the JVM Tool Interface, an interface for interposing an "agent" library which is commonly used to instrument bytecode at load time.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VMIL'12, October 21, 2012, Tucson, Arizona, USA.
Copyright © 2012 ACM 978-1-4503-1633-0/12/10. . . $10.00

These mechanisms are inadequate. JPDA[1] is a usable basis for debuggers, but for dynamic analyses offers inherently limited performance and expressiveness. Meanwhile, our experiences building the DiSL instrumentation framework [11] using JVMTI-supported bytecode instrumentation show that common use cases cannot be realised without risking the introduction of show-stopping bugs, including deadlock and VM crashes. These problems can be worked around only by reducing the scope of observation.

We do not claim to be the first to observe these difficulties. In this paper our intention is to highlight them as a deeper issue. They are not simply quirks or "gotchas" for tool authors to be aware of; they are real obstacles to expanding the range and quality of tools available to programmers. By collecting the problems, underlining their severity, and characterising the requirements and design space for an eventual solution, we hope to advance the agenda of high-quality tool construction for managed runtimes. In summary, our contributions are:

- to identify seven design problems with instrumentation-based tools, illustrated with practical examples gathered during the development of DiSL;
- to survey the spectrum between *internal* and *external* observation, considering the JVM's relative strengths in these two modes;
- to sketch a set of requirements and possible approaches, motivated by existing literature, for a safe, efficient observability mechanism combining the benefits of internal and external observation.

Our latent position is that the JVM is not sacred. Considerable effort among researchers—ourselves included—is expended on building tools which exhibit good properties using only standard JVM interfaces. Much of this effort is wasted, because it ignores the real problem: the design of general, high-performance observability mechanisms is an open research challenge.

We begin by reviewing the JVM's observability facilities.

[1] Our canonical references for JPDA and JVMTI are the guides supplied by Oracle, as retrieved on 2012/8/16 from http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/. Note that strictly speaking, JVMTI is *part of* JPDA; when we refer to JPDA, we are more precisely referring to its other two constituent interfaces: JDI and JDWP.

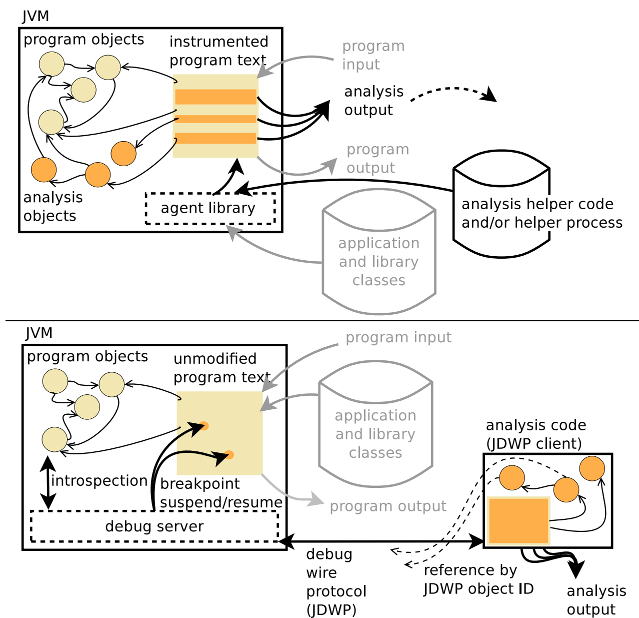

Figure 1. Internal observation by instrumentation (top) versus JPDA-based external observation (bottom)

2. Observability on the JVM

Like physical systems, software systems exhibit a tension between *observation* and *perturbation*: one cannot observe a system without affecting it somehow. This informs the design of VM-level observation mechanisms. The two mechanisms offered by the JVM platform—JVMTI and JPDA (contrasted in Fig. 1)—approach perturbation differently.

JVMTI allows construction of tools by linking a native library, called an "agent", into the VM. This library interposes on various VM events. Significant among these is class loading, where replacement code may be supplied by the agent. JVMTI's design deliberately emphasises tool construction by bytecode instrumentation: its documentation[2] notes that "this interface does not include some events that one might expect... [but] instead provides support for bytecode instrumentation, the ability to alter [the] bytecode instructions which comprise the target program". To minimise perturbation, the same document also suggests that agents should be "controlled by a separate process which implements the bulk of a tool's function without interfering with the target application's normal execution". Avoiding perturbation therefore becomes the tool author's problem.

