Tracing Algorithmic Primitives in RSqueak/VM

Lars Wassermann, Tim Felgentreff, Tobias Pape,
Carl Friedrich Bolz, Robert Hirschfeld

Technische Berichte Nr. 104

des Hasso-Plattner-Instituts für
Softwaresystemtechnik
an der Universität Potsdam

Universität Potsdam

Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für
Softwaresystemtechnik an der Universität Potsdam

Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam | 104

Lars Wassermann | Tim Felgentreff | Tobias Pape
Carl Friedrich Bolz | Robert Hirschfeld

Tracing Algorithmic Primitives in RSqueak/VM

Universitätsverlag Potsdam

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.dnb.de/ abrufbar.

Universitätsverlag Potsdam 2016
http://verlag.ub.uni-potsdam.de/

Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0)331 977 2533 / Fax: 2292
E-Mail: verlag@uni-potsdam.de

Die Schriftenreihe **Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam** wird herausgegeben von den Professoren des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam.

ISSN (print) 1613-5652
ISSN (online) 2191-1665

Online veröffentlicht auf dem Publikationsserver der Universität Potsdam
URN urn:nbn:de:kobv:517-opus4-91277
http://nbn-resolving.de/urn:nbn:de:kobv:517-opus4-91277

Zugleich gedruckt erschienen im Universitätsverlag Potsdam:
ISBN 978-3-86956-355-8

When realizing a programming language as virtual machine (VM), implementing behavior as part of the VM, as primitive, usually results in reduced execution times. But supporting and developing primitive functions requires more effort than maintaining and using code in the hosted language since debugging is harder, and the turn-around times for VM parts are higher. Furthermore, source artifacts of primitive functions are seldom reused in new implementations of the same language. And if they are reused, the existing application programming interface (API) usually is emulated, reducing the performance gains. Because of recent results in tracing dynamic compilation, the trade-off between performance and ease of implementation, reuse, and changeability might now be decided adversely.

In this work, we investigate the trade-offs when creating primitives, and in particular how large a difference remains between primitive and hosted function run times in VMs with tracing just-in-time compiler (JIT). To that end, we implemented the algorithmic primitive BitBlt three times for RSqueak/VM. RSqueak/VM is a Smalltalk VM utilizing the PyPy RPython toolchain. We compare primitive implementations in C, RPython, and Smalltalk, showing that due to the tracing JIT, the performance gap has lessened by one magnitude to one magnitude.

Contents

1 Introduction

Dynamically typed programming languages are usually implemented as process virtual machines (VMs). With the increasing popularity of these languages, VM implementation techniques have evolved to the point where most mature VMs utilize some form of dynamic compilation, such as a method just-in-time compiler (JIT), or a tracing JIT. Another large part of a VM are basic behavioral blocks, so called *primitives*. Some of these primitives are unavoidable, because their behavior cannot adequately be expressed in the implemented language. But some primitives, named *algorithmic primitives* henceforth, exist only because primitive functions evaluate faster than identical functions implemented in the hosted language (the language implemented by the VM).

Supporting and developing primitive functions requires more effort than maintaining code in the hosted language since debugging is harder, and the turn-around times for VM parts are higher. In contrast to programs implemented in the hosted language, source artifacts of primitive functions are seldom reused in new implementations of the same language. And if they are reused, the existing API usually is emulated, reducing the performance gains.

Additionally, when using tracing dynamic compilation, primitives should be visible to the trace recorder, to not circumvent the speed gains. Making existing primitives visible increases the effort to introduce such systems, and to implement new primitives in VMs with tracing JIT.

As a result, having algorithmic primitives being part of the VM impedes language reimplementation and VM maintenance.

In a tracing JIT, some of the optimizations from ahead-of-time compiler (AOT) work particularly good, and are well suited to optimize traces of primitives. This reduces the run time gap between behavior implemented as primitive and hosted. Especially type specialization, inlining, and loop unwinding are applicable when applied to former primitives, because the traces are volatile.

In this work, we investigate what trade-offs exist around primitives, and in particular how large a difference remains between primitive and hosted function execution times, when applying those optimizations.

In order to compare performance and implementation effort, we implemented two algorithmic primitives for RSqueak/VM [10] on the different abstraction levels available. RSqueak/VM is a Smalltalk VM using the PyPy RPython toolchain, and as such comes with a tracing JIT.

The first primitive, fill, is a simple function which sets all fields of an numeric array to the same constant value. The second, more complex primitive is BitBlt, which takes bits from one two dimensional bitfield and merges them with a rectangular part of another bitfield. BitBlt allows for bit-precise positioning and cropping. We implemented BitBlt not only in RPython and Smalltalk, but also connected and reused an existing C implementation.

Besides providing wall times and relative run times of repeated execution of those two primitives on the current standard Squeak VMs, we informally evaluate the other trade-offs. In particular, the run time difference has shrunk by one magnitude to one magnitude.

While there are attempts to reduce the amount of primitives in PyPy [17], and to reimplement existing primitives in JavaScript[27], to our knowledge there are no publications showing success, or explaining the reasons.

Recent achievements in tracing JIT optimization techniques have closed the speed gap between primitive functions and their hosted equivalents to one magnitude, because they work especially well on former primitives. As a result, and given the benefits of hosted language implementation, the decision to add another primitive to a VM might now be decided adversely.

Our experiences with improving RSqueak/VM and implementing primitives multiple times allow us to make the following contributions:

- We discuss possible trade-offs when implementing behavior as primitive in contrast to in the hosted language.

- We quantify how large a runtime gap is remaining between well optimizable functions implemented as primitive in contrast to in the hosted language, and thus in how far speed is still a trade-off for algorithmic primitives.

Outline The remainder of this work is structured as follows. Chapter 2 introduces into the background. It describes different types of primitives, and goes into detail about the trade-offs to weight when implementing behavior as primitive. Based on these trade-offs, Chapter 3 highlights the benefits of a reduced number of primitives. It explains tracing JIT optimization techniques, and in how far meta-tracing differs from tracing traditional. Chapter 4 discusses implementation details about the VM and the two primitives used for evaluation, fill and BitBlt. The performance of both primitives is measured in Chapter 5, which also presents our experiences during implementation. The work ends with related work from both, academia and industry, presented in Chapter 6 and the conclusion and future work in Chapter 7.

2 Building Blocks of Behavior in Virtual Machines

This chapter explains the concept of virtual machines (VMs), why they are chosen as implementation pattern for dynamically typed programming languages, and what functions they encompass. We will discuss what types of built in behavior are part of VMs, and trade-offs when implementing behavior as part of the VM in detail.

Vocabulary

In the context of this work, a function is a group of statements, which may be ordered. It has zero or more arguments, and a return value. Upon invocation, the statements are executed with regard to their order, if any. Depending on language semantics, unknown identifiers are looked up in the list of arguments, local variables, scope, and global state. The language semantics also determine the return value. It might be the value of the last statement, a value indicated by a special keyword/statement, or a neutral value, like null, nil, or None. Methods are a specialization of functions. They have an additional implicit argument, the receiver, and additional scope, such as the receivers named instance variables, among other. Whenever talking about functions in context of generic programming languages, this explicitly includes methods, in this work.

The functions this work focuses on are explained in Section 2.2. Different language communities have created different jargons, calling them *primitives* in Smalltalk, *built-ins* in Python, or *natives* in Java. In this work, we adhere to the Smalltalk jargon.

Especially in the implementation and evaluation part of this work, we reference the PyPy project. The PyPy project consist of a Python interpreter, and a tool chain. The tool chain translates a subset of Python, RPython, to C, and optionally generates a tracing dynamic compiler. In order to avoid confusion, we refer to the Python interpreter as the PyPy interpreter, and to the toolchain as PyPy. The Python subset compiled by the PyPy toolchain is named RPython, for *reduced* or restricted Python.

2.1 Virtual Machines

Traditionally, programming languages were compiled to machine code, either via assembler, or, less commonly, via C. This results in fast binaries, if there is enough information, such as type information, for the compiler to optimize. Creating such a compiler for dynamic languages is hard, and it usually yields bad performance since few information is available at compile time.

In contrast to that, the currently more dominant way to implement programming languages with garbage collection and dynamic types is splitting the implementation into a compiler and an 'execution engine', the VM. The compiler translates source code to an intermediate representation, which is subsequently executed by the VM. The compiler can be quite simple, because the language's behavior is defined by the VM.

In its basic version, the VM can be implemented as an interpreter stepping through the intermediate representation. The VM can collect runtime information to enable optimization of the dynamic parts.

Besides allowing simpler implementations, splitting the language implementation into compiler and VM also yields the advantages of smaller program representation and better portability.

A VM is mainly responsible for the execution and memory management. Depending on the execution model of the programming language that may include stack management, a fetch-decode-execute cycle, calling primitive behavior, and exception handling. In the case of a simple stack-based bytecode interpreter for a dynamically typed object oriented language, this means that during execution, the VM has a method with bytecodes, and a frame. The frame gives access to the receiver, the object the methods self-references are bound to, local variables, the current program counter, and a stack. Stepping through the bytecodes, the interpreter pushes onto and pops from the stack according to each bytecode, and sends messages. Message sends include looking up the correct method to invoke, building a frame for it and filling the frame with receiver and arguments.

Besides execution, a VM usually manages memory. The VM allocates space, allows read and write access, resolves references, and collects garbage.

2.2 Primitive Building Blocks

When invoking behavior in languages implemented as VM, there are ways to call functions which are not implemented in the hosted language (the language implemented by the VM). Some of those functions are grouped into libraries and called using a foreign function interface (FFI). Other functions are part of the runtime. They are implemented as parts of the interpreter, in linked libraries, or by some extension mechanism provided by the VM. This second group of functions consists of *primitives*. Usually, there is a big set of primitives implemented as part of the VM.

If a primitive is used often, it might have a shortcut. In bytecode based implementations, this means reserving a bytecode which will directly invoke the according behavior. This reduces the number of function names to be saved near the call site, and may elide the lookup of the function name, thus saving space and time. Depending on the language implementation, such bytecodes might be the only way to achieve some behavior, e.g. integer addition on the Java virtual machine (JVM)[30].

