Towards Reducing the Need for Algorithmic Primitives in Dynamic Language VMs Through a Tracing JIT

Tim Felgentreff Tobias Pape
Lars Wassermann Robert Hirschfeld
Hasso Plattner Institute,
University of Potsdam
{firstname.lastname}@hpi.uni-potsdam.de

Carl Friedrich Bolz
Software Development Team,
King's College, London
cfbolz@gmx.de

Abstract

When implementing virtual machines, besides the interpreter and optimization facilities, we have to include a set of primitive functions that the client language can use. Some of these implement truly primitive behavior, such as arithmetic operations. Other primitive functions, which we call *algorithmic primitives*, are expressible in the client language, but are implemented in the VM to improve performance.

However, having many primitives in the VM makes it harder to maintain them, or re-use them in alternative VM implementations for the same language. With the advent of efficient tracing just-in-time compilers we believe the need for algorithmic primitives to be much diminished, allowing more of them to be implemented in the client language.

In this work, we investigate the trade-offs when creating primitives, and in particular how large a difference remains between primitive and client function run times in VMs with tracing just-in-time compiler. To that end, we extended the RSqueak/VM, a VM for Squeak/Smalltalk written in RPython. We compare primitive implementations in C, RPython, and Smalltalk, showing that due to the tracing JIT the performance gap can be significantly reduced.

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

Keywords Algorithmic Primitives, Tracing JIT, Squeak/Smalltalk

ICOOOLPS'15,, July 06 2015, Prague, Czech Republic.

ACM [to be supplied]. . . $15.00.
http://dx.doi.org/10.1145/2843915.2843924

1. Introduction

Dynamically typed programming languages are traditionally implemented as virtual machines (VMs). As the popularity of VM-based languages has increased, VM implementation techniques have evolved to the point where most mature VMs utilize some form of just-in-time (JIT) compilation, with two main technologies contending at the moment: method JITs [12], which generally compile methods at a time into machine code, and utilize techniques such as inlining and branch pruning to simplify the code or merge multiple methods; and tracing JITs [1], which record the actions of the program including branches taken and methods dispatched into a specific *trace* and compile it into machine code.

A large part of VMs are basic behavioral blocks of the implemented language, called *primitives*, natives, or builtins. Some of these primitives are inherently required, because their behavior cannot adequately be expressed in the implemented (*client*) language. But some primitives, which we call *algorithmic primitives*, exist only for performance reasons — since the implementation (*host*) language is usually lower-level and easier to optimize than the client language of the VM, it is assumed that primitives implemented in the lower-level host language are often faster than if they were implemented in the client language. We discuss this in more detail in Section 2.

Many VM-based languages have more than one VM implementation[1]. Some of those language implementations are not (only) source compatible, but may also share an instruction set — typically bytecode — or a standard library. As a result, every primitive in the reference implementation has to be re-implemented in all the other implementations. Furthermore, the primitive implementations cannot usually be debugged from the high-level language — if a bug is suspected in a primitive implementation, the debugging process has to cross language boundaries and can become more cumbersome.

[1] Python, Jython, and PyPy; Ruby, JRuby, and Topaz; HotSpot, Dalvik, and Jikes; . . .

Thus, we argue that it is desirable to reduce the number of primitives to ease VM development and maintenance.

In this work we investigate the trade-offs around implementations of algorithmic primitives. Our investigation was prompted by a trend that we have seen in other work to move behavior that used to be implemented as part of the VM into the client language (Section 5) and makes the following contributions:

- We discuss possible trade-offs when implementing behavior as primitive in contrast to in the client language in Section 3.
- We quantify the performance gap between functionality implemented as primitive in contrast to in the client language in Section 4.

2. Primitive Building Blocks in Virtual Machines

Virtual machines primarily provide a runtime for a client language. In addition to that, they typically provide certain pieces of code that are to be invoked from the client language. The most common primitive behaviors are often available as part of the instruction set, e. g. the integer addition bytecode on the Java virtual machine (JVM). Other primitives often expose the same calling interface as functions of the client language; primitive invocation is hence typically transparent to user code. Depending on the language, primitives are sometimes more selective regarding what arguments they can handle and cannot be inspected in the same way as ordinary methods, given such reflective facilities exist in the client language.

In languages including Python, Ruby, Java, and JavaScript, calling primitives is like calling any function, the only difference being that the source code is not given in the client language. Primitives in such languages are usually documented in text form, explaining their purpose, signature, and peculiarities such as possible side-effects. In Self [6], primitives are indicated by their selectors — methods whose slot name starts with underscore (_) call a primitive function.

To explore and evaluate the trade-offs for implementing primitive functions, we use Squeak/Smalltalk running on top of the RSqueak/VM [3], an open-source[2] implementation that uses the RPython meta-tracing just-in-time compiler (JIT) toolchain which also powers PyPy [4].

In Squeak/Smalltalk, a method can have an annotation that indicates a primitive call by number or name. When a method is to be executed that is denoted as primitive, the VM dispatches to the primitive function instead of running the method's code. Only if the primitive fails, either because it is not provided by the VM or because the primitive code indicates a failure, the VM invokes the Smalltalk method. Commonly, the Smalltalk code either generates a runtime exception or emulates the primitive behavior.