Meanwhile, JDPA[3] "goes to great pains to avoid the execution of any code in the debuggee virtual machine" because in-process analysis "interferes with the behavior being analyzed... for example: ... competition for resources [means that] deadlock can occur" and that "many operations can only be reliably performed in a suspended virtual machine". As a result, the interface is relatively constrained in both expressiveness and performance: the wire protocol supports only a fixed set of queries, many of which execute only on suspended threads or a suspended VM. Although arbitrary analysis computations could be performed externally in the debugging process (and effectively this is what debugger-based Java expression evaluators do), implementing such an evaluator is a nontrivial undertaking, and the continual need to suspend and resume parts of the VM severely reduces overall performance. Most JVMs fall back to unoptimised or deoptimised execution of code observed by a debug client.

We first discuss various practicalities of bytecode instrumentation; JPDA is discussed subsequently.

3. Current practice

Most dynamic analysis tools for the JVM work by bytecode instrumentation, using JVMTI (or, rarely, performing the instrumentation offline). We call this *internal observation*: a single process contains both program and analysis.[4] Although JVMTI's documentation endorses a separate-process approach to minimise perturbation, to our knowledge only a small minority of instrumentation-based tools actually follow such a design. Simplicity and performance are likely explanations; certainly, these motivated DiSL's initial single-process design. Remote processes require marshalling and copying code, with its associated development and runtime overheads. In contrast, processing within local instrumentation does not incur these overheads, and benefits from JIT optimisations. In this section we review a series of problems encountered while building and using DiSL, which we believe are inherent to internal observation on today's JVM.

3.1 An example analysis

Consider a simple tool for identifying fields that are immutable (or likely to be) in a Java program, suggesting to the programmer that they could be made final. Clearly, we should instrument bytecode performing field writes, and record them as a set of per-field per-class "mutable" flags. Fields whose flag remains unset are likely to have immutable semantics, so could be made `final`. Such a tool was constructed in DiSL and has been used in published work [16]. Unfortunately, even simple instrumentations exhibit subtle problems; in the remainder of this section we describe several problems that this example and/or comparably simple instrumentation-based analyses easily encounter.

3.2 Deadlock of non-wait-free analyses

Fig. 2 shows a simplified set of stack traces that we have observed while using our immutability analysis. The analysis data structure, a `WeakKeyIdentityHashMap`, is protected

[2] Retrieved on 2012/8/16 from `http://docs.oracle.com/javase/6/docs/platform/jvmti/jvmti.html`.

[3] Quotations are from Sun's Frequently Asked Questions on the Java Platform Debugger Architecture, retrieved on 2012/8/16 from `http://java.sun.com/javase/technologies/core/toolsapis/jpda/faqs.jsp`.

[4] We note that often, as with DiSL, the instrumentation itself is done in a separate process spawned by the agent.

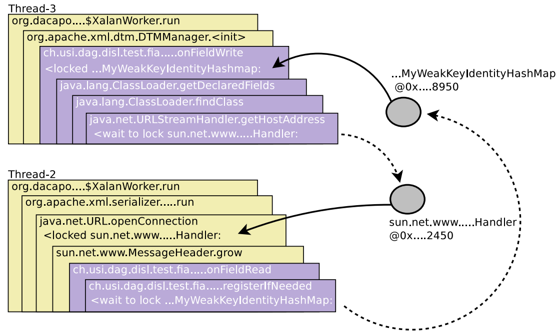

Figure 2. Deadlock between instrumentation and program. Purple (darker) boxes represent calls resulting from instrumentation; beige (lighter) boxes are in the base program.

by a lock, for which at least two threads are contending. Thread-3 is running user code that performs a field access, so the instrumentation locks the analysis structure. It then queries the target object's java.lang.Class to allocate some per-class state. In turn, this queries the class loader., which makes calls to the I/O library, requiring a lock on a Handler object. Unfortunately, Thread-2 has acquired a lock on the same Handler object and, owing to a field access in the same critical section, has itself called into the analysis and is waiting on the analysis lock. This is a classic deadlock: the two threads are contending for the same pair of locks but in opposite orders. Under instrumentation there is no way to enforce a global locking order, because the emergent ordering of locking operations depends on the implementation details of the instrumented code—which the tool author cannot reasonably know. One solution is to use only wait-free code in instrumentation, or to ensure that any lock taken out by instrumentation is a leaf lock (not held during any other locking operation). However, this requirement is even stronger than it first appears; we consider it shortly.

