How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time

An Experience Report of Using the RPython Toolchain

Tim Felgentreff Tobias Pape Patrick Rein Robert Hirschfeld

Hasso Plattner Institute

{firstname.lastname}@hpi.de

Abstract

In this paper we present our experience in letting students develop RSqueak/VM, a fast virtual machine for executing Squeak/Smalltalk, over the course of multiple introductory VM courses and projects. Our experience indicates that the way RSqueak/VM is constructed makes it easy for developers with little or no prior exposure to VM development to construct a Squeak runtime that is very nearly competitive to the professional grade Cog VM, which is used by Squeak and other flavors of Smalltalk. In this experience report we discuss the benefits and drawbacks of RSqueak/VM as a research vehicle for developing and evaluating Smalltalk language extensions without having to learn the deep secrets of VM development.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—code generation, optimization

Keywords Tracing JIT, Squeak/Smalltalk, RPython

1. Introduction

We present our experience in creating RSqueak/VM [4, 5], a fast virtual machine (VM) that can execute unmodified Squeak/Smalltalk images. RSqueak/VM was originally based on the SpyVM [1] and is written using the RPython VM toolchain and leverages its meta-tracing just-in-time compiler (JIT) framework [2]. RSqueak/VM runs faster in some benchmarks than the standard Cog VM that is commonly used for Squeak and is competitive in many others. The RSqueak/VM JIT inherits its platform support from RPython, and can thus target x86 (both 32 and 64 bits), PowerPC64, ARMv6 and ARMv7 (32bit), and IBM's Z architecture (specifically its current incarnation s390x running 64 bit Linux).

In this report we share our development experience with RSqueak/VM. In particular, we believe that the way RSqueak/VM is implemented makes it possible to achieve good performance even though development almost exclusively takes place during part-time student projects and in introductory VM courses. Furthermore, most of the students that develop RSqueak/VM have never taken VM courses before, have no prior experience with Squeak VM development, and have little knowledge of Squeak internals besides using the Squeak development tools. We see this in contrast to the Cog VM that is being developed by full-time developers and that requires a good knowledge of the Squeak object format, its garbage collector, its meta-programming facilities, as well as C and assembly programming to participate in optimization tasks on the VM level. However, RSqueak/VM is not production ready and is worse in terms of stability and performance in many areas than the highly optimized Cog. It also only supports Squeak, whereas Cog can also run Pharo and Cuis[1].

A goal for RSqueak/VM is to support experiments that extend the Squeak system or the language in particular for researchers without prior experience in VM development. In our experience, the RSqueak/VM architecture and toolchain makes it easy for new students to experiment with new features in the VM while achieving very good performance. Among such features are software transactional memory (STM) (which gives RSqueak/VM the ability to scale Squeak to multi-core systems), replacing VM plugins with pure Squeak code (which allows RSqueak/VM to execute Balloon and BitBlt Slang-code directly in Smalltalk without requiring a translation to C), or supporting multiple Squeak image formats in the same VM (starting with the Squeak

[1] There is no fundamental reason for not supporting Pharo and Cuis on RSqueak — both of these dialects are simply missing fallback code for primitives that are optional in Squeak and not implemented at all in RSqueak, so execution fails when these primitives are hit. One example is primitive 105 which replaces part of a string with another string. On RSqueak, the fallback code is actually faster in many cases than the primitive so we omitted the primitive implementation. Pharo removed the fallback code, failing in a non-recoverable way early during startup.

1.2 release all the way to the new Spur-format Squeak 5.0 release) without sacrificing run-time performance.

RSqueak/VM takes many of its ideas from the PyPy project [10], with which it shares its implementation language and development toolchain RPython. VMs written in RPython automatically get garbage collection and can use the generic meta-tracing JIT library. Also, RPython is a true subset of Python. This also means that, like Squeak, the VM can be developed, simulated, and debugged in a high-level language (Python). Just like the Squeak VM is developed using Slang (a subset of Smalltalk), we think this lowers the barrier of entry for new VM developers. The simulation of RSqueak/VM extends all the way to the JIT, so even optimizations in that area do not require any knowledge of assembly language or the machine-level object format. We thus argue that RPython is a more friendly VM development platform for novices than Slang/VMMaker. In particular, many optimizations and platform interfaces can be developed using high-level RPython application programming interfaces (APIs).

Although the Cog VM offers very good performance and its development process clearly works well when a full-time VM developer is available, we argue it is not ideal for researchers that want to conduct language experiments and evaluate them with respect to performance, but do not have the time or experience to work on low-level performance optimizations in the VM. We think RSqueak/VM can make the following contributions in this area:

- RSqueak/VM is written in RPython and thus offers a flexible, object-oriented language as implementation platform, where extensions can make use of VM level polymorphic implementation classes and object-oriented intercession. This makes it easy to add new primitives, create new VM-level objects, or conditionally extend the interpreter.
- RSqueak/VM benefits from RPython's high-level JIT annotations that allow the developer to express assumptions such as branch probability or functional dependencies and change frequencies of certain variables and have those annotations automatically translated into optimizations of the JIT. This makes it easy to achieve good performance when developing extensions that change the semantics of the language such as Context-oriented Programming (COP) or Aspect-oriented Programming (AOP).

2. Squeak VM Development so far

A fast VM is a core concern for many Smalltalk implementations to make the Smalltalk language competitive in real-world application scenarios. Although the success of a language also depends on many other factors, newcomers to Squeak often use simple, well known programming tasks to evaluate the feel of developing in the system and the effectiveness of its runtime. If execution is slow, people might not be swayed by arguments about the beauty and expressiveness of the Smalltalk language and instead move on to other programming languages.

