Building Debuggers and Other Tools: We Can "Have it All"

Position Paper ICOOOLPS '15

Michael L. Van De Vanter
Oracle Labs
michael.van.de.vanter@oracle.com

Abstract

Software development tools that "instrument" running programs, notably debuggers, are presumed to demand difficult tradeoffs among *performance*, *functionality*, *implementation complexity*, and *user convenience*. A fundamental change in our thinking about such tools makes that presumption obsolete.

By building instrumentation directly into the core of a high-performance language implementation framework, tool-support can be *always on*, with confidence that optimization will apply uniformly to instrumentation and result in near zero overhead. Tools can be always available (and fast), not only for end user programmers, but also for language implementors throughout development.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Debuggers, Interpreters, Optimization, Runtime environments

Keywords Virtual Machine, Instrumentation, Optimization, Debugging, Tools

1. Introduction

A time-honored lament among users of new programming languages is that supporting *tools* (profilers, debuggers, coverage analyzers, etc.) typically arrive late, if ever. Moreover, the tools that finally arrive are likely to exhibit design compromises that cost productivity, for example:

- Compilers running at high optimization levels are unlikely to support tools, making it difficult to observe bugs and other phenomena in long-running or production environments.
- Tools that do operate at higher optimization levels are likely to suffer functional limitations and may become unreliable.
- Dedicated compiler support is often required, increasing expense and decreasing tool portability (and thus availability).
- Tools may only be available in compromised runtime modes, for example the JVM's performance-limiting `-Xdebug` option, making them unavailable in routine situations.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICOOOLPS '15, July 6, 2015, Prague, Czech Republic.
Copyright © 2015 ACM 978-1-nnnn-nnnn-n/yy/mm. . . $15.00.
http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

2. Roadblocks

Why is it so difficult to have tools that are as good and timely as our programming languages? Why can't we "have it all"?

2.1 Tribes

One perspective is historical and cultural. Concerns about program execution speed (utilization of *expensive machines*) came long before concerns about software development rate and correctness (utilization of *expensive people*).

Our legacy is that people who write compilers and people who build developer tools essentially belong to different *tribes*, each with its own technologies and priorities[1]. More significantly, each has its own *mindset*: one trafficks in small performance improvements, the other in human nature. A good metaphor would be that tool builders are the user interface designers for the technologies built by compiler/runtime specialists: in other words *front-end* vs. *back-end* technologies.

Another legacy is that the tribes tend to be organizationally distant, often not even in the same companies.

2.2 Technology

Tribal distinctions aside, front-end and back-end technologies could most politely be described as "not well aligned". For example, a typical compilation pipeline immediately abstracts away the information least relevant to execution: comments, whitespace (i.e. layout), variable names, etc. This is done in the name of both performance and convenience for language implementors. However, these are precisely the *most important* things about code that human readers rely upon [7].

This example points at a larger observation at the heart of the problem. Much of the information needed by developer tools is necessarily related to the human programmer perspective, and this is often the information most difficult to extract from a highly optimized running program.

3. Two Strategies

Tools do manage to get built, however, and the approaches taken for access to runtime information fall predictably into the two camps: one where performance is paramount, and the other where tools matter most.

3.1 Reengineering Execution State

Conventional performance-focused implementations require that tools, debuggers in particular, *reengineer* the execution state of a halted program from a highly optimized runtime representation.

[1] Any provocative dichotomy has exceptions, in this case the creators of languages/environments such as Self, Smalltalk, Lisp and others. They are in fact exceptional and are the intellectual ancestors of much of this work.

This usually requires additional information, typically provided through a side-channel in the name of efficiency, and that raises issues of availability, consistency, and (expensive) complexity.

Especially in the realm of debugging, we find a deep, long-standing, and much studied tension in the disconnect between performance optimization and the increasingly complex and expensive task of reengineering state. In the limit, debugging becomes less functional, then less reliable, and finally impossible. This is the legacy of a "speed first, tools later" mindset.

Research underway will lighten the burden on the compiler writer [5], but the task remains burdensome, and the resulting information is restricted to what's explicitly produced during static compilation.

3.2 Generated Tools

Tool builders approach the problem from the opposite perspective. Contemporary "language workbenches" expedite the development of languages (often small and domain-specific) and their supporting tools together, usually driving the development from grammars and other formal descriptions.

For example, Wu et al. propose a framework for source level debugging that includes shared, language-agnostic code together with description-driven generation of language-specific features [10]. Unfortunately the technique only applies to a particularly limited language implementation technique.

The Spoofax language workbench generates debuggers in a more general way and is also description-driven [4]. Following an approach with some precedent [3], additional code (call-outs to the debugging support library) is added to each program before compilation. This technique, potentially expensive, requires that the decision to debug be made before any compilation.

4. Opportunity

"Having it all" clearly calls for different thinking, perhaps a complete break with the established tribal disconnect. A language implementation platform under development at Oracle Labs embodies a new way of thinking about dynamic language optimization, and this opens the door for new ways of thinking about tool development.

4.1 Truffle/Graal

Truffle is a platform under development by the Labs' Virtual Machine Research Group (VMRG) for constructing high performance implementations of dynamic languages. A Truffle-based implementation is expressed as an abstract syntax tree (AST) interpreter written in Java[2], to which the framework (including the *Graal* compiler) applies aggressive dynamic optimizations that include type specialization, inlining, and many other techniques [11].

4.2 Deoptimization

Central to Truffle/Graal is the ability to optimize code, through partial evaluation and inlining, based on *speculation*: carefully managed assumptions about the future behavior of a program that are likely, but not guaranteed, to remain true. The critical companion to speculation is the ability to revert to AST interpretation when an assumption ceases to hold, and in particular to do so *without loss of program execution state*. So-called "deoptimization" was first developed to support builtin debugging in the Self language [3].

Deoptimization is not completely free: the information needed to reconstruct execution state and the guards that check assumptions both incur costs. However, mature and efficient techniques for these have been developed during the years since Self, for example in both in the Graal compiler [1] and its predecessor in the VMRG's Maxine VM [9]. Although scenarios remain where the ability to deoptimize precludes some kinds of optimizations, the compromise has proven more than worthwhile.

4.3 Truffle Instrumentation

The development of Truffle's instrumentation support represents a break with tradition. It is designed for generality, to expedite the construction of any tools that require dynamic access to runtime execution state [8] [2] and for any Truffle-implemented language. More importantly, it has also been designed in close collaboration with the Truffle team so that it has both minimal impact on program optimization and maximum exposure to those same optimizations.

Such close alignment does not come easily; adjustments on both sides have been (and continue to be) needed as the platform evolves. For example, until the addition of one particular optimization in 2014, the instrumentation nodes attached to a particular method's AST could be shared automatically by all clones of the AST. That changed, precipitating a complete reimplementation of the instrumentation framework, as well as an API redesign.

The outcome is a productive synergy between the two aspects of the platform: optimization enables instrumentation at near-zero cost, and early tool support helps other aspects of platform development.

4.4 Instrumentation Events

The client model for Truffle instrumentation is the interception of *execution events* at particular program locations, for example the event "the AST node implementing a statement is about to be executed". Instrumentation is dynamic: any number of tools can independently *register* and *unregister* interest in any events at any number of important program locations.

The current (but still evolving) event API offers three levels of access to execution state:

- The simplest notification reports that an AST node implementing some piece of program syntax is about to be executed, or has just completed executing with a resulting *value*. Some tools need nothing more, for example a simple code coverage analyzer.
- A more complex notification carries references to the event's AST node and current frame. With these references, Truffle APIs make available additional aspects of program execution state. This is sufficient for many tools, including most aspects of a language-independent debugger, to be described below.
- A third API permits a client tool to attach an AST fragment at a program location with the guarantees that it will be executed upon certain execution events and that it will be subject to full optimization. This is not intended to interact directly with program execution, but rather to enable optimized execution of tool-related code that has traditionally been expensive, for example trace functions and breakpoint conditions.

The platform code under development to support instrumentation is largely shared, requiring only that specific *adapters* be supplied with each language implementation. An early example is a mechanism for identifying the AST nodes that correspond to syntact constructs of interest to programmers, such as statements, assignments, exceptions, etc. Another example is a suite of methods for printing language-specific information, such as program values and field or method names.

[2] Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

4.5 The Truffle Debug Engine

Although early versions of Truffle instrumentation have been used for profiling and code coverage, the most ambitious client to date is the Truffle "debug engine". Using the instrumentation API, the engine can set breakpoints of various flavors, can step (In, Out, and Over), navigate up and down the stack, inspect frame contents, and evaluate expressions.

Most of the debugging code is shared across languages, with some debugging-specific adapters needed for each. These currently include identification of nodes where stepping should halt, as well as methods for starting program execution and evaluating expressions in a halted context. The Debug Engine itself includes no special facilities for examining detailed execution state such as object contents, but relies at present on the (also evolving) Truffle runtime API.

Specializations of the debug engine have been created for JavaScript, Ruby, R, and a simple demonstration language. A simple client/server framework is available to test these specializations, using a language-agnostic command line client. An experimental branch of the NetBeans IDE uses the engine for debugging Truffle-implemented JavaScript code.

5. Status

Results to date are encouraging. An early experiment with a very limited form of instrumentation convinced us that close cooperation with Truffle optimization could lead to extremely low overhead [6]. The framework continues to be built out in many directions, along with bug fixes, adaptations to optimization changes, and extensions to client functionality.

A notable aspect of Truffle instrumentation is the part of the API that is *not present*: anything having to do with optimized execution. The API and most of the framework implementation deal with runtime state *only* as if the AST interpreter were executing as ordinary Java code.

The paramount goal for the framework and most client-supplied code is that it be optimized fully by Truffle, together with the program ASTs, until such time as access is required to state unavailable in optimized code. When that occurs, the instrumentation framework triggers the same kind of deoptimization that Truffle uses when a speculative assumption ceases to be true. Other than managing occasions for deoptimization, the instrumentation framework and clients never interact with optimized code.

Adding a breakpoint, for example, triggers deoptimization on a method since it modifies instrumentation code attached to the method's AST. Should that code run long enough to be optimized, then the breakpoint (along with a conditional expression) will be Truffle-optimized and might continue executing optimized until the breakpoint is activated and the program halts under debugger control.

We have yet to find any reason why Truffle instrumentation cannot be always on.

6. Conclusions

We aim to "have it all". Language implementations will be delivered to programmers with a robust, dynamic, instrumentation framework that can be available for debugging or other tools at any time, without serious compromise. Language implementors will have functional debuggers, profilers, and other tools available continuously from "hello world" status through to completion.

Acknowledgments

I am indebted to members of the Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz for creating the language implementation technologies that make this work possible. Very helpful comments on early versions of this paper were contributed by Michael Haupt, Chris Seaton, Mario Wolczko, and Thomas Würthinger. Yuval Peduel and the anonymous reviewers contributed addtional comments of great value.

References

[1] Gilles Duboscq, Thomas Würthinger, and Hanspeter Mössenböck. Speculation without regret: reducing deoptimization meta-data in the Graal compiler. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java platform: Virtual machines, Languages, and Tools (PPPJ '14)*. ACM, New York, NY, USA, 187-193.

[2] Michael Haupt and Hans Schippers. A Machine model for Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP) 2007*. Lecture Notes in Computer Science Volume 4609, 2007, pp 501-524, Springer Verlag.

[3] Urs Hölzle, Craig Chambers, and David Ungar. Debugging Optimized Code with Dynamic Deoptimization. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, 1992.

[4] Ricky Lindeman, Lennart CL Kats, and Eelco Visser. Declaratively defining domain-specific language debuggers. In *ACM SIGPLAN Notices*. Vol. 47. No. 3. ACM, 2011.

[5] Sukyoung Ryu and Norman Ramsey. Source-Level debugging for multiple languages with modest programming effort. *Proceedings of the 14th international conference on Compiler Construction*. Springer-Verlag, 2005.

[6] Chris Seaton, Michael L. Van De Vanter, and Michael Haupt. Debugging at Full Speed. In *Proceedings Workshop on Dynamic Languages and Applications DYLA '14*. Edinburgh, (June 2014)

[7] Michael L.Van De Vanter. The Documentary Structure of Source Code. *Information and Software Technology*, Volume 44, Issue 13, 1 October 2002, pp. 767-782.

[8] Michael L. Van De Vanter, Chris Seaton, Michael Haupt, Thomas Würthinger, and David Leibs. A Flexible, High-Performance, Multi-Language Instrumentation Framework. *In Preparation* 2015.

[9] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization* 9, 4, Article 30 (January 2013).

[10] Hui Wu, Jeff Gray, and Marjan Mernik. Grammar–driven generation of domain-specific language debuggers. *Software: Practice and Experience* 38.10 (2008): 1073-1103.

[11] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, Mario Wolczko. One VM to Rule Them All. In *Proceedings of Onward!*, ACM Press, 2013.