3.3 State corruption of non-reentrant code

Instrumentation can cause non-reentrant code to be invoked reentrantly, leading to state corruption. Consider a common requirement in instrumentation: to print out a message to the console. Normally a println() implementation avoids calling itself, so need not be reentrant. Suppose that it models a state machine, as in the sketch of Fig. 3. This code is perfectly correct; however, if we introduce some instrumentation to the definition of copySome() which prints out a message, the state machine will be advanced prematurely by a reentrant println() invocation, only to resume the first println() in the wrong state (causing an assertion failure at line 7). Note that the problem arises from interleaving *within a single thread*, so thread-safe code still exhibits this problem.

3.4 Calling methods

Our last two problems suggest that perhaps *any method* called on base program state from instrumentation is dangerous. We might therefore say: don't make any such calls!

```
1  /* inside a non-reentrant method,
2     perhaps java.io.PrintStream.println()...  */
3  try {
4    this.state = PENDING; // non-reentrant state machine
5    while (pos != len) pos = copySome(in, out, pos, len);
6  } finally {              /*   ↖ now makes reentrant call! */
7    assert this.state == PENDING; // fails following reentrant call
8    this.state = CLEAR;
9  }
```

Figure 3. Non-reentrant code corrupted by instrumentation

Indeed, to avoid deadlock (by wait-free instrumentation or "only leaf locking", as in §3.2) we must enforce this rule, because the waiting and locking behaviour of arbitrary methods is unknown. Unfortunately, enforcing this rule severely limits our expressiveness, because calling methods is indispensible in many circumstances. Suppose we want to analyse use of a library API making pervasive use of a user-defined type like Date or Currency as a method argument. Collecting contextual information about an event (for example, the month of the Date being passed) invariably means calling methods (on the Date object). Aggregating by such information also entails method calls—such as equals(), compareTo() or hashCode()—made by the aggregating container. All of these methods are entitled to perform locking. Although in these cases we could perhaps use JNI to make raw (private) field accesses instead of method calls, targeting private interfaces is little more desirable in instrumentation than in normal code. Method calls are also necessary to perform I/O, which invariably risks contention for per-VM data structures such as file handle tables.

3.5 Plausible instrumentation crashes the VM

It would be a convenient facility to instrument Object.<init>, because this captures all object initialization events. Unfortunately, doing so on at least one popular JVM (namely HotSpot) crashes the JVM.[5] Similarly, adding fields to Object might be an efficient way for an analysis to associate additional state with each object, but this also crashes HotSpot. The underlying problem is that there is no specification about what instrumentations are required to be supported by the VM; it is *undefined* whether this is a bug in HotSpot. (We emphasise that *all* problems described in this section, although inevitably triggered using specific JVM or library implementations, are not implementation-specific problems. Rather, sharing the JVM between instrumentation and user code creates unavoidable risk of bad interactions.)

3.6 Bytecode verification failure

Our immutability analysis needs to determine whether a given field write occurs during the execution of the target

[5] Actually, whether HotSpot crashes depends on precisely what instrumentation does, but without any obvious pattern. For example, we found that constructing a String with the += operator reliably crashed the VM, but constructing one from a literal did not.

object's constructor (meaning the field may be immutable) or afterwards (meaning it must be mutable). Unfortunately, to determine which field is being written, our analysis requires a reference to the containing object; if the constructor is still executing, this is an uninitialized object, and passing a references to it is conservatively forbidden by the bytecode verifier. We are forced to run this analysis with verification turned off. In general, objects which are not yet initialized may nevertheless be of interest to an analysis, but such analyses are not accommodated by the bytecode verifier.

3.7 Coverage underapproximations

DiSL supports instrumentation of the entire class library, not just user-supplied code. Indeed, instrumenting the sensitive code found deep in the libraries has helped expose the problems we have encountered. (However, all of them *could* arise purely in user code.) To allow the same library classes to be both *instrumented* and *used by* instrumentation without infinite recursion, a "bypass" is used: a thread-local flag records whether execution is currently at the base level (the program) or meta-level (the code inserted by instrumentation). Each method body is duplicated in both arms of an if–else construct testing this flag, with only the "false" (base level) copy being instrumented. In this way, helper calls made by the analysis into library code (such as containers) are not themselves analysed.[6] In general, we seek to avoid both this *over-analysis* (analysing the analysis, possibly causing infinite regress) and also *under-analysis* (loss of coverage, e.g. if we instead omitted to instrument the class library). Unfortunately, this thread-local bypass is only approximate; it avoids neither over- nor under-analysis. A simple example of under-analysis is the static initializer of a class which is used by the base program, but now also used *earlier* by instrumentation: its initializer will be run uninstrumented, when in fact it would later have been run by the base program and should therefore be analysed. The bypass is also active on all execution before the main() method, to avoid perturbing the load order of core classes (which is critical to VM bootstrapping), so these classes' static initializers are also not covered.

