GraalSqueak

Toward a Smalltalk-Based Tooling Platform for Polyglot Programming

Fabio Niephaus
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
fabio.niephaus@hpi.uni-potsdam.de

Tim Felgentreff
Oracle Labs
Potsdam, Germany
tim.felgentreff@oracle.com

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany
hirschfeld@hpi.uni-potsdam.de

Abstract

Polyglot programming provides software developers with a broader choice in terms of software libraries and frameworks available for building applications. Previous research and engineering activities have focused on language interoperability and the design and implementation of fast polyglot runtimes.

To make polyglot programming more approachable for developers, novel software development tools are needed that help them build polyglot applications. We believe a suitable prototyping platform helps to more quickly evaluate new ideas for such tools.

In this paper we present GraalSqueak, a Squeak/Smalltalk virtual machine implementation for the GraalVM. We report our experience implementing GraalSqueak, evaluate the performance of the language and the programming environment, and discuss how the system can be used as a tooling platform for polyglot programming.

CCS Concepts • **Software and its engineering → Runtime environments**; **Interpreters**; *Integrated and visual development environments.*

Keywords Squeak/Smalltalk, virtual machines, Truffle, GraalVM, polyglot programming, tools, live development

ACM Reference Format:
Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2019. GraalSqueak: Toward a Smalltalk-Based Tooling Platform for Polyglot Programming. In *Proceedings of the 16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357390.3361024

1 Introduction and Background

Polyglot programming is the practice of writing code in multiple programming languages in the same software project. Therefore, it gives software engineers a much broader choice in terms of software libraries and frameworks they can use for building applications. The widespread use of mechanisms, such as Foreign Function Interfaces (FFIs) and Inter-process Communication (IPC), demonstrates that it is desirable and sometimes even necessary to be able to call out to software written in other languages.

In the last years, polyglot runtime environments have become more and more popular. The .NET framework [25] and GraalVM [29] are among the most mature systems that support the execution of multiple programming languages including the interaction between them. Moreover, a lot of research has focused on how languages can be combined and executed efficiently. The .NET framework, for example, compiles all languages to its Common Intermediate Language (CIL) [6], which is then executed on its Common Language Runtime (CLR). GraalVM, on the other hand, requires that all languages are implemented in Truffle [29], its language implementation framework. Languages implemented in this framework generate the same kinds of Abstract Syntax Trees (ASTs) as an intermediate representation, which are then executed and optimized by the Graal Just-in-time (JIT) compiler. Hence, languages can be integrated by mixing ASTs of different languages [27].

Nonetheless, the availability of appropriate technologies enabling fast execution of polyglot code is not sufficient to make polyglot programming practical. We believe software development tools supporting developers in writing polyglot code are just as important. Therefore, the goal of our work is to improve the polyglot programming experience. For this, we want a platform that allows fast prototyping of tools for polyglot programming.

Modern Integrated Development Environments (IDEs) provide comprehensive sets of tools for various languages. These tool sets are powerful on the one hand. On the other hand, they are often hard to extend or to adjust and are therefore not a good choice when it comes to prototyping new tooling ideas. Instead, we decided to use a Smalltalk programming system, which has proven to be a great environment for tooling experiments with its support for live, interactive software development [24]. Its incremental compilation provides short feedback loops as programs including tools can be modified while they are being used. More importantly, Smalltalk is also a programming language and therefore operates on the same level as all other languages supported by a polyglot runtime.

Squeak/Smalltalk is an open Smalltalk system directly derived from the Smalltalk-80 language specification [9]. This specification includes a small and well-defined bytecode set, which can be implemented in a few thousands lines of code [8]. Since we had prior experience implementing a Squeak/Smalltalk virtual machine (VM) in the RPython language implementation framework [7], we decided to implement a VM for Squeak/Smalltalk in Truffle for GraalVM.

On the one hand, such frameworks allow language implementers to use another high-level language and useful components, such as garbage collectors or caching mechanisms, to implement fast VMs for dynamic programming languages. On the other hand, they have to make certain design decisions upfront and hence enforce certain implementation styles and are usually designed to support a specific kind of interpretation model. Truffle, for example, is a framework for implementing AST interpreters. Implementing a bytecode interpreter for Squeak/Smalltalk in it is therefore an interesting challenge on its own.

In this paper, we make the following contributions:

1. Present and evaluate a new VM implementation for Squeak/Smalltalk
2. Show that GraalSqueak provides a solid foundation for interactive polyglot development
3. Share our experience of using Truffle and GraalVM and discuss their limitations with regard to Squeak/Smalltalk

Outline The remainder of this paper is organized as follows. In the next section, we present our approach for GraalSqueak, followed by implementation details in Section 3. Then, in Section 4, we evaluate the performance of GraalSqueak with two different types of benchmarks. Then, in Section 5, we discuss GraalSqueak and elaborate on limitations. Afterwards, we compare GraalSqueak to related systems and discuss other related work in Section 6. Finally, in Section 7, we give an overview of future work and conclude the paper.

2 Approach

GraalSqueak is inspired by many different virtual machines. OpenSmalltalk-VM serves as the reference implementation because it is the default VM for Squeak/Smalltalk and other Smalltalk dialects such as Cuis or Pharo. We previously worked on RSqueak/VM, which is written in the RPython, PyPy's language implementation framework. In many ways, RPython is similar to Truffle [12]. Apart from transferring our knowledge from one to the other frame, it was also necessary to learn and understand many Truffle specifics. SOMns, a Newspeak implementation derived from the Simple Object Machine (SOM) and written in Truffle, gave us ideas for various implementation strategies, for example with regard to method dispatching or efficient data representations. Since the core of SqueakJS, another Squeak/Smalltalk VM written in JavaScript, is maintained in a single file in around 8400 LOC, it was a great resource to understand VM details that are rather hard to find in all other implementations. Although outdated, the same is true for Potato, a Java-based VM for an older version of Squeak/Smalltalk.

2.1 Building the Bytecode Interpreter

After porting the image reader of RSqueak/VM to Java, we started with the implementation of the bytecode interpreter for Squeak/Smalltalk. Since the Truffle framework is designed for building AST interpreters, this turned out to be challenging. Initially, we decided to implement a decompiler that generates Truffle ASTs from Squeak/Smalltalk bytecode. In order to be fully compatible with the specification, however, we shifted away from the decompiler approach and instead transformed bytecode into linear ASTs. In [17], we demonstrated that GraalVM and Truffle are able to provide high run-time performance for this special kind of ASTs, which are also used in other Truffle interpreters such as Sulong [22].

2.2 Supporting the Programming Environment

An interpreter is not sufficient to support Squeak/Smalltalk's programming environment. The VM also needs to support a number of essential primitives as well as plugins in order to draw the user interface (UI) of Squeak/Smalltalk. Therefore, we needed to implement additional primitives, for example for process scheduling or object access. This step also enabled us to interact with Squeak/Smalltalk's compiler, which in turn allowed us to run Squeak/Smalltalk's SUnit test cases. From this point on, we were able to continue the development of GraalSqueak in a test-first development style. Eventually, most kernel tests were passing, so we decided to focus on supporting the BitBlt and Balloon plugins, which are used for drawing. This again turned out to be challenging, because it required windowing and graphics to be used in Truffle — something other language implementations do not have to use. Initially, we followed the approach used in RSqueak/VM for supporting the two rendering plugins: Instead of porting Balloon and BitBlt to Java, we run their Smalltalk sources on the VM level, similar to how the VM simulator [15] runs them in the image. This allowed us to get both plugins working with relatively low effort. However, it took a significant time to warm up the simulation code as further discussed in Section 3.