The Squeak virtual machine was created as a simple bytecode interpreter [8]. The initial goals for the VM were to be reasonably efficient, while still being easily extensible so that continued VM development could take place as the Squeak-dialect of Smalltalk evolves. To that end the VM and its plugins were developed largely in Slang, a subset of Smalltalk amendable to static compilation C, so that VM developers could use the Smalltalk environment to develop, simulate, and debug the Squeak VM. Originally, only some platform specific code was hand-written in C.

Since early 2011, the open-source Squeak VM has a branch to develop the Cog variant of the VM. Cog is a very high performance virtual machine for Squeak, Pharo, Cuis, and Newspeak that adds a stack-to-register-mapping just-in-time compiler, inline message caching, and optimizations for Smalltalk's first-class context objects. Much of Cog's performance comes from compiling Smalltalk methods by stitching pieces of hand-written and optimized assembler for a number of Squeak primitives and bytecodes together. Because the original Squeak object format was not developed for this use case, the complexity of some primitives was too high to convert them into assembler. For example, the variable object header format meant to improve interpreter performance and memory requirements made Cog's assembler for probing the inline cache larger than it should be, costing additional performance on every send. As a result, Squeak's object format was changed to a new object format called *Spur* that (among other optimizations) allows more primitives to be implemented in optimized assembler and that simplifies the inline cache code [9].

The communities around Squeak, Pharo, and Cuis continue to evolve their various Smalltalk dialects. This includes evolving the core class library (such as changing the internal structure of collections or the hierarchy of numeric classes) adding mechanisms that can be used to improve the expressiveness of the language (like AOP or COP), or evolving the tools to offer more powerful features and introspection capabilities. As these mechanisms and language extensions evolve during use, new execution paths in the VM become apparent that need to be optimized, and old optimizations become superfluous.

Due to the evolving nature of the systems that use the Cog VM, the malleability with which the original Squeak interpreter was created is still important to continue supporting language-level experiments. As we argued above, one important question for any language feature is whether it will have an impact on the efficiency of the system as a whole. However, if new developers cannot easily adapt the VM it is hard to judge weather a new language extension can eventually be optimized appropriately. A problem we see for new developers with the approach Cog and Spur take is the direct mapping to assembler and intimate knowledge of the object layout that is required to make the VM fast. This leads to a situation where some performance-critical parts of the VM are no longer easily understandable by just reading the Slang code (reducing the benefit of simulating the VM in Smalltalk) and where new developers interested in working on the VM and extending it with new VM-level objects or primitives must consider how these will translate into assembler if performance is relevant to the experiment. Given that a goal of the Squeak VM is to run on a multitude of platforms, this means considering not only x86 assembler, but also possibly ARM, MIPS, PowerPC, or more architectures in the future.

* * *

The rest of this paper presents the current state of RSqueak/VM architecture and performance, and what we found to be interesting aspects of RSqueak/VM development as compared to the traditional way of building VMs. Section 3 presents the core object model. Section 4 describes how we use strategies to deal with different object layouts efficiently. Section 5 discusses performance optimizations to simulation of primitives in Smalltalk and presents benchmarks as a general overview of RSqueak/VM performance. Section 6 discusses two example projects by students that worked on RSqueak/VM and Section 7 summarizes our findings.

3. RSqueak/VM Model Classes

RSqueak/VM is architected in an object-oriented fashion. At its core is a class simply called `Interpreter` which implements the bytecode loop by providing a method for each Squeak bytecode. The second core class is the so-called `ObjectSpace`, which takes on the role of the `InterpreterProxy` from Slang/Squeak VM in that it has a globally unique instance that provides access to often needed objects and convenience functions. Both of these classes are straightforward and we can find similar expressions of their behaviors in the standard Squeak VM, as well as the JSqueak/Potato VM[2] and SqueakJS [6].

All Smalltalk objects in RSqueak/VM are represented as instances in the *model class* hierarchy, presented in Figure 1. Squeak (and most other Smalltalks that we are aware of) use a subclassing mechanism that allows developers to control the storage layout of a class's instances. Ordinary objects store pointers, others store bytes or words, or hold on to their pointers weakly. In addition to that, we have special classes known to the VM that cannot store anything, but are represented as immediates through pointer tagging, and compiled methods, which use a hybrid of byte and pointer storage.

In RSqueak/VM, the different types of objects are represented in the class hierarchy of the model classes. Any Squeak object is represented at run-time by a sub-instance of the VM implementation class `W_Object`. This abstract class contains default implementations for some methods such as getting the Squeak class of an object, accessing its contents, or unwrapping the Squeak contents into RPython types. RSqueak/VM provides eight concrete subclasses: `W_SmallInteger`, `W_Float`, `W_Character` represent the immediate and (generally) not mutated objects of their Smalltalk equivalents. `W_LargePositiveInteger1Word` represents unsigned positive integers that happen to fit in a machine word (we elaborate on this optimization in Section 5). `W_PointersObject`, `W_BytesObject`, and `W_WordsObject` represent the different storage types that Smalltalk developers can use when subclassing. Finally, `W_CompiledMethod` represents the special format of Squeak's CompiledMethod that mixes pointer and byte fields.

One nice effect of the abstraction using model classes is that features can be implemented in an object-oriented manner. For example, accessing fields is simply implemented in subclass-specific `at0` and `atput0` methods. Even a feature as specific as `become` is easily implemented for objects of the same structure: rather than doing a full GC scan and changing all pointers, we merely need to exchange the RPython model objects' fields. For example, our `become` implementation for a variable words object that is represented on RPython as a `W_WordsObject` simply needs to exchange the fields of these objects: a words array, an optional object to represent the words array as a native C-level array (used for interoperability with existing Squeak plugins), the object hash, and the class reference. The latter two fields are implemented in the abstract model classes `W_AbstractObjectWithIdentityHash` and `W_AbstractObjectWithClassReference`, respectively, which `W_WordsObject` inherits from.

```
# from W_WordsObject
def _become(self, w_other):
    assert isinstance(w_other, W_WordsObject)
    self.words, w_other.words = w_other.words, self.words
    self.native_words, w_other.native_words = w_other.native_words, self.native_words
    W_AbstractObjectWithClassReference._become(self, w_other)

# from W_AbstractObjectWithClassReference
def _become(self, w_other):
    assert isinstance(w_other, W_AbstractObjectWithClassReference)
    self.w_class, w_other.w_class = w_other.w_class, self.w_class
    W_AbstractObjectWithIdentityHash._become(self, w_other)

# from W_AbstractObjectWithIdentityHash
def _become(self, w_other):
    assert isinstance(w_other, W_AbstractObjectWithIdentityHash)
    self.hash, w_other.hash = w_other.hash, self.hash
```

[2] http://sourceforge.net/projects/potatovm/, accessed May 19, 2016

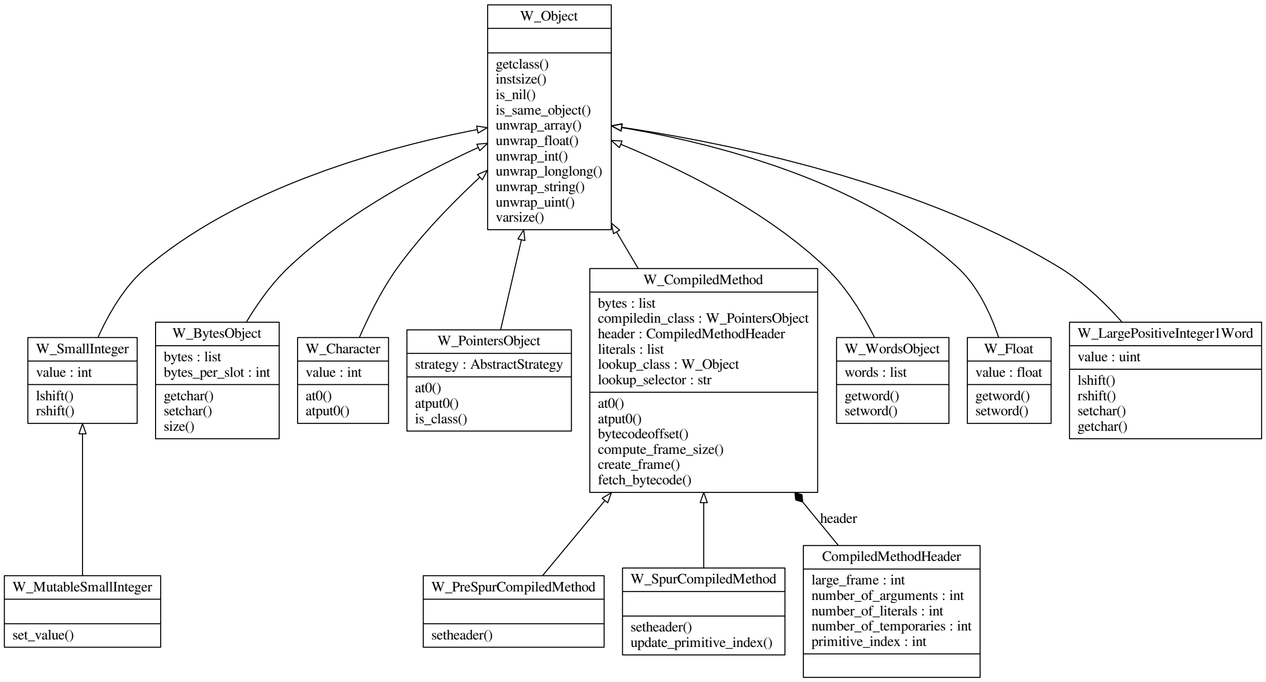


Figure 1: The Core Model Classes in RSqueak/VM

Immediate Classes All Squeak VMs use pointer tagging for `SmallInteger`, and recently the Spur object memory format in both 32- and 64-bit have introduced tagging also for `Characters` and `Floats`. In RSqueak/VM we do not tag pointers, but we do represent those immediate types with special VM-level classes. These classes inherit directly from `W_Object` and they each have only one field to hold their value. RPython provides a special declaration for values that are expected to change rarely or that are even immutable as a hint for the JIT.

`SmallIntegers` in RSqueak/VM are marked as immutable. This gives the JIT the opportunity to replace object allocations for `W_SmallIntegers` with their respective integer value and allow it to produce code for integer-heavy loops where only those integers that escape the loop must be allocated on the heap. In return, the assembler does not have to check each pointer access for tag bits—providing a speedup that in many cases outweighs the cost of allocating a small number of `W_SmallInteger` objects at the end of the loop.

We further reduced the cost of allocations by allowing mutations to `SmallIntegers` that are stored in pointer object fields. Most integers that escape loops do so because they are stored in another object's field. If the objects in those fields are immutable, the VM must allocate a new object. Instead, whenever a `W_SmallInteger` is stored into a pointers object, we wrap its value in a subclass `W_MutableSmallInteger` which does not have the immutability annotation. In cases where a field already contains such a mutable integer, the JIT can avoid allocation and simply update the value.

`Floats` in most Squeak images are implemented as objects that store two words, which together make a double precision floating point value. Although it happens rarely and (as far as we know) never in production code, this means that `Floats` can be mutated by writing to their word fields directly. To accommodate this, `W_Float` objects in RSqueak/VM were usually only annotated as quasi-immutable. This still allows the JIT to emit fast assembler instructions by speculating that a `Float`'s value never changes, but requires the insertion of a few additional *guard* instructions before each float access.

The final immediate class introduced with Spur are `Characters`. Like the previous two types, we wrap characters and mark them as immutable.

* * *

In our experience, structuring the core model of RSqueak/VM in this object-oriented fashion made it easier for new developers to contribute to its development. We had the same experience with SqueakJS. However, in RPython the additional performance that can be achieved by a few well-placed JIT annotations allows even new developers to extend the VM and immediately achieve very good performance. The most important JIT annotations so far have been `elidable` and `promote`. The former tells that JIT that a certain RPython method is *pure*, that is, its return value will be the same if all arguments are the same. The second tells the JIT that an object is likely to be constant in a given execution trace. The students, just knowing this abstract description, used these annotations extensively in many parts of the VM when they knew that many objects that are theoretically mutable in Squeak (method dictionaries, compiled methods, literals, strings, ...) nonetheless are unlikely to change. Thus, these annotations completely avoid any infrastructure for caching (such as method caches) and offload this work to the JIT.

Another lesson we learned from avoiding pointer tagging was that it made the transition to Spur much easier— adding support for immediate float and character values was as simple as creating a new subclass akin to `W_SmallInteger`. At no point did we have to worry about checking pointer accesses to go through (un-)tagging macros, so mistakes by less experienced developers caused by forgetting these are impossible by design. This is especially important since most RSqueak/VM developers have no prior experience with tagging or other traditional VM optimization techniques.

4. Strategies to Avoid Changes to the Object Layout

The original Squeak VM uses a simple model of object representation where the in-memory layout of objects corresponds to the Squeak meta-protocol. This strategy has advantages (such as making the VM simpler) but comes with the drawback that the evolution of the VM is tied to the evolution of the image.

This tight coupling makes it difficult to support Squeak on an alternative VM with competitive performance, and it makes it near impossible for the standard VM to support different image formats. This means that even as the VM evolves and becomes faster, older images and architectures that are no longer supported are left to use older VMs.

With RSqueak/VM, we wanted to avoid this tight coupling. To still get good performance, we employed a combination of strategies [3] and shadows [1] to translate in-image object layouts into formats that can be flexibly used and optimized in RPython.

Strategies Strategies are a technique that was originally presented for PyPy to avoid memory allocations for simple objects stored in lists. In PyPy, this is useful to improve both the memory and runtime overhead of homogenous lists, for example lists that store only integers and floats.

Shadows Shadows were used in both SpyVM [1] (the precursor to RSqueak/VM) and JSqueak/Potato[3]. Some special objects such as classes, method dictionaries, and context parts, which look like ordinary objects to the image, store information in their fields that is important during lookup, object creation, or method activation. Classes, for example, store a class header as an ordinary tagged integer that encodes information such as the class type or the size of instances. When the SpyVM first needed to access that information, it would attach a shadow to the object in question, decode the header into explicit fields in that shadow, and subsequently direct all read and write operations through that shadow, which acts as an adapter to provide a more VM-friendly interface. In Potato, shadows were used to represent large positive integers also as Java big integers for primitive operations.

4.1 Storage Strategies in RSqueak/VM

In Squeak, many objects are mostly empty, that is, they are filled with `nil` values. We found that, for example, most method dictionaries of metaclasses are empty, yet take up 10 words plus header, the default size of a Squeak `Array` instance. While this could be corrected in-image, strategies can provide a mechanism to improve the memory overhead of such objects. All pointers objects in RSqueak/VM start out with a storage strategy that always returns `nil` on every read, without actually allocating the memory required. If the object is never written to, no allocation need ever take place. If it is written to or used in a specific context, we initiate a storage transition.

All subclasses of `StorageStrategy` in RSqueak/VM can be converted into one another by first reading all fields from the old storage and then writing them to the new storage. We use a Squeak-specific set of strategies with different trade-offs with regards to memory and access performance. Commonly used strategies are `FloatOrNil` and `SmallIntegerOrNil`. These are useful for homogenous collections and can store floats and integer values untagged by using a C-level int or float array. A special NaN tag and the largest word sized integer are used to indicate that a position really hold a Squeak `nil`, rather than a float or integer.

Figure 2 shows how storage strategies can reduce memory overhead. These measurements were taking by simply opening a Squeak 5.0 release image, invoking the *About Squeak* dialog, and closing the image. Specifically, we can see that about 22 % of all objects with on average nine slots remain in the `AllNil` strategy, saving the allocation of nine words of memory for each of these objects. Counting all fields of all objects in the image, over half of all slots do not have memory allocated for them, because their objects remain in the `AllNil` strategy. The figure also shows that 12 % of all objects (and 6 % of all fields) only contain integers or `nil` values, and so for these no `W_SmallInteger` objects are created, but the integer values are stored unwrapped even without tagging.

The shadows from SpyVM are now special kinds of storage strategies in RSqueak/VM that objects can transition into. The first time a pointers object is used as a class (for example, by sending a message to an instance of that class), the VM converts all fields in the class object into a more suitable (but less memory efficient) form by eagerly decoding information from the class header and storing each piece of information separately. This makes development easier by providing a dedicated VM-level class with a class-specific

[3] `http://potatovm.blogspot.de/2008/07/large-integer-handling-in-potato.html`, accessed May 19, 2016

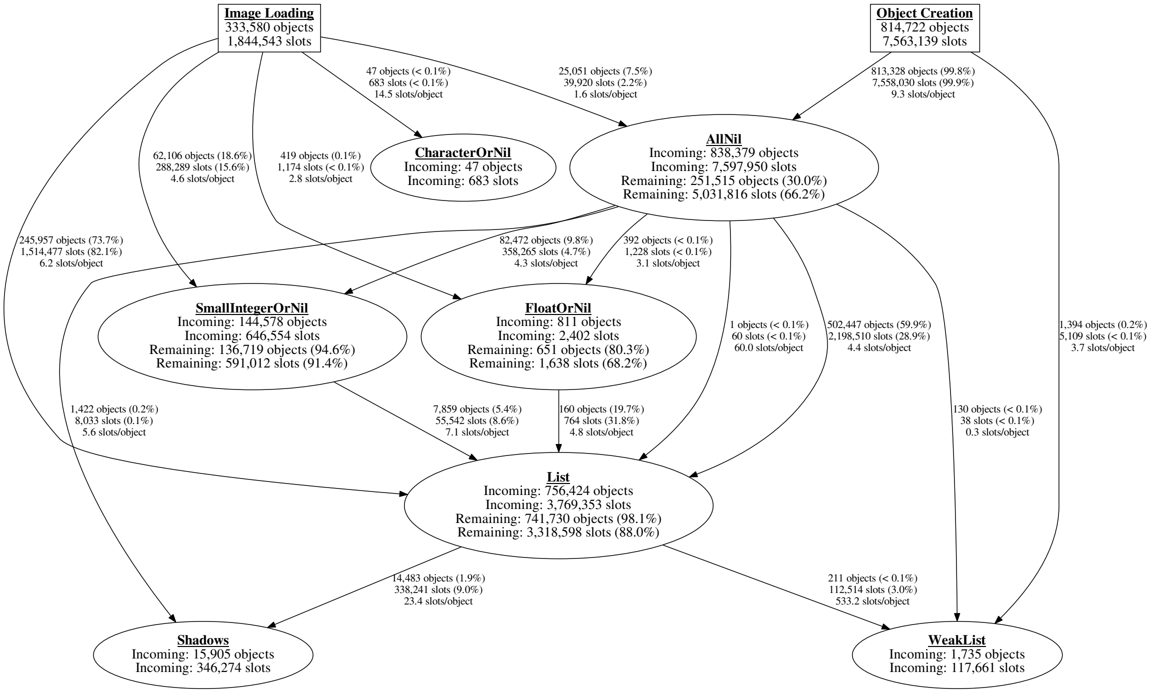


Figure 2: Storage Strategies and Transitions when Running Squeak

interface for tasks such as lookup and object creation. There are similar special strategies for context parts and method dictionaries.

While it may seem like a lot of overhead to have a storage array, a strategy, and a model object for each Smalltalk object, the JIT that is generated for the RPython sources removes almost all of these allocations in loops. For example, fields of context parts are mapped to registers or they are omitted entirely if they are found to not escape the loop.

4.2 Shadows for the Meta-object Protocol

Shadows make the implementation of RSqueak/VM more complex than we would like in some cases. A major reason for this complexity (and some lost performance that comes with it) is how Squeak's meta-object protocol is used in the system. In particular the "special" objects that are ordinary objects from the point of view of the VM can and are manipulated as such. This includes method dictionaries, method literals, compiled methods, strings, large integers, floats, classes. This mutability is actually used by the development tools. For example, method dictionaries are manipulated by writing directly into the underlying arrays that implement the dictionary, but there is no explicit VM callback that passes this object. Thus, RSqueak/VM uses a special shadow to notice such manipulation to selectively invalidate JIT code involving those method dictionaries. Without such special shadows, it is very easy to have too coarse grained caching in the JIT, leading to frequent invalidation of machine code during development.

Shadows abstract from the Squeak object layout for these special objects to provide object-oriented, RPython-level methods for tasks such as method lookup or temporary variable access. A benefit of them is that, since these shadows are still ordinary RPython objects, they can be returned and handed around as such. This makes, for example, the implementation of Squeak's meta-object protocol for accessing the current context trivial to implement. All bytecodes are implement as methods on the class `ContextPartShadow`. The bytecode to access the active context simply pushes self on to the stack.

5. Simulated Primitives in RSqueak/VM

As we have previously presented [5], RSqueak/VM simulates many primitives that have fallback code available in Smalltalk. Some of these primitives are no longer implemented in Spur or at least were found not to provide a performance benefit anymore in some cases[4]. Some primitives that RSqueak/VM simulates, however, do not have fallback code, but are implemented in Slang. The most important of these are the BitBlt and Balloon primitives that are used to render

[4] For example, there was recent discussion on the Squeak VM mailing list about findings that some primitives for large integers have now become slower than the JIT-ed Smalltalk code. See: `http://forum.world.st/The-Inbox-Kernel-dtl-1015-mcz-td4889787.html`, accessed May 19, 2016

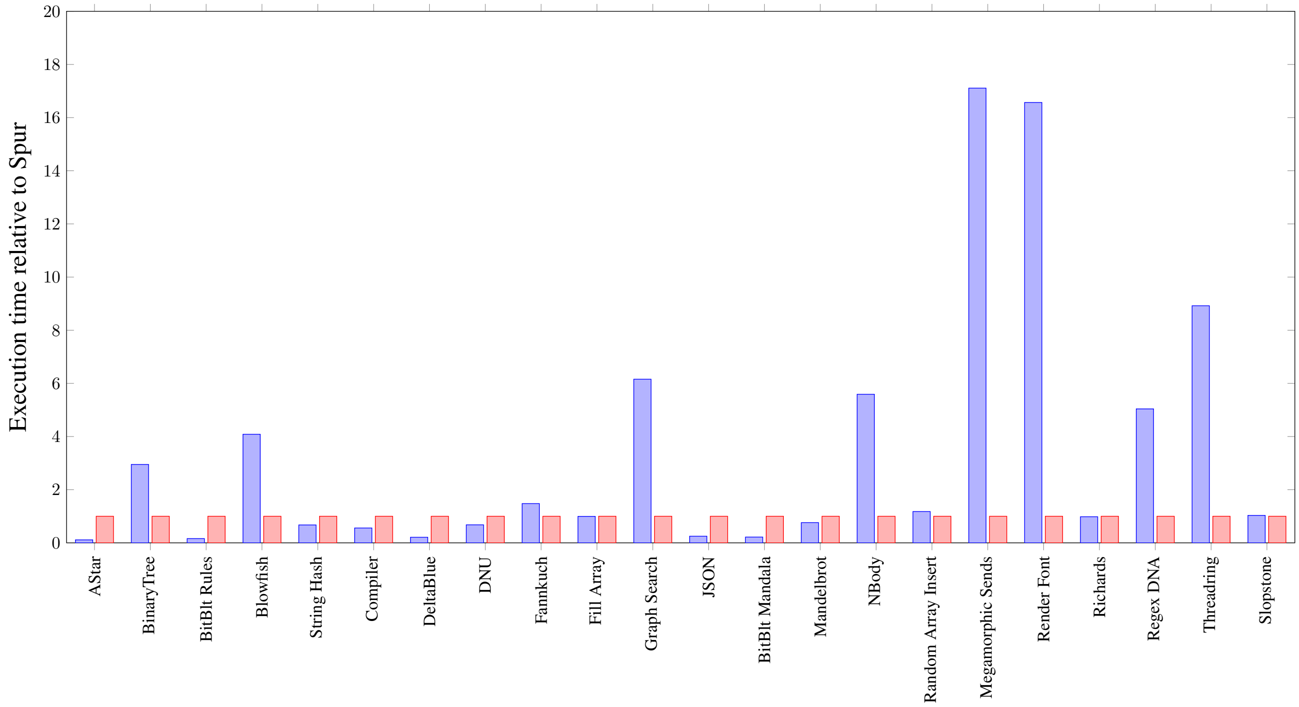


Figure 3: Relative runtime of various benchmarks for RSqueak/VM (blue) and Spur VM (red).

the entire Squeak user interface and have to be fast to allow using the environment.

The simulation of BitBlt uses Slang code available in the VMMaker packages, which neccessitates that any image that is used on RSqueak/VM must include that package in some form. An interesting result of simulating these primitives is that any changes to the BitBlt code are immediately visible — but if the changes break the system's ability to render a debugger this feature can also cause frustration. We have found Context-oriented Programming (COP) [7] to be a good fit here as it allows for very fast turn-around times when developing code that is central to the function of the environment and still keep the environment itself protected.

Optimizing for Primitive Simulations

We have previously shown that the performance of simulating BitBlt primitives on RSqueak/VM is within one order of magnitude of the performance of the standard Squeak VMs binary plugin and simulating Balloon primitives is within two orders of magnitude. [5]

The BitBlt code does calculations on word-sized chunks of the display and the forms it renders to. When the Slang code is translated to C, these calculations are all done on word-sized unsigned integers. However, in-image, they will usually be done on large positive integers. Since RSqueak/VM does not ship with a LargeInteger plugin to make these calculations fast, the fallback code is used that causes byte-wise access to the integer, resulting in very slow 8-bit operations.

LargePositiveInteger in Squeak is a byte class. This storage type was chosen to minimize the memory overhead that would come with word objects, but the performance of that storage layout is not ideal when we have to do arithmetic for BitBlt operations. To avoid both the maintenance burden of a large integer plugin and the bad performance of the large integer operations fallback code, RSqueak/VM uses a special VM-level class to represent small, word-sized large positive integers. Larger integers are still represented as byte objects (with the associated performance properties), but word-sized integers are stored directly as unsigned integer values in a field of the `LargePositiveInteger1Word` class. That class otherwise simulates the semantics of byte objects. When such objects are used in arithmetic, the usual path in Squeak is to fail the `SmallInteger` primitive and use double dispatch to call into the primitives provided by the LargeInteger plugin. In RSqueak/VM, all arithmetic primitives succeed also for the special one-word positive integers, allowing fast arithmetic on any integer that fits into a machine word.

This optimized representation of word-sized positive integers has improved the relative performance of the simulation on RSqueak/VM compared to the C plugins. For the BitBlt Blue Book benchmark that renders the first 16 combination rules to the screen, the simulation in RSqueak/VM is now faster than the generated plugin in C by about one order of magnitude. For Balloon, our previous worst case rendering performance for TrueType fonts was more than 180 times slower than the Balloon plugin compiled to C [5]. The optimization detailed above has reduced this to less than 20 times.

A disadvantage of this architecture are more checks when converting RPython integers into Smalltalk integers and back: the VM-level class that has to be used depends on whether the result fits into a small integer, a word-sized unsigned integer, or needs to be represented as bytes. However, these classes then provide a unified interface so the rest of the code can ignore the distinction.

* * *

Figure 3 presents various benchmarks specific to Squeak and taken from the computer language shootout. The entire suite of benchmarks is available online[5]. The benchmark results represent the average of the middle 95 % measured over 100 benchmark runs and normalized to the Spur Squeak VM version 16.18.3692. The RSqueak/VM version we used was c6487cb7d1. All benchmarks were run on an otherwise idle Ubuntu machine with an Intel Core i5-3470 @ 3.2 GHz and 8 GB of RAM. The benchmarks show that while RSqueak/VM is much slower for some applications, it is competitive and sometimes even faster than Spur for others. The areas where RSqueak/VM is much slower are of three types: First, benchmarks such as Blowfish and font rendering rely on in-image fallback code where the Spur VM uses a plugin that was compiled to C and is linked into the VM. Second, benchmarks such as Graph Search and Binary Trees are tree-recursive, and the RPython JIT is bad at optimizing tree-recursive algorithms. Third, benchmarks such as megamorphic sends or NBody do not work well with the JIT, because it speculates that most call sites have very few receiver types.

6. Inexperienced Developers and RSqueak/VM

We argue that the RSqueak/VM architecture makes it easy for new developers to extend the VM even without much guidance. While it is difficult to evaluate this, we want to present anecdotal evidence to support our claim.

6.1 Adding Support for Objects as Methods

Objects as methods were introduced in Squeak as a mechanism to change lookup semantics from within the image. The contract is that, when the VM encounters an ordinary object (rather than a compiled method) in the method dictionary during lookup, it sends it the special selector `#run:with:in:` providing the original selector, arguments, and receiver.

Objects as Methods were added in an introductory VM development seminar by four students. The seminar lasted for three months with about one day per week spent working on the project.

Changes to RSqueak/VM For current images, implementing this in RSqueak/VM is straightforward. The method `sendSelector` implemented in the interpreter calls `lookup` on the receiver's class. If the result of the lookup is not a compiled method, the arguments are taken off the stack, wrapped in a Smalltalk array, and the selector, argument list, and receiver are used as arguments for a new send.

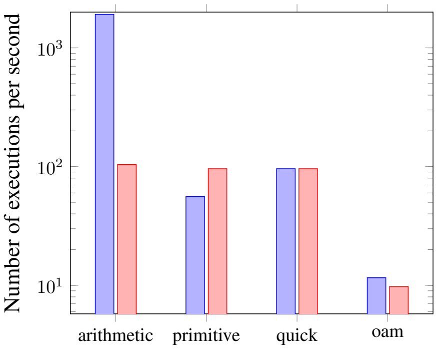


Figure 4: Objects as Methods: Send performance comparison between RSqueak/VM (blue) and the Spur Squeak VM (red).

One issue for the students was to take care of older images. The `#run:with:in:` selector is a special object in images that support objects as methods. With RSqueak/VM, we support images before that as well, so the students hat to make sure that the VM does not crash trying to fetch that selector and sending it.

Benchmarks Figure 4 shows the performance of the students' implementation. We compare the send performance for a method doing simple arithmetic (returning the result of 1 + 1), a method that calls the identity hash primitive, a method that simply returns self, and an object as method that also returns the receiver.

Without any prior experience or specific optimization, the students achieved roughly the same performance in this benchmark as the standard Squeak VM, and have the same comparative slowdown of about one order of magnitude. For us, this demonstrates that experiments with lookup, for example, could be done cheaply on RSqueak/VM and be evaluated immediately without having to spend much time on performance.

6.2 Multi-core with Software Transactional Memory

RPython includes a module to aid developers in creating VMs that use STM for concurrency. STM is a lock-free concurrency mechanism in which each thread executes on its own transactional view of the global memory space. At select times (or by default after each bytecode), the transactions from different threads are considered for merging into the global view. If there are any conflicts, some threads roll back and re-execute the set of operations since the last merge. This transparent approach to concurrency is interesting to let Squeak processes run truly parallel on different cores of a multi-core machine with only minimal changes to the image.

[5] `http://hpi.de/swa/squeaksource/BenchmarkRunner`, accessed May 20, 2016

The work to add STM concurrency was done as a master's project by four students. The project lasted for four months with about one and a half days per week spent working on the VM.

Changes to RSqueak/VM The core adjustments to support STM were inside the VM and are completely transparent for the user of the image. Since an STM implementation is available as an RPython library, all that was required was a change to the scheduler (which is a special object in RSqueak/VM) and a change to the bytecode loop in the interpreter to commit the transaction. Additionally, the students added primitives so that Smalltalk code can influence the behavior of the STM. One primitive added the capability to explicitly launch a process as a separate operating system thread rather than a green thread that is scheduled cooperatively. With STM's automatic transaction boundaries this primitive already allows concurrent execution of threads without any further changes to the Smalltalk code. However, there are cases were the application relies on the fact that a bigger group of changes atomically executed either completely or not at all. To let the application mark the beginning and the end of such atomic blocks, the students furthermore added two more primitives.

Benchmarks Figure 5 shows the students' implementation on two parallel benchmarks with different numbers of threads. The first benchmark calculating the sum of all numbers was run twice: first summing up the numbers to 10,000,000 and then to 1,000,000,000. This benchmark is embarrassingly parallel, but it shows that the parallelism can actually pay off for sufficiently large problems if run with the same number of threads as there are cores in the underlying machine. For the smaller run, the overhead of thread and transaction management never pays off however. Nevertheless, it does appear that unmodified multi-threaded algorithms in Smalltalk can in principle benefit from transparent parallelization with STM.

On the other hand, the Mandelbrot benchmark shows the limits of a transparent integration. This benchmark does calculations on distinct ranges of the same Smalltalk array. Without any further annotations, these lead to a lot of conflicts, rollbacks, and re-calculations in the different threads, a problem that gets worse as more threads are added. Here, the additional primitives to control atomic access must be used in the Smalltalk code.

7. Discussion and Conclusions

These and other student projects showed us that it is possible for students with no prior VM development experience to extend RSqueak/VM and create extensions that are worth including in the main branch of development without much further refactoring.

While the changes to the code base were straightforward thanks to abstractions we have in RSqueak/VM architecture, the students disliked other aspects of the development process.

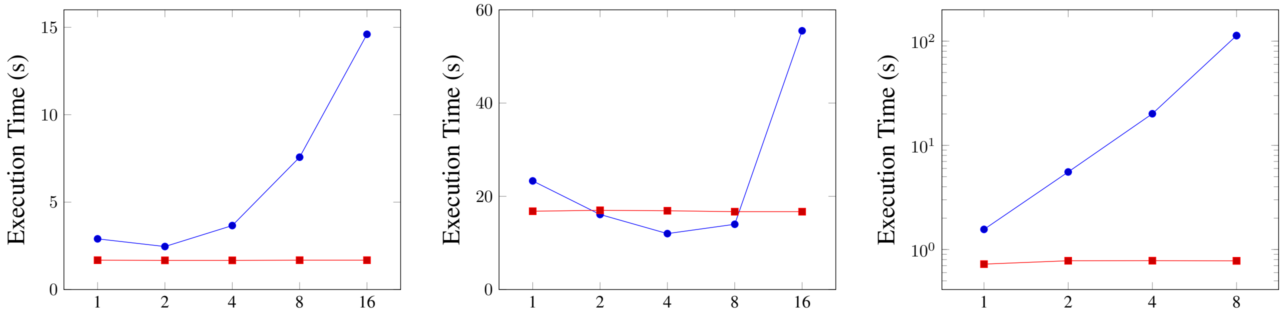
One problem with RPython compared to Slang is that it requires a full program translation to compile to C. Whereas Slang is a Smalltalk-subset that maps very directly to C and can thus be translated without much analysis, RPython's higher level abstractions require full program control flow analysis to generate C type annotations. Although RPython (just like Slang) can be run in interpreted mode for development, the fact that it has to load and simulate the entire image (including many primitives) makes this very slow for RSqueak/VM. The full translation often requires between five and ten minutes, and these long roundtrip times led to a tendency of bigger code changes ("Before we compile, let's also add this"), lost flow ("What where we doing before?"), and different compiled interpreters in parallel testing ("How is this version different from the others?"). As a consequence, while the students found it easy to extend the VM in appropriate places, they found it harder to test and correct errors than on the Cog VM.

Another limitation of RSqueak/VM is that it only works with Squeak. Fallback code for optional primitives has diverged in the different Smalltalk flavors that use the standard Squeak VM and limited time has prevented us from pursuing Pharo and Cuis support. We are planning to attempt to add support for those two systems in another student project.

Despite those limitations, we feel that RSqueak/VM meets our goals for a research VM. First, it is easy to introduce new developers to the code base because although RPython is a subset of Python, it is much more flexible than (for example) Slang and well-known object-oriented patterns in the codebase make the introduction easier. Second, the high-level JIT abstractions provided by RPython allow developers to achieving promising performance even without any knowledge of the underlying machine, by encoding domain knowledge directly.

Acknowledgments

We want to thank all of our students who contributed to RSqueak/VM development: Lars Wassermann, Anton Gulenko who wrote their master's theses about aspects of the VM. Patrick Rein, Johannes Henning, Toni Mattis, Eric Seckler who added experimental support for STM. Sebastian Gerstenberg, Sven Knebel, Jakob Reschke, Patrick Siegler who implemented support for objects as methods. Franz Liedke, Cornelius Bock, Sven Köhler who experimented with tagged versus wrapped integers. Hubert Hesse, Marcel Pursche, Benjamin Siegmund, Hannes Würfel who contributed a large number of benchmarks. Bastian Kruck, Matthias Springer, Dustin Glaeser, Jakob Reschke who added support for the Spur format and worked on using Smalltalk code to write the image to disk. Daniel Werner, Fabio Niephaus, Philipp Otto, Felix Wolff, Jan Graichen who continue to experiment with other data layouts and data persistence for RSqueak/VM. We also want to acknowledge the authors of the original SpyVM from which RSqueak/VM was initially derived for



(a) Parallel Sum 10^7 (b) Parallel Sum 10^9 (c) Mandelbrot

Figure 5: Different Benchmark Runs with (blue circles) and without STM (red squares)

their first demonstration that a Smalltalk VM could be written in a short time using RPython: Carl Friedrich Bolz, Armin Rigo, Adrian Kuhn, Adrian Lienhard, Nicholas D. Matsakis, Oscar Nierstrasz, Lukas Renggli, and Toon Verwaest.

References

[1] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week-implementing a smalltalk vm in pypy. In *Self-Sustaining Systems*, pages 123–139. Springer, 2008.

[2] C. F. Bolz, A. Cuni, M. Fijałkowski, and A. Rigo. Tracing the meta-level: Pypy's tracing jit compiler. In *Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, pages 18–25. ACM, 2009.

[3] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, pages 167–182. ACM, 2013.

[4] T. Felgentreff, T. Pape, R. Hirschfeld, A. Gulenko, and C. F. Bolz. Language independent storage strategies for tracing JIT based VMs. In *Proceedings of the 11th ACM Symposium on Dynamic Languages*, DLS '15, pages 119–128. ACM, 2015. doi: 10.1145/2816707.2816716.

[5] T. Felgentreff, T. Pape, L. Wassermann, R. Hirschfeld, and C. F. Bolz. Towards reducing the need for algorithmic primitives in dynamic language VMs through a tracing JIT. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS'15. ACM, 2015.

[6] B. Freudenberg, D. H. Ingalls, T. Felgentreff, T. Pape, and R. Hirschfeld. Squeakjs: A modern and practical smalltalk that runs in any browser. In *Proceedings of the 10th ACM Symposium on Dynamic Languages*, DLS '14, pages 57–66, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3211-8. doi: 10.1145/2661088.2661100.

[7] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[8] D. H. H. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. *SIGPLAN Not.*, 32(10):318–326, Oct. 1997. ISSN 0362-1340.

[9] E. Miranda and C. Béra. A partial read barrier for efficient support of live object-oriented programming. In *Proceedings of the 2015 International Symposium on Memory Management*, ISMM '15, pages 93–104. ACM, 2015. ISBN 978-1-4503-3589-8. doi: 10.1145/2754169.2754186.

[10] A. Rigo and S. Pedroni. PyPy's approach to virtual machine construction. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, OOPSLA '06, page 944–953. ACM, 2006. ISBN 1-59593-491-X. doi: 10.1145/1176617.1176753.