[2] https://github.com/HPI-SWA-Lab/RSqueak

In the example below, whenever the method `bitXor:` is called, line 3 declares that the primitive function `primDigitBitXor` from the `LargeIntegers` module should be called. The method body is *fallback code* — it is only run if calling the primitive fails. Here, the Smalltalk code implements the same behavior as the primitive — the primitive call is only a performance optimization. We call this an *algorithmic primitive*.

```
1 bitXor: n
2   | norm |
3   <primitive: 'primDigitBitXor' module:'LargeIntegers'>
4   norm := n normalize.
5   ↑self
6       digitLogic: norm
7       op: #bitXor:
8       length: (self digitLength max: norm digitLength)
```

The original Squeak VM [13] is written in a subset of Smalltalk called *Slang*. This code is statically translated into C and compiled to create the virtual machine. We chose Squeak to evaluate the trade-offs of implementing primitives in the VM, because the code for most algorithmic primitives already exists in a form that is executable in the client language, since Slang is a subset of Smalltalk.

The motivation of Slang is enabling Smalltalk developers to develop the VM using a familiar language and familiar tools. However, in practice the limits of Slang compared with Smalltalk proper can get in the way of this ideal; Slang does not support polymorphism — or even instances, for that matter — it supports only primitive uses of block closures for branching, and its semantics for array access and bit operations are subtly different, since they are mapped to those of C. Furthermore, although the Slang code is executable, to distribute changes to the VM it still needs to be translated to C and compiled for different platforms. For changes in pure Smalltalk, on the other hand, a convenient update mechanism exists in-image. It would thus be useful to move algorithmic primitives out of the VM, since development already happens at the Smalltalk level, and cutting out the translation to C not only removes compilation and makes distribution of the changes easier, it also serves to remove the restrictions of Slang when implementing the algorithms.

Removing algorithmic primitives from the VM would also serve to reduce code-duplication in Squeak. Many methods (such as `bitXor:` above) already have *fallback code* that implements the same behavior as the Slang code, often written in more idiomatic Smalltalk. The RSqueak/VM and other alternative VMs could automatically share the fallback and the Slang code. Only if none exists or it is inherently necessary to provide a primitive on the VM level, the RSqueak/VM would have to re-implement the primitive function. There are many ways to group primitives into those that require re-implementation and those that do not — we present here a grouping that we have found useful to make that decision while implementing the RSqueak/VM.

Memory Primitives This set of primitives is closely related to the memory design of the programming language. In object-oriented languages, memory primitives may be used to create new objects, reflectively access their fields or their size, trigger garbage collection, or manipulate their pointers. Depending on the language, not all of these are available as primitives, and their function may only be accessible through bytecodes.

Mathematical Primitives We count in this group of primitives arithmetic operations (including addition and multiplication, trigonometric and exponential functions, and numeric equality and inequality), as well as binary operations including or, xor, bit-shift, and bit-invert. These are implemented directly on the CPU and can be accessed through machine instructions.

System Call Primitives Most VMs provide access to system resources such as I/O devices, the system clock, or a random number generator to the client language. These VMs typically abstract application programming interfaces (APIs) of different operating systems to allow operating system independent access to system resources. These functions are usually implemented in the standard library of the operating system and the VM only needs to convert arguments and return values appropriately.

Foreign Function Interface Most VMs offer a way to interface with existing libraries. The greatest common denominator among foreign function interfaces (FFIs) are C calling conventions. A FFI usually requires explicit naming of the library to find the functions in, and marshalling for function arguments and demarshalling of return values. While in primitives, the callee is responsible for type conversions, in a FFI call, the caller has to explicitly convert types.

VM Specific Primitives Depending on the implementation, the VM might offer additional entry points, for example to register callbacks for mouse and keyboard handling, for green threading, timed behavior, or reflection on VM internals such as garbage collection information.

Algorithmic Primitives Finally, some primitives implement behavior that can be expressed in the client language, but doing so was considered to be too slow. This group of primitives can include, for example, operations on arbitrarily large integers, string and regular expression operations, or sorting, searching, and cryptographic algorithms.

***

The last group of primitive is of particular interest to implementers of an alternative VM, because it may be possible to omit them and still get a functional implementation of the language. Primitive functions are generally considered to be faster than the same function implemented in the client language, but one common goal of VM re-implementations is to use current optimization techniques to improve the execution speed of the client language. In this work, we use the RSqueak/VM to explore if current tracing JIT techniques improve the execution speed of pure Smalltalk to the point where some algorithmic primitives may be replaced with client language implementations with acceptable performance.

3. Trade-Offs Between Implementation Locations For Language Algorithms

Implementing behavior in the client language rather than as a primitive can allow easier implementation and debugging, better code reuse, and changeability — in short all the benefits that the high-level client language ought to have over more low-level languages. However, doing so may adversely affect speed, security, and subtle execution semantics, which can be a concern for often used functions.

However, changeability and performance are probably of highest importance among the six.

Ease of Implementation Most VMs provide a high-level programming language implemented in a lower-level language. They incur a performance overhead for the promise of easier implementations through higher expressiveness, better abstractions and more powerful tools. Likewise, implementing an algorithm in a higher-level, VM-based language typically requires less code, has shorter round-trip times than in a lower-level, ahead-of-time (AOT)-compiled language, and allows using advanced debugging tools that take advantage of the language abstraction and the introspection capabilities of the VM. In contrast, implementing behavior as algorithmic primitives requires rebuilding the VM during development. To debug algorithmic primitives, developers have to debug both in the client language as well as the host language, likely using different tools at different abstraction levels that may not be built to be used in conjunction.

Reuse Different VMs implementing the same language can share all code implemented in the client language. Hence, re-implementations of an existing VM need to only re-implement fewer primitives. Moreover, improvements made to the standard library can be shared directly across different VMs without additional adaptations for each VM.

Code reuse can also prevent subtle bugs from creeping into different implementations of the same functionality. In Squeak, for example, some primitives are optional — while implemented in the standard VM, fallback code in Smalltalk implements that same behavior. This is useful to get alternative implementations working quickly, but is also a source of hard to track bugs when code changes in the primitive implementation are not adequately reflected in the fallback code. A similar situation arises in JavaScript, where not all VMs implement the same set of features. Pure-JavaScript libraries, commonly called *polyfills*, are used to support older or non-compliant browsers.

Changeability Changing behavior implemented as a primitive usually requires recompiling of the VM. Since rebuilding requires additional sources and a build environment, this discourages improving primitives in general and negatively affects the round-trip times between editing the code and seeing the changed behavior. In order to see the results of a VM change, it has to be recompiled and externally tested. If the source is edited in the client language, the VM can recompile that code and the primitive can be tested more quickly.

Speed Reduced execution time is the main reason for creating primitives which could also be implemented in the client language. The reasoning is that every method call needs a new frame, which (depending on the language) may mean costly method lookups, saving frame and stack pointer, copying the arguments to the new frame, initializing local variables, or repeated boundary checking, boxing, and unboxing of basic values used in the computation of the algorithm.

On the other hand, inlining JITs with escape analysis can omit many of those operations if they can look into the implementation of the algorithm, which is not possible if the algorithm is implemented as a primitive. Thus algorithmic primitives prohibit these kinds of optimizations across their boundaries.

Security Many dynamic languages allow *monkey-patching*, a practice by which behavior in other libraries including the language's standard library can be overwritten by user code. Fixing the behavior in form of a primitive may prohibit maliciously monkey-patched code from affecting security critical algorithms. The computation within a primitive is usually hidden from the client language: it cannot be reflectively inspected, and usually a thread cannot be interrupted within a primitive method. In a closed source VM, the behavior of the primitive can be documented only externally, not as part of the code. This may make it harder to exploit security issues in the algorithm, but is unlikely to. The benefit may be undone because the lower-level VM implementation language may be prone to different, potentially more security critical issues such as buffer overflows and out-of-bounds memory accesses which are harder to occur within a managed VM runtime.

Behavior Some behavior cannot be easily expressed in the client language or its semantics differ from an implementation as primitive. Properties or behavior such as endianness and overflow, bit-wise operations, or string encoding, that are naturally platform specific in a low-level language such as C, have to be explicitly implemented to behave in the way that is correct for the given platform if they are implemented in the client language. In this case, the client implementation may be more complex and thus more error-prone than an implementation as a primitive.

4. Measuring the Performance Gap

We explore the performance impact of running common algorithmic primitives in pure Smalltalk and give explanations of our results.

4.1 Algorithmic Primitives in Squeak

We present six benchmarks: generating a DSA key, filling an array with a constant value, filling a string with a constant character, generating the SHA of a long string, compositing the screen, and rendering TrueType fonts. These are meant to show real-world usage of primitives that RSqueak/VM executes in pure Smalltalk: large integer arithmetic, preparing memory with constant values, string operations, compositing 32 bit graphics including transparency, and rendering graphics with Bézier shapes.

All of these primitives have implementations in Slang that are translated to C and compiled into the *Cog* VM, a Squeak VM with a single-stage method JIT, and into the Interpreter Squeak VM. On RSqueak/VM, all but the compositing and rendering primitives use fallback code written in pure Smalltalk that runs when the primitive fails. The compositing and rendering primitives in RSqueak/VM execute the Slang code directly, which is a little more expensive as the Slang simulation simulates C semantics for things like array pointers and bit-wise operations. We explicitly measure compiled C code against running pure Smalltalk code here — implementing the primitives in the client language is only an option if performance is still acceptable[3].

4.1.1 Large Integer Arithmetic

Most dynamic languages have a way of transparently overflowing from the machine word size to a larger integer type and so does Squeak. The `SmallInteger` class represents 32 bit tagged integers — operations that exceed the tagged range return a `LargePositiveInteger` or `LargeNegativeInteger`. The arithmetic on small integers is handled using *arithmetic primitives*. Large integers are represented as lists of bytes and arithmetic on them is handled using specially crafted primitive functions. However, the algorithms for byte-wise operation are also available in Squeak.

In the RSqueak/VM, we avoid having to implement large integer primitives in two ways: first, we use an optimized internal representation for large positive integers that fit into an unsigned 32 bit number. Byte-wise access is simulated, but arithmetic is done on the unsigned value. Second, we optimistically call the small integer primitives even for large integers that can fit into a 64 bit integer. Quite many do, and thus we pay only the price of converting from the unsigned 32 bit integer or the byte representation to a 64 bit value. For the remaining operations, we run the fallback code.

[3] Running the simulated code on the Cog VM is between one and two orders of magnitude slower than RSqueak for these benchmarks, with the interpreter even worse, but that is not the comparison we are interested in for this work.

The implementation in Squeak is based on byte-wise operations. There are a number of different large integer libraries, each with their own balance between speed and implementation complexity. We observed that very large integers are currently rarely used in Squeak, and the Slang code is not well optimized for them. Given this, we consider an implementation that is sufficiently fast for word-sized to double-word sized integers to be acceptable. Omitting large integer primitives also allows to easily evolve the algorithms or change the representation of large integers, should performance for larger numbers be an issue.

As a benchmark for large integers, we generate keys according to the Digital Signature Algorithm [16].

4.1.2 Filling an Array

Squeak has a number of primitives to quickly fill collections with the same value. One such primitive is number 105, which for arrays and strings has the following definition:

```
replaceFrom: start to: stop with: rep startingAt: repStart
    <primitive: 105>
    super replaceFrom: start to: stop
          with: rep startingAt: repStart
```

`SequenceableCollection`, a superclass of both types, has a generic implementation in Smalltalk:

```
replaceFrom: start to: stop with: rep startingAt: repStart
    | index repOff |
    repOff := repStart - start.
    index := start - 1.
    [(index := index + 1) <= stop] whileTrue: [
      self at: index
           put: (rep at: repOff + index)]
```

Regarding our trade-offs, omitting this primitive comes down to ease of implementation. Squeak has specialized arrays for words, bytes, and characters, besides arrays that hold pointers to arbitrary objects. These all implement the same interface with respect to element access, and thus can share the Smalltalk implementation. However, the primitive implementation has to explicitly handle each possible array type, because polymorphic dispatch is unavailable at that level, making the primitive implementation more cumbersome. We measure the impact of omitting this implementation by filling variable sized arrays and strings with constant values.

4.1.3 The SHA Algorithm

Squeak implements the Secure Hash Algorithm (SHA) from the U.S. government's Secure Hash Standard (SHS) [15], based on an implementation by Bruce Schneier [17]. The secure hash standard was created with 32 bit hardware in mind, so all arithmetic in the hash computation must be done modulo 2^{32}. The Squeak implementation was originally created as pure-Smalltalk and uses objects to simulate 32 bit hardware registers. Squeak's SHA implementation explicitly avoids using large integers. Due to the known size limit on the numbers, specially crafted objects are faster than overflowing into large positive integers (strengthening our previous argument that large integer arithmetic, if left in Smalltalk, could be optimized based on the use-cases of developers). The default Squeak VM includes a set of primitives to directly generate the hash in C, which translates into a three orders of magnitude speedup.

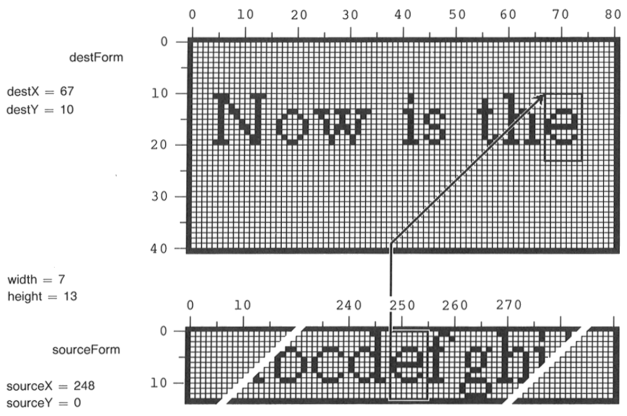

Figure 1. BitBlt transfers rectangular areas from one array to the other. In this image, a paragraph is composed from letters copied from the font form [10, p. 335].

Implementing this algorithm as a primitive made sense to get the significant speedup on the original Squeak VM. However, we argue that the performance of running the pure Smalltalk code is much improved with the tracing JIT of RSqueak/VM. Another consideration may be that ease of implementation and changeability is no argument for maintaining the implementation in Smalltalk: the algorithm is fixed and a canonical implementation exists in C. Then, again it may be easier to study and understand the code in Smalltalk.

4.1.4 Transferring Bits to the Screen

BitBlt is an algorithm that combines one block of bits with another according to certain rules, and was developed to speed up graphical operations for one bit displays [11], but later extended to handle up to 32 bit of color [13]. BitBlt has several rules for combining blocks of bits for blending or replacing, as well as combining bit-areas of different color depths. The Squeak screen is entirely composed of `Form` objects that are composited together using BitBlt and drawn to the screen. Drawing to the screen is the same as drawing to any other form. A primitive is used to designate a form as the display, and the VM is expected to mirror it to the screen. Forms are *word objects*, arrays which hold only unsigned integers. As an example (Figure 1), to display a text paragraph, the required letters are copied from a form that has the entire pre-rendered font, to a paragraph form. The letters are also used a mask for syntax highlighting by blending with

Table 1. All benchmark results. We give means of the execution time along with a 95 % confidence interval.

Benchmark	RSqueak	Cog	Interpreter
	mean	mean	mean
arrayFillArray	586 ±7 ms	378 ±2 ms	920±58 ms
arrayFillString	1079 ±80 ms	419 ±2 ms	945±12 ms
mandala	25 ±3 ms	12 ±1 ms	14 ±0 ms
renderFont	4514 ±45 ms	17 ±0 ms	25 ±0 ms
dsaGen	3794±377 ms	411±54 ms	405±52 ms
shaLongString	862 ±19 ms	47 ±1 ms	58 ±1 ms

a uni-colored form. Finally, the paragraph form is then copied to the display form.

BitBlt was originally published as Smalltalk code [10]. It is now mostly in Slang code, but some fast path optimizations were only done in C, with the Slang code having only stubs that delegate to the slow code paths. Furthermore, the Slang code for BitBlt uses custom `CArray` objects that simulate C array semantics — these allow fast access to arrays when translated to C, but incur additional overhead during simulation in Smalltalk. Finally, although word objects contain unsigned 32 bit integers, reading from them in Smalltalk returns either `SmallIntegers` or `LargePositiveIntegers`, depending on their size. This increases the type variety and thus puts additional pressure on the optimizing JIT. Concerning our trade-offs, the version history of BitBlt demonstrates that indeed this is a primitive that changes relatively often, as new optimizations or blending modes are introduced. However, it is also crucial that the BitBlt operations run fast enough to provide a smooth user interface.

4.1.5 Font Rendering

Squeak includes Balloon, a rendering engine for 2D and 3D graphics. The 2D engine provides anti-aliasing and vector graphics in software. It does most of its rendering using a contiguous chunk of memory and makes heavy use of matrix transformations to draw curves and polygons. Once it finishes rendering, it calls BitBlt to transfer the actual bits from the work buffer to the screen. Balloon is very flexible and provides 44 primitive functions that can be combined to show SVGs, draw TrueType fonts, play Flash 3 animations, and render Bézier curves.

Implementing Balloon in the VM or running it in Smalltalk must weigh performance against changeability. The algorithms used in the Balloon Engine are heavy on numeric operations and their performance benefits from being implemented in the VM. At the same time, Balloon is a rather large module with more than 5 000 lines of Slang code (for comparison, all large integer primitives comprise less than 1 400 lines, and the SHA primitives comprise less than 250 lines of code). Re-implementing Balloon is thus a major burden on an alternative VM implementation. As to changeability, although Balloon set out to support multimedia formats, progress on updating it for current formats has stalled, with Flash support at version 3. One reason for this may be that the code is not in Smalltalk (and not even fallback code exists), and thus experimenting with extensions is difficult. To evaluate this trade-off, we measure the performance of rendering TrueType glyphs.

Running Primitives

We ran our benchmarks on a 64 bit Ubuntu 14.10, running a 3.16.0 Linux kernel. For translation, we used PyPy revision 223e8c97aec from February 19, 2015, and 32 bit GCC 4.8.2. The revision of RSqueak/VM was e235be9, the version of the Cog VM was 4.0-2585, and the version of the Squeak Interpreter VM was 4.10.2-2614. The benchmark ran on an Intel Core i7-4650U at 1.7 GHz with 3.94 GiB of RAM. The benchmarking script and image is publicly available online.[4] The Squeak image version is 4.6-14318.

We report wall times measured *in-system* using the Smalltalk method `Time class>>#millisecondsToRun:`. We remove the impact of JIT warm-up from our measurements by running each benchmark 200 times in a freshly booted image, and using the last 20 results [2]. Except for the font rendering benchmark, all benchmarks run headless. All benchmarks also run single-threaded. We show the arithmetic mean of all runs along with bootstrapped [7] confidence intervals for a 95 % confidence level.

Table 1 gives the raw benchmark results and Figure 2 gives the results in terms of relative speed against the standard Interpreter VM. We can see from the benchmarks that RSqueak/VM has a larger variance in general, as compared to the other two VMs. The last column of Figure 2 also shows that the mean slowdown of RSqueak/VM over all benchmarks is just under a factor of six, and thus within an order of magnitude. RSqueak/VM achieves by far the worst performance in the Balloon benchmark.

4.2 Explaining the Gap

The time differences between Interpreter VM and Cog are likely a result of Cog's method JIT, which is very simple: it compiles each method separately and has only very limited support for advanced features such as method inlining or escape analysis.

arrayFillArray and arrayFillString These two benchmarks shows that the overhead of filling large chunks of memory with the same value is small. The overhead is

[4] `https://github.com/HPI-SWA-Lab/RSqueak-Benchmarking`

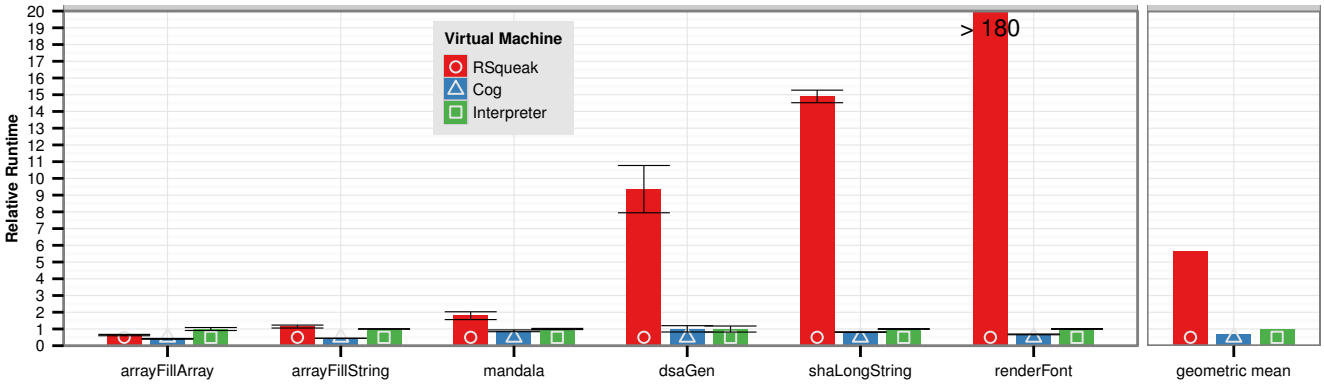

Figure 2. Algorithmic primitives benchmarks, with execution time normalized to Interpreter. Lower is better.

mostly due to additional guards in the compiled code from RSqueak/VM. In contrast to the C code, the JIT cannot rely on the fact that the Smalltalk methods used in the primitive implementation would not change. Furthermore, another, higher-priority thread could interrupt the benchmark and exchange the array or string that is filled using the `become:` method to exchange pointers, whereas a VM primitive on Cog and the Interpreter VM cannot be interrupted. It may be possible to further minimize the number of guards with structural changes to RSqueak/VM.

mandala and renderFont These benchmarks have very different slowdown, but mostly suffer from the same types of overhead. All three tested VMs allocate Smalltalk frames initially on the stack and only copy them to the heap if they are accessed reflectively from Smalltalk. In these benchmarks, this happens only for RSqueak/VM, because each primitive call to BitBlt in the `mandala` benchmark or to Balloon in the `renderFont` benchmark sets up the Slang code for simulation by copying arguments from the sender context into an `InterpreterProxy` object, the Slang interface to various Smalltalk facilities. Thus, every call into the method that sets up the Slang code is slow, currently. Second, both of these benchmarks rely on fast arithmetic. Although the tracing JIT can generally remove allocations of integer objects so that few object allocations are needed, allocations remain around entering inner loops and Slang code. Thus, besides the impact of simulating arithmetic for some large integers, RSqueak/VM suffers additional overhead and GC pressure from allocating numeric objects. Third, the Slang code was written with the intention of being translated to C. It uses special arrays and bit-operations that simulate C semantics in Smalltalk, incurring additional overhead.

The reason why `mandala` is only about a factor of two slower than the standard VM, whereas `renderFont` is three orders of magnitude slower, is that one BitBlt operation corresponds to only one call to the Slang code which then does all the work. One Balloon operation, in contrast, comprises dozens of very short calls into Slang code, each of which forces a few stack frames to the heap and a few numeric objects to be fully allocated. However, these problems may be greatly diminished if the primitive were not *simulated* in Smalltalk, but rather written in pure Smalltalk, without the overhead of an `InterpreterProxy` or simulating C semantics for array access and bit-wise operations.

dsaGen and shaLongString Both of these benchmarks suffer primarily from missing large integer primitives. Although the values used generally do not overflow into more than 32 bit, the fallback implementations for missing primitives use byte-wise and bit-wise access. As mentioned in Section 4.1.1, RSqueak/VM internally stores 32 bit positive numbers as unsigned integers and simulates byte access. In this case, however, this incurs additional overhead.

4.2.1 Threats to Validity

Re-implementing former primitives in Smalltalk exposes them to the regime of the garbage collector, which, for the RSqueak/VM, we cannot currently disable selectively. This might increase running times for re-implemented primitives, because all basic types, including integers, are boxed, and arithmetic operations repeatedly create those. On the other hand, if objects are created within a trace and not referenced from outside, the tracing JIT removes those unnecessary heap allocations. Therefore, the number of created objects after warm-up should be quite low. On Cog and the Interpreter VM, integers that fit within 31 bit are tagged and thus do not create pressure on garbage collector.

The RSqueak/VM is far from being a mature Squeak implementation and several features of the standard VM are still missing. Squeak defines a model of green threading, including interrupts triggered outside the VM, e. g. by mouse movements across the window, or due to timing. While Cog and the Interpreter VM support those, the implementation in the RSqueak/VM is still incomplete. We prevent the biggest of those scheduling related performance deteriorations by running headless, but the heartbeats used for timing are still active in all three VMs.

Finally, the presented benchmarks were written with a particular set of primitives in mind, but they exercise a significant portion of the Squeak core and should be considered macro-benchmarks. Thus, they should be taken to demonstrate heavy usage of specific algorithmic primitives within an application, rather than assessments of a single primitive feature.

5. Related Work

Other modern VMs with JITs have moved algorithmic primitives out of the VM and into the client language. The reasons given are mostly due to performance, but the benefits to ease of implementation and debugging that we have mentioned also apply.

5.1 JavaScript DOM-Operations

JavaScript is a dynamically typed programming language used in web-browsers. All JavaScript runtimes used in major browsers use some version of dynamic compilation. Some browser vendors have recently re-implemented at least some parts of their Document Object Model (DOM) operations in JavaScript to make them visible to the JIT, or plan to do so. Unfortunately, there have been few publications about this particular change, and none with performance results.

TraceMonkey [9], the JavaScript JIT formerly used in Firefox, also traces DOM-operations that were re-implemented in JavaScript[5]. Internet Explorer 10 supposedly executes DOM operations in JavaScript[6] and for Chromium, there are plans to re-implement the DOM-API in pure JavaScript[6]so the V8 engine can compile methods including DOM operations as one. For all these browsers, implementing the DOM-API in C++ meant that every call to DOM functions resulted in a runtime switch.

The re-implementation of browser specific DOM libraries in JavaScript improves the overall performance because it makes them visible to the JIT, as we described in Section 3. It allows the inlining of DOM operations instead of exiting just-in-time compiled code and having to assume that all arguments can escape. The re-implementations were not done for increased code clarity or reuse.

5.2 PyPy Python Interpreter

The general way to introduce primitive behavior to Python outside of the VM are C-extensions. While the PyPy Python interpreter tries to maintain C-extension compatibility, they explicitly recommend re-implementing extensions in Python[7]. The RPython-based Topaz Ruby implementation mirrors this sentiment in their recommendations when to implement functionality in Ruby and when in the VM[8], but lacks data.

The reasons for re-implementation the PyPy project gives are twofold. First, a missing visibility to the tracing infrastructure results in dropping out of traces for any calls to C-extension functions. Second, the Python C-extension API allows access to garbage collection implementation specific details. CPython uses a reference counting garbage collector. Since PyPy does not, the counters have to be emulated, which can impact performance.

5.3 Maxine Snippets

Writing primitive behavior in the same language as the non-primitive user code has also been done in the Maxine VM [18]. This Java VM is itself written in a slightly augmented Java. The implementation of primitives is hence also done in Java via so-called *Snippets*, small pieces Java code that gets translated to machine code using the existing JIT during the compilation of the VM. However, at execution time, the thusly specified primitives are indistinguishable from C or Assembler written ones. They can neither be inspected nor changed.

6. Conclusions and Lessons Learned

Through this work we have gained insights into some of the trade-offs mentioned in Section 3. The points raised here are not limited to implementing algorithmic primitives, but some are specific to meta-tracing JITs and PyPy.

Regarding the development process, we found that debugging primitives at the Smalltalk level generally works well. BitBlt presented the most problems. Debugging Smalltalk requires the Smalltalk debugger to be rendered, but all rendering depends on the BitBlt primitive. Once we had a working version, however, we were able to incrementally tweak the BitBlt implementation from within Smalltalk, easily reverting changes if they broke code paths. Overall, the Smalltalk BitBlt-implementation was both hardest and easiest to debug — hardest when rendering issues prevented the debugger from showing, because then we were unable to inspect the state of the system, and easiest when the issues were relatively minor (such as color issues with BGR versus RGB pixel formats), because we could directly inspect and fix these issues in the Squeak debugger without even having to restart the image.

Reusing Smalltalk code to replace primitives has made development of the RSqueak/VM easier. There are several primitives in Squeak that have a correct Smalltalk implementation in case the primitive fails. The Smalltalk code for the primitives we presented has been in development for a long time — the BitBlt code originated with Smalltalk-80 [10], the `replaceFrom:to:with:startingAt:` method is too old to have a correct timestamp, and the large integer operations

[5] `https://brendaneich.com/2008/08/tracemonkey-javascript-lightspeed/` (retrieved March 2, 2015)

[6] `http://www.chromium.org/blink#architectural-changes` (retrieved March 1, 2015)

[7] `http://doc.PyPy.org/en/latest/faq.html#do-cpython-extension-modules-work-with-PyPy` (retrieved March 3, 2015)

[8] `https://github.com/topazproject/topaz/issues/486` (retrieved April 7, 2015)

and hashing code are each more than 20 years old. Even in a community as small (compared to more mainstream languages like Ruby and Python) as Squeak/Smalltalk, there exist multiple different VM-implementations besides the one we already presented: the RoarVM [14] for running on multiprocessor systems, SqueakJS [8] which runs in the browser, Potato[9] for running on the JVM. To keep them compatible with the standard implementation as it evolves requires considerable effort from the VM maintainers, and thus most VMs have fallen behind. This effort can be lowered by replacing VM primitives with pure Smalltalk code.

