Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools

Michael L. Van De Vanter[1], **Chris Seaton**[2], **Michael Haupt**[3], **Christian Humer**[4], **and Thomas Würthinger**[5]

1 **Oracle Labs**
`michael.van.de.vanter@oracle.com`
2 **Oracle Labs**
`chris.seaton@oracle.com`
3 `haupt@acm.org`*
4 **Oracle Labs**
`christian.humer@oracle.com`
5 **Oracle Labs**
`thomas.wuerthinger@oracle.com`

Abstract

Software development tools that interact with running programs, for instance debuggers, are presumed to demand difficult tradeoffs among performance, functionality, implementation complexity, and user convenience. A fundamental change in thinking obsoletes that presumption and enables the delivery of effective tools as a forethought, no longer an afterthought.

We have extended the open source multi-language Graal platform with a language-agnostic Instrumentation Framework, including (1) low-level, extremely low-overhead execution event interposition, built directly into the high-performance runtime; (2) shared language-agnostic instrumentation services, requiring minimal per-language specialization; and (3) versatile APIs for constructing many kinds of client tools without modifying the VM.

A new design uses this framework to implement debugging services for arbitrary languages (possibly in combination) with little effort from language implementor. We show that, when optimized, the service has no measurable overhead and generalizes to other kinds of tools.

It is now possible for a client in a production environment, with thread safety, to dynamically insert into an executing program an instrumentation probe that incurs near zero performance cost until actually used to access (or modify) execution state. Other applications include tracing and stepping required by some languages, as well as platform requirements such as the need to timebox script executions. Finally, opening public API access to runtime state encourages advanced tool development and experimentation with much reduced effort.

1998 ACM Subject Classification D.3.4 Programming Language Processors

Keywords and phrases Virtual Machine, Instrumentation, Optimization, Debugging, Tools

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.?

1 Introduction

A frequent complaint from users of new programming languages (or new implementations of old languages) is that runtime-based *tools* (especially debuggers and profilers) typically

* work done while at Oracle Labs

arrive much later, if ever. When tools do arrive, they compromise programmer productivity in many ways, for example:

- Aggressive code optimization obscures bugs and compromises tool functionality in long-running or production environments.
- Compiler extension for tool support increases complexity and decreases tool portability (and thus availability).
- Tool support constrains optimization, which discourages enabling potentially useful tool support in production environments.

Tools that enhance software development productivity are too often treated as an *afterthought* (e.g. JVMPI, JVMTI for Java [18]), but this was not always so. Gabriel reminds us that some of the earliest and most influential programming languages, for example Lisp (1965) [16], Smalltalk (1980) [10], and Self (1989) [1], were actually programming *systems* [9]. Their implementations exhibited little distinction between language and tools. The shift in focus toward compilers as separate artifacts (which optimize utilization of *expensive machines*) and away from programming tools (which optimize utilization of *expensive people*) came at a cost, reflected in the complaints recited above. Van De Vanter argues that working programmers should once again "have it all" [26]: modern languages, high performance, and productive tools as a fundamental property of run-time environments, without compromising either of them.

We believe we can bridge that divide and "deliver it all" by integrating highly flexible language-agnostic instrumentation and tool support deeply into a high-performance multi-language runtime. A system of this generality has multiple stakeholders, each with priorities and requirements that must be addressed:

- *Platform performance engineers* must grant access (both read and write) to execution state without imposing extra implementation complexity on the core platform. That access must be always enabled, serve many purposes, have no cost when unused, and incur near-zero overhead (beyond client intervention) in fully optimized code.
- *Language implementors* must enjoy the benefit of working tools (both for eventual customers and for themselves during language development) at the cost of supporting a very small set of language-specific APIs.
- *Tool builders* must be presented with simple, highly flexible APIs for specifying and capturing execution events. During event handling, many aspects of execution state must be available through language-agnostic APIs, with language-specific adjustments available when needed, e.g. for understanding and displaying names, values, etc. It must be possible to inject fragments of program code, for example breakpoint conditions, that will be subject to the same optimizations as surrounding program code. Finally, there must be access to a library of shared services, for example implementing debugger primitives and aggregating runtime information such as coverage.

The *Truffle Instrumentation and Debugging Framework* meets these requirements by extending the open source Graal platform [30]. Graal supports developing high performance programming language implementations that leverage a great deal of common infrastructure. We consider Graal the first truly *polyglot* high performance language implementation platform. Our expanded vision for instrumentation-based tools is to deliver a highly productive, polyglot "Programming Environment out of the box", with minimal per-language cost, for Graal-based language implementations.

The contributions of this paper include:

1. A novel design for a language agnostic instrumentation framework, implemented by extension to the Graal platform, that meets all stakeholder requirements and serves many purposes. Those purposes include, in addition to conventional tools already mentioned, tool-related functionalities that are expected to be built into the implementations of some languages. The core platform also benefits from this extension, for example trivially making it possible to timebox program execution and support sampling-based analyses.
2. New APIs in the Graal platform that implement those requirements. Most notably, it is now possible for a client in a production environment, with thread safety, to dynamically insert into an executing program an instrumentation *probe* that incurs near zero performance cost until actually used to access (or modify) execution state. A probe can also be used to inject code fragments (e. g. breakpoint conditions) subject to full optimization as if the original source had been rewritten. It is also possible for clients, using Truffle APIs, to customize compilation of their code when needed.
3. Case studies showing the effectiveness and practicality of these extensions for many purposes. They now support features required in some specific language execution environments, some platform features such as timeboxing execution, measurement services in development such as code coverage and profiling, and an always-on "debugging service" that we call the Debugging API.

The Debugging API is the most developed client of Truffle instrumentation. It provides debugging services for the NetBeans IDE [19], the platform's own REPL-style shell, and other experimental systems. It also as serves as motivation and a running example throughout the remainder of this paper.

Debugging API services operate uniformly across all Truffle implemented languages, even when are used together via language interoperation features of the Graal platform. Execution can be halted, either at first possibility or at breakpoints. Breakpoints, specified by source lines or by text regions, can be configured with *conditions*, as *one shots*, or with *ignore counts* and can be dynamically enabled/disabled. Access to suspended execution state includes location and cause of suspension, stack frames, local variables in any frame, among others. Clients can evaluate code fragments in the context of any stack frame and resume execution as either *Continue*, *Step In*, *Step Over*, *Step Out*, or *Kill*.

Section 2 begins with background on the Graal platform's essential features, followed in Section 3 by a summary of the key insights and design ideas that make the Instrumentation and Debugging Framework possible. Section 4 describes how the framework provides access to dynamically specified execution events, together with a performance evaluation that demonstrates minimal run-time overhead. Section 5 describes the per-language support that implementors must provide to support the tools built so far, together with a summary showing that the burden is quite low relative to the traditional cost of providing tools. Section 6 describes APIs that allow many kinds of tools to be built without additional modification to the runtime. Section 7 evaluates the overall suitability of the Instrumentation and Debugging Framework for its intended goals by reviewing case studies where it is used. Section 8 reviews significant related work, section 9 summarizes the current and future outlook for the project, followed by conclusions in Section 10.