In most languages, there is no difference between calling a primitive or calling a hosted function, i.e. primitive invocation is transparent. In the following, we show three different ways how primitives can be invoked, and how the VM knows which primitive to invoke, and how they are documented.

In Smalltalk [20], a method can have an annotation, called pragma. The pragma can indicate a primitive by number [20, pg. 52], or by module-name-pair. When calling a method with primitive annotation, the primitive function is run instead of the method's body. The VM invokes the method's body only if the primitive fails, e.g. because the parameters were of the wrong types. It may contain error handling behavior, or the exact same behavior the primitive does, expressed in Smalltalk. Named primitives may return an error code upon failing, which is then available as temporary (local) variable.

In file-organized languages, such as Python, calling primitives is like calling an imported function. The only difference is that the imported function is not implemented in the hosted language. Primitives in such languages are usually documented in text form, explaining a functions signature, arguments, side-effects, and return value.

In Self [37], primitives are indicated by their selectors (slot names). Every access to a slot whose name starts with underscore (_) should call a primitive function. If that function fails, the VM raises an error to be caught by the caller.

In the following paragraphs, we present a grouping of primitives based on their behavior. The groups are ordered according to the necessity for having them in a VM. We discuss them only briefly. A more detailed explanation of a taxonomy, and the sequence would exceed the scope of this work.

Memory Primitives This set of primitives is closely related to the memory design of the programming language. In object-oriented languages, memory primitives provide means to create new objects, access their fields, their size, and trigger garbage collection. The behavior supplied by these primitives might be implemented as bytecodes, because they are used frequently.

Mathematical Primitives While arithmetics and boolean operations can be implemented using memory access, branching, and constants, they are used often and have a magnitudes faster implementation in the actual machine. This group of primitives consists of basic arithmetic operations such as addition, multiplication, and inverses, but also trigonometric functions, modulo, truncation, exponential functions, numeric type conversion, and root functions. The binary value primitives are *and*, *or*, *xor*, bit-shift, sometimes also *not* ($\neg a \equiv a \oplus 1_{32}$ for 32 bit integers). Another bigger subgroup of mathematical primitives are comparison operations, such as *smaller*, *smaller or equal*, *equality*, *identity*, and their inverses.

Foreign Function Interface FFI Usually, VMs offer a way to interface with existing libraries. The greatest common denominator among 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 FFI, the caller has to explicitly convert types.

System Calls Most process VMs offer a way to access system resources, such as the file system, system clock, or the network. Some offer an abstracted API trying to merge similar concepts of different operating systems, giving access to the greatest common denominator of functionality. This way, they try to promote operating system independent programming. The goal of a programming language determines, whether this set of primitives includes input and output (I/O)-operations, or not.

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, or for timed behavior. Furthermore, the collected runtime information might be available to the hosted language, i.e. accessible using primitives.

Algorithmic Primitives Some behavior can be sufficiently expressed in the hosted language, but is never the less available as primitive. Primitives are generally considered to be faster than the same function implemented in the hosted language. This work focuses on this group of primitives, showing why current optimization techniques are well suited and in how far the execution time gap has closed as a result. Reasons for implementing behavior as primitive, or in the hosted language, are explored in the following chapter. Common examples are copying blocks of memory from one collection to another, simple loops which are run regularly, sorting algorithms, and integer arithmetics for arbitrarily large numbers.

2.3 Trading in the Basement

Whenever programmers make the decision to implement a function as a primitive, they have to weigh the behavior, speed, and security against changeability, reuse, and ease of implementation. In the following, we discuss each of these trade-offs in detail.

2.3.1 Primitive Behavior

Some behavior cannot be expressed in the hosted language. Therefore, such behavior has to be implemented as part of the VM. Among others, this may be as primitive, as bytecode, or as node in an abstract syntax tree.

For example, starting the garbage collection of the VM the program is hosted on is behavior which is part of the VM. And because of its low frequency of being called, it is likely implemented as primitive.

2.3.2 Speed

The goal of reduced execution time is the main reason for creating primitives which could be implemented in the hosted language. The reasoning behind the reduced runtime is that every method call needs a new frame. Even if that does not mean creating a heap object, it means saving frame and stack pointer, copying the arguments (including the receiver, for object oriented languages) to the new frame, and initializing local variables. Depending on language semantics, implementing behavior primitive may also remove costly method lookups, repeated access boundary checks, and repeated boxing/unboxing of basic values such as integers, floats, or strings.

Speed concerns can also be a reason to not implement behavior as primitive. Tracing optimizations record every operation in the hosted language, but operations done by the VM in primitives is not visible. Implementing behavior outside of the visibility of the optimization machinery forbids it to take those operations into account, forcing it to create safe borders whenever leaving traces for primitives. On traces exits, the VM has to create a valid frame for the primitive to operate in, and, if applicable, box the arguments and intermediate results. If a VM utilizes tracing optimizations, it needs to be able to trace primitives. The easiest way to make a primitive visible to the tracing machinery is not having it, i.e. implementing it in the hosted language.

2.3.3 Programming Process

Developing a VM has different properties regarding the development process and the ease of debugging, relative to the hosted language.

First, the round-trip times may differ. The round-rip time is measured between editing the code and seeing changed behavior. The round-trip time depends on features such as incremental compilation, and the general speed of the VM. In order to see the results of a VM change, the VM has to be recompiled, and then, the program needs to run again to the point of failure. If the source of the hosted program changes, the VM only has to rerun, skipping the compilation step.

During debugging, if the VM is developed in a language which itself translates to C, it might have the problem of translucent abstractions. Even though programmers writes code on one abstraction level, they need to be aware of specifics of lower abstraction levels. And when they debug on that lower level, they might have to match the translated sources with their own code, further increasing the effort required, and thus impeding debugability.

Last, the steps taken within a primitive are usually hidden from the hosted language. This has two implications. First, this means a stepwise debugger can but step over the primitive, since it has no intermediate representation to step through. Second, the primitives source code is hidden in the VM sources, and usually also in a different programming language. Thus, they require explicit documentation, and are not easily understandable.

2.3.4 Reuse

Most widely used languages have more than one implementation. Examples are CPython, IronPython, and PyPy, all of which implement Python, MRI (or CRuby), JRuby, Rubinius, and Topaz, which implement Ruby, and the Cog VM, RSqueak/VM, and RoarVM, which implement Squeak Smalltalk. Even if different VMs might not be compatible at the level of intermediate representation, any behavior not implemented in the hosted language needs to be supplied by the VM. Subsequently, any code which is not part of the original VM increases the amount of work needed to reimplement that language. In other words, any complex behavior implemented in the hosted language based on a small set of basic primitives can be taken and used directly, in contrast to programs which rely on hot loops implemented primitive, or C-compatibility for VM interfaces. Furthermore, a VM reimplementation may decide to not adhere to C-interfaces, or to reduce the number of primitives. In such cases, changes to the existing codebases of libraries are required.

Unfortunately, such changes break compatibility with existing language implementations, leading to either increased effort to offer two versions of popular libraries with only small differences for the language implementations, or a decision to only support one of the VMs.

2.3.5 Changeability

When behavior is implemented as primitive, changing it usually requires recompiling the VM, or the according extension. Since rebuilding requires additional sources and a build environment, this not only affects the round-trip times, but also encumbers changing primitives at all.

On the other hand, there might be behavior which should not be subject to change. By fixing that behavior in form of a primitive, it is possible to assert a certain behavior by controlling the VM binary.

Summary

For dynamic programming languages, virtual machine (VM) is the most prominent pattern of implementation. VMs have several building blocks, of which basic behavioral building blocks are the focus of this work. There are different ways to access such primitive behavior, of which all known to us are shown above. Implementing behavior as primitive usually results in reduced execution times, sacrificing readability, changeability, reuse, and ease of implementation.

The following chapter gives reasons why reimplementing primitives in the hosted programming language may be wanted, and why it is feasible when using tracing dynamic compilation.

3 Reimplementing Algorithmic Primitives

Due to the advent of tracing just-in-time compilers (JITs), we argue that it is feasible to reimplement primitives which where created solely for execution time improvements in dynamically typed languages. We argue that some optimization techniques used in ahead-of-time compilers (AOTs) can be used, and used more efficiently, in tracing JITs.

3.1 Reducing the Number of Primitives

In every VM language implementation, there is behavior which cannot be expressed in the hosted language. Therefore, all such language implementations have primitives, if they do not opt to not offer that behavior. Unfortunately, having many primitives introduces some problems.

There are few languages which have just one implementation. Some of those language implementations are not source but immediate representation compatible, e.g. bytecode compatible. As a result, every primitive in the reference implementation has to be mirrored in all the other implementations. First, a small set of primitives alleviates reimplementation. Second, behavior which is not implemented as primitive will work without additional effort.

Since primitives are part of the VM, there is no symbolic stepwise debugging into primitives, to our knowledge. And if there is the goal to understand the primitive, it's source is usually not collocated with the primitives signature definition.

This missing information about atomic steps within primitives also means, that tracing JITs are blind to primitives. In order to inline a primitives behavior, it has to provide it some other way. Coping with this problem only increases the effort to introduce a tracing JIT to a VM, or to introduce new primitives to a VM with tracing JIT. Any defects (bug sources) identified within primitives may have longer change-cycles, both in round-trip time during development, and for publishing.

In conclusion, we argue that reducing the number of primitives will alleviate VM development, and portability between the hosted language implementations.

3.2 Tracing Hot Loops

In a VM with JIT, a dynamic compiler compiles hot (frequently executed) code to machine code and the VM caches it for reuse. A method JIT compiles blocks of code, i.e. functions. A tracing JIT does not adhere to the structures supplied by the language, but records traces (sequences of operations), and compiles those.