3.8 Reference handler over-analysis

The bypass flag avoids over-analysis within a single thread. However, when work is passed between threads, it cannot help. A notable example is reference handling. Many analyses make heavy use of WeakReferences, to track program objects without preventing their collection. Unfortunately, the task of appending cleared WeakReferences to their intended ReferenceQueue is usually implemented by a shared reference handler thread, implemented in class library code and therefore subject to instrumentation. All work done by this code is analysed, even though *some* of these references are due to the analysis rather than the base program. The problem is exacerbated when the analysis running in the reference handler thread itself allocates WeakReferences, hence creating yet more work for the reference handler, hence more allocations. This cycle is rate-limited by the lifetime of the WeakReference target, but can still exhaust memory. In affected applications of DiSL we have worked around this manually excluding the reference handler thread from analysis using an if-test in each inserted snippet. However, this turns overanalysis into underanalysis: reference processing for the base program is no longer analysed.

4. External observation

Our immediate plans for improving DiSL rest on a new design strategy: "execute as little code as possible in the observed process". However, this statement begs two important questions: how little is possible, and will this really solve our problems? In this section we review the current options for external observation of JVMs, and also consider related designs not currently implemented by most JVMs.

4.1 Options for external observation of JVMs

Native code The closest vantage point "outside" a JVM is from native code in the same process. Some JVMTI-based systems such as hprof [10] perform analysis in native code to reduce perturbation. Unfortunately, most potential problems remain in some form. The biggest effect of shifting analysis to native code is to reduce coverage (since native code is not itself observed) and so reduce the likelihood of hitting a problem. However, native code is not intrinsically safer; deadlock (§3.2) and reentrancy (§3.3) remain issues as presented earlier, and gathering contextual data (§3.4) will still generally require calls back into Java code.

Separate process The JVMTI documentation recommends doing most analysis from a separate process. However, in such a design, inserted bytecode must still be used to *collect* data and transmit it to a remote process using some IPC mechanism (such as a ring buffer in shared memory). So, while storage and computational processing are done in a separate process, the design does not fundamentally prevent any of the same problems from occurring. In particular, simply collecting data can easily require a method call; if so, then this call must be made within the observed process.

JVM debugging interfaces As outlined in §2, JPDA facilities designed primarily for the construction of interactive debuggers can also support a variety of dynamic analysis tools (including Caffeine [6] a trace collector and query engine, and the first version of the PROSE aspect weaver [14]). However, these tools run slowly: most JVM implementations run unoptimised code when a debugger is attached. (Even HotSpot's "full-speed debugging" works by dynamic deoptimisation of the debugged code.[7]) Furthermore, we

[6] This technique is called "polymorphic bytecode instrumentation" [12]. We previously believed it to offer adequate separation of meta-levels, until further experience uncovered the problems in §3.7 and §3.8.

[7] Described in a HotSpot white paper, retrieved from http://www.oracle.com/technetwork/java/whitepaper-135217.html on 2012/8/17.

note that debug clients acquire the ability to query VM state thanks to the presence *within* the VM of a debug server (talking JDWP[8]). This is therefore arguably not external observation at all! *Pure* external observability requires that observing a program's execution involves adding *no* code in the target process. This is not supported by any Java platform specification. Significantly, JVMs need not publish their data representations or stack frame layouts, so cannot be observed from memory dumps or peek/poke-style interfaces.

4.2 Pure external observation

Some existing JVMs provide additional support for pure external observation of program state. We first discuss this support, then discuss a real-world use case.

4.2.1 Vendor extensions

Some JVM vendors have added limited forms of pure external observability using custom interfaces. HotSpot provides a set of "SA tools", for Serviceability Agent, a "Sun private component… developed by engineers… debugging HotSpot [who] then realized that SA could be used to craft serviceability tools for end users". In particular, SA "can expose Java objects as well as HotSpot data structures both in running processes and in core files".[9] SA tools include the jmap memory-map dumper, the jstack stack tracer, the jhat heap dump analyser, and others.