2 Background: the Truffle-Graal platform

The Graal project began as an advanced Java-in-Java compiler developed for the Maxine Research VM [28]. When paired with the Truffle library for implementing programming languages as self-optimizing Abstract Syntax Tree (AST) interpreters [31], Graal became a productive *platform* for creating very high performance programming language implementations. A core of reusable host services greatly simplify *guest language* implementations, including dynamic compilation, automatic memory management, threads, synchronization primitives, and a well-defined memory model. Truffle-implemented languages currently include Javascript, R [25], Ruby [23], and Python [27], among others. Figure 1 summarizes the structure of a deployed guest language program.

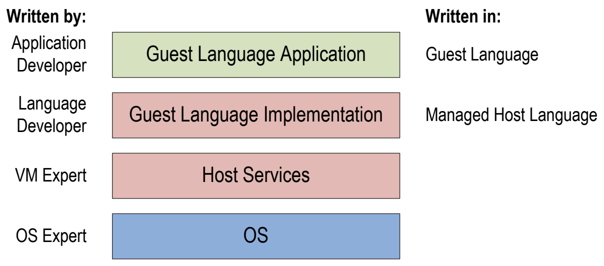

Figure 1 System structure of a guest language implementation utilizing host services to build a high-performance VM for the guest language.

A guest language implementation begins with an *AST interpreter*, a straightforward but traditionally poor-performing technique. *Truffle* ASTs replace optimize specific nodes dynamically by replacement with more specialized versions. This can be done safely, even in multi-threaded execution environments. For example, the newly created Truffle AST in Figure 2 is populated by uninitialized ("U") nodes. Nodes transition during execution by replacement to type-specific nodes, such as the ones exclusively for Integer ("I" in the figure). Each specialization is accompanied by a *guard* that verifies on each execution the validity of the specialization. A guard failure transitions a specialized node to a more generic version ("G") that handles all possible cases. Truffle ASTs are required to stabilize after a finite number of node replacements, as suggested by the permissible "Node Transitions" inset in Figure 2.

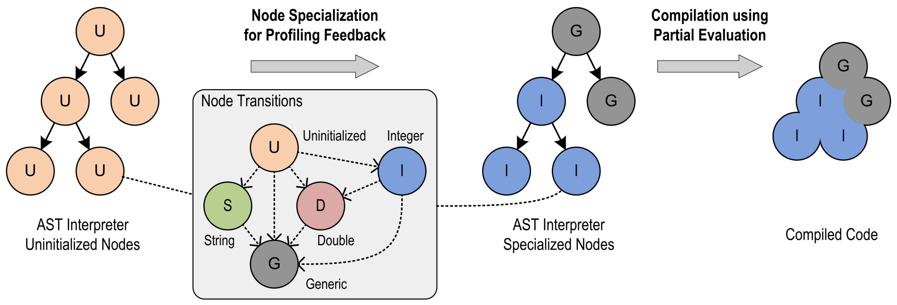
U
U
U
U
U
AST Interpreter Uninitialized Nodes
Node Specialization for Profiling Feedback
Node Transitions
Uninitialized
Integer
U
I
S
D
String
Double
G
Generic
G
I
G
I
I
AST Interpreter Specialized Nodes
Compilation using Partial Evaluation
G
I
G
I
I
Compiled Code

Figure 2 Truffle ASTs speculate and optimize ...

Whenever an AST stabilizes, Graal *dynamic optimizes* [3] its interpreter, using *partial evaluation* [8] to produce highly optimized and specialized machine code. Guard failure in optimized machine code triggers dynamic *deoptimization* [13], transferring execution back to the interpreter without loss of execution state, as shown in Figure 3. Dynamic deoptimization frees the compiler to apply speculative optimizations [6] more aggressively, while avoiding compilation overhead for not-yet-seen or slow-path cases.

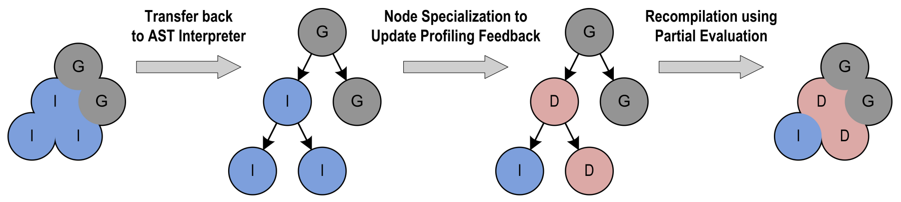
Transfer back to AST Interpreter
Node Specialization to Update Profiling Feedback
Recompilation using Partial Evaluation
G
I
D

Figure 3 … and transfer to interpreter and reoptimize

Figure 4 summarizes the components of the Graal platform. A recent Java extension[1] allows hosting Truffle-Graal on an unmodified JVM.

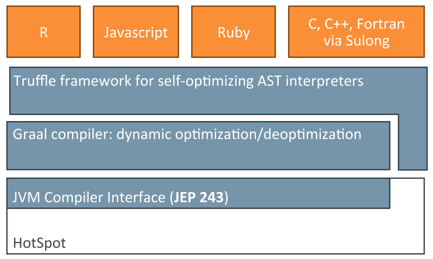
R
Javascript
Ruby
C, C++, Fortran via Sulong
Truffle framework for self-optimizing AST interpreters
Graal compiler: dynamic optimization/deoptimization
JVM Compiler Interface (JEP 243)
HotSpot

Figure 4 Graal technology stack

3 Truffle Instrumentation design overview

The goal of the Truffle Instrumentation and Debugging Framework is to significantly simplify the construction of tools needing dynamic access to execution state in a very high performance runtime environment. The most difficult challenge is to do this without sacrificing performance. Inspiration for a new approach to this challenge comes from two sources: Self [1] and Aspect Oriented Programming (AOP) [14].

Dynamic optimization, pioneered by Self, confounded traditional approaches to debugging. Self debugging was eventually enabled through development of the ability to *deoptimize* dynamically without loss of execution state [13]. Self's breakpoint strategy became: deoptimize the method and rewrite its internal representation to include a debugger call. The runtime may eventually reoptimize, and breakpoint removal is just another rewrite.

AOP is a family of metaprogramming systems that were performance-limited in practice by lack of support for efficient manipulation of runtime state. A solution was developed in the form of a generalized *interposition model*[2] for language VMs [12].

The Graal platform already provides many of the otherwise challenging low-level services needed to support optimized instrumentation.

- Safe AST rewriting of executing methods can be used to interpose via dynamic insertion of instrumentation nodes.
- Deoptimization can be invoked by instrumentation when needed.

[1] `http://openjdk.java.net/jeps/243`
[2] The only implementation of this model was an experiment that demonstrated its general applicability to various AOP models without concern for performance [22].

- Execution state (when deoptimized) is completely accessible as structures in the Java-implemented interpreter, for example stacks, frames, and frame slots.
- AST node executions can be interpreted as instrumentation events, using highly optimized platform abstractions and implementation techniques.

Meeting the varied and sometimes conflicting goals for Truffle Instrumentation depends on a clear separation of concerns into three areas with very different priorities, strategies, and measures of success.

First, execution event reporting must have near-zero overhead. This can only be accomplished by building interposition directly into the lowest levels of the Graal platform and by making performance the absolute priority. Section 4 presents more details, including performance measurements that demonstrate near-zero cost for the instrumentation machinery itself.

Second, the supporting framework for clients (including some internal to the platform) to access and interpret execution events must require minimal support from each language implementation. Section 5 describes the language-specific adaptations that have been added so far to support clients, for example the Debugging API. A summary of instrumentation-imposed requirements for language implementors shows that the burden is modest.

Finally, a client-facing API must provides convenient access to these language-agnostic services, designed to make construction of many different kinds of tools possible. Among many kinds of flexibility, it must be possible for multiple clients of instrumentation be simultaneously active without mutual interference. Section 6 presents more details. A summary of clients built using the framework, several by independent parties, demonstrates the effectiveness of the client API.

Figure 5 summarizes extensions to the Graal platform that implement these ideas.

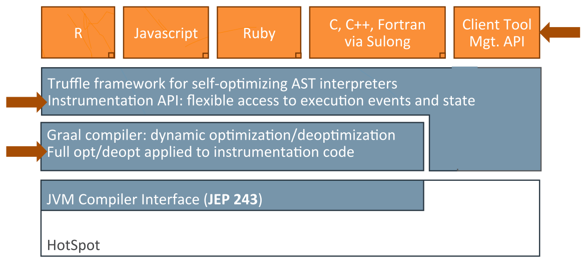
R
Javascript
Ruby
C, C++, Fortran via Sulong
Client Tool Mgt. API
Truffle framework for self-optimizing AST interpreters
Instrumentation API: flexible access to execution events and state
Graal compiler: dynamic optimization/deoptimization
Full opt/deopt applied to instrumentation code
JVM Compiler Interface (JEP 243)
HotSpot

Figure 5 Graal extensions for instrumentation (arrows)

4 Low overhead access to runtime state

This section describes the implementation of low-level platform extensions needed to support the Truffle Instrumentation and Debugging Framework: the dynamic capture and reporting of execution events with access (both reading and writing) to execution state. Subsequent sections describe client APIs and use cases, from the perspective of language implementors (Section 5) and tool builders (Section 6).

Minimal overhead is the highest priority, achieved using three pervasive strategies. First, keep functionality simple, leaving clients to make their own performance tradeoffs. Second, implement functionality using Truffle nodes for maximum optimization. Finally, defer extra node allocations and deoptimizations as long as possible.

The explanation is organized in a bottom-up fashion, treating these topics in turn:

- Enabling instrumentation at a program location by dynamically inserting a *probe*.
- Routing events from a probed location to an interested client by creating a *subscription*.
- Specifying program locations for a subscription by creating a *query*.
- Patching a guest language program at a probed location via *code injection*

The section concludes with a performance evaluation that demonstrates the low overhead achieved by these performance-critical parts of the framework.

4.1 Probes

Truffle Instrumentation takes place at specific AST nodes that have language-provided metadata associated to support instrumentation (more about this in Section 4.3). The framework prepares a node for instrumentation by inserting two additional nodes, as shown in Figure 6.

- A language-provided *wrapper node* that acts as a proxy for its child and reports events; and
- A language-agnostic *probe node* that propagates events to clients.

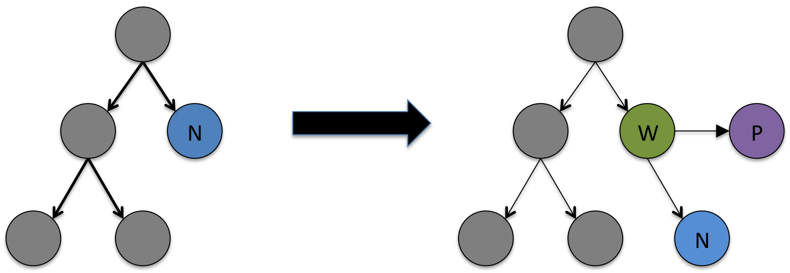

Figure 6 Probing a node

The wrapper reports an event to the probe node just before the child executes and another event just after. Listing 1 summarizes those event signatures, discussed in more detail in Section 6.

```
void onEnter(EventContext, Frame)
void onReturnValue(EventContext, Frame, Object)
void onReturnExceptional(EventContext, Frame, Throwable)
```

Listing 1 Event signature

Truffle AST's thread safety ensures that probing and un-probing can be dynamic and lazy in order to minimize memory footprint.

Truffle node replacement implicitly deoptimizes any compilations that depend on its methods. However, the platform will eventually reoptimize a modified AST, including instrumentation nodes. Optimization eliminates event propagation code that ends up doing nothing. For example, the newly inserted probe in Figure 6 has no clients and incurs zero time overhead when fully optimized.

4.2 Subscriptions

Truffle Instrumentation connects an interested client to a probed node by inserting an additional node to manage the subscription. Figure 7 shows a probed node with an attached

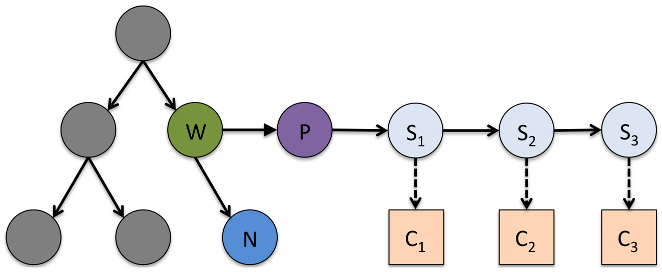

Figure 7 A probed node with three chained subscriptions

"chain" of three subscription nodes, one for each of three independent clients. When a subscription node receives an event from its parent, it propagates it first with an ordiary Java method call to its client and then to its child (successor in the chain) before returning.

As with wrappers and probes, subscription nodes can be inserted and removed safely, at the cost of deoptimization. Event propagation methods will be aggressively optimized, possibly eliminated if clients do nothing.

4.3 Queries

Each Truffle subscription applies to a set of program locations specified by a *query*. These specify criteria such as particular sources, source kinds, and line numbers, for example "`line 42 in mysource.js`". A stepping debugger or a coverage tool might use the query "`every statement`". Section 6 describes the API for queries. Language implementations "markup" AST nodes with instrumentation meta-data needed for matching program locations: precise source attribution plus symbolic tags such as "statement" (more detail in Section 5.1).

A new subscription is represented by a *binding* that dynamically maintains its set of matching nodes, updated for example when new ASTs appear. Moreover, subscriptions can appear and disappear arbitrarily. The implementation challenge is to maintain subscription nodes, as shown in Figure 7, in the presence of changing subscriptions and a changing collection of ASTs.

A brute force implementation walks every AST whenever the binding set changes. It also reviews every binding whenever a new AST is created. Wrapper, probe, and subscription nodes are added and removed as needed.

A first optimization relies on the static fact that the source code specified for an AST root node subsumes the source for all children. It can often be determined that a query with source specifications could not match any nodes in an AST by checking the root (pseudocode `rootCheck` in Listing 2);

The next optimization relies on the dynamic fact that probed nodes are unlikely in many situations to be executed before another binding set change, if ever. The optimized strategy is to *invalidate* any probe[3] whose subscription set is out of date, an inexpensive operation that does not modify the AST. Probes check validity before routing every event, also an inexpensive operation in fully optimized code. Only when a validity check fails will dependent code be deoptimized so that the subscription chain can be updated (and validity reset) before continuing (`checkSubscriptions` in Listing 2).

[3] Each probe node uses a Truffle `Assumption` object to instruct Graal that the chain of subscription nodes (Figure 7) should be treated as `final`. The compiler inserts a very efficient guard in optimized machine code that does not deoptimize anything until a call to `Assumption.isValid()` fails.

```
def rootCheck(AST, binding)
    // could binding match any source locations in AST?

def addAST(AST)
    for all bindings
        if rootCheck(AST, binding)
            for all nodes in AST
                if match(node, binding)
                    probe = probe(node)
                    invalidate(probe)

def addBinding(binding)
    for all ASTs
        if rootCheck(AST, binding)
            for all nodes in AST
                if match(node, binding)
                    probe = getProbe(node)
                    if probe == null
                        probe = probe(node)
                    invalidate(probe)

def removeBinding(binding)
    for all ASTs
        if rootCheck(AST, binding)
            for all nodes in AST
                if match(node, binding)
                    assert probed(node)
                    invalidate(probe)

def checkSubscriptions(probe)
    if invalid(probe)
        remove all subscription nodes
        node = getNode(probe)
        for all bindings
            if match(node, binding)
                addSubscription(probe, binding)
        if number of subscriptions > 0
            validate(probe)
        else
            remove(probe)
```

Listing 2 Subscription maintenance pseudocode

4.4 Code injection

The type of subscription described in Section 4.2 propagates event notification to clients via ordinary Java method calls. Clients are given access to execution state in the form of Java APIs (described in Section 6) for implementation objects such as stacks, frames, and slots. This access is essentially *outside* the guest language execution context.

A second form of subscription, *code injection*, allows clients to provide fragments of guest language code to be executed *inside* the guest language environment, as shown in Figure 8. This kind of subscription propagates an event by simply executing, in the context of the probed node, a client-provided AST fragment. Injected code becomes part of the AST and can be fully optimized together with the surrounding code. Neither return values nor exceptions are allowed to affect guest language execution.

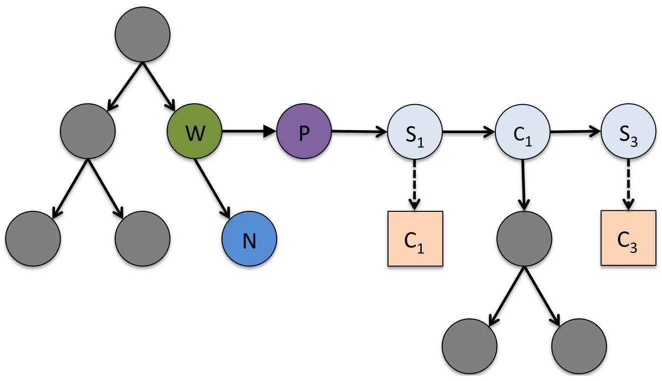

Figure 8 A probed node with injected code

Important use cases for code injection include optimizeable breakpoint conditions and tracing code, both of which are evaluated for performance in the following section and discussed in later sections.

4.5 Evaluation: runtime overhead

We evaluated performance characteristics of the Instrumentation and Debugging Framework with two use cases in the Ruby programming language [7]: implementing `set_trace_func`, a standard feature of the language, and building a simple debugger that sets breakpoints.

Our experimental system is a Sun X4-2 server running two Intel Xeon E5-2660 Ivybridge CPUs with 8 cores and 16 hardware threads each at 2.20GHz, 256 GB of RAM, and running Oracle Linux Server 6.5. We used Graal `05845`, based on OpenJDK 1.8.0_111 with JVMCI 0.23.

We compared TruffleRuby, the Graal platform implementation, against JRuby [17], an implementation using conventional JVM technology. Both implementations live in the same repository at commit `379e8`.

We used a simple Mandelbrot program from the Computer Language Benchmarks Game.

In JRuby `set_trace_func` is disabled by default, so we first measured baseline performance with tracing disabled. We ran it for four minutes and took the mean average of the iteration times for the last two minutes to very generously allow for warmup. In TruffleRuby `set_trace_func` is always enabled, so tracing was already enabled in that first measurement.

Our first experiment used Ruby's `set_trace_func` feature, which installs a callback to be run each time the interpreter arrives at a new source line. With both implementations in turn, we ran four additional measurements:

- with `set_trace_func` enabled but unused;
- then with an empty callback installed;
- then with the call back replaced by one that increments a global variable; and finally
- with all callbacks removed.

Table 1 shows results with mean average time per iteration shown in seconds (so lower is better) and one standard deviation as an error. We can see that the baseline performance of TruffleRuby is an order of magnitude better than JRuby when tracing is disabled.

Table 1 Performance times for `set_trace_func`, lower is better

	Disabled	Before	Empty	Increment	After
JRuby	0.555 ±0.004	15.928 ±0.062	125.371 ±0.0	338.526 ±0.0	16.707 ±0.047
TruffleRuby	0.044 ±0.001	0.044 ±0.001	0.085 ±0.001	2.096 ±0.006	0.044 ±0.0

Table 2 Performance times Ruby debugging, lower is better

	Disabled	Before	Not-taken	Conditional	After
JRuby	0.555 ±0.004	14.39 ±0.725	37.503 ±0.023	45.368 ±0.03	39.004 ±0.082
TruffleRuby	0.044 ±0.001	0.044 ±0.001	0.044 ±0.0	0.044 ±0.0	0.044 ±0.0

TruffleRuby tracing is always enabled, so there is no impact on performance until a callback is installed. In JRuby, however, enabling tracing disables the just-in-time to bytecode compiler, which does not support tracing.

Installing an empty trace callback reduces performance in JRuby by another order of magnitude on top of the existing slowdown, but only slows TruffleRuby to half speed. A callback that increments a global variable slows both JRuby and TruffleRuby, but in this state TruffleRuby is still 100x faster than JRuby. When the callback is removed entirely TruffleRuby performance returns to the baseline level, but JRuby performance appears to suffer permanently.

An empty trace in TruffleRuby still has an impact because the Ruby logic for calling any block has some overhead, such as checking the mutable class of the block for the correct way to call it. If the trace instrument is modified to not call the trace block then the generated machine code for methods is exactly the same with the instrument installed or not.

In a second experiment, we modified the same Mandelbrot program to start a debugger which we used to insert breakpoints. In JRuby we used the *ruby-debug* module, a Java extension promoted as a faster alternative to historical Ruby debuggers. In TruffleRuby we implemented the same functionality using the Instrumentation and Debugging Framework.

Similar to the previous experiment, we measured performance with debugging disabled, then with debugging enabled but no breakpoints installed. We then installed a breakpoint on a line on a branch that is not taken as the benchmark runs. Next we install a conditional breakpoint on a line that is taken, but for which the condition is never true. Finally, we removed all the breakpoints.

Similar to the previous experiment, results in Table 2 show that simply enabling debugging decimates JRuby performance. Installing a breakpoint that is never reached further reduces performance even though it is not actually in the path of execution. The extra work to test the breakpoint condition adds a further penalty. After all breakpoints are removed performance seems to remain permanently reduced. In TruffleRuby there is no performance impact to enabling debugging, as it is enabled by default. A breakpoint on a line never reached has no impact on performance, and the expression in the conditional breakpoint is inlined into the generated machine code for the method and is not measurable. When all breakpoints are removed, performance remains at the level as if a debugger had never been attached.

These two experiments show that the Instrumentation and Debugging Framework allows us to implement language features and a debugger that can be always enabled with no impact on peak temporal performance, that have very low impact on performance when in use, and revert to no impact on performance when no longer in use.

5 API extensions for language implementors

The Truffle Instrumentation and Debugging Framework, much like the underlying Graal platform, derives much of its advantage from services that are *language-agnostic*: applicable to many languages. Programming is not language-agnostic, however. Productive tools must account for variations among languages. Truffle instrumentation extends the APIs for language implementors by adding requirements for language-specific support.

The platform's language-agnostic instrumentation support has been designed to keep those requirements to an absolute minimum. The intention is to make instrumentation support "cost effective" for language implementors. With minimal extra effort, they will themselves enjoy the benefit of working tools throughout development. The ultimate beneficiaries are eventual end users.

This section describes Truffle instrumentation APIs from the perspective of additional requirements for language implementors. It concludes with a summary of those requirements that demonstrates a much lower level of effort than traditionally needed for per-language tools of this complexity.

5.1 Markup

The Graal platform, including instrumentation, treats all AST nodes very much the same. This is an enormous advantage for execution and optimization, but not for people. Programmers think of code interchangeably as either *source text* or language-specific *program elements* (e.g. statements, expressions, blocks), depending on the mental task at hand, but never as ASTs [2]. Truffle language implementors support human tool users by "marking up" language-specific AST nodes with both textual and structural meta-data needed for the low-level query matching implementation mentioned in Section 4.3.

Source Attribution Every Truffle AST node that represents a syntactic program element provides source information by overriding `Node.getSourceSection`. A `SourceSection` describes precisely the corresponding text location, makes the text available if needed, and supports two-way mapping between AST and code. That supports translating a line number to a node when creating a breakpoint, for example, and to translate a node to a code display when a program halts. Truffle *source attribution* is more thorough than needed for traditional debugger support, which is typically limited to statement line numbers. Clients can access program elements at any level of detail so that, for example, a debugger could easily be configured to step through every operation in a complex expression.

Tags A debugger must decide where (i.e. which node) to halt when stepping. Language implementors supply this information by associating a symbolic *tag* with those nodes by overriding `Node.isTaggedWith(Class<?> tag)` to return *true* for `StatementTag`. Other tags currently in use include `CallTag` and `RootTag`. More tags will be added as the suite of built-in services grows. For example, a recent experiment explored how tagging AST nodes where program values are managed could be used to track dynamic data dependencies.

5.2 Visibility

Clients of the Debugging API can enumerate the stack frames and frame slots using only the Graal platform's language-agnostic abstractions. Language implementations, however, sometimes use such abstractions for purposes that correspond to nothing useful or intelligible to a programmer. Truffle instrumentation introduces the notion of *internal* elements and invites language implementors to identify instances that would be counterproductive to reveal under normal circumstances. Examples of internal elements include:

- *Sources* that contain no code sensible to a programmer. For example implementations of language *builtins*, should normally not appear in debugging sessions (designated during creation by applying `Source.Builder.internal()`).
- *Frames* that correspond to no visible program call. For example Ruby implements most control constructs as method calls, which should not appear in ordinary stack traces (designated by the source associated with the frame being marked as internal). Whether a frame is designated internal currently derives from the marking of its associated source.
- *Frame slots* that hold implementation-related state. For example, certain intermediate values might appear in slots that correspond to no local variable (designated by information in the language-provided `FrameFormat`).

It is important that an *internal* designation be independent of other considerations and easily changed for improved usability. It reflects a *judgement* that programmers would not benefit (and might be mystified) seeing such an element during ordinary interaction. Instrumentation clients are free to ignore the designation, however. For example a special debugger mode for Truffle language implementors might reveal internal elements.

5.3 Presentation

Debuggers and other tools often display program information beyond simple source text, for example names and values. How such information should be appear is often language-specific and sometimes a matter of *convention*: i.e. what programmers *expect*.

As with the *internal* designation, it is important that language-specific *presentation* be independent of other considerations and easily modified to increase usability. Examples of language-specific presentation that might appear in a debugging session include:

- *Method/Procedure Name* for every AST. The language implementation overrides the method `RootNode.getName()` for this purpose only, used most commonly by the debugger in backtrace displays.
- *Language Name* at every node. Every AST root has a `language` property assigned via annotation `@NodeInfo`. This becomes important in polyglot debugging sessions, where a backtrace may contain frames in multiple languages.
- *Local Variable Name* for every member of a frame. This is currently provided by `FrameSlot.getIdentifier()`, but a more flexible source is under development in the language-provided `FrameFormat`.
- *Values*, for example from execution results or in frame slots. Language values are known only as instances of `Object` in Graal platform services. A debugger must display special values (e.g. each language's `null` value) appropriately and use language conventions for numbers, especially floating point. Each language implementation overrides `TruffleLanguage.toString(C context, Object value)` to produce a simple string for any language value.

- *Objects.* More detail can be extracted from values using Truffle's support for *language interoperation* [11]. This is a message-based protocol that allows very low overhead cross-language calls among Truffle-implemented languages, and partial access to *foreign objects.* That support is expressed in a message protocol, for example to deal with primitive values (`Message.IS_BOXED`, `Message.UNBOX`), with object references (`Message.IS_NULL`), and with object fields (`Message.READ`, `Message.WRITE`, `Message.INVOKE`).

5.4 Eval and patch

A more direct form of interaction by debuggers is to evaluate a newly provided fragment of language code as if it were *injected* into a running program. In one case the injection is virtual, where the fragment is executed separately but in the context of a halted program. In the other case the fragment becomes part of the program.

Eval Debuggers, especially in the context of REPL-style interaction, allow programmers to "eval" code fragments in the context of a halted execution and display the result. Ideally this accounts also for the lexical context and works at any frame in the current stack. Language implementations prepared to evaluate program fragments in halted contexts override `TruffleLanguage.evalInContext(String, Node, Frame)`. Not all languages are prepared to do this, of course, but it has been straightforward for dynamic languages that have an `eval` operator.

Patch Debuggers are expected to support breakpoints with dynamically assigned *conditions.* These are expressions in the debugged language that must be evaluated each time execution reaches a breakpoint location. Implementing this feature often degrades performance, both for the expression and surrounding code. Truffle instrumentation provides a solution: a language-agnostic mechanism that injects into a probed location a fully optimizable AST fragment (Section 4.4). Language implementations support injection via override of `TruffleLanguage.parse(Source, Node, String...)`.

5.5 Other requirements

Certain kinds of errors, both syntactic and runtime, are inherently language-specific. The current convention for reporting those errors, a language-provided string wrapped in an exception, has proven inadequate and is being replaced by a more complete report. The new report will also include, among other language-specific diagnostics, a stack trace in the same language-agnostic format used in the debugging API.

Low-level event capture requires any instrumentable node must be prepared to generate an instrumentation wrapper for itself (Section 4.1). It must be language-specific so that, in addition to capturing events, it can act as a proxy for the node it wraps. Node implementations provide wrappers by implementing the interface method `Instrumentable.factory()`, which creates appropriate wrapper instances on demand. Platform machinery is in place to generate wrapper classes automatically in straightforward cases.

5.6 Evaluation: language-specific support requirements

Table 3 summarizes the API requirements for Truffle language implementations to support instrumentation fully. Five are met by existing language requirements without modification. Source attribution needs to be more complete than is typical in runtime environments, and each language's `eval` operator needs to be adapted to the framework API. The remaining requirements are new to the platform and are still evolving.

Table 3 Summary: Language-dependent Instrumentation Support

Existing Language Support – Unmodified
`@NodeInfo.language()`
`TruffleLanguage.parse(Source, Node, String...)`
`FrameFormat`
`FrameSlot.getIdentifier()`
Foreign object interoperation protocol
Existing Language Support – Extended
`Node.getSourceSection()`
`TruffleLanguage.evalInContext(String, Node, Frame)`
New Language Support
`Node.isTaggedWith(Class<?> tag)`
`RootNode.getName()`
`Source.Builder.internal()`
`TruffleLanguage.toString(Context, Object)`
`Instrumentable.factory()`

6 Versatile client API

Building tools that need dynamic access to execution state often requires understanding a great deal about VM internals. In many cases tools cannot be built at all without tool-specific VM modifications. This section summarizes the client API of Truffle Instrumentation and Debugging Framework, designed to lower that burden.

Clients are both internal to the Graal platform (e.g. to support language implementation features or timeboxing) and external (e.g. debuggers, code analyzers, and experiments). A greatly simplified subset of the Debugging API implementation (the Graal platform's "debugging service") serves as a running example.

A good measure of the effectiveness of this API appears in Section 7, which describes the diverse set of clients that currently depend on Truffle instrumentation.

Instruments An instrumentation *client* extends class `Instrument` and is installed by placement on the class path of a Truffle execution environment. Listing 3 contains the skeleton for a simple instrumentation-supported debugging service. The framework activates `Instruments` on demand (method `onCreate()`), when requested by a client, and deactivated (method`onDispose()`) when no clients remain. A newly activated instrument receives the argument `Env` for access to basic features of the current execution environment, for example i/o streams and instrumentation support.

```
@Registration(id = "debug-service")
public final class DebugService extends Instrument {
    @Override
    protected void onCreate(Env env) {
        // Set up debugging support
        }
        @Override
        protected void onDispose(Env env) {
            // Release resources
        }
       public void setBreakpoint(SourceSectionFilter,
                                 ExecutionEventListener) {
           ...
       }
       ...
}
```

Listing 3 Skeleton debugging service

Listing 4 sketches a simple scenario for a client tool. It starts by requesting access to the debugging service from the Truffle execution environment (lines 1-2). It then sets up access to some code by creating a `Source` instance, using a builder that supports many other options (line 3). Lines 4-7, which set a breakpoint and execute the code, will be described in more detail one line at a time.

```
1.  PolyglotEngine engine =...; // A Truffle execution environment
2.  DebugService debugService =
        (DebugService) engine.getInstruments().get("debug-service");
3.  Source mySource =   // Access to program source
        Source.newBuilder("myProgram.xx").build();
4.  SourceSectionFilter location = ...;// Specify location in mySource
5.  ExecutionEventListener callback =...;// Called when program halts
6.  debugService.setBreakpoint(location, callback);
7.  engine.eval(mySource);  // Run program, expect a callback
```

Listing 4 Debugging scenario

Filters Clients request events in general by describing a set of program locations with an instance of `SourceSectionFilter`. This class is the client implementation of instrumentation *queries*, described in Section 4.3. Listing 5 shows how the example client on line 4 describes the location for a breakpoint ("the statement at line 42 in `myprogram.xx`") using some of the options available in a builder.

```
4.  SourceSectionFilter location =   // Specify location in mySource
      SourceSectionFilter.newBuilder()
        .sourceIs(mySource).
        .lineIs(42)
        .tagIs(StandardTags.StatementTag)
        .build();
```

Listing 5 Describe breakpoint location

All builder specifications must be satisfied for the filter to match a program location. In addition to tags (any number can be specified), filters can currently be specified by (any number of) specific source names, by (any number of) languages, by line ranges, by character ranges, and whether a source has been marked as "internal".

A filter can match any number of locations. For example a coverage tool might request an event at every statement execution by creating a filter that only specifies that tag. The builder pattern ensures that additional kinds of specifications can be supported in the future without breaking API compatibility.

Event Listeners The Instrumentation framework delivers event notifications from a probed program location via calls to instances of `ExecutionEventListener`, using the event signatures mentioned in Section 4.1. Listing 6 shows how the example client's callback listener, created on line 5, captures execution state just *before* the specified statement is executed. Program execution continues when the callback method returns.

```
5.  ExecutionEventListener callback =   // Called when program halts
      new ExecutionEventListener() {

        public void onEnter(EventContext context, Frame frame) {
           // Handle breakpoint halt
        }
        public void onReturnValue(EventContext context,
                          Frame frame, Object result) {}
        public void onReturnExceptional(EventContext context,
                          Frame frame, Throwable exception) {}
    };
```

Listing 6 Breakpoint callback receiver

Execution state Event notifications include information about the *context* of the event. The `EventContext` argument appearing in Listing 6 provides the static context, which includes:

- the `Node` instance where the event takes place, which allows access to the whole AST,
- `Source` information for the `Node`, including language, source, text location, and concrete text, and
- tags associated with the `Node`.

The `Frame` argument provides dynamic context, including method locals.

Subscription Returning now to the debugging service implementation first sketched in Listing 3, the revision in Listing 7 shows how it creates a breakpoint in response to the call on scenario line 6. The debugging service calls the instrumentation framework (`attachListener()`) to create a *subscription*, described in Section 4.2. That subscription will notify the client callback whenever execution reaches the specified program location, which may happen during execution of the code taking place via the call on line 7 in the scenario.

```
@Registration(id = "debug-service")
public final class DebugService extends Instrument {
    Instrumenter = instrumenter;
    EventBinding binding;

    @Override
    protected void onCreate(Env env) {
        instrumenter = env.getInstrumenter();
    }

    public void setBreakpoint(SourceSectionFilter location,
                              ExecutionEventListener callback) {
       binding = instrumenter.attachListener(location, callback);
    }
    ...
}
```

Listing 7 Debugging service sets a breakpoint

The resulting `EventBinding` instance is a handle that can be used to cancel the subscription, which otherwise remains active for the lifetime of the instrument. Any number of subscriptions can be created (and disposed) at any time. A filter is immutable and can be shared by many subscriptions; likewise a listener can participate in many subscriptions.

Instrumentation errors Exceptional return from any `ExecutionEventListener` method is treated as an implementation failure by instrumentation clients. The instrumentation framework captures exceptions, reports them out-of-band, and allows guest language execution to continue.

Code injection Breakpoints created in the above scenario are unconditional. Adding conditions requires Truffle instrumentation support for code injection, described in Section 4.4. The general approach is to convert the condition (a guest language textual expression) to an AST fragment and attach it to the program AST where it will be evaluated each time execution arrives. When fully optimized by the platform, the performance effect is equivalent to the source code having been rewritten at each injected location and eventually optimized by the Graal platform. The following paragraphs demonstrate how the debugging service does this.

A Truffle subscription for code injection requires a *factory* to produce ASTs, an implementation of the interface in Listing 8.

```
interface ExecutionEventNodeFactory {
    ExecutionEventNode create(final EventContext context);
}
```

Listing 8 A factory for AST fragment injection

Truffle instrumentation invokes the factory lazily, the first time execution reaches each specified program location. The resulting AST, whose root extends `ExecutionEventNode`, is then patched into the AST as shown earlier in Figure 8, where it will be executed each time program execution reaches the probed node. The factory is invoked lazily so the static context of each location can be considered, for example to bind variables.

Simply patching in the conditional expression would have no effect, however. As noted in Section 4.4, the instrumentation framework ignores return values from injected code. Instead, the debugging service returns a new instance the node class shown in Listing 9.

```
    class CondBreakNode extends ExecutionEventNode {
        @Child Node conditionNode;

        CondBreakNode(Node node) {
            conditionNode = node;
        }

        @Override
        public void onEnter(Frame frame) {
            if ((Boolean) condition.call(frame)) {
                // Handle conditional program halt
            }
        }
    }
```

Listing 9 Conditional breakpoint node

The field `conditionNode` holds the AST fragment that evaluates the conditional expression, which the debugging service creates by delegation to the appropriate language implementation. The `@Child` annotation is one of the Truffle conventions that enables aggressive AST optimizations. For brevity, this listing ignores handling of exceptional or non-boolean returns.

Finally, Listing 10 shows a revised implementation of the example debugging service, this time with a method for creating conditional breakpoints.

```
@Registration(id = "debug-service")
public final class DebugService extends Instrument {
    Instrumenter = instrumenter;
    EventBinding binding;

    @Override
    protected void onCreate(Env env) {
        instrumenter = env.getInstrumenter();
    }

    public void setBreakpoint(SourceSectionFilter location,
                                     final String condition) {
        ExecutionEventNodeFactory factory =
           new ExecutionEventNodeFactory() {
             public ExecutionEventNode create(EventContext context) {
                Node condNode =...; // parse condition in context
                return new CondBreakNode(condNode);
            }
        };
       binding = instrumenter.attachFactory(location, factory);
    }
    ...
}
```

Listing 10 Debugging service with conditional breakpoints

To recapitulate, when a debugging client requests a conditional breakpoint:

- The debugging service creates an `ExecutionEventNodeFactory` that holds the text of the condition;
- Instrumentation creates a new subscription via a call to `attachFactory()`.
- The factory, when called, first creates an AST to evaluate the condition in context (by delegation to the language implementation) and then returns a new `CondBreakNode` that wraps that AST.
- Truffle instrumentation attaches the new `CondBreakNode`, where it is executed immediately and on every subsequent execution.
- Each time the `CondBreakNode` is executed, it evaluates the condition and notifies the debugger if `true`.

Source events Many kinds of events other than execution events at AST nodes are important to tools built upon the Instrumentation and Debugging Framework. For example, the listener in Listing 11 can be used to create a subscription that notifies each time a `Source` is newly loaded into the runtime. That subscription can also be filtered, for example by MIME types.

```
public interface LoadSourceListener {
    void onLoad(LoadSourceEvent event);
}
```

Listing 11 A Listener for Source events

It is also possible to *query* for `Source` instances that have already been loaded, optionally subject to a filter.

Evolution We anticipate that, as the Graal platform continues to evolve, we will add subscription support for more kinds of events. In every case this will result from negotiated tradeoffs among complexity, runtime overhead, and API generality for clients. The API design allows for this growth without breaking backward compatibility.

7 Results

A significant measure of success for any software framework is the number (and diversity) of clients. This section evaluates the versatility of the public Truffle instrumentation APIs by reviewing software that now depends on it.

Although originally conceived as a support layer for programming tools, Truffle instrumentation now supports a growing number of platform features. For example, very few lines of instrumentation code were needed to address an unanticipated program requirement: the need to timebox programs by terminating execution when a specified amount of time has passed.

A novel and growing feature of the execution Graal platform is low overhead *language interoperation* [11], which Truffle instrumentation supports almost transparently.

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 as demonstrated in Section 4.5.
- A feature of the R language is 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.

The Truffle Debugging API is now a core service of the Graal platform. Two clients within the framework depend on it: the NetBeans IDE [19] (via a specially developed JPDA adapter) and a REPL-style shell with debugging commands. Breakpoints are implemented as simple instrumentation event subscriptions, and breakpoint conditions are fully optimizable. Other tools have been prototyped and are expected to be added eventually to a suite of default platform tools: code coverage and profiling.

A growing number of tools that depend on Truffle instrumentation have been developed the third parties, including some that were unknown to the authors of this paper until published. Here are two examples. 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 [21]. A masters thesis at the University of Tartu (Estonia) used Truffle instrumentation for three implementations (of increasing generality) of dynamic method reloading for Truffle-implemented languages [20].

8 Related work

Self [1] also implemented debugging using the abstractions of the underlying system. For example, a breakpoint was a "`self halt`" statement inserted dynamically into a running program. The Truffle Instrumentation and Debugging Framework generalizes that strategy to many languages, by extending Truffle ASTs with an interposition model, and to many kinds of concurrently non-interfering tools through a generalized API.

Self also pioneered full service debugging in dynamically optimized code. The compiler was extended to store just enough information to enable *deoptimization* when needed, restoring full access to execution state without loss of information [13]. Deoptimization is essential to Graal's ability to speculate [5], and Truffle instrumentation relies upon it as well.

Tool access to Java execution state is often implemented by rewriting bytecode. Early work in dynamic bytecode modification acknowledged its potential application to tools [4]. In the context of the Graal platform, however, bytecode level support would be at the wrong level of semantic abstraction. Truffle Instrumentation clients are written in Java, the same level as the implementation (host) language, and events are expressed as node execution, which maps naturally to guest language semantics. Finally, Truffle Instrumentation's support for multiple, non-interfering clients would be awkward at best to support via bytecode rewriting.

The Maxine Inspector is a dedicated debugger and heap inspector for the metacircular Maxine Research VM [28]. The Inspector is also built using abstractions of the VM implementation, with which it shares a considerable amount of code. This approach provided excellent functionality but does not generalize past debugging the VM itself.

There have been attempts to automate the delivery of debuggers and other tools in language-agnostic frameworks. Some impose important constraints on language implementation, for example the framework for debugging proposed by Wu et al. [29]. Others lack the performance advantages of close runtime integration, for example debuggers generated by the Spoofax language workbench [15].

Observing execution events in managed runtime environments generally requires intrusive VM modifications, for example *meta-programming* infrastructure for Aspect Oriented Programming. Those modifications typically target specific AOP approaches as narrowly as possible in order to minimize performance cost and complexity. This limitation was addressed by an abstract model for built-in VM support based on highly flexible *interposition* [12]. Truffle instrumentation also uses interposition, realized in the context of Truffle-Graal by dynamic injection of nodes into ASTs under interpretation.

An early experiment with the Graal platform demonstrated the potential of node-based interposition, prototyped by a simple in-language debugger for Ruby [24].

9 Status and future work

We plan to extend the kinds of execution events that can be captured beyond the few mentioned in this paper. Possibilities include other syntactic elements such as expressions, but also events that do not always have syntactic counterparts such as exceptions and object allocation. New possibilities for the Debugging API include richer access to language-specific variations in use of stack frames, as well as the ability to pop frames and reenter execution. 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.

10 Conclusions

We have shown that the Graal platform for creating high performance language implementations can also deliver an open-ended suite of high performance, very low overhead developer tools for those languages with minimal extra per-language effort. Three platform extensions, part of the Truffle Instrumentation and Debugging Framework, make this possible. First, a low level, flexible, and efficient event reporting mechanism provides access to execution state suitable for many purposes. Second, extensions to the platform's language implementation API provide just enough per-language adaptation to allow the creation of a general purpose language-agnostic framework for building tools. That framework, together with tools such as debuggers that use it, works for all platform-supported languages, either singly or when used in combination via the platform's language interoperation features. Finally, a versatile new client API simplifies greatly the construction of tools that need some kind of dynamic access to language execution state. That access is provided by built-in support that is itself subject to the aggressive optimization proved by the underlying Graal platform.

We have yet to find any reason why Truffle instrumentation should not be always enabled, as well services that use it such as the Debugging API.

Acknowledgements. We are 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. Mario Wolczko and David Leibs of Oracle Labs provided foundational insights and encouragement throughout. Martin Entlicher of Oracle's NetBeans team integrated the IDE with the Truffle Debugging API and provided helpful feedback and assistance. Jaroslav Tulach facilitated our NetBeans collaboration and provided ongoing API guidance. David Piorkowsky added a valuable human factors touch to the Instrumentation API design during a 2014 internship. Tom Rodriguez provided helpful insight about JVMTI. Sarah Mount, Yuval Peduel, and Laurence Tratt provided valuable feedback on drafts of this paper.

References

1 C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '89, pages 49–70. ACM Press, 1989.

2 Françoise Détienne and Frank Bott. *Software Design–Cognitive Aspect.* Springer Science & Business Media, 2002.

3 L P Deutsch and A M Schiffman. Efficient Implementation of the Smalltalk-80 System, 1984.

4 Mikhail Dmitriev. Application of the hotswap technology to advanced profiling. In *Proceedings of the European Conference on Object-Oriented Programming*, volume 2, 2002.

5 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, pages 187–193, New York, NY, USA, 2014. ACM Press.

6 Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a

dynamic compiler. In *VMIL '13: Proceedings of the 7th ACM workshop on Virtual machines and intermediate languages*, 2013.

7 David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, Inc., 2008.

8 Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12(4):381–391, 1999.

9 Richard P Gabriel. The structure of a programming language revolution. In *Proceedings of the ACM international symposium on New ideas, new paradigms, and reflections on programming and software*, Onward! '12, pages 195–214. ACM Press, 2012.

10 Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

11 Matthias Grimmer. High-performance language interoperability in multi-language runtimes. In *Proceedings of the Companion Publication of the 2014 ACM SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '14, pages 17–19, New York, NY, USA, 2014. ACM Press.

12 Michael Haupt and Hans 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.

13 Urs Hölzle, Craig Chambers, and David 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.

14 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242. Springer, 1997.

15 Ricky T. Lindeman, Lennart C.L. Kats, and Eelco Visser. Declaratively defining domain-specific language debuggers. In *Proceedings of the 10th ACM International Conference on Generative Programming and Component Engineering*, GPCE '11, pages 127–136, New York, NY, USA, 2011. ACM Press.

16 John McCarthy. *LISP 1.5 programmer's manual*. MIT press, 1965.

17 Charles Nutter, Thomas Enebo, et al. *JRuby: The Ruby Programming Language on the JVM*, 2016. URL: `http://jruby.org`.

18 Kelly O'Hair. *The JVMPI Transition to JVMTI*, 2004. URL: `http://www.oracle.com/technetwork/articles/java/jvmpitransition-138768.html`.

19 Oracle. *Welcome to NetBeans*, 2016. URL: `http://netbeans.org`.

20 Tõnis Pool, Allan Raundahl Gregersen, and Vesal 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.

21 Gülfem Savrun-Yeniçeri, Michael L Van De Vanter, Per Larsen, Stefan Brunthaler, and Michael 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.

22 Hans Schippers, Michael Haupt, and Robert Hirschfeld. An implementation substrate for languages composing modularized crosscutting concerns. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 1944–1951, New York, NY, USA, 2009. ACM.

23 Chris Seaton. *Specialising Dynamic Techniques for Implementing The Ruby Programming Language*. PhD thesis, University of Manchester, 2015.

24 Chris Seaton, Michael L Van De Vanter, and Michael 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.

25 Lukas Stadler, Adam Welc, Christian Humer, and Mick Jordan. Optimizing R language execution via aggressive speculation. In *Proceedings of the 12th Symposium on Dynamic Languages*, DLS 2016, pages 84–95, New York, NY, USA, 2016. ACM.

26 Michael 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.

27 Christian Wimmer and Stefan Brunthaler. Zippy on truffle: A fast and simple implementation of python. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity*, SPLASH '13, pages 17–18, New York, NY, USA, 2013. ACM.

28 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):30:1–30:24, 2013.

29 Hui Wu, Jeff Gray, and Marjan Mernik. Grammar-driven generation of domain-specific language debuggers. *Software: Practice and Experience*, 38(10):1073, 2008.

30 Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario 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.

31 Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing AST interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages*, DLS '12, pages 73–82, New York, NY, USA, 2012. ACM Press.