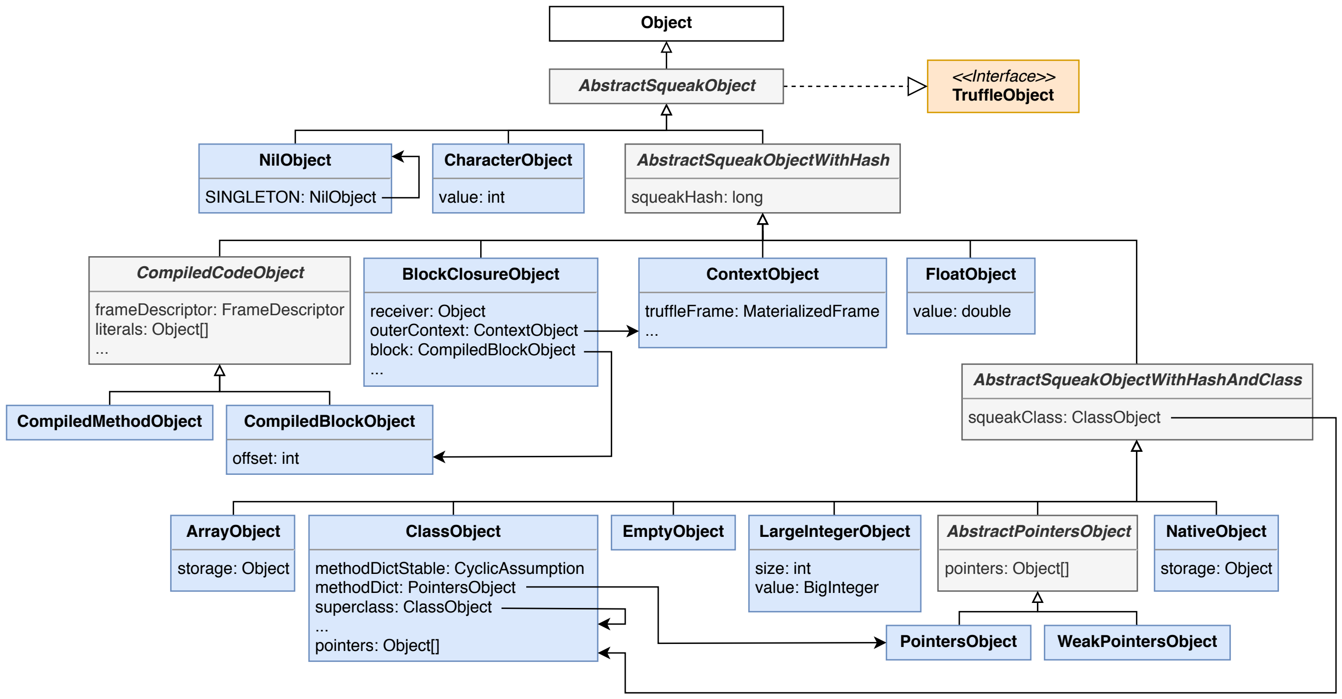


Figure 1. The object model of GraalSqueak. Smalltalk objects mapped to Java primitives have no corresponding implementation class in the diagram.

2.3 Smalltalk Language Features

Up until this point, some language features were mocked in the VM to avoid errors. For a correctly working system, however, it was necessary to support essential languages features correctly. One challenge was to fully yet efficiently support Squeak/Smalltalk's activation records. Activation records are exposed as `Context` objects in Smalltalk and can be used to implement language features such as exception handling, among others. However, Truffle `Frames`, which must be used for managing activation records in the framework, cannot be modified arbitrarily. Although the sender field of context object can be changed, which allows manipulation of the control flow, the caller of a Truffle frame cannot. We encountered a similar situation during the implementation of RSqueak/VM's bytecode interpreter and were able to work around this problem in a similar way: We introduced different representations for `Context` objects, which allows GraalVM to apply its optimizations in common cases [18]. Another challenge was to support the `allInstances` method, which returns all instances for a given behavior or class. The problem here is that this functionality is also not supported by Truffle out-of-the-box and there is not way to instrument GraalVM's garbage collectors to collect instances. Again, we were able to apply a similar implementation strategy used in RSqueak/VM: Similar to a garbage collector, we walk all objects in the heap manually. For this, the `specialObjectsArray` of Squeak/Smalltalk is used as a root from which we start to trace all pointers and collect the objects matching the given class.

3 Implementation

In this section, we highlight noteworthy implementation details and explain which tools we used for the development of GraalSqueak. The sources of GraalSqueak are available on GitHub[1].

3.1 Object Model

An important part of a virtual machine is its object model. This part of a VM not only needs to correctly model the objects of its guest language. Usually, it also needs to make a trade-off between run-time performance and memory consumption. In case of GraalSqueak, an efficient object model is even more important as it must represent Squeak/Smalltalk objects with Java objects. The OpenSmalltalk-VM, on the other hand, uses the object format of the image file and can therefore copy the image file directly into memory.

The most efficient way for representing objects in Truffle is to use Java primitive data types for specific groups of guest language objects. In GraalSqueak, we decided to use Java's `boolean` type for representing Squeak/Smalltalk's `true` and `false` values, `char` for `Character`, `long` for `SmallInteger`, and `double` for `SmallFloat` objects. We explicitly decided against `int` and `float` as we wanted to target 64-bit systems and the

[1] https://github.com/hpi-swa/graalsqueak/

ranges of their `SmallInteger` and `SmallFloat` exceed those of `int` and `float`. But more importantly, with the number of Java classes used for object representation, the number of Truffle specializations and consequently code complexity increases, sometimes dramatically. As an example, if we decided to use `int` for 32-bit `SmallInteger` objects, the node for integer addition would need four specializations (for supporting the addition of an `int` and an `int`, an `int` and a `long`, a `long` and an `int`, as well as a `long` and a `long`) plus appropriate guards.

Moreover, Truffle provides a `DynamicObject` class, which can be used for representing objects that have a variable number of members at run-time. This infrastructure is not only helpful for implementing languages such as JavaScript or Python. It is also able to optimize the representation of such objects with the aid of shapes [26]. However, different workloads and benchmarks have suggested that Squeak/Smalltalk does not benefit from this performance optimization. Most Squeak/Smalltalk objects have only few slots and are therefore small in size. Truffle's `DynamicObjectBasic`, the open source implementation of `DynamicObject`, allocates at least three `long` fields and four `Object` fields. Furthermore, the size of a Squeak/Smalltalk object is set at allocation time and cannot change, so we decided to represent them with Java's `Object[]` type instead.

Figure 1 shows GraalSqueak's object model for all objects that are not represented by a Java primitive type. The base class for all Squeak/Smalltalk objects is `AbstractSqueakObject`, which implements the `TruffleObject` interface as required by the framework. All Squeak/Smalltalk objects provide a hash (see `squeakHash` of `AbstrctSqueakObjectWithHash`). However, certain objects have an intrinsic hash, such as integers, booleans, or any of the following special objects and hence `squeakHash` is unnecessary. The `nil` object is, similar to `true` and `false`, a well-known object and can therefore be represented by a singleton as part of its `NilObject` class. In Squeak/Smalltalk, `Character` objects store code points as an unsigned 30-bit value. Therefore, the `CharacterObject` class is needed to represent characters that exceed the range of Java `char`.

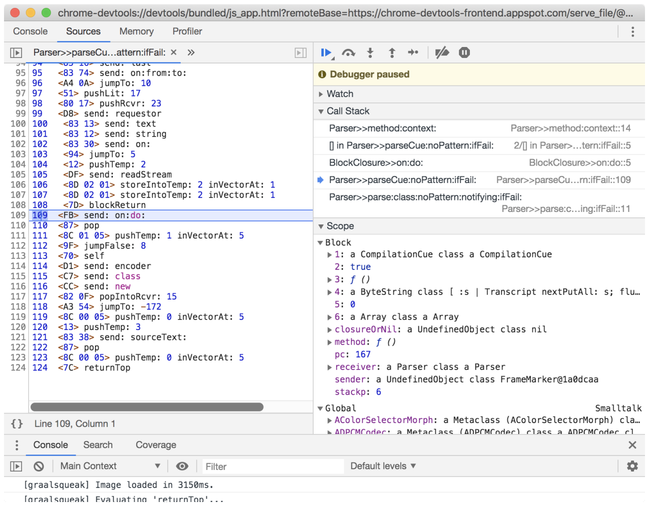
Similar to the `CompiledCode` hierarchy in Squeak/Smalltalk, GraalSqueak uses three classes to represent methods and blocks. Since these objects contain information needed for context objects, Squeak/Smalltalk activation records, they each hold a `FrameDescriptor` which is required for Truffle's activation records infrastructure. `BlockClosureObject` is used for block closures and holds references to its receiver, its block, its outer context, and other objects. If requested in Squeak/Smalltalk code, activation records are allocated on the heap using `ContextObject`, which references a `MaterializedFrame` provided by the Truffle framework. `BoxedFloat64` objects are represented by `FloatObject`. These objects only need a `squeakHash` as well as a `value`.

Finally, `AbstractSqueakObjectWithHashAndClass` is for the group of objects with a dedicated class. `ArrayObject` represents arrays and applies optimizations similar to storage strategies [2]. This improves both memory efficiency as well as performance as homogeneous arrays of primitive data types are maintained without boxing. For Squeak/Smalltalk class objects, GraalSqueak uses `ClassObject`, which maintains different assumptions about the stability of a class. It also maintains the five slots of a `ClassDescription` in separate fields. Moreover, it has a `pointers` array for addition slots used for `Class`, `Metaclass`, and `TraitBehavior` (and its subclasses), all of which are subclasses of `ClassDescription`. `EmptyObject` is used for representing objects without any slots, while `LargeIntegerObject` manages its `value` using a Java `BigInteger` for large integer arithmetic. Objects with instance variables are maintained in instances of `PointersObject` or `WeakPointersObject` depending on whether their references are hard are weak. Similar to `ClassObject`, both store their values in a `pointers` array defined in `AbstractPointersObject`. In addition to the behavior of `PointersObject`, instances of `WeakPointersObject` wrap values in a `WeakReference` to correctly work with Java's garbage collector. Finally, `NativeObject` represents byte, short, word, or double-word objects. A `ByteString`, for example, is stored in `storage` as a `byte[]`. Since the class of such an object can change, its storage must be converted appropriately as well.

For all leaf classes of the object model, there is a set of nodes for accessing corresponding objects in a way compatible with the Graal compiler and partial evaluation. As benchmarks in Section 4 will demonstrate, this object model yields competitive run-time performance.

3.2 Improving Responsiveness

Although Squeak/Smalltalk requires less than 40 different bytecodes to run, it needs far more primitives (300+) to be supported by the VM. We implemented most of these primitives manually by reading through the sources of the OpenSmalltalk-VM and other VM implementations. As mentioned in Section 2, the responsiveness of the programming environment was quite poor due to the simulated BitBlt and Balloon plugins, which are used for drawing. Moreover, warmup was a problem as responsiveness only slowly improved over time. With GraalVM's memory and CPU profiler, we found out that a lot of objects were allocated as part of the simulation plus the Graal compiler needed a significant amount of time to compile the simulation code. Consequently, we decided to port both plugins to Java. For this, we translated the source code of Balloon and BitBlt with the Slang compiler to C and then manually ported the C code to Java. This took us about three days and as a result, both ported plugins added approximately 9K SLOC to GraalSqueak, which had at this point around 20K SLOC in total. Nonetheless, this additional code resulted in much better


chrome-devtools://devtools/bundled/js_app.html?remoteBase=https://chrome-devtools-frontend.appspot.com/serve_file/@...
Console
Sources
Memory
Profiler
Parser>>parseCu...attern:ifFail:
Debugger paused
Watch
Call Stack
Scope
Line 109, Column 1
Console
Search
Coverage
Main Context
Filter
Default levels

```
95   <83 74> send: on:from:to:
96   <A4 0A> jumpTo: 10
97   <51> pushLit: 17
98   <80 17> pushRcvr: 23
99   <D8> send: requestor
100   <83 13> send: text
101   <83 12> send: string
102   <83 30> send: on:
103   <94> jumpTo: 5
104   <12> pushTemp: 2
105   <DF> send: readStream
106   <8D 02 01> storeIntoTemp: 2 inVectorAt: 1
107   <8D 02 01> storeIntoTemp: 2 inVectorAt: 1
108   <7D> blockReturn
109  <FB> send: on:do:
110  <87> pop
111  <8C 01 05> pushTemp: 1 inVectorAt: 5
112  <9F> jumpFalse: 8
113  <70> self
114  <D1> send: encoder
115  <C7> send: class
116  <CC> send: new
117  <82 0F> popIntoRcvr: 15
118  <A3 54> jumpTo: -172
119  <8C 00 05> pushTemp: 0 inVectorAt: 5
120  <13> pushTemp: 3
121  <83 38> send: sourceText:
122  <87> pop
123  <8C 00 05> pushTemp: 0 inVectorAt: 5
124  <7C> returnTop
```

```
Parser>>method:context:                          Parser>>method:context::14
[] in Parser>>parseCue:noPattern:ifFail:         2/[] in Parser>...tern:ifFail::5
BlockClosure>>on:do:                             BlockClosure>>on:do::5
Parser>>parseCue:noPattern:ifFail:               Parser>>parseCu...rn:ifFail::109
Parser>>parse:class:noPattern:notifying:ifFail:
                                                 Parser>>parse:c...ing:ifFail::11
```

```
Block
  1: a CompilationCue class a CompilationCue
  2: true
  3: ƒ ()
  4: a ByteString class [ :s | Transcript nextPutAll: s; flu...
  5: 0
  6: a Array class a Array
  closureOrNil: a UndefinedObject class nil
  method: ƒ ()
  pc: 167
  receiver: a Parser class a Parser
  sender: a UndefinedObject class FrameMarker@1a0dcaa
  stackp: 6
Global                                                     Smalltalk
  AColorSelectorMorph: a Metaclass (AColorSelectorMorph) cla...
```

```
[graalsqueak] Image loaded in 3150ms.
[graalsqueak] Evaluating 'returnTop'...
```

Figure 2. Debugging the parser of Squeak/Smalltalk in Chrome DevTools.

responsiveness of the programming environment as well as an improved warmup time. More importantly, it removes the dependency from the in-image simulation code, which means that standard images, including the Squeak/Smalltalk 5.2 release image, can be opened in GraalSqueak. In Section 4, we discuss how we measured UI performance of GraalSqueak in more detail.

3.3 Tools Used to Develop GraalSqueak

The programming experience is not only important for the users of a language. Since language implementations are rather complex software projects, appropriate tools and a good programming experience are just as important. Since Truffle allowed us to implement a Squeak/Smalltalk VM in Java, we were able to use sophisticated Java IDEs such as Eclipse and IntelliJ IDEA for the development of GraalSqueak. One key feature to mention is the ability to debug a running GraalSqueak instance, which is just as convenient as the custom-built simulation infrastructure of OpenSmalltalk-VM [15]. For us, this debugging capability was especially useful for implementing numerous primitives for Squeak/Smalltalk. While OpenSmalltalk-VM, RSqueak/VM, and other VMs must be recompiled after each change, we were able to adjust the behavior of a primitive in the VM at run-time. Moreover, debugging our Java code also helped to understand and fix misbehavior in the bytecode interpreter.

However, sometimes it was necessary to debug on the language or bytecode level rather than on the level of the language implementation. Before the Squeak/Smalltalk environment and its debugger were supported by GraalSqueak, we were able to use GraalVM’s support for the Chrome Debugging Protocol with relatively low effort. Figure 2 shows a screenshot of the Chrome debugger connected to a GraalSqueak instance. In this particular session, we are debugging through bytecode of Squeak/Smalltalk’s parser. This integration supports various ways of stepping, evaluation of code, as well as inspection of call stack and scopes.

Furthermore, it is necessary to understand how the Graal compiler optimizes code of a guest language. For this, GraalVM provides many different command-line flags that enable printing of various debug and compiler information. In addition, GraalVM also provides the Ideal Graph Visualizer tool. This tool makes it possible to inspect Truffle ASTs of a language, call trees, and Graal compiler graphs for each compilation unit. Figure 3 shows the call graph of `Integer>>benchFib`, a recursive implementation of the Fibonacci algorithm in Squeak/Smalltalk, after the profiling stage. Switching between different stages reveals information on the costs of nodes and the cost model used in the Graal compiler [10], the different characteristics of ASTs, as well as information on method inlining. In case of `Integer>>benchFib`, for example, we can find out that the compiler applied splitting [28] to the method, which caused inlining of two levels of the recursion. Hence, the Ideal Graph Visualizer is an indispensable tool when working on the performance of GraalSqueak.

4 Evaluation

In this section, we evaluate GraalSqueak using two different types of benchmarks. First, we discuss results of several well-established language benchmarks and compare GraalSqueak with other VMs. Then, we assess the performance of the Squeak/Smalltalk programming environment running two different workloads.

4.1 Benchmarking Language Performance

For comparing GraalSqueak with other VMs, we ran the Are We Fast Yet benchmark suite [11] with ReBench[2] and with different configurations. Each configuration runs 250 iterations of each benchmark in a fresh process. The first configuration ran the benchmarks on GraalSqueak using the community edition of GraalVM, the second using the enterprise edition of it. The next two configurations ran them on top of OpenSmalltalk-VM, the state-of-the-art Squeak/Smalltalk VM, and RSqueak/VM. Lastly, we also ran the benchmark suite on SOMns. More information on the different versions used for the benchmarks is listed in Appendix A. We excluded the CD benchmark because it is currently not fully functioning on GraalSqueak due to its highly recursive nature. All configurations and benchmarks were executed on Debian 9 with Linux kernel version 4.9.168-1+deb9u3 and on the same dedicated hardware, a Dell PowerEdge 2950 (CPU: 2.33 GHz Intel Xeon CPU E5410; Memory: 8x4 GB

[2]https://github.com/smarr/ReBench

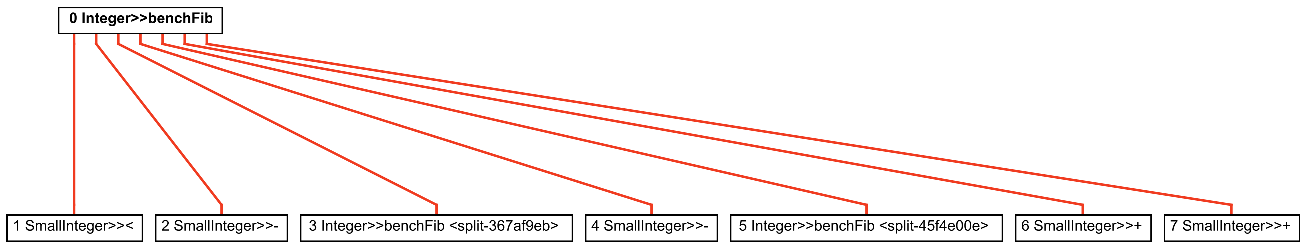


Figure 3. Call tree of `Integer>>benchFib` after profiling as visualized by GraalVM's Ideal Graph Visualizer.

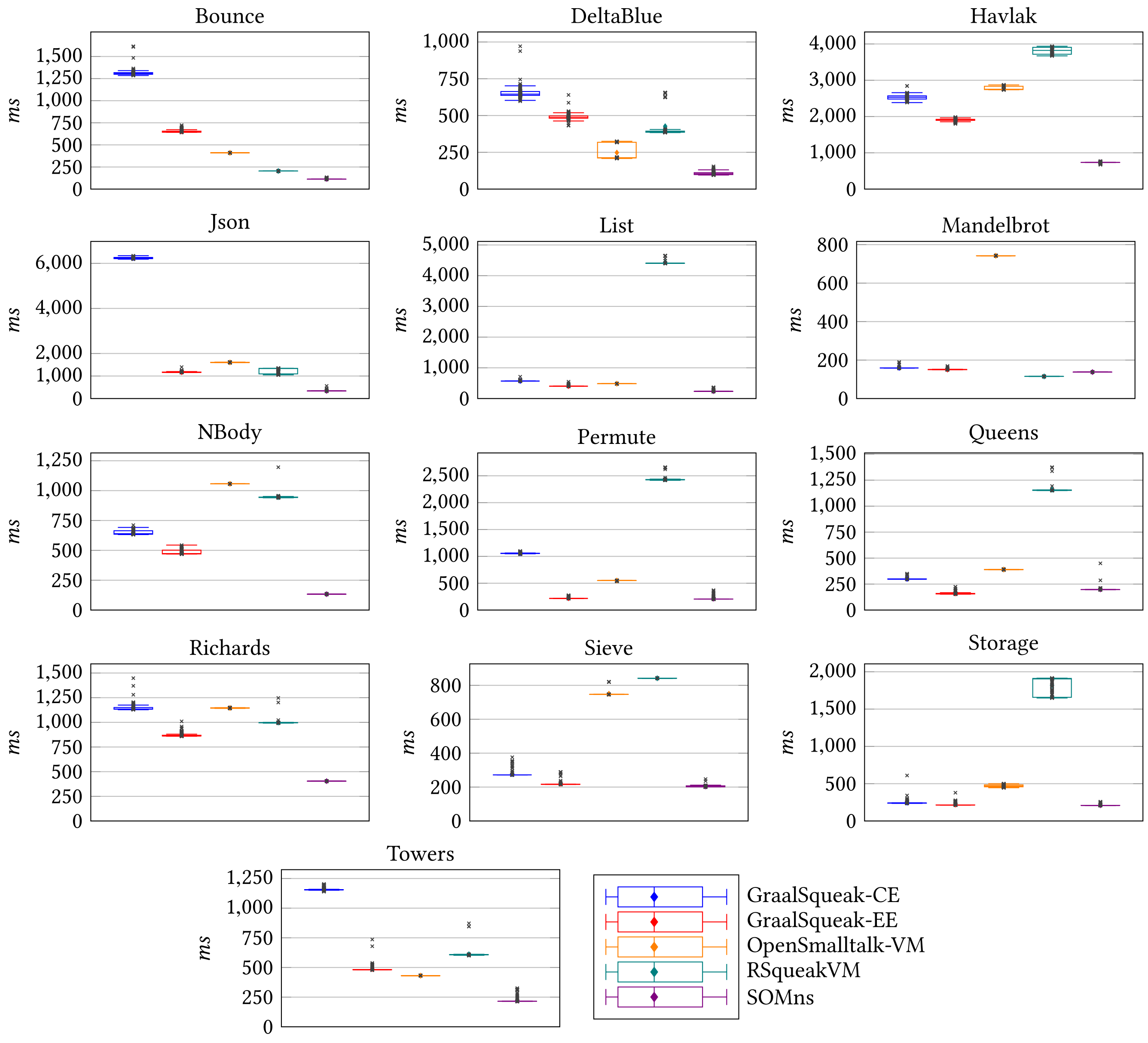


Figure 4. Are We Fast Yet benchmark results in milliseconds per VM (lower is better). Each Tukey boxplot [14] visualizes the quartiles of 200 iterations per VM (after 50 iterations for warmup). The median is highlighted with a symbol. The ends of the whiskers represent the lowest and highest datum within 1.5 IQR of the corresponding quartile.

DDR2-667 MHz SDRAM ECC) with hyper-threading, Intel Turbo Boost, and Intel P-States disabled.

The benchmarks results are shown in Figure 4. From the 250 iterations of each benchmark, we removed the first 50 iterations to remove warmup behavior from the box plots. As Figure 8 in Appendix A shows, GraalSqueak warms up within the first few iterations and all other VMs show similar behavior. Furthermore, we checked that a steady state was reached during each benchmark.

GraalSqueak always performs better on the enterprise edition of GraalVM than on its community edition. This is expected, as the enterprise edition, the commercial version of GraalVM, applies additional optimizations and optimizes more aggressively in general. However, sometimes the difference in performance can be significant, especially when looking at Bounce, Json, and Towers. In the case of Json, the performance on GraalVM EE is more than three times better than on GraalVM CE. Out of all 13 benchmarks, however, GraalSqueak on GraalVM CE is still faster than OpenSmalltalk-VM in seven benchmarks, while GraalSqueak on GraalVM EE is faster in 10 of them. Although SOMns is often much faster than all other VMs, GraalSqueak on GraalVM EE is able to achieve similar performance in six benchmarks. In the Permute and Queens benchmarks, our VM is even slightly faster than SOMns on a custom-built GraalVM CE. RSqueak/VM, on the other hand, is faster than GraalSqueak on GraalVM EE in Bounce, DeltaBlue, and Mandelbrot. Considering this is the first time we compare GraalSqueak with other VMs, it is good to see that it already provides competitive language performance.

4.2 Benchmarking the Programming Environment

The performance of the Squeak/Smalltalk programming environment is just as important as pure language performance. To evaluate this, we prepared two Squeak/Smalltalk images with different workloads and measured the frame rate of the environment. Both of these benchmarks ran on a 13-inch MacBook Pro from 2018 (CPU: 2.7 GHz Intel Core i7; Memory: 16 GB 2133 MHz LPDDR3).

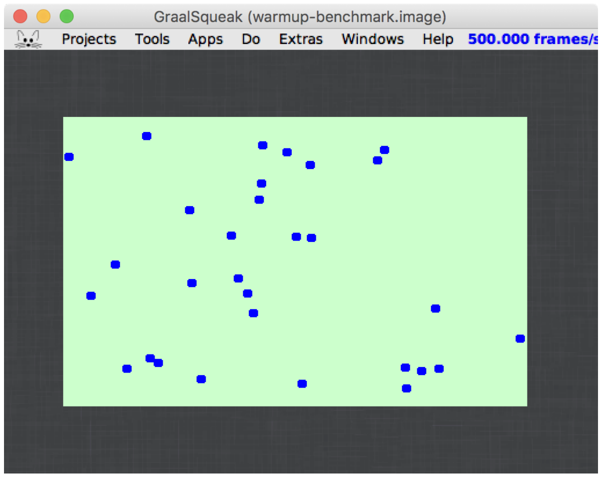
In the first image, we ran the bouncing-atoms simulation that comes with Squeak/Smalltalk to measure warmup and peak performance of the UI. For this, we enabled the high-performance mode in the preferences of the image. Otherwise, the frame rate is throttled to 50 fps. In a next step, we opened a `BouncingAtomsMorph` in the middle of the display as well as a `FrameRateMorph` in the menu bar. Furthermore, we slightly modified the `FrameRateMorph`, so that it prints the determined frame rate and the time to the console. A screenshot of this benchmark image is shown in Figure 5a. Finally, we saved the image and opened it again with GraalSqueak on GraalVM CE and EE as well as with the OpenSmalltalk-VM for three minutes. During that time, we logged the frame rate and ensured that the image does not receive any keyboard or mouse events from the outside.

Figure 5c shows the frame rates of all three VM setups. GraalSqueak is able to outperform OpenSmalltalk-VM on GraalVM CE within the very first second, and on GraalVM EE in the third second. In the first half of the second minute, the frame rate of GraalSqueak stabilizes at 500 fps, which is the maximum the `FrameRateMorph` can measure as the morph's time resolution is limited to 2 ms. Around the same time, the Graal compiler no longer optimizes Smalltalk methods, which suggests that warmup has completed. OpenSmalltalk-VM, on the other hand, reaches 59.88 fps in the first second and then stays at a little less than 60 fps for most of the time. Instead of showing warmup behavior, we see occasional drops in the frame rate down to up to 29.88 fps. We believe this is due to garbage collection. In GraalSqueak, on the other hand, we see random drops in performance during warmup. These drops are much bigger in absolute difference, which makes the simulation look as if it hangs for a subsecond. Furthermore, GraalSqueak running on the community edition of GraalVM warms up slightly faster than when running on its enterprise edition. This is in line with how both flavors are advertised: Compared with GraalVM CE, the enterprise edition performs additional optimizations and optimizes more aggressively in general, which results in longer compilation times, but more efficient code.

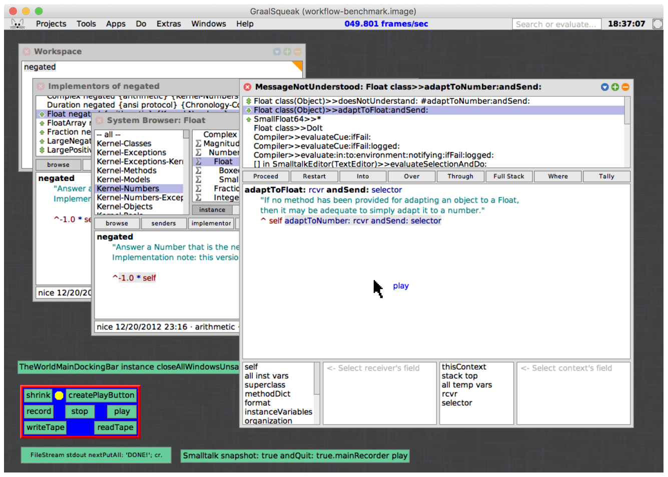
At this point, it is also important to mention that 500 fps means the UI was rendered into the display buffer 500 times per second. The actual window was rendered at a lower frame rate and in a separate Java-level UI thread. Nonetheless, the point of this benchmark was to compare warmup behavior and peak performance of Squeak/Smalltalk's rendering machinery. Under normal circumstances, it is more reasonable to throttle the frame rate, which Squeak/Smalltalk does by default.

The second UI performance benchmark evaluates a more realistic workload. For this, we recorded mouse and keyboard events of a user interaction with development tools using the `EventRecorderMorph` of Squeak/Smalltalk. Figure 5b shows a screenshot of what the image looks like during the interaction: We started with a workspace, then opened the implementors for a method selector. Next, we opened an implementation of such a method in a system browser. We then evaluated code which in turn opened a debugger. Afterwards, we closed all windows and repeated the workflow after waiting for around 20 s, 70 s, and 160 s. These pauses are meant to give GraalSqueak some additional time for JIT compilation. To make the benchmark as reproducible as possible, we prepared another image which automatically replays the recorded input events on startup. Again, we use a `FrameRateMorph` to measure the frame rate over time.

The results are shown in Figure 5d. Although the frame rate is throttle to 50 fps, OpenSmalltalk-VM draws the programming environment at around 46 fps when idling. Similar to the previous benchmark, the state-of-the-art Smalltalk VM

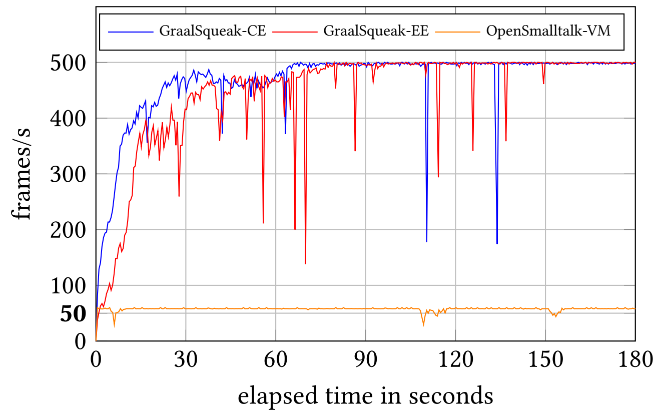


(a) Screenshot of a `BouncingAtomsMorph` running on GraalSqueak.

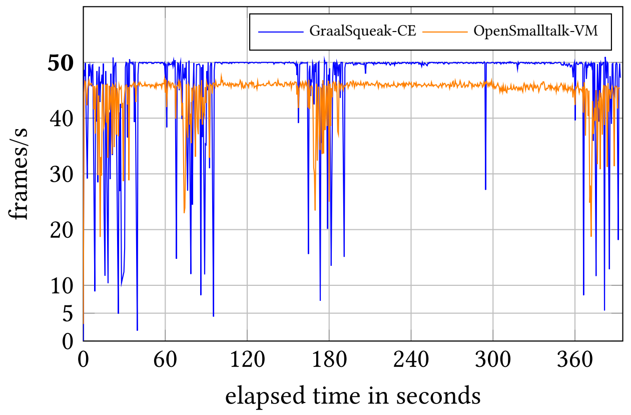


```
negated

Duration negated {ansi protocol} {Chronology-Co
Float negat
FloatArray
Fraction ne
LargeNegat
LargePositiv

-- all --
Kernel-Classes
Kernel-Exceptions
Kernel-Exceptions-Kern
Kernel-Methods
Kernel-Models
Kernel-Numbers
Kernel-Numbers-Excep
Kernel-Objects

Complex
Magnitud
Number
Float
Boxe
Smal
Fractio
Intege

negated
    "Answer a Number that is the ne
    Implementation note: this versio

    ^-1.0 * self

nice 12/20/2012 23:16 · arithmetic

MessageNotUnderstood: Float class>>adaptToNumber:andSend:
Float class(Object)>>doesNotUnderstand: #adaptToNumber:andSend:
Float class(Object)>>adaptToFloat:andSend:
SmallFloat64>>*
Float class>>DoIt
Compiler>>evaluateCue:ifFail:
Compiler>>evaluateCue:ifFail:logged:
Compiler>>evaluate:in:to:environment:notifying:ifFail:logged:
[] in SmalltalkEditor(TextEditor)>>evaluateSelectionAndDo:

adaptToFloat: rcvr andSend: selector
    "If no method has been provided for adapting an object to a Float,
    then it may be adequate to simply adapt it to a number."
    ^ self adaptToNumber: rcvr andSend: selector

play

self
all inst vars
superclass
methodDict
format
instanceVariables
organization

<- Select receiver's field

thisContext
stack top
all temp vars
rcvr
selector

<- Select context's field

TheWorldMainDockingBar instance closeAllWindowsUnsa

FileStream stdout nextPutAll: 'DONE!'; cr.

Smalltalk snapshot: true andQuit: true.mainRecorder play
```

(b) Screenshot of an UI workflow running on GraalSqueak.



(c) Benchmark results of the bouncing-atoms simulation.



(d) Results of the UI workflow benchmark.

Figure 5. Two different UI benchmarks and corresponding benchmark results.

does not show any noticeable warmup behavior. When interacting with different UI elements, the frame rate becomes somewhat unstable. The lowest frame rate measured for OpenSmalltalk-VM in this benchmark was 18.69 fps. However, there is no particular difference with regard to the frame rate between the different iterations of the workflow.

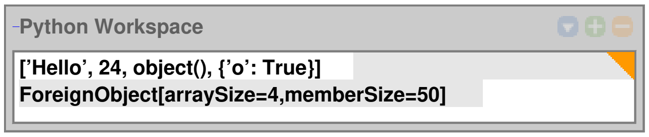
GraalSqueak on GraalVM CE, on the other hand, is in fact able to reach 50 fps. At the same time, the performance cliffs are much more extreme. This manifests in noticeable pauses when interacting with the UI elements of Squeak/Smalltalk. In the first iteration of the workflow, for example, the frame rate drops down to 1.87 fps, the lowest frame rate measured, but then recovers to 47.80 fps in the next measurement. As the following runs of the workflow illustrate, GraalSqueak's behavior slightly improves over time. Nonetheless, it is unable to provide the same UI responsiveness that the OpenSmalltalk-VM can provide even after five minutes. Apart from that, the UI performance randomly dropped to 27.13 fps in the 294th second for exactly one measurement. We can only guess, but we believe this is due to garbage collection or to further optimizations by the JIT compiler.

5 Discussion

In this section, we show how GraalSqueak can be used as a tooling platform for polyglot programming and discuss its limitations.

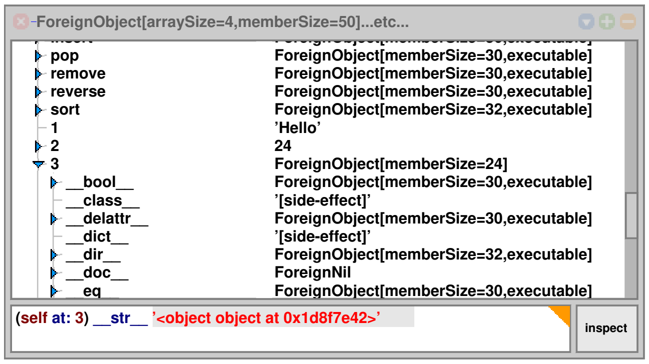
5.1 Tooling Platform for Polyglot Programming

The goal of GraalSqueak is to provide a tooling platform for polyglot programming. In particular, we want to use a Smalltalk environment for prototyping and experimenting because it provides short feedback loops and therefore faster iterations when incorporating user feedback. For this, GraalSqueak needs to integrate GraalVM's polyglot Application Programming Interface (API). This API can be used, for example, to evaluate code of all languages supported by



```
['Hello', 24, object(), {'o': True}]
ForeignObject[arraySize=4,memberSize=50]
```

Figure 6. Interactively evaluating Python code in a polyglot workspace.



```
pop          ForeignObject[memberSize=30,executable]
remove       ForeignObject[memberSize=30,executable]
reverse      ForeignObject[memberSize=30,executable]
sort         ForeignObject[memberSize=32,executable]
1            'Hello'
2            24
3            ForeignObject[memberSize=24]
  __bool__     ForeignObject[memberSize=30,executable]
  __class__    '[side-effect]'
  __delattr__  ForeignObject[memberSize=30,executable]
  __dict__     '[side-effect]'
  __dir__      ForeignObject[memberSize=32,executable]
  __doc__      ForeignNil
  __eq__       ForeignObject[memberSize=30,executable]

(self at: 3) __str__ '<object object at 0x1d8f7e42>'
```

Figure 7. Exploring a Python object in Squeak/Smalltalk.

GraalVM, retrieve various meta-information of objects of different languages, and to send messages to them.

In a first step, we wanted to make the workspace, inspector, and explorer tools of Squeak/Smalltalk work with the polyglot API. A workspace allows interactive evaluation and exploration of code, while the two other tools are for live object inspection.

Initially, we added support for the polyglot API through a new `PolyglotPlugin` in GraalSqueak. This plugin exposes every functionality of the underlying API. Exposing this API directly allows us to experiment with it from within Squeak/Smalltalk and without modifying our VM. Then, we introduced a new `TruffleObject` class in Squeak/Smalltalk. All foreign objects are represented as instances of this class. This made it possible to map Squeak/Smalltalk's meta-object protocol onto the protocol of objects that Truffle provides. In Truffle, non-primitive objects of a language can have members (for dictionary-like objects) or a size (for array-like objects) or both. Similarly, objects in Squeak/Smalltalk can have instance variables or/and are indexable. Both concepts fit together nicely, so the mapping between Smalltalk and Truffle objects was straightforward to implement.

Next, we subclassed existing Squeak/Smalltalk tools to create new versions of them that support foreign objects. Figure 6 shows our `PolyglotWorkspace`, a subclass of `Workspace`. This tool can be used to interactively evaluate code. In this case, the workspace's title hints that it is in Python mode and can therefore be used to run Python code. Instead of evaluating code through Squeak/Smalltalk's compiler, this new workspace uses the evaluation infrastructure of the polyglot API. Its context menu allows switching between all languages supported by the current GraalVM setup. In addition, Truffle uses a special bindings object to facilitate the exchange of objects between languages. Instead of having to use the polyglot API, this object is directly accessible in a workspace as an appropriate and language-specific import statement is automatically prepended before code evaluation.

Furthermore, instances of `TruffleObject` are opened in a new `PolyglotInspector` when object inspection is requested. Again, only few modifications were necessary to make it compatible with non-Smalltalk objects. In Squeak/Smalltalk, classes maintain information such as the number of instance variables. Therefore, we needed to override those methods of `Inspector` which accessed the class of the inspected object for this type of information. Instead, the object itself must be queried. Python objects, for example, can have arbitrary attributes not defined in their class. Figure 7 shows an `ObjectExplorer` opened on the object created in Figure 6. This tool re-uses the same infrastructure the inspector uses, so no additional modifications were needed to make it work. In this example, the explorer shows the list of attributes of the Python `list` as they are retrieved through Truffle's member API. The list's content is displayed below the members, the tool retrieves all these items through Truffle's array API.

This basic support for polyglot programming shows the potential of GraalSqueak. As a first new tool, we also built a polyglot notebook system [20] and are currently working on other tools that further support the development of polyglot applications.

5.2 Limitations

Although GraalSqueak is fully compatible to the Squeak/Smalltalk 5.2 release image and outperforms OpenSmalltalk-VM in some benchmarks, it also comes with limitations which are now discussed in more detail.

Partial Evaluation and Smalltalk An important limitation to mention is the conceptional mismatch between Squeak/Smalltalk and the way the Graal compiler works. In Squeak/Smalltalk, there is a language-level process scheduler. Processes are implemented as green threads. For this to function correctly, an interrupt handler checks for user interrupts and semaphores every couple of milliseconds and triggers a process switch if necessary. Typically, these checks for interrupts are performed after a certain number of message sends or loop iterations. This way, it is possible to exit an infinite recursion or an endless loop for example. In GraalSqueak, however, interrupt checks are only performed as part of Squeak/Smalltalk's idle process. Consequently, interrupts and corresponding process switches cannot be triggered in any other code and might not even trigger at all if the image is busy, for example when the bouncing-atoms simulation is running. The main reason for this is that performing interrupt checks in message sends and loops dramatically reduces the ability of the Graal JIT to perform its optimizations. In case no interrupt has fired as part of a method activation, for example, compiled code would be similar to what is compiled at the moment. However, if a process switch is triggered in another method or loop and goes through the compiled code, the current Truffle frame needs to be materialized, which in turn causes deoptimization. As a result, control-flow often has to transfer from compiled code back to the interpreter. Sometimes, the compiled methods are even invalidated and thrown away. The compiler might decide to re-compile them at a later point in time, but it is likely that they will also be deoptimized and invalidated again by a future process switch. Therefore, the system would never reach a steady state as the Graal compiler is busy optimizing and deoptimizing some methods over and over again.

One idea to overcome this problem is to use a Java thread for every Squeak/Smalltalk process. But this makes it very hard to support arbitrary sender modifications, especially when the sender of a Smalltalk activation record is set to another record which is part of a different Java process.

We also tried to exclude certain code paths from compilation which are known to trigger process switches. However, this only reduced the frequency of invalidations in the compiler.

Furthermore, it is unclear if Java continuations, which are currently being worked on as part of Project Loom [21], could help to solve this problem. Since continuations are not an uncommon feature of programming languages, it would be reasonable to have proper support for them in Truffle.

Memory Consumption and CPU Usage Our benchmarks from Section 4 focused on run-time performance, but did not include memory consumption and CPU usage. The OpenSmalltalk-VM is able to load a Squeak/Smalltalk image from file directly into memory. GraalSqueak, on the other hand, needs to allocate Java objects instead. Consequently, it cannot represent Squeak/Smalltalk objects as memory-efficient as OpenSmalltalk-VM. For a 60 MB image file, for example, the Java heap of GraalSqueak is about 220 MB in size, while OpenSmalltalk-VM is able to run the same image with a total memory usage of just under 100 MB (including memory for garbage collection). The default garbage collectors in Java require far more memory to deal with the large number of object allocations. While the heap is almost four times larger than the image file, the total memory consumption of GraalSqueak is about 2 GB. Moreover, GraalVM uses multiple threads for Truffle compilation and they always need to start from scratch when an image is opened. Therefore, the CPU usage is, especially at image startup, much higher compare with OpenSmalltalk-VM and other Smalltalk VMs. Although not supported by Truffle yet, GraalSqueak would greatly benefit from the ability to persist and re-use compiled code caches.

With native image, GraalVM also supports ahead-of-time (AOT) compilation of Truffle languages. These native images provide better startup times and lower memory footprints. An experimental build of GraalSqueak using this technology only needs around 700 MB in total to operate the 60 MB image file. This, however, is still far more compared with the OpenSmalltalk-VM. Furthermore, GraalSqueak's performance is further reduced due to the basic semi-space copying garbage collector that currently comes with native image. In the future, better garbage collector algorithms and more sophisticated profile-guided optimizations may further improve performance.