Specifying this kind of mechanism in the Java platform, ideally using compiler-generated descriptive debugging information (rather than "baked in" VM-specific knowledge used by the SA tools) would be a step forward, in allowing external observation without deoptimised execution and also in post-mortem cases. Fig. 4 illustrates. JVMs built on native compiler back-ends, such as gcj [2] for gcc or VMKit [5] with LLVM [9], already inherit this ability. However, just as method calls were preferable to digging for fields with JNI (§3.4), access to raw fields is less useful than the ability to isolate bytecode-based instrumentation would be.[10]

4.2.2 Use case: DTrace on Java

A cutting-edge application of external observation is found in DTrace [3], a dynamic tracing tool designed for safety, high coverage, and performance appropriate for use on production systems. DTrace primarily targets native code at both user and kernel levels. All analysis code runs in the kernel, sandboxed within an interpreted virtual machine subject to various load- and run-time checks.[11]

[8] http://docs.oracle.com/javase/1.5.0/docs/guide/jpda/jdwp-spec.html

[9] From "Serviceability in Hotspot", retrieved on 2012/8/15 from http://openjdk.java.net/groups/hotspot/docs/Serviceability.html.

[10] Pure external observers of Java will require notification when objects are moved by the collector; a portable protocol for this is also required.

[11] DTrace arguably then does *internal* observation of kernel code. It avoids deadlock and reentrancy problems (§3) using a wait-free analysis path which mutates only private state and may not itself be instrumented.

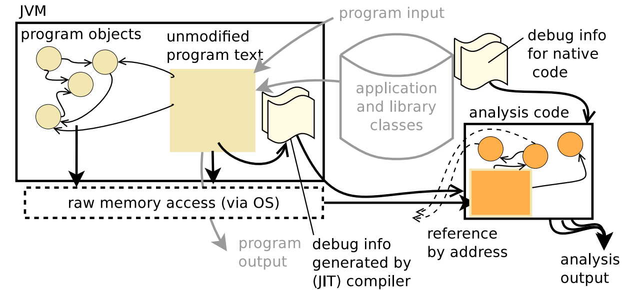

Figure 4. Pure external observation using descriptive debugging information

An essential design feature of DTrace is that little information is propagated proactively from the analysed program to DTrace. Rather, DTrace kernel code extracts state from the observed program (such as the stack trace, current function arguments, and data gathered from walking data structures), using memory access and debugging information much like a native debugger. In this way, probes can be enabled and disabled without the base program's involvement, and unwanted probe data can be discarded at source. This relies on the ability of DTrace kernel code to decode the data structures and stack frames of target code, so cannot be supported with existing JVM observability mechanisms. Existing portable solutions for running DTrace in Java code (including the JVMTI-based dvmti[12] provider, and the BTrace[13] bytecode instrumentation systems) are forced to proactively *marshal* data into predictable form, negating the "discard at source" feature and adding slowdown even for disabled probes (an overhead avoided by most DTrace providers). Recent versions of HotSpot now contain a built-in DTrace provider, which permits a more optimised but VM-specific approach (analogous with the "Serviceability Agent", §4.2.1), reaffirming our position that the specified observability mechanisms are not sufficient.

5. How to fix it

We remain committed to the approach of running analysis outside the JVM as far as possible. On the current JVM platform this dooms us to limitations. We believe that the design of VM-level mechanisms for fast, safe observation is an open challenge, and specifically that an optimal synthesis of internal and external observation is yet to be achieved. Here we sketch some ideas and requirements for such mechanisms.

Inlined guards and other "safe" instrumentation The dynamic compilation available in JVMs should allow us to achieve a *better* isolation/performance trade-off than in native code. Some code really is effect-free and can safely be inserted as instrumentation, where it can be optimised. This could, for example, avoid redundant traps in DTrace for false

[12] http://kenai.com/projects/dvm/

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

predicates (§4.2.2). Useful guidance could come from purity analyses already performed by JIT compilers.

Isolated bytecode Executing analysis against snapshots of program state is a convenient abstraction. Object-level copy-on-write snapshots have already been demonstrated by work on asynchronous assertions [1]. The same approach could allow instrumentation bytecode to execute "as if" in the target process, but in an effect-free fashion. The resulting "sandboxed bytecode evaluator" could be a candidate for replacing JDWP, much as JVMTI uses bytecode instrumentation to replace various utility calls in its predecessor JVMPI [10].