A VM with tracing JIT usually tracks not only the frequency at which a code block is run, but also type and branching information. Once the JIT finds a sequence or loop it is confident will be run again often, the JIT compiles it to machine code. Whenever the program goes into that loop again, instead of interpreting the intermediate representation, the VM jumps into the compiled trace. If, for some reason, the path taken with the new arguments leads to different control flow, the JIT faults back to the intermediate representation. In order to ensure that the control flow is the same for different arguments, guard expressions are inserted into the trace asserting types, values, or value-bounds. If the same guard expression is violated repeatedly, the JIT might compile an additional trace starting at that side exit, and stitch together those traces, creating trace trees [18].

In order to keep all traces in memory with low access delay, unused traces are discarded to allow new traces to be compiled. Usually, tracing JITs have a long warm-up time. The VM counts type specific at possible entry points, and in order to generate the trace, it runs the trace in an interpreter recording every operation. Algorithmic primitives have properties such that their traces will optimize well. They usually have narrow type signatures and behave rather static.

3.3 Optimizing traces

The information gathered by tracing JITs are so similar to those available to AOTs, that the most common optimization techniques used there, can also be used in tracing JITs. Additionally, the JIT can use some optimizations not available or feasible to AOTs for two reasons.

First, the generated traces are volatile. That means that the space-time trade-off can be decided differently for optimizations such as loop unwinding or inlining. A tracing JIT only applies them in hot loops, and drops them when those cool, whereas after ahead-of-time compilation, the inflated code stays and consumes space.

Second, a tracing JIT can always go back to interpretation if an assumption turns out to be false. This is especially important for constant folding and branch pruning.

In the following sections, we discuss a number of optimizations which are used in both, AOT and tracing JITs, namely function folding and constant propagation, type specialization, inlining, loop unwinding, loop peeling, and branch pruning.

3.3.1 Folding Functions and Propagating Constants

There are groups of functions which will always yield the same result, given the same arguments [7], or which can be parenthesized differently due to associativity properties. Examples for foldable functions are arithmetic operations and true functions. The compiler can remove the function call and directly assign the result, when it sees such a function, and can prove that the arguments are constant. When a variable has a constant value, the compiler can substitute references to it with its value.

While constant folding is also applicable in an AOT, tracing JITs can assume more values to be constant and thus elide more functions calls. Most branching also adds to the number of constant values.

In a VM with a tracing JIT, there might be the possibility to indicate to the tracing machinery, that a variable should be guarded and its value can assumed to be constant when compiling a trace. To our knowledge, if at all, this hint is only available in the interpreter implementation. This hint is called *promotion* [35], or 'The Trick' [28] and originates in the partial evaluation community. When used, it increases the number of constants and thus the efficiency of function folding. Promotion can be used to implement type specialization.

3.3.2 Specializing for Types

One of the main problems when trying to compile dynamically typed languages is the absence of type information. But, as Garret et al. [19] have shown, even in object oriented dynamically typed languages, the types are rather stable, i.e. the same messages are send to objects of only few different types. Since a tracing JIT records all operations, it can also record types for both, callee and arguments. If the tracing JIT supports promotion, type specialization is but a special case of promotion.

As a result of type specialization, method lookups may be cached, boxed types can be substituted by their unboxed version within traces, and some object layout guards can be removed.

3.3.3 Inlining Functions

When a compiler inlines a function, it substitutes the function call by the operations the function has. As a result, the execution engine, central processing unit (CPU) or VM, does not need to look up the correct function and build a frame. Inlining is only possible, if the function lookup can be dropped because of type information gathered or inferred.

When a tracing JIT creates a trace, it tracks every basic operation from the beginning of the loop until its end. This means that the JIT inlines any visible functions. Example for invisible (and thus not inline-able) functions are those in loaded libraries, FFI-calls, and depending on the recording also primitives which are not explicitly made visible.

AOTs always have to balance the cost of inlining with the anticipated speedup. If the compiler inlines a function, it has to copy the function body to every call site. Inlining is desirable for medium length functions with very few call sites and single instruction functions with many call sites. But if the compiler cannot conclusively identify every call site, it has to preserve the original function, increasing the copy-count by one, which virtually prohibits inlining long functions. Moreover, the expected speedup is small, if a function is called few times, e.g. only during initialization. Therefore, the space cost is not feasible for functions which are not called in loops. Unfortunately, identifying those hot call sites is nontrivial.

3.3.4 Unwinding Loops

Whenever the body of a loop is small compared to the overhead of maintaining the loop, both AOT and JIT may unwind loops. They can either unroll the loop completely, copying the body for every iteration after one another, or only partially, increasing the loop step length and replicating the body accordingly.

Loop unwinding requires the number of iterations of the loop to be constant, or multiple of some constant. Providing this constraint is usually the biggest challenge when implementing loop unwinding.

3.3.5 Peeling Loops

Loop peeling is a special form of loop splitting. Loop splitting is an optimization technique where one loop is split into multiple, which iterate over different contiguous part of the original loops range. As a result, the body of the loop can be optimized with different range constraints. Loop peeling means that the compiler splits off only the first or last iteration.

Analyzing the loop body to identify where to split is time-consuming. Therefore, loop splitting is feasible only in AOT.

But peeling of the first iteration helps to identify loop invariants in an easy and fast way, as shown by Ardö et al. [2]. As a result, this version of loop-peeling is used by tracing JITs. Once the JIT identified loop invariants, it can remove repeated guard expressions and reuse values from the last iteration.

3.3.6 Pruning Branches

A tracing JIT optimizes for one trace. That means, it does not allow branching. When state results in changed control flow, a guard expression at the missing branch will fail. Either the interpreter will pick up at that point, or if the same guard expression fails frequently another trace may continue. The effect of pruning all unlikely branches is that the sophisticated prefetching mechanisms available in modern CPUs will only fail in guard expressions. Every next opcode is just the next in memory. As a side effect, branches which are not taken do not use up memory.

AOTs have a similar mechanism. By analyzing code, they try to identify likely and unlikely branches, then they reorder those alternatives in such a way that the likely branches are written to memory first. From a compilers point of view, this means switching one conditional jump to the other, and then moving some memory around. The effect is similar, since prefetching mechanisms can identify constant offset jumps and follow them, and the likely branch ends with a constant offset jump opcode skipping the unlikely alternative. Nevertheless, assumptions which can be made because of the branching, are not valid once the control flow merges, while traced computations usually can only merge at the loop start. As a result, the constraints introduced by a branch are also valid after they merged, improving constant propagation and function folding optimizations.

3.4 Meta-tracing Just-In-Time Compiler

With PyPy [8] and SPUR [6], there exist two toolchains which come with a meta-tracing JIT. Instead of tracing the program implemented in the hosted programming language, they trace an interpreter executing said program. To allow the meta-tracing framework to compile traces, which are efficient for the hosted language, those frameworks generally rely on annotations in the interpreter.

With respect to reimplementing primitives, annotated and restructured meta-tracing JITs [9] should behave very similar to tracing JITs. With promoting, type-specialization can be achieved. The other optimizations are the same when tracing or meta-tracing. For meta-tracing, there are some annotations, which support constant folding by declaring VM functions to be true functions, and which support loop unwinding by explicitly telling the JIT to do so.

Apart from that, meta-tracing moves the visibility problem and as a result solves it for the interpreter-implementor. Any primitives implemented as part of the traced interpreter are visible. Nevertheless, the behavior which the interpreter is combined from, the primitives it is based on, still suffers from the visibility problem.

Summary

Implementing primitives in the hosted programming language increases readability, changeability, reusability, and ease of implementation. Due to the advances in tracing JIT techniques, the speed-up when creating primitives might not warrant sacrificing all these benefits.

Arguing in favor of reimplementation, we show some of the common optimization techniques which can be used by both, ahead-of-time compilers (AOTs) and tracing just-in-time compilers (JITs), and why a dynamic compiler might actually have an edge in using those techniques.

The next chapter describes implementations of a simple and a complex primitive, both of which are part of the RSqueak/VM research VM.

4 An Example

In this chapter, we describe two algorithmic primitives, BitBlt and Fill. Both are part of modern Squeak Smalltalk VMs. The implementation we show here is part of RSqueak/VM. We describe both primitives, since they are of different complexity and optimize differently. For BitBlt, we describe three implementations.

4.1 RSqueak/VM

RSqueak/VM[1] is a Smalltalk-80 VM. It is a reimplementation of the Squeak Interpreter VM [10], which in turn is a reimplementation of the Smalltalk-80 VM [26]. RSqueak/VM aims to be a fast enough alternative to the existing Squeak Smalltalk VMs, in order to be used and extended in research.

RSqueak/VM uses the tracing JIT generation tool chain which is part of PyPy[2]. PyPy translates an interpreter implemented in RPython to C, and offers the possibility to also generate a tracing JIT for said interpreter. As a result, RSqueak/VM offers three levels of abstraction for implementing primitives. From high to low abstraction, those are Smalltalk, the hosted language, RPython, the VM-implementation language, and C, accessible via RPython FFI.

4.2 Filling an Array

The fill primitive[3] sets every element of an array to a constant value. The array may be an array containing words, or bytes. It is a simple representative for the group of algorithmic primitives. Listing 4.1 illustrates how a primitive-calling method looks, and what the alternative Smalltalk implementation with identical behavior is.

This primitive can only be called on byte- and word-arrays, therefore the receiver types are limited. The argument is always an integer. If the receiver is a byte array, the argument needs to fit in one byte. As a result, the arguments types and range are bounded. On the other hand, the implementation for the VM [Listing 4.2] has to check these assumptions, because the VM-language requires type-safety.

Listing 4.2 presents the RPython code implementing fill. Every primitives in RSqueak/VM use a decorator. The decorator manages the primitive table. Every

[1]http://bitbucket.org/PyPy/lang-smalltalk (accessed April 1, 2015).
[2]http://www.pypy.org (accessed April 1, 2015).
[3]Fill's primitive number is 145.

Listing 4.1: The Smalltalk declaration of primitive fill field in SequenceableCollection, together with a correct implementation in case the primitive fails

```
atAllPut: aNumber
    <primitive: 145>
    1 to: self size do: [:index | self at: index put: aNumber]
```

Listing 4.2: RSqueak/VM's fill field primitive

```
@expose_primitive(145, unwrap_spec=[object, pos_32bit_int])
def func(interp, s_frame, w_arg, new_value):
    if isinstance(w_arg, model.W_BytesObject):
        if new_value > 255:
            raise PrimitiveFailedError
        for i in xrange(w_arg.size()):
            w_arg.setchar(i, chr(new_value))
    elif (isinstance(w_arg, model.W_WordsObject)
            or isinstance(w_arg, model.W_DisplayBitmap)):
        for i in xrange(w_arg.size()):
            w_arg.setword(i, new_value)
    else:
        raise PrimitiveFailedError
    return w_arg
```

primitive has to tell it at which number to register. The decorator can manage the stack for the primitive. It checks that there are all the arguments required, retrieves them, and checks their types, and unwraps any boxed basic type arguments for the primitive function. The decorator also pushes the returned value onto the stack. In case the primitive fails, by raising a PrimitiveFailedError error, the decorator resets the stack to its initial state and tells the VM to invoke the alternative behavior.

In this case, the decorator registers the function in slot 145, it ensures that there are two arguments, an unspecified object and an integer. Upon success, the decorator pushes the receiver, `w_arg`, onto the Smalltalk stack.

The primitive itself has to switch on the first arguments type, the receivers type, to check for the correct upper bound of new_value. The replacement loop in lines 6 f. and 10 f. is identical to the Smalltalk implementation, although replicated for each type because of different interfaces.

4.3 Transfering Bits

BitBlt is an algorithm which combines one rectangular block of bits in a conceptually two dimensional array to another. The conceptually two dimensional arrays are wrapped in form objects which remember width and height. The unique feature of

BitBlt is that it supports bit-wise specification of source and target rectangle. To that end, BitBlt rotates and combines consecutive words of memory.

BitBlt has its roots in graphical operations for one bit displays. 'The first instance of this operation was coded by Dan Ingalls and Diana Merry at Xerox PARC in 1975'[21]. An example is copying letters from the font-specification to a paragraph form, which in turn is copied in parts to the world form, which is the display. Figure 4.1 illustrates this. BitBlt not only supports replacing target with source words, but also combining them, or ignoring both, i.e. filling with a color. Current BitBlt implementations, like WarpBlt[26], can also operate on more information per pixel, for example with 2 bit for red, green, blue, and alpha, 8 bit in total. Colorful pixels require more combination rules, allowing for additive or subtractive mixing of colors, or expansion of 1 bit form values to 16 bit colors. We measured only 1 bit BitBlt performance.

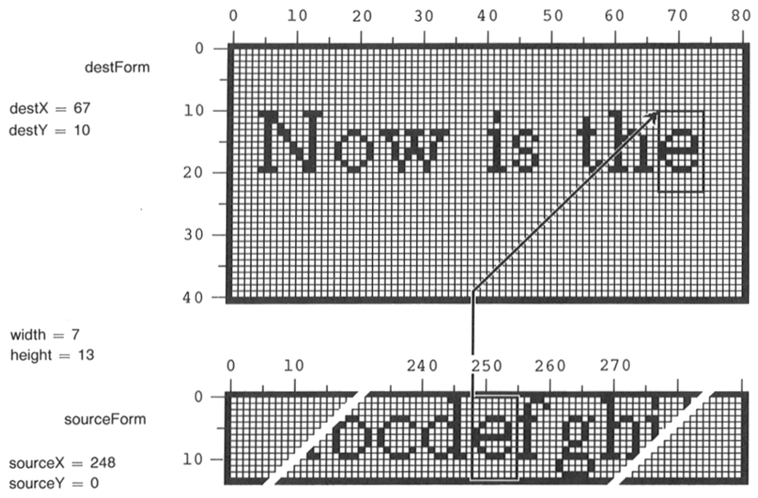

Figure 4.1: BitBlt transfers rectangular areas from one array to the other, accurate to the bit. In this image, a paragraph is composed from letters copied from the font form[20, pg. 335].

In contrast to BitBlt's semantic, Squeak Smalltalk implements BitBlt not as a message-send to forms, but wraps the algorithm in a BitBlt object which has a field for every argument. In order to invoke the primitive, the program needs to send #copyBits to a correctly initialized BitBlt object.

In Smalltalk-80, copying to the screen is the same as copying to any other form, except that one of the forms has been told prior that it should represent the display. The RSqueak/VM implementation transparently expands values and writes to display-memory, if it writes to the display form. Nevertheless, 1 bit display forms maintain a cached copy of the screen, to enable faster access, and because the display memory supplied by the Simple DirectMedia Layer library (SDL) is sometimes filled with zeros.

4.3.1 …in Smalltalk

The Smalltalk implementation mirrors the BitBlt simulation published in the Bluebook [20]. Due to the age of the Bluebook, there are some features missing, like additional combination rules, having the source form as an optional argument, or switching from 16 to 32 bit word size. Unfortunately, implementing BitBlt in Smalltalk has two problems, which also impede tracing.

First, Smalltalk has a different semantic when it comes to bit-shifts and some arithmetic operations. If any of these operations over- or underflow in a machine-oriented language, the result is still of the same range as the arguments, only that a global flag is set which indicates that the last result is incomplete. In contrast, the Smalltalk VM transparently creates LargePositiveInteger, which have a larger range, but arithmetics with such integer is magnitudes slower.

Second, the primitive integer type SmallInteger only covers the 31 bit range. As a result, 32 bit integer are LargePositiveInteger. In RSqueak/VM, we added a special type for 32-bit LargePositiveInteger, which are used transparently in place of LargePositiveInteger, and are supported by the fast arithmetic primitive functions. Nevertheless, this increases the type variety when tracing. Resulting in several very similar traces for the same loop, which the VM switches between according to the result types.

4.3.2 …in RPython

For caching information on objects that are used by RSqueak/VM, and to use the optimized data structures supplied by RPython, RSqueak/VM uses shadows [10]. Shadows are RPython objects hidden from Smalltalk, which are attached to Smalltalk objects. Shadows are used because in Squeak Smalltalk, any object can be used as class, or as context, if only it has enough slots and those are filled with valid data.

Just like most objects which are used by RSqueak/VM, the RPython implementation of BitBlt uses these shadows. We attach shadows to objects used as BitBlt, or as source or target form. In contrast to other shadows, if adding them fails, the primitive fails instead of stopping the VM. Form shadows are the only shadows which might detach from an object.

The BitBlt shadow behaves like any BitBltSimulation object from a Squeak image. It checks the source and target form, reduces the actual clipping rectangle taking into account the targeted rectangle and the actual sizes, prepares bit-masks to escape the first and last word of each line, calculates whether to start at the top or bottom, left or right, and copies the bits.

The form shadow allows direct access to the forms information required during the copy loop: depth, width, height, and offset in x and y. While the original implementation uses direct memory access, we switched to indirect accesses in order to allow switching different BitBlt implementations on the fly. The C implementation might force layout changes to boxed arrays.

The algorithm as such is a port of our updated Smalltalk version of the original Bluebook code. The only difference are some type annotations and value coercions.

The latter are required to be able to run the RPython code on a Python VM and have identical behavior to the translated program.

4.3.3 ...in C

The C implementation differs from the other two implementations because we reuse code which is part of the current Interpreter VM. In Squeak Smalltalk, whenever a primitive is not numbered but named, the VM looks up the primitive function in the accordingly named shared library. If the library is present, the VM loads it, initializes it, and then sets the interpreter. The interpreter is a structure containing the interface between VM and library. It is in effect a function table.

In RSqueak/VM, we replicated this structure, providing the functions needed by BitBlt. For building the C-interface, we used the decorator idiom also found in RSqueak/VM's primitives and many other PyPy projects.

The decorator has multiple arguments, which specify input and output of the function it decorates. The decorator fetches and checks the arguments. The loop for adapting the arguments is written and annotated in such a way, that PyPy can unroll it and remove unused branches during the initial translation.

In the end, our VM creates the so called InterpreterProxy structure from all the functions registered by name and signature.

While some of the functions of the InterpreterProxy used by BitBlt are trivial to implement, there is one particular function which required more effort since it breaks the abstractions: firstIndexableField. When given a words- or bytes-array, firstIndexableField should return the pointer where the collection starts, in order to allow the plugin direct memory access. Unfortunately, this is not possible with RPython lists, requiring manual memory management for such collections and conversion between the two formats.

The plugin itself is written in Slang [26]. Slang is a small subset of Smalltalk which can be translated to C directly. It allows for only a restricted set of methods and types. While Slang looks like Smalltalk and mostly behaves the same way when interpreted as Smalltalk, there are two constructs required for its execution that hindered using the sources of the current BitBlt-plugin as basis for the Smalltalk and RPython implementation. First, Slang allows for inlined C code, which cannot be understood by Smalltalk. Second, the aforementioned InterpreterProxy is the receiver of plenty of messages in Slang.

Summary

This chapter gives two example of primitives implemented on different abstraction levels in the RSqueak/VM Smalltalk VM. The first example fills a field with a constant value. It is a simple and easy primitive with a simple implementation on both abstraction levels supplied, Smalltalk and RPython.

The second example, BitBlt, copies blocks of memory accurate to the bit. BitBlt is the biggest algorithmic primitive found in RSqueak/VM. This chapter described

BitBlt implementations in Smalltalk, and RPython, and how we connected an exiting Slang implementation to RSqueak/VM.

The next chapter evaluates both primitives presented here with respect to two existing Squeak Smalltalk VMs, and gives some insights into the problems of implementing primitives.

5 Measuring the Gap

This chapter consists of two parts. First, we show in how far tracing former primitives reduces their execution times, and how large a gap remains to VM-level primitives. For this, we present numbers measured on the RSqueak/VM. Second, we discuss what influence implementing behavior as primitive has on the implementation effort, focusing on programming process, reuse, and changeability.

5.1 Running Primitives

We measured the speed of execution for both primitives shown in the previous chapter on an idle 64-bit Ubuntu 12.04 with an 3.2.0 Linux kernel. The translation toolchain is PyPy bleeding edge revision 858acb7c946f from August 8, 2013, building on GCC 4.6.3 in a 32 bit chroot. The machine has an Intel Core i7-3520M 2.9 GHz CPU and 7.53 GiB RAM. The VMs all run single-threaded. We used RSqueak/VM in revision 9db538bbab54. All Benchmarks run in a modified Squeak 2.2 mini-image[1], as far as possible.

We measure wall times. For measuring time, we use the Smalltalk facilities, namely `Time class>>#millisecondsToRun:`. As the message name indicates, the resolution is milliseconds. When measuring execution times of JITs, there is always the question whether to take into account the time the JIT needs to identify a hot loop. As Bolz and Tratt [9] argue, there is gain in measuring both times. While JIT-implementors often take the stance that in long-running processes, warming up times are irrelevant, others argue that most processes are short-running. Therefore, we measured the Smalltalk primitives twice, in order to show the penalty involved when not warming up the JIT.

Smalltalk uses green-threading. It provides a semi-cooperative scheduler with several priorities. As long as there are no interrupts, a thread can run as long as it wants to. If there is an interrupt resulting in an higher priority process to continue, the scheduler pre-empts the current process. For every measurement, we start the normal VM, but tell the main user interface (UI)-loop to stop itself after initialization. Stopping the UI-process removes any influence the operating system (OS)'s window manager might have. Prior to starting the image, RSqueak/VM can inject an additional process, which starts the benchmark. The benchmark process will be scheduled once the main process terminated itself. The benchmarks print their re-

[1]http://bitbucket.org/pypy/lang-smalltalk/src/9db538bbab54/images/minibluebookdebug.image (accessed April 1, 2015).

Table 5.1: BitBlt benchmark on a RSqueak/VM, minimum runtime and arithmetic mean runtime with standard deviation (in ms)

n	C	RPython	Smalltalk warmed-up	Smalltalk cold
100	6	28	444	1480
	6.66 ± 0.48	29.6 ± 1.77	458 ± 17.8	1510 ± 23.8
1000	73	293	1300	3550
	76.5 ± 8.50	299 ± 10.6	1330 ± 26.7	3620 ± 55.6
20000	884	5710	20300	24600
	896 ± 20.7	579 ± 111	20600 ± 117	24800 ± 112

sults to the command line using a set of primitives originally introduced to alleviate debugging.

All measured primitives evaluate fast. Usually, they evaluate faster than one millisecond, the measurement resolution. Therefore, we measure the time of execution in a loop. The loop for both our benchmarks contains only the call to the primitive, because general interpretation of Smalltalk might be a magnitude slower, especially when creating new objects. In case of BitBlt, we trigger several different BitBlt operations. Because every call to the according primitive is so fast, the overhead of maintaining the loop is also part of the measured time. Nevertheless, the influence should be minimal since the JIT dynamically optimizes the loop in all cases. The measure n gives the number of times the loop is repeated.

In order to have a baseline, we took measure in both, a classic Smalltalk Interpreter VM version 3.7-7, the newest Interpreter VM which still can run the 2.2 image, and a recent version of the Cog VM, which has a method JIT and generally is the fastest VM for Squeak Smalltalk around. Because current Cog versions do not run Squeak 2.2, the benchmarks were run in a modified Squeak 4.4 image.

5.2 Taking Measure

Table 5.1 and Figure 5.1 both show the same data, the minimum wall time to n times loop through some BitBlt operations. If we switch off the JIT in RSqueak/VM, just computing the first data point takes 24.8 s, 370 times longer than the RPython run time. Therefore, we did not collect those.

Figure 5.4 and Table 5.2 show the relative running times for the Interpreter VM, the Cog VM, and the RSqueak/VM. The numbers are based on the minimal wall times within the 30 samples for the largest n. The base times are the C-reference times, except for the RSqueak/VM Fill data point, because we have only an RPython primitive implementation. We decided to use the minimum of the sample times for the same, largest n. The minimum wall times are closest to actual running times, with minimal time spent on garbage collection or scheduling.

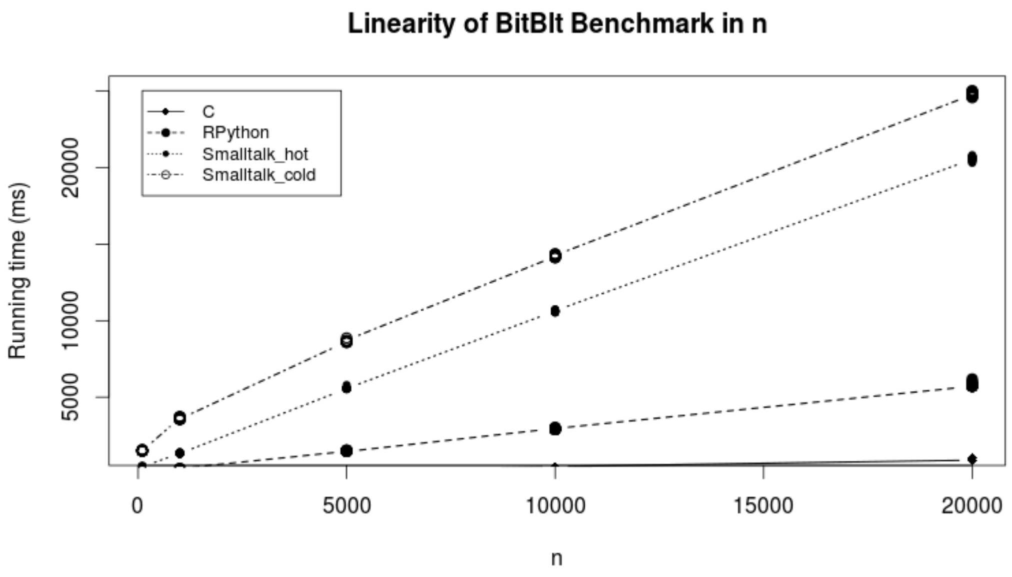

Figure 5.1: Absolute Running Times of the BitBlt benchmark (less is better)

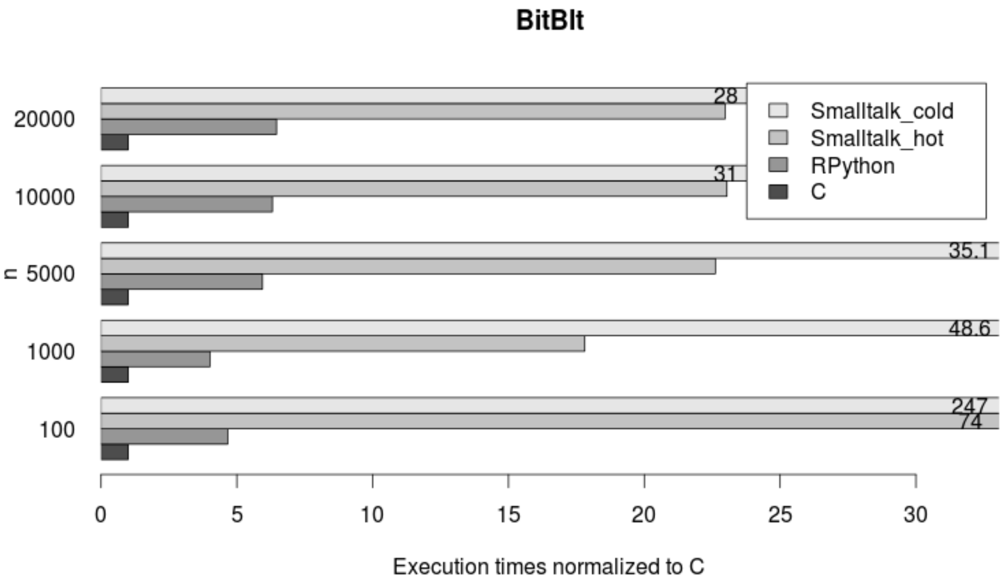

Figure 5.2: Relative Running Times of the BitBlt benchmark (less is better)

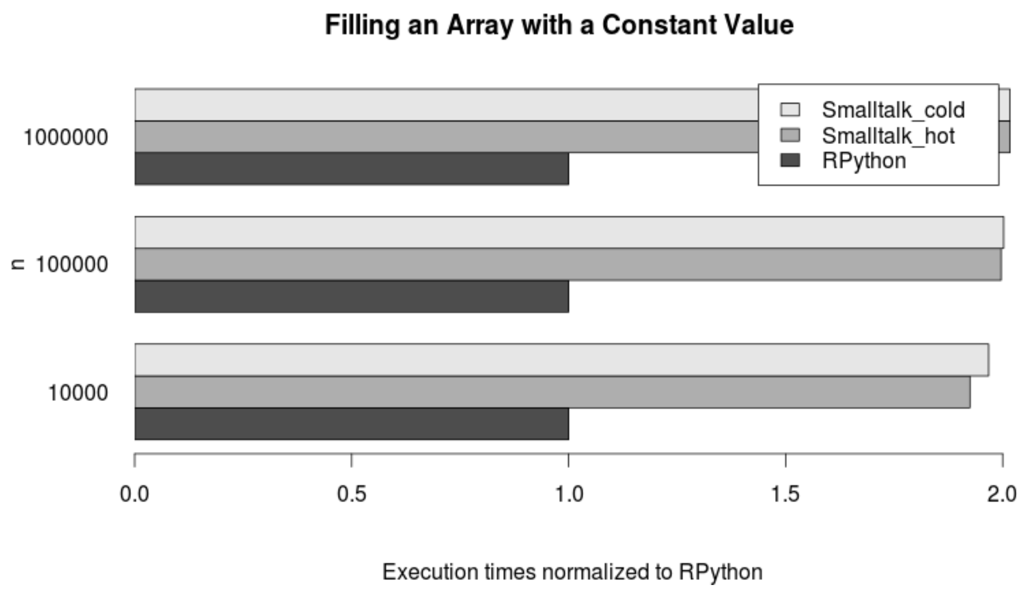

Figure 5.3: Relative Running Times of the Fill benchmark. (less is better)

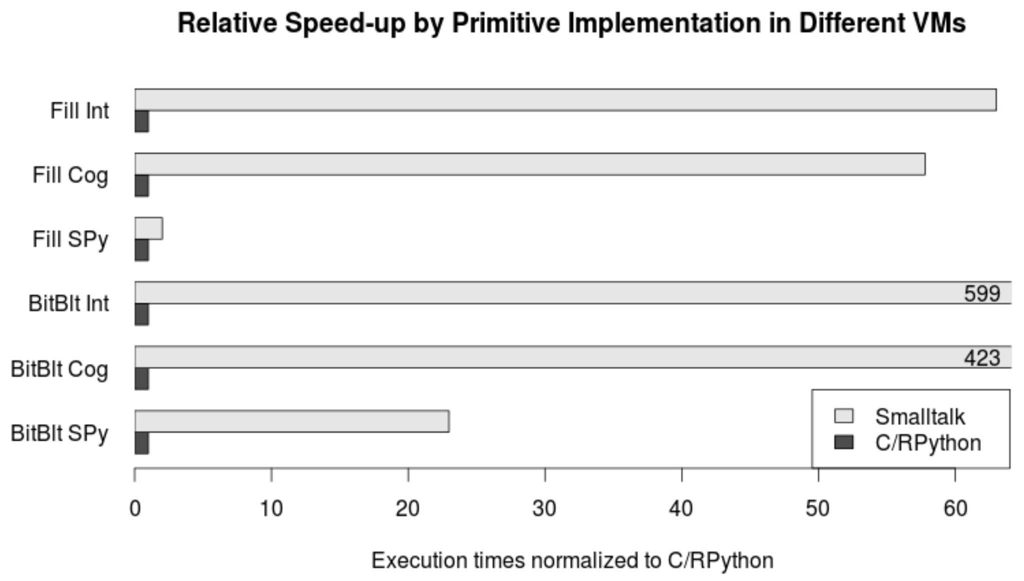

Figure 5.4: Comparison of relative run times when implementing behavior as primitive for the Interpreter VM, the Cog VM, and the RSqueak/VM, for the two examples presented in Chapter 4. (less is better)

Table 5.2: Absolute and relative running times for the Interpreter VM, the Cog VM, and the RSqueak/VM for the two example primitives (in ms). The RSqueak/VM Fill running times are relative to the Cog Fill C time.

	C		RPython		Smalltalk	
Interpreter Fill	2,863	1 ×			180,387	63 ×
Cog Fill	2,894	1 ×			167,256	58 ×
RSqueak/VM Fill	-		8,418	3×	16,979	6 ×
Interpreter BitBlt	650	1×			389,663	599 ×
Cog BitBlt	796	1×			336,494	423 ×
RSqueak/VM BitBlt	884	1×	5,711	6×	20,313	23 ×

Figures 5.2 and 5.4 display a wide value range. We decided to cut the x-axis at 33 and 65 respectively in order to get a good enough resolution for the smaller values. The added numbers give the exact height of the cut bars.

The comparison Figure 5.4 shows that the relative speed improved from 63 and 57 to two, or 599 and 417 to 25 respectively, resulting in a speed up between 31 and 18.

5.3 Explaining the Gap

RSqueak/VM is still in development. While some of the simple, but effective changes explained by Bolz and Tratt [9] have been applied, there is still room for improvement. This can be seen by the base lines in Table 5.2. Unexpectedly, the RPython implementation performs rather badly. One reason might be problems with repeatedly accessing memory and checking bounds every time. Additionally, the RSqueak/VM is still missing type specialized arrays, which were implemented in other PyPy VMs at considerable effort.

The time differences between Interpreter VM and Cog are likely a result of Cogs method JIT. It compiles the benchmarking loop, although the overhead for this loop is negligible compared to the compilation time.

Figure 5.1 shows that after an initial increase for further optimizations, the overhead for compilation is a more or less constant cost. For the first two data points, the dynamic compilation takes longer than the actual execution, nevertheless, without compilation, the resulting times would hardly fit in this graphs range. Figure 5.1 also shows that after an initial compilation, once the traces are all there, any more tracing information gives no additional speed-up. While the dynamic compilation overhead seems quite large for these numbers, it is in all cases less than 5 s.

During the BitBlt Benchmark for n = 1000 in Figure 5.2, the relative running times seem to favor the RPython implementations. But, just like the Smalltalk implementations, the RSqueak/VM JIT also optimizes the benchmarking loop. This additional overhead increases the runtime by the compilation time, thus decreasing the ratios.

Filling an array is a simple primitive in that it does not span multiple functions. While the expectation was that it would yield similar results for method and tracing JIT, the difference is one magnitude. The reason is that Cog cannot yet inline code, let alone primitives. As a result, the Cog code is only slightly faster than the naive interpreter. The magnitude between the Interpreter VM's C implementation and RSqueak/VM's traces can be explained by repeated guards, checking the array's bounds. In C, those bounds are checked only if the programmer specifically asks for it, and not in the loop, but rather before. Those repeated guards are not an inherent problem of PyPy, but a sign for missing hints or structural changes in RSqueak/VM.

5.4 Contesting the Results

One problem with PyPy is that you cannot deactivate the garbage collection. This might increase running times for reimplemented primitives, because all basic types, including integer, 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 JIT removes those unnecessary heap allocations. Therefore, the number of created objects once traces exist should be quite low. Additionally, the benchmarks run the garbage collector between any two measurements. On Cog and the Interpreter VM, integer within 31 bit range are tagged, therefore they are not allocated on the heap. As a result, the object overhead is minimal.

A second ambiguity arises from the youth of RSqueak/VM. 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, there are still bugs in RSqueak/VM's implementation. We prevent the biggest of those scheduling related performance deteriorations by stopping the process responsible for UI prior to benchmarking, but the heartbeats used for timing are still active in all three VMs.

5.5 Lessons learned

This section evaluates the trade-offs explained in Section 2.3, namely Programming Process, Reuse and Changeability. The points raised here are not limited to implementing algorithmic primitives. Some of them are specific to meta-tracing or PyPy.

Programming Process Developing and debugging primitives for the research VM RSqueak/VM has three major problems. First, Python is slow compared to the translated artifact. Even when using the PyPy Python interpreter, reaching the defect might take several minutes. In case of BitBlt, the basic case of drawing the initial windows takes several minutes when using both primitive and non-primitive implementation. As a result, the programmer either retranslates the VM, or waits for interpretation, in order to observe a code change.

When deciding to retranslate, there is a second problem. PyPy so far does not support incremental translation, and due to its structure is unlikely to support it any time soon [29]. Any source changes require retranslation of the whole VM sources. For small VMs like RSqueak/VM, this takes about seven minutes. But translating the PyPy Python interpreter takes about 45 minutes in the setup described above.

Third, and most problematically, behavior differs between the levels of computation. If a primitive works when interpreted as Python, it may not work when translated. Additionally, there might be problems only when using dynamic compilation, which leads to different behavior when translating the VM with and without JIT. While some of those problems are bugs in the PyPy toolchain, there is an inherent problem with changed number semantics. In Python, overflowing arithmetic operations yield results of different types, while in RPython, potentially overflowing operations have to be checked explicitly. Additionally, changed number types might propagate through the flow analysis and extend types far from the problem source.

With regard to the two example primitives, the Fill primitive suffered only from the translation round-trip problem. But as the code shows (see Figure 4.2 for the code), the primitive implementation is simple and did not require debugging. On the other hand, implementing BitBlt illustrated some interesting problems around the different abstraction levels.

The BitBlt implementation in C was the easiest to debug, because we could use traditional C tools, and the debug symbols mapped to our source artifacts quite well. This unfortunately stopped at the bridge to RPython, namely the InterpreterProxy. There are still memory problems which only occur during dynamic compilation. Debugging these with C-tools means working with debugging symbols for the generated sources. The mapping back to RPython is non-trivial, impeding any attempts at understanding what the infects might have been.

For the BitBlt RPython implementation, the necessity to debug on a low level of abstraction was less of a problem. It was the last of the three attempts to implement BitBlt and the source code mirrors the Smalltalk implementation. The main problems were behavioral differences between RPython and Python, namely out of bounds array access. Negative and large indices are well-defined in Python. Thus, they raise no objection when interpreting RPython as Python. But once translated, errors and segmentation faults occurred, requiring repeated time-consuming translations.

The Smalltalk BitBlt was the first among the implementations. Debugging Smalltalk requires the Smalltalk debugger to be rendered. But rendering depends on the BitBlt primitive. As a result, we developed using an existing Smalltalk VM and just reopened the image in RSqueak/VM for every increment. As a result, the round-trip times were the shortest compared to the other implementations, because we did not need to recompile the VM. Now, that we have an alternative, improving upon the current Smalltalk BitBlt using one of the alternatives is possible, using different BitBlt classes, or using different copyBits-methods. As a side effect, the Smalltalk BitBlt-implementation is both hardest and easiest to debug. Hardest, because the image does not just stop on error, but rather tries to create a debugger window, which might fail and obscure the actual error source, and easiest, because we can use the

Squeak IDE. Switching might swap between an in-image Smalltalk version and a primitive implementation, or between two Smalltalk implementations.

Since Smalltalk compiles method-wise and allows to change classes in a running system, the round-trip time is the method compilation time: a few hundred milliseconds.

Reuse While we show two examples of behavior implemented as primitive here, the code for the Smalltalk comparison existed and has been used for a long time. The BitBlt code used here originates from the Smalltalk Bluebook [20], published in 1983. The '#atPutAll'-Method is too old to have a correct timestamp.

More interestingly, there exist several primitives in Squeak, which have a correct Smalltalk implementation in case the primitive fails. This allows us to open newer images in older VMs, which might be missing those primitives. Also, there exists a plugin for named primitives, called MiscPlugin, which does not have to be present for the image to run. Of the primitives having an alternative implementations, only one has an implementation in the RSqueak/VM: #replaceFrom:to:with:startingAt:, which replaces a continuous block of values from one collection with values from a second collection. In a minimized image, this primitive is used to alter immutable symbols after creation. The alternative implementation would require several private methods, which have been removed in the minimized image.

Even in a community as small as Squeak/Pharo Smalltalk, there exist multiple different VM-implementations, all but one of which would benefit from a small set of primitives. The Cog VM is an improvement of the Interpreter VM, adding the JIT. As such, there is one main VM in two variants for Squeak Smalltalk. But then there are the Roar VM [31], which explored primitives for parallel computation for VMs, the Potato Virtual Machine Project[23], running on the JVM, and now RSqueak/VM, as a research vessel built with PyPy.

Changeability All arguments against primitive behavior given in Section 2.3.5 apply. Changing the RPython primitives required a current checkout of the PyPy and RSqueak/VM sources, and the correct command line arguments to initiate translation. Being able to change the primitives requires superficial knowledge of the general VM structure, and knowledge of the decorator used for registering primitives. The PyPy translation works out of the box, if done in a 32 bit-environment.

The additional part needed for the C implementation are the external named plugins compiled for the Interpreter or Cog VM. Setting up an environment to do that is more involved [33]. But once the VM compiles, you only need to change two configuration files to compile plugins to shared libraries.

The Smalltalk implementations on the other hand can be changed within seconds, without requiring any tools or sources other than those coming with a standard Squeak image.

Summary

The gap in execution times between primitives and hosted-language behavior is still one magnitude. But, due to the techniques presented, it has also closed by one magnitude.

Second, we described our experiences when implementing primitives for the RSqueak/VM as well as in the hosted language. The debugging experience differs greatly, mostly because of translucent abstraction and semantic differences in RPython and Python. Changing a former primitive is easiest and fastest in Smalltalk.

The next chapter presents some projects which promote reimplementing primitives in the hosted languages, as well as other attempts to speed up dynamic language execution.

6 Related Work

This chapter presents efforts and academic work related to primitive reimplementation, and the advancement of JIT technologies. The first two sections describe work taking the same stance as this takes. The following two section describe different approaches to algorithmic primitives. The remaining sections present important academic achievements in creating JIT, and their evolution.

6.1 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. Most of the browser vendors have already, reimplemented their Document Object Model (DOM) operations in JavaScript, or plan to do so. There have been few publications about this particular change, and none with performance results.

Mozillas JavaScript JIT, TraceMonkey[1] [18], also traces DOM-operations. Either, the primitives were reimplemented in JavaScript, or otherwise added into the view of their tracing JIT [16]. Neither performance measurements nor details have been published.

Supposedly, in Internet Explorer 10, all DOM operations were reimplemented in JavaScript [27]. Unfortunately, we have found no primary source confirming that.

For Chromium[2], there are plans to reimplement the DOM-API in JavaScript [27]. Chromium uses the V8[3] JavaScript engine. V8 compiles JavaScript to machine code, and patches fast-paths for property access during runtime. The DOM-API is currently implemented in C++. Every call to DOM functions result in a runtime switch.

In contrast to the work presented here, the reimplementation of browser specific DOM libraries in JavaScript improves the overall performance. Reimplementing those libraries allows the tracing machinery to inline DOM operations, instead of exiting the traces and escaping all arguments. Those reimplementations are not done because of increased code understandability, or reuse.

[1]http://wiki.mozilla.org/JavaScript:TraceMonkey (accessed April 1, 2015).
[2]http://www.chromium.org/ (accessed April 1, 2015).
[3]http://developers.google.com/v8/intro (accessed April 1, 2015).

6.2 PyPy Python Interpreter

The general way to introduce new primitives to Python are CExtensions. While the PyPy Python interpreter tries to maintain CExtension compatibility, they explicitly recommend reimplementing extensions in Python [1, 17, comment by M. Fijałkowski].

The reasons for reimplementation is twofold. First, a missing visibility to the tracing infrastructure results in dropping out of traces for any calls to CExtension functions. Second, the Python CExtension API allows access to garbage collection implementation specific details. CPython uses a reference counting garbage collector. Since PyPy does not, the counter have to be emulated, which can be quite costly. On the plus side, once the primitives are implemented in Python, there is the posibility to alternatively use other Python implementations like Jython.

6.3 Quicktalk

Ballard et al. [5] proposed to enable primitive definitions in a Smalltalk dialect called Quicktalk, in the trailer of a primitive-calling method definition in stead of the pragma. As a result, the primitives implementation and the call site are collocated in the sources. The dialect should behave just like Smalltalk, if copied to a methods body, thus enabling to use the existing tools. The focus were performance improvements in stable applications. Quicktalk solves some of the issues with programming process and costs of change to primitives, but it increases the specification for the VM, introduces another technology, and has some behavioral restrictions which prohibits it's use to replace existing algorithmic primitives.

6.4 Multiple Implementations of the Same Primitive

In contrast to replacing the primitive by a hosted function, there is the posibility to maintain two implementations. In Squeak/Smalltalk, this is realized using pragmas [20, pg. 52] to indicate to the VM that the function is equivalent to a primitive. But if the primitive fails for any reason, the VM invokes the methods body, which might contain the same behavior. This way, reimplementations of Squeak Smalltalk may choose not to implement some primitives, while at the same time old VMs are still fast enough, which might not be the case when completely replacing primitives by hosted functions.

With JOHN [34], Samimi reported about a prototype language which facilitated multiple implementations for the same function, using different solving strategies, with different running times. Given a simple and readable, but slow solution, he proposed adding faster implementations, with the intuitive implementation to be the reference for behavior.

In urbit's Nock [32, 38], access to the underlying platform is intentionally impossible, prohibiting any references to primitives. Instead, optimizing VMs are supposed to analyze the source code and when identifying a known program, to replace it by faster primitives called 'jets', which should be behavioral identical.

All of these aproaches have the problem of maintaining multiple versions of the same behavior, and that those behaviors have to stay synchronous, especially for corner cases.

6.5 Tracing Dynamic Languages

Chang et al. [12] were the first to describe a tracing JIT for a dynamic language, ActionScript, which is a superset of JavaScript. Besides tracing, they explore the impact of the bytecode set, especially bytecode complexity, and type information on execution speed. They conclude that despite good performance results in benchmarks, a tracing compiler cannot overcome a slow interpreter for typical web applications.

Gal et al. [18] expands on the idea of tracing dynamic languages. They explore the type specializations possible, and they introduce the notion of trace trees, stitching together traces at side exits and loop-ends, allowing for traces through nested loops as well as prepending to existing traces. Their implementation and evaluation example is TraceMonkey, a JavaScript tracing JIT.

6.6 Meta-Tracing Dynamically Typed Languages

With SPUR [6] and PyPy [8], there exist two meta-tracing projects. In meta-tracing, instead of tracing the application code, the JIT traces the runtime executing said application. As a result, the JIT is agnostic to specific semantics of the implemented programming language.

PyPy [8] is a toolchain for creating VMs in RPython. For a interpreter implementation, it can generate a tracing JIT. To optimize traces, it relies on annotations and hints [2]. Some structural changes of the interpreter might also yield better performance [9].

SPUR is a collection of JITs, which run Microsofts Common Intermediate Language (CIL) [15]. On top of CRL, there runs a JavaScript compiler and runtime implemented in C#. In contrast to PyPy, SPUR removes any interpreter in the chain, having a system solely based on JITs. To our knowledge, there have been no language implementations besides the JavaScript runtime using SPUR.

In SPUR, the JavaScript primitives are implemented as part of the runtime in C#. Prior to starting the VM, the runtime is translated to CIL and then further to machine code. Both artefacts are kept, because the primitives should remain visible to the CIL tracing JIT. In their conclusion, they specifically attribute the performance to being able to trace through the runtime, i.e. their primitives, thus enforcing the point taken by the JavaScript implementations in Section 6.1.

6.7 The Beginnings of Tracing

Dynamo [4] pioneered the utilization of runtime-information to identify and optimize hot paths for machine code. Because Dynamo optimizes machine code, there are no primitives. A slightly more recent example for a binary optimization system is Transmeta Code Morphing [13].

Based on a successor of Dynamo, Sullivan et al. described the first meta-tracer [36]. They traced through an artificial interpreter with Dynamo RIO, using hints to indicate the interpreter loop and backward jumps.

YETI [39], a Java VM, applied the techniques from Dynamo to a statically typed language, trying to improve the performance of interpretation. YETI identified hot paths, translating methods, partial methods and traces. Among others, they applied inlining and indirect jump elimination. Zaleski et al., the YETI team, focused on creating a VM which could be gradually improved to a tracing VM, fostering the integration of interpreter and trace-compiler.

6.8 Method JITs

As a predecessor to Tracing JITs, there exist method JITs [14]. Instead of inlining every operation along a trace, a method JIT compiles frequently used blocks of behavior, like methods, or loop-bodies, to machine code. In his work on the history of dynamic translation systems, Aycock classified method JITs as third generation simulators, and tracing JITs as fourth generation simulators [3].

As an optimization to method compilation, the JIT may cache lookups inline. Hölzle et al. [24] describe method lookup caches, and in SELF, property lookup was optimized [11]. Because of the missing type information, method JITs usually have problems with extensive inlining. In method JITs, loops are not compiled to a single block of machine code in general.

One of the reference VMs during evaluation, Cog VM, has a method JIT. While the RSqueak/VM tracing JIT performed much better than the Cog VM method JIT when tracing former primitives, RSqueak/VM performs worse in more general benchmarks like binarytrees, nbody, or threadring.[4] The likely reason is the maturity of Cog and youth of RSqueak/VM.

In retrospective, an intermediate step to type spezialization is explored in the SELF system [25]. They proposed specializing on the receiver types, as collected by an inline cache, generating specialized code for different receiver types.

[4]http://speed.bithug.org/comparison/ (accessed April 1, 2015).

6.9 Theoretical Soundness

In [22], Guo et al. provide a model for mathematical soundness of trace optimizations, proving bisimilarity, confluence, and determinacy. They show that AOT optimizations and trace optimizations are different sets, with neither being a subset of the other. In particular, they show that traditional forward optimization are sound, but dead store elimination, a backward data-flow optimization removing variables which are never read, is unsound.

7 Conclusion

In this work, we studied the impact dynamic trace compilation has on the decision to implement behavior as primitive. When implementing behavior as primitive, programmers trade changeability, reuse, and ease of implementation for speed, and security. And due to the advent of tracing JITs, the speed argument has lost weight.

We discussed which AOT optimizations are particularly well suited for tracing JITs in general, and for former primitives in particular. We showed that for the two primitives fill and BitBlt, the runtime gap has closed by one magnitude.

There is still one magnitude of speed difference remaining, to be closed by future work. However, 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 on primitives, VMs with tracing JIT, and RSqueak/VM can go into several directions.

None of the JavaScript-projects have published any data. Gathering and discussing that data may be fruitful task, even if it just strengthens the arguments against primitives.

RSqueak/VM offers some points of improvement with high potential. Neither the fact that objects cannot change in size after creation, nor that there might be a set of objects, which can be assumed constant during trace generation are exploited by the VM so far.

The current Squeak/Pharo libraries are build with the assumption that SmallInteger have a range of 31 bit. Neither is this exploited in RSqueak/VM, nor is known to us where the image will break when deciding to change that range and switch from tagged integers to boxed integers.

While we compared several BitBlt implementations, those were not identical, in that the C implementation is younger, but includes several optimizations. Iterating over the Smalltalk implementation and updating it might lessen the gap for this data point.

References

[1] Alex. *NumPy Follow up*. May 2011. URL: http://morepypy.blogspot.de/2011/05/numpy-follow-up.html (visited on 2015-04-01).

[2] H. Ardö, C. F. Bolz, and M. Fijałkowski. "Loop-aware optimizations in PyPy's tracing JIT". In: *SIGPLAN Not.* 48.2 (Oct. 2012), pages 63–72. ISSN: 0362-1340. DOI: 10.1145/2480360.2384586.

[3] J. Aycock. "A brief history of just-in-time". In: *ACM Comput. Surv.* 35.2 (June 2003), pages 97–113. ISSN: 0360-0300. DOI: 10.1145/857076.857077.

[4] V. Bala, E. Duesterwald, and S. Banerjia. "Dynamo: a transparent dynamic optimization system". In: *SIGPLAN Not.* 35.5 (May 2000), pages 1–12. ISSN: 0362-1340. DOI: 10.1145/358438.349303.

[5] M. B. Ballard, D. Maier, and A. Wirfs-Brock. "QUICKTALK: a Smalltalk-80 dialect for defining primitive methods". In: *SIGPLAN Not.* 21.11 (June 1986), pages 140–150. ISSN: 0362-1340. DOI: 10.1145/960112.28711.

[6] M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. "SPUR: a trace-based JIT compiler for CIL". In: *SIGPLAN Not.* 45.10 (Oct. 2010), pages 708–725. ISSN: 0362-1340. DOI: 10.1145/1932682.1869517.

[7] C. F. Bolz, A. Cuni, M. Fijałkowski, M. Leuschel, S. Pedroni, and A. Rigo. "Runtime feedback in a meta-tracing JIT for efficient dynamic languages". In: *Proceedings of the 6th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*. ICOOOLPS '11. Lancaster, United Kingdom: ACM, 2011, 9:1–9:8. ISBN: 978-1-4503-0894-6. DOI: 10.1145/2069172.2069181.

[8] 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. Genova, Italy: ACM, 2009, pages 18–25. ISBN: 978-1-60558-541-3. DOI: 10.1145/1565824.1565827.

[9] C. F. Bolz and L. Tratt. "The impact of meta-tracing on {VM} design and implementation". In: *Science of Computer Programming* (2013). ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2013.02.001.

[10] C. 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: *Self-Sustaining Systems*. Edited by R. Hirschfeld and K. Rose. Volume 5146. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pages 123–139. ISBN: 978-3-540-89274-8. DOI: 10.1007/978-3-540-89275-5_7.

[11] C. Chambers, D. Ungar, and E. Lee. "An efficient implementation of SELF a dynamically-typed object-oriented language based on prototypes". In: *SIGPLAN Not.* 24.10 (Sept. 1989), pages 49–70. ISSN: 0362-1340. DOI: 10.1145/74878.74884.

[12] M. Chang, E. Smith, R. Reitmaier, M. Bebenita, A. Gal, C. Wimmer, B. Eich, and M. Franz. "Tracing for web 3.0: trace compilation for the next generation web applications". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. VEE '09. Washington, DC, USA: ACM, 2009, pages 71–80. ISBN: 978-1-60558-375-4. DOI: 10.1145/1508293.1508304.

[13] J. C. Dehnert, B. K. Grant, J. P. Banning, R. Johnson, T. Kistler, A. Klaiber, and J. Mattson. "The Transmeta Code Morphing™ Software: using speculation, recovery, and adaptive retranslation to address real-life challenges". In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. CGO '03. San Francisco, California: IEEE Computer Society, 2003, pages 15–24. ISBN: 0-7695-1913-X.

[14] L. P. Deutsch and A. M. Schiffman. "Efficient implementation of the smalltalk-80 system". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pages 297–302. ISBN: 0-89791-125-3. DOI: 10.1145/800017.800542.

[15] ECMA. *International standard ECMA-355, Common Language Infrastructure*. June 2006.

[16] B. Eich. *TraceMonkey: JavaScript Lightspeed*. Aug. 2008. URL: https://brendaneich.com/2008/08/tracemonkey-javascript-lightspeed/ (visited on 2015-04-01).

[17] *FAQ: Do CPython Extension modules work with PyPy?* 2013. URL: http://doc.PyPy.org/en/latest/faq.html#do-cpython-extension-modules-work-with-PyPy (visited on 2015-04-01).

[18] 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". In: *SIGPLAN Not.* 44.6 (June 2009), pages 465–478. ISSN: 0362-1340. DOI: 10.1145/1543135.1542528.

[19] C. D. Garret, J. Dean, D. Grove, and C. Chambers. *Measurement and application of dynamic receiver class distributions*. Technical Report CSE-TR-94-03-05. University of Washington, 1994.

[20] A. Goldberg and D. Robson. *Smalltalk 80: the language and its implementation.(The Blue Book)*. Addison-Wesley, 1983. ISBN: 0-201-11371-6.

[21] L. J. Guibas and J. Stolfi. "A language for bitmap manipulation". In: *ACM Trans. Graph.* 1.3 (July 1982), pages 191–214. ISSN: 0730-0301. DOI: 10.1145/357306.357308.

[22] S.-y. Guo and J. Palsberg. "The essence of compiling with traces". In: *SIGPLAN Not.* 46.1 (Jan. 2011), pages 563–574. ISSN: 0362-1340. DOI: 10.1145/1925844.1926450.

[23] M. Haupt. *The Potato Virtual Machine Project*. 2008. URL: http://potatovm.blogspot.de/ (visited on 2015-04-01).

[24] U. Hölzle, C. Chambers, and D. Ungar. “Optimizing dynamically-typed object-oriented languages with polymorphic inline caches”. In: *ECOOP'91 European Conference on Object-Oriented Programming*. Edited by P. America. Volume 512. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1991, pages 21–38. ISBN: 978-3-540-54262-9. DOI: 10.1007/BFb0057013.

[25] U. Hölzle and D. Ungar. “Optimizing dynamically-dispatched calls with run-time type feedback”. In: *SIGPLAN Not.* 29.6 (June 1994), pages 326–336. ISSN: 0362-1340. DOI: 10.1145/773473.178478.

[26] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. “Back to the future: the story of Squeak, a practical Smalltalk written in itself”. In: *SIGPLAN Not.* 32.10 (Oct. 1997), pages 318–326. ISSN: 0362-1340. DOI: 10.1145/263700.263754.

[27] P. Irish and P. Lewis. *Blink Architectural Changes*. 2013. URL: http://www.chromium.org/blink#architectural-changes (visited on 2015-04-01).

[28] N. D. Jones, C. K. Gomard, and P. Sestof. *Partial evaluation and automatic program generation*. Prentice Hall, 1993. ISBN: 0-13-020249-5.

[29] W. Leslie. *Parallel Building*. July 2013. URL: http://mail.python.org/pipermail/PyPy-dev/2013-July/011599.html (visited on 2015-04-01).

[30] T. Lindholm and F. Yellin. *The Java virtual machine specification*. 3rd. Prentice Hall, Feb. 15, 2013. ISBN: 978-0133260441.

[31] S. Marr. “Supporting Concurrency Abstractions in High-level Language Virtual Machines”. PhD thesis. Pleinlaan 2, B-1050 Brussels, Belgium: Software Languages Lab, Vrije Universiteit Brussel, 2013. ISBN: 978-90-5718-256-3.

[32] M. Moldbug. *Chapter 0: Introduction and philosophy*. 2013.

[33] M. Peck. *Building the VM from scratch using git and cmake vmmaker*. Apr. 2011. URL: http://marianopeck.wordpress.com/2011/04/10/building-the-vm-from-scratch-using-git-and-cmakevmmaker/ (visited on 2015-04-01).

[34] H. Samimi. “JOHN - A Knowledge Representation Language”. In: Viewpoints Research Institute, 2008.

[35] D. Schneider and C. F. Bolz. “The efficient handling of guards in the design of RPython's tracing JIT”. In: *Proceedings of the sixth ACM workshop on Virtual machines and intermediate languages*. VMIL '12. Tucson, Arizona, USA: ACM, 2012, pages 3–12. ISBN: 978-1-4503-1633-0. DOI: 10.1145/2414740.2414743.

[36] G. T. Sullivan, D. L. Bruening, I. Baron, T. Garnett, and S. Amarasinghe. “Dynamic native optimization of interpreters”. In: *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*. IVME '03. San Diego, California: ACM, 2003, pages 50–57. ISBN: 1-58113-655-2. DOI: 10.1145/858570.858576.

[37] D. Ungar and R. B. Smith. “Self: The power of simplicity”. In: *SIGPLAN Not.* 22.12 (Dec. 1987), pages 227–242. