Building Flexible, Low-Overhead Tooling Support into a High-Performance Polyglot VM

Extended Abstract

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

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

Keywords Virtual Machine, Instrumentation, Optimization, Debugging, Tools

1. Introduction

The disconnect between programming languages and the developer tools needed to make them useful has grown wide since the days of *programming systems* with integral tool support, such as Self (1989), Smalltalk (1980), and Lisp (1965). Tools now are typically an *afterthought*: expensive to develop, delivered late if ever, and arrive with undesirable performance trade-offs.

The time has come to build modern programming systems. We can do that by embedding flexible, reusable, low-overhead instrumentation and tool support deeply into modern Virtual Machines. We can "have it all" (Van De Vanter 2015), but only if we can re-establish close collaboration between language engineers (who optimize utilization of *expensive machines*) and tool builders (who optimize utilization of *expensive people*).

2. Truffle-Graal

The Truffle Instrumentation and Debugging Framework is a tightly coupled extension under development for the open source *polyglot* (multi-language with interoperability) Graal platform (Würthinger et al. 2013). The primary goal for the framework is to simplify construction of tools needing dynamic access to execution state in this very high performance runtime environment. A secondary goal is to encourage advanced tool development and experimentation by offering public API access. The difficult challenge is to do all this without impact on runtime performance.

Inspiration for a new approach to this challenge comes from two sources: a generalized interposition model developed to support Aspect Oriented Programming in language VMs (Haupt and Schippers 2007) adapted to AST interpretation, combined with dynamic optimization and deoptimization from Self (Hölzle et al. 1992). It is now possible for an Instrumentation client to dynamically (and safely) insert into an executing program an instrumentation probe that incurs near zero performance cost until actually used to access (or modify) execution state.

The framework includes:

1. Low-level, extremely low-overhead execution event interposition, built directly into Graal's high-performance runtime;
2. The ability to dynamically inject code fragments (for example breakpoint conditions) into running code where they will be fully optimized together with surrounding code;
3. Reusable language-agnostic instrumentation services, requiring minimal per-language specialization, that operate smoothly across language *interoperation* boundaries; and
4. Versatile APIs for constructing many kinds of client tools without VM modification.

We have demonstrated that in fully optimized code the framework itself has near-zero overhead (Seaton et al. 2014). We have yet to find any reason why Truffle instrumentation should not be permanently enabled, as well as the services that use it.

3. Technology

Functional aspects of the Instrumentation and Debugging Framework exploit Truffle's fundamental execution model: AST interpretation implemented in a high-level programming language (Java). Runtime events of interest to tools, for example execution of a statement, correspond to transfer of interpreter control from one AST node (parent) to another (child) and back. The framework captures events by inserting a *wrapper node* between parent and child that acts as a *proxy* with respect to guest language semantics. The wrapper also reports two execution events: one just before the child

MoreVMs'17 April 3, 2017, Brussels, Belgium

executes (where a debugger might suspend execution) and another just after (where a tool might trace value propagation).

Clients of the framework can install any number of *event listeners*. Each listener receives event notifications from (possibly multiple) locations specified by a combination of source location and symbolic *tags* on nodes (provided by language implementors) that identify useful AST sites such as statements and expressions. Event notification provides access to execution state (e.g. current stack) via Java APIs that are shared by all Truffle language implementations and are thus largely *language-agnostic*. Clients may also provide fragments of guest language code to be evaluated in the lexical context of event sites, for example as breakpoint conditions.

Performance aspects of the Instrumentation and Debugging Framework exploit Truffle's fundamental optimization model by implementing as much framework code as possible as nodes embedded in running ASTs. Low-level instrumentation support is written to be indistinguishable from the executing program from the perspective of dynamic optimization. Instrumentation branches that have no effect, for example disabled breakpoints, compile away completely in the fast path. Instrumentation (or client) code that cannot be supported in the fast path triggers *deoptimization*, which is an essential capability of the platform's optimization strategy. Memory footprint is minimized by making all instrumentation-related AST modifications lazy, carried out only when execution reaches a site marked as inconsistent with respect to current client requests, again using fundamental features of the platform's optimization model.

4. Applications

Although originally conceived as a support layer for programming tools, Truffle Instrumentation now supports a growing number of platform features.

The Truffle Debugging API is a core service of the Graal platform, dependent on instrumentation services and developed in close collaboration with platform developers. Truffle language implementors are asked to provide a small amount of language-specific information, including how to display certain program elements such as values and class names, and identifying which stack frames and slots are implementation artifacts and should be hidden.

The public API is growing. Support for frame popping and reentry is currently under development, as well as data-collection APIs for coverage and profiling. Two clients internal to the project now depend on the Debugging API: the NetBeans IDE (via a specially developed JPDA adapter) and a REPL-style shell with debugging commands.

Some otherwise difficult language features have been addressed by instrumentation.

- A Ruby programmer may at any time call `set_trace_func`, which requires that a specified block of code be executed dynamically before each statement in the running program. This feature is notorious for confounding performance, but can run fully optimized using Truffle instrumentation.
- The R language includes an interactive shell that must be prepared at any time to turn on *stepping* through specified methods, which is easily addressed using techniques similar to those used in the platform's Debugging API.

Sometimes the framework serendipitously proves helpful to collaborating platform engineers. For example, a very few lines of instrumentation code recently implemented an unanticipated program requirement: the need to timebox programs by terminating execution when a specified amount of time has passed. A *remote agent* for platform management under development is conveniently implemented within the instrumentation framework, where it dynamically provides data gathering, debugging, and other services with extremely low overhead.

A growing number of tools that depend on Truffle instrumentation have been developed by third parties. A PhD dissertation at UC Irvine used a very early version of Truffle instrumentation to build a low-overhead framework for event profiling, applied it to Truffle implementations of Python and Ruby, and performed cross-language comparisons of benchmark implementations (Savrun-Yeniçeri et al. 2015). A masters thesis at the University of Tartu (Estonia) used Truffle instrumentation for three (increasingly general) implementations of dynamic method reloading for Truffle-implemented languages (Pool et al. 2016).

5. Status and future work

We plan to extend the kinds of execution events that can be captured beyond the few mentioned here. Possibilities include other syntactic elements such as expressions, but also events that do not always have syntactic counterparts such as exceptions or object allocations. Work is underway on other platform services, including ones that gather data such as coverage, profiling, and data dependency. Finally, we expect to continue supporting and encouraging experimentation with tools that might otherwise require prohibitively difficult VM modification.

Acknowledgments

The Virtual Machine Research Group at Oracle Labs and the Institute of System Software at the Johannes Kepler University Linz created the language implementation technologies that make Truffle Instrumentation possible.

References

M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, pages 501–524,

Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-73588-7, 978-3-540-73588-5.

U. Hölzle, C. Chambers, and D. Ungar. Debugging optimized code with dynamic deoptimization. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '92, pages 32–43. ACM Press, 1992.

T. o. Pool, A. R. Gregersen, and V. Vojdani. Trufflereloader: A low-overhead language-neutral reloader. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '16, 2016.

G. Savrun-Yeniçeri, M. L. Van De Vanter, P. Larsen, S. Brunthaler, and M. Franz. An efficient and generic event-based profiler framework for dynamic languages. In *Proceedings of the International Conference on the Principles and Practice of Programming in Java*, PPPJ '16, pages 102–112. ACM Press, 2015.

C. Seaton, M. L. Van De Vanter, and M. Haupt. Debugging at full speed. In *Proceedings of the Workshop on Dynamic Languages and Applications*, DYLA '14, pages 2:1–2:13, New York, NY, USA, 2014. ACM Press. ISBN 978-1-4503-2916-3.

M. L. Van De Vanter. Building debuggers and other tools: We can have it all. A Position Paper. In *Proceedings of the Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, ICOOOLPS '15. ACM Press, 2015.

T. Würthinger, C. Wimmer, A. Wöß, L. Stadler, G. Duboscq, C. Humer, G. Richards, D. Simon, and M. Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 187–204, New York, NY, USA, 2013. ACM Press. ISBN 978-1-4503-2472-4.