Free association Maintaining *per-object* analysis state is currently done using associative mappings (e.g. keyed on WeakReferences). We noted (§3.5) that adding fields to Object would be a useful alternative. Meanwhile, adding fields to *every* object could be wasteful if only certain objects are of interest. Fast disjoint metadata implementations using virtual memory techniques have appeared in recent work [8, 13]. A useful addition to instrumentation libraries could be to specify the availability of an associative container keyed on object identity with a strong performance contract.

Meta-level separation as a VM service The concept of software-isolated processes or "isolates" is well developed [4, 7] and could be the basis of an isolated metalevel. A distinction from the normal case is that information flow *in one direction* must be permitted. Fitting a suitable design onto the JVM would at least require eliminating shared threads (cf. §3.8). (We note that DTrace's in-kernel virtual machine, described in §4.2.2, is another instance of software isolation, i.e. with respect to the wider kernel.)

Record/replay correctness It is currently a difficult task to actually test that an analysis does not unduly perturb the program it is observing. Some performance effect is always expected, and in the case of instrumentation, the path taken through the program will necessarily be modified too. An intuitive requirement is that we should be able to erase the analysis parts of the path and find the base program path otherwise unchanged. A useful approximation of this criterion might be available from record/replay systems such as that of Saito [15]: a replay log from an uninstrumented run should be replayable against the instrumented code without divergence, and producing the same output (as well as additional output from the analysis). This will likely require some support from the VM to tolerate execution differences without causing divergence, e.g. concerning garbage collection: the instrumented program will allocate more memory and collect more often.

6. Conclusions

We have shown, with examples, that bytecode instrumentation poses severe and unavoidable dangers as the basis of tool construction, yet no other Java platform mechanism is adequate. We have motivated the open challenge of designing a efficient, isolated observation mechanism suitable for JVMs, and have provided some initial design sketches.

Acknowledgments

The research presented here has been supported by the Swiss National Science Foundation (project CRSII2_136225), by the European Commission (Seventh Framework Programme grant 287746) and by the Czech Science Foundation (project GACR P202/10/J042). The authors thank their fellow DiSL authors and contributors: Aibek Sarimbekov, Petr Tůma, Yudi Zheng, Andreas Sewe.

References

[1] E. E. Aftandilian, S. Z. Guyer, M. Vechev, and E. Yahav. Asynchronous assertions. In *Proc. OOPSLA '11*. ACM.

[2] P. Bothner. Compiling Java with GCJ. *Linux Journal*, 2003.

[3] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proc. USENIX ATEC '04*. USENIX Association.

[4] G. Czajkowski and L. Daynés. Multitasking without compromise: a virtual machine evolution. In *Proc. OOPSLA '01*. ACM.

[5] N. Geoffray. *Fostering Systems Research with Managed Runtimes*. PhD thesis, Paris, France, September 2009.

[6] Y.-G. Gueheneuc, R. Douence, and N. Jussien. No Java without Caffeine: A tool for dynamic analysis of Java programs. In *Proc. ASE '02*. IEEE, 2002.

[7] G. C. Hunt and J. R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, Apr. 2007.

[8] S. Kell and C. Irwin. Virtual machines should be invisible. In *Proc. VMIL '11*, SPLASH '11 Workshops. ACM.

[9] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*. IEEE Computer Society, IEEE.

[10] S. Liang and D. Viswanathan. Comprehensive profiling support in the Java virtual machine. In *Proc. COOTS '99*, COOTS'99, Berkeley, CA, USA. USENIX Association.

[11] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: a domain-specific language for bytecode instrumentation. In *Proc. AOSD '12*. ACM.

[12] P. Moret, W. Binder, and E. Tanter. Polymorphic bytecode instrumentation. In *Proc. AOSD '10*. ACM.

[13] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic. Softbound: highly compatible and complete spatial memory safety for C. In *Proc. PLDI '09*. ACM.

[14] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. AOSD '02*. ACM.

[15] Y. Saito. Jockey: a user-space library for record-replay debugging. In *Proc. AADEBUG '05*. ACM.

[16] A. Sewe, M. Mezini, A. Sarimbekov, D. Ansaloni, W. Binder, N. Ricci, and S. Z. Guyer. new Scala() instance of Java: a comparison of the memory behaviour of Java and Scala programs. In *Proc. ISMM '12*. ACM.