Being able to change the primitives in Smalltalk also lowered the barrier to inspecting and thinking about improvements to the implementation. In contrast, changing the RSqueak/VM primitives requires a current checkout of the PyPy and RSqueak/VM sources, and build environment setup for 32 bit compilation. Similarly, changing the Interpreter VM requires an SVN checkout of the C sources and the Slang code[10]. In either case, changing the primitives requires superficial knowledge of the general VM structure and some familiarity with C or RPython.

***

In this work, we studied the impact dynamic trace compilation has on the decision to implement behavior as primitive. When deciding whether to implement behavior as primitive, programmers weigh ease of implementation, reuse, and changeability against speed, security, and platform behavior. Our results show that due to the advances of tracing JITs and VM implementation techniques, the speed argument has lost some of its weight, but there is still about one magnitude of speed difference remaining. However, we are confident that for some primitives this difference can be removed almost entirely, and we argue that the costs of implementing behavior primitive are palpable and not increasing the primitive count, or even actively reducing it, is worth considering.

Future work can go in several directions. RSqueak/VM itself has some potential for improvement. First, neither the fact that objects cannot change in size after creation, nor that there are some objects and methods which can be assumed constant during tracing are exploited by the VM so far. Second, although the standard Squeak VM is limited to 31 bit integers due to tagging, RSqueak/VM is not. However, the wider range is not currently exploited nor do we know if the image will break when the small integer range changes. Third, we are working to fully apply *strategies* [5] to RSqueak/VM. Many full objects used in primitive operations (such as forms, rectangles, and points) have only fields of the same types. Using strategies, these may be stored unboxed and thus help the JIT eliminate more object allocations and improve performance further.

[9] `http://sourceforge.net/projects/potatovm/` (retrieved March 29, 2015)

[10] `https://wp.me/p1sRgQ-2Y` (retrieved March 29, 2015)

References

[1] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *ACM SIGPLAN Notices*, volume 35, pages 1–12. ACM, 2000.

[2] C. F. Bolz and L. Tratt. The impact of meta-tracing on VM design and implementation. *Science of Computer Programming*, 98:408–421, 2013. ISSN 0167-6423.

[3] C. F. Bolz, A. Kuhn, A. Lienhard, N. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the future in one week — implementing a smalltalk vm in pypy. In R. Hirschfeld and K. Rose, editors, *Self-Sustaining Systems*, volume 5146 of *Lecture Notes in Computer Science*, pages 123–139. Springer Berlin Heidelberg, 2008. ISBN 978-3-540-89274-8.

[4] 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*, ICOOOLPS '09, pages 18–25. ACM, 2009. ISBN 978-1-60558-541-3.

[5] C. F. Bolz, L. Diekmann, and L. Tratt. Storage strategies for collections in dynamically typed languages. In *ACM SIGPLAN Notices*, volume 48, pages 167–182. ACM, 2013.

[6] C. Chambers, D. Ungar, and E. Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Not.*, 24(10):49–70, Sept. 1989. ISSN 0362-1340.

[7] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*, chapter 5. Cambridge, 1997.

[8] B. Freudenberg, D. H. 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.

[9] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6):465–478, June 2009. ISSN 0362-1340.

[10] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983. ISBN 0-201-11371-6.

[11] L. J. Guibas and J. Stolfi. A language for bitmap manipulation. *ACM Trans. Graph.*, 1(3):191–214, July 1982. ISSN 0730-0301.

[12] U. Hölzle. Adaptive optimization for SELF: reconciling high performance with exploratory programming. Technical report, Stanford University, 1995.

[13] 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.

[14] S. Marr. *Supporting Concurrency Abstractions in High-level Language Virtual Machines*. PhD thesis, Software Languages Lab, Vrije Universiteit Brussel, Pleinlaan 2, B-1050 Brussels, Belgium, 2013.

[15] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. National Institute for Standards and Technology, Gaithersburg, MD, USA, Apr. 1995. URL http://www.itl.nist.gov/fipspubs/fip180-1.htm.

[16] National Institute of Standards and Technology. *FIPS PUB 186-2: Digital Signature Standard (DSS)*. National Institute for Standards and Technology, Gaithersburg, MD, USA, Jan. 2000. URL http://www.itl.nist.gov/fipspubs/fip186-2.pdf.

[17] B. Schneier. *Applied Cryptography: Protocols, algorithms, and source code in C*. John Wiley & Sons, 1996.

[18] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, java. *ACM Trans. Archit. Code Optim.*, 9 (4):30:1–30:24, Jan. 2013. ISSN 1544-3566.