Tools and Language Interoperability Furthermore, there are limitations with regard to language integrations. The fact that GraalSqueak allows the execution of the Squeak/Smalltalk programming environment on the same level as all other languages has many advantages. But it also means that some features are hard if not impossible to support. For example, Squeak/Smalltalk's debugger implements its stepping functionality by executing bytecode after bytecode. However, it can only step its own bytecodes and not bytecodes or even ASTs of other languages. Moreover, its UI process must not be interrupted when the debugger is used. When running and debugging non-Smalltalk code, the interrupt handler is paused and therefore, Squeak/Smalltalk's UI is not functional. Future work might investigate if it is possible to use Squeak/Smalltalk's debugger to debug code running in a different Java-level thread. In addition, it is unclear what should happen if non-Smalltalk code calls out to Smalltalk code in which a UI component is triggered. At the moment, GraalSqueak detects syntax errors and debugger requests in this case and throws an error instead. Further, it would also be possible to do language interoperability between two different Squeak/Smalltalk images opened on the same GraalVM. This would allow for interesting experiments but is currently not correctly supported by GraalSqueak due to static state on VM level. Also, it is unclear whether the similar objects from different images (e.g., the `nil` object) should have the same identity or not. Lastly, saving an image in GraalSqueak persists all Squeak/Smalltalk objects in the memory file format. However, it is unable to persist heap objects from other languages.

6 Related Work

In this section, we discuss related work and compare it with GraalSqueak.

OpenSmalltalk-VM and Sista The OpenSmalltalk-VM [15] is a state-of-the-art Smalltalk VM and the default for Squeak/Smalltalk. Variations of it include Sista [1] which stands for "Speculative Inlining SmallTalk Architecture". Instead of optimizing code purely on VM-level, the VM provides an API which can be used from inside a Smalltalk environment to retrieve profiling information. This information can then be used to apply optimizations at the image level, which can also be persisted when saving the image.

RSqueak/VM RSqueak/VM [3] is an alternative interpreter for Squeak/Smalltalk and written in the RPython language implementation framework. Therefore, it leverages the same meta-tracing JIT compiler that is also used in PyPy to optimize its bytecode loop. Squimera, a multi-language runtime system [19], is based on interpreter composition of RSqueak/VM, PyPy, and Topaz. Similar to GraalSqueak, it provides tools with support for multiple languages. However, RPython is not designed to support language interoperability and therefore does not provide a polyglot API. In GraalSqueak, on the other hand, we can build on top of this API.

SOMns SOMns [13] is an implementation of the Newspeak language in Truffle. Since it is completely file-based, it does not provide compatibility with image-based Newspeak and traditional Smalltalk systems. GraalSqueak, on the other hand, aims at being fully compatible with existing Squeak/Smalltalk images. This also includes language features such as sender modifications or support for various VM plugins. As a consequence, we needed to find ways to support these features in Truffle.

Tools for GraalVM GraalVM provides an instrument API, which allows instrumentation of AST nodes. Based on this API, an implementation of the Chrome Debugging Protocol [4] is maintained as part of GraalVM's code base. This integration makes it possible to debug through polyglot code including Squeak/Smalltalk bytecode using an appropriate debugger client, such as the Chrome browser or Visual Studio Code. Similarly, an implementation of the Language Server Protocol (LSP) is currently under development [23]. Once officially supported, the LSP integration in Truffle could provide common features such as code completion and goto definitions out-of-the-box. For that, language implementations must provide appropriate hooks, for example to find the scopes of a language. In an early prototype, we were able to use this in combination with GraalSqueak for auto-completing Squeak/Smalltalk code.

Eco Eco is a prototype editor with support for language compositions [5]. The editor provides languages boxes which allow nesting of code written in different languages. Its incremental parser is able to continuously parse polyglot code written in the editor and to maintain an up-to-date AST of the program. Furthermore, some of this work has focused on combining the editor with the GraalVM ecosystem [16].

The Eco editor is integrated much deeper with the underlying runtime as it uses a special parser while GraalSqueak supports an entire programming system and makes it possible to build new tools on top of GraalVM's polyglot API.

7 Conclusion and Future Work

In this paper, we reported our experience implementing GraalSqueak, a Squeak/Smalltalk VM implementation written in Truffle. Alternative implementations including OpenSmalltalk-VM and RSqueak/VM as well as tools provided by GraalVM helped in this process. Nonetheless, we needed to find ways to support language-specific features that are not straightforward to implement in the Truffle framework. Furthermore, we evaluated the performance of both the language and the Squeak/Smalltalk programming environment with different benchmarks. We were able to demonstrate that GraalSqueak provides competitive performance compared with OpenSmalltalk-VM, a state-of-the-art VM for Smalltalk. Moreover, we demonstrated how this system can be used as a tooling platform for polyglot programming and discussed limitations of the current implementation.

We plan to use GraalSqueak for further research on tools for polyglot programming. For this, we are currently exploring approaches for polyglot code editors and different integrations of APIs for language interoperability. These new ideas and tools for polyglot programming could then be evaluated by means of user studies. In future work, the limitations of GraalSqueak need to be investigated further. In particular, it would be interesting to find a proper way to support Squeak/Smalltalk's interrupt handler and polyglot object heaps. Moreover, we believe cross-language benchmarks are needed to evaluate runtime performance over language boundaries in polyglot systems such as GraalVM.

A Benchmark Environment and Warmup

Table 1 lists the different configurations of our benchmark environment. Figure 8 shows the warmup behavior of GraalSqueak observed during the execution of the Are We Fast Yet benchmark suite.

Table 1. Overview of the configurations of our benchmark environment (cf. Figure 4 and Figure 5).

Configuration Name	Git Commit	Squeak/Smalltalk Image	GraalVM Version
GraalSqueak	9d565518	Squeak5.2 18229 64bit	GraalVM CE/EE 19.0.2
OpenSmalltalk-VM	15341b57	Squeak5.2 18229 64bit	n/a
RSqueak/VM	d33005c8	Squeak5.1 16494 32bit	n/a
SOMns	8256b0d5	n/a	graal-jvmci-8 1.8.0_181 (1076dfbd)

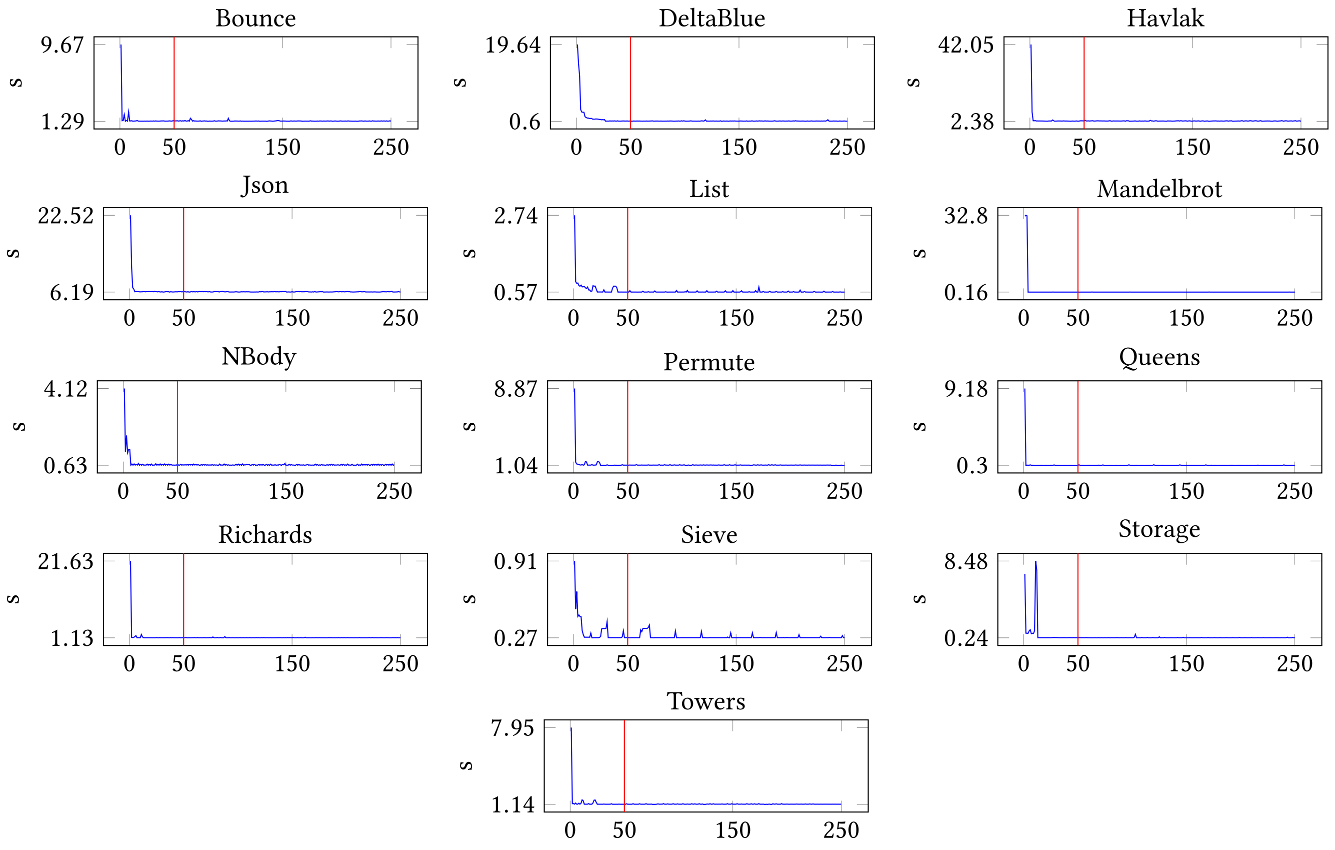


Figure 8. Runtime in seconds of the first 250 iterations of GraalSqueak on GraalVM CE 19.0.2 running the Are We Fast Yet benchmark suite (cf. Figure 4). Y-Axis shows maximum and minimum seconds of each data series. Warmup behavior can be observed in the first few iterations while performance has always reached a somewhat steady state with the 50th iteration (highlighted with a red line).

Acknowledgments

We gratefully acknowledge the financial support of Oracle Labs[3], HPI's Research School[4], and the Hasso Plattner Design Thinking Research Program[5].

References

[1] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. 2017. Sista: Saving Optimized Code in Snapshots for Fast Start-Up. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (ManLang 2017)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3132190.3132201

[2] Carl Friedrich Bolz, Lukas Diekmann, and Laurence Tratt. 2013. Storage Strategies for Collections in Dynamically Typed Languages. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '13)*. ACM, New York, NY, USA, 167–182. https://doi.org/10.1145/2509136.2509531

[3] Carl Friedrich Bolz, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, Armin Rigo, and Toon Verwaest. 2008. *Back to the Future in One Week — Implementing a Smalltalk VM in PyPy*. Springer-Verlag, Berlin, Heidelberg, 123–139. https://doi.org/10.1007/978-3-540-89275-5_7

[3]https://labs.oracle.com/
[4]https://hpi.de/en/research/research-school.html
[5]https://hpi.de/en/dtrp/

[4] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. 2018. Fast, Flexible, Polyglot Instrumentation Support for Debuggers and other Tools. *The Art, Science, and Engineering of Programming* 2, Article 14 (2018), 30 pages. Issue 3. https://doi.org/10.22152/programming-journal.org/2018/2/14

[5] Lukas Diekmann and Laurence Tratt. 2014. Eco: A Language Composition Editor. In *Software Language Engineering (SLE)*. Springer, 82–101. https://doi.org/10.1007/978-3-319-11245-9_5

[6] ECMA-335 2012. *ECMA-335: Common Language Infrastructure (CLI)*. Technical Report ECMA-335. Ecma International, Geneva, Switzerland. Also ISO/IEC 23271.

[7] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies (IWST '16)*. ACM, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/2991041.2991062

[8] Bert Freudenberg, Dan Ingalls, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2014. SqueakJS: A Modern and Practical Smalltalk That Runs in Any Browser. In *Proceedings of the 10th ACM Symposium on Dynamic Languages (DLS '14)*. ACM, New York, NY, USA, 57–66. https://doi.org/10.1145/2661088.2661100

[9] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley Longman, Boston, MA, USA.

[10] David Leopoldseder, Lukas Stadler, Manuel Rigger, Thomas Würthinger, and Hanspeter Mössenböck. 2018. A Cost Model for a Graph-based Intermediate-representation in a Dynamic Compiler. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2018)*. ACM, New York, NY, USA, 26–35. https://doi.org/10.1145/3281287.3281290

[11] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-language Compiler Benchmarking: Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016)*. ACM, New York, NY, USA, 120–131. https://doi.org/10.1145/2989225.2989232

[12] Stefan Marr and Stéphane Ducasse. 2015. Tracing vs. Partial Evaluation: Comparing Meta-compilation Approaches for Self-optimizing Interpreters. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2015)*. ACM, New York, NY, USA, 821–839. https://doi.org/10.1145/2814270.2814275

[13] Stefan Marr, Carmen Torres Lopez, Dominik Aumayr, Elisa Gonzalez Boix, and Hanspeter Mössenböck. 2017. A Concurrency-Agnostic Protocol for Multi-Paradigm Concurrent Debugging Tools. In *Proceedings of the 13th ACM SIGPLAN International Symposium on Dynamic Languages (DLS'17)*. ACM, 12. https://doi.org/10.1145/3133841.3133842

[14] Robert McGill, John W. Tukey, and Wayne A. Larsen. 1978. Variations of Box Plots. *The American Statistician* 32, 1 (1978), 12–16. https://doi.org/10.2307/2683468

[15] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two Decades of Smalltalk VM Development: Live VM Development Through Simulation Tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL 2018)*. ACM, New York, NY, USA, 57–66. https://doi.org/10.1145/3281287.3281295

[16] Sarah Mount and Laurence Tratt. 2017. Simple Visualisation of Profiling Data. Project report.

[17] Fabio Niephaus, Tim Felgentreff, and Robert Hirschfeld. 2018. GraalSqueak: A Fast Smalltalk Bytecode Interpreter Written in an AST Interpreter Framework. In *Proceedings of the 13th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS '18)*. ACM, New York, NY, USA, 30–35. https://doi.org/10.1145/3242947.3242948

[18] Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2019. Efficient Implementation of Smalltalk Activation Records in Language Implementation Frameworks. In *Proceedings of the 3rd International Companion Conference on Art, Science, and Engineering of Programming (Programming '19)*. ACM, New York, NY, USA, Article 6, 3 pages. https://doi.org/10.1145/3328433.3328440

[19] Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and Marcel Taeumel. 2018. Live Multi-language Development and Runtime Environments. *The Art, Science, and Engineering of Programming* 2, Article 8 (2018), 30 pages. Issue 3. https://doi.org/10.22152/programming-journal.org/2018/2/8

[20] Fabio Niephaus, Eva Krebs, Christian Flach, Jens Lincke, and Robert Hirschfeld. 2019. PolyJuS: A Squeak/Smalltalk-based Polyglot Notebook System for the GraalVM. In *Proceedings of the 3rd International Companion Conference on Art, Science, and Engineering of Programming (Programming '19)*. ACM, New York, NY, USA, Article 24, 6 pages. https://doi.org/10.1145/3328433.3328434

[21] Oracle Corporation and/or its affiliates. 2019. *Loom - Fibers, Continuations and Tail-Calls for the JVM*. https://openjdk.java.net/projects/loom/

[22] Manuel Rigger, Matthias Grimmer, Christian Wimmer, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Bringing Low-level Languages to the JVM: Efficient Execution of LLVM IR on Truffle. In *Proceedings of the 8th International Workshop on Virtual Machines and Intermediate Languages (VMIL 2016)*. ACM, New York, NY, USA, 6–15. https://doi.org/10.1145/2998415.2998416

[23] Daniel Stolpe, Tim Felgentreff, Fabio Niephaus, and Robert Hirschfeld. 2019. Language-independent Development Environment Support for Dynamic Runtimes. In *Proceedings of the 15th ACM SIGPLAN International Symposium on Dynamic Languages (DLS '19)*. ACM, New York, NY, USA, 11. https://doi.org/10.1145/1122445.1122456

[24] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. 2014. Interleaving of Modification and Use in Data-driven Tool Development. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014)*. ACM, New York, NY, USA, 185–200. https://doi.org/10.1145/2661136.2661150

[25] Thuan L. Thai and Hoang Q. Lam. 2003. *.NET Framework Essentials* (3rd ed.). O'Reilly Media.

[26] Andreas Wöß, Christian Wirth, Daniele Bonetta, Chris Seaton, Christian Humer, and Hanspeter Mössenböck. 2014. An Object Storage Model for the Truffle Language Implementation Framework. 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, 133–144. https://doi.org/10.1145/2647508.2647517

[27] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. 2017. Practical Partial Evaluation for High-performance Dynamic Language Runtimes. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 662–676. https://doi.org/10.1145/3062341.3062381

[28] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. 2012. Self-optimizing AST Interpreters. In *Proceedings of the 8th Symposium on Dynamic Languages (DLS '12)*. ACM, New York, NY, USA, 73–82. https://doi.org/10.1145/2384577.2384587

[29] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. 2013. 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)*. ACM, New York, NY, USA, 187–204. https://doi.org/10.1145/2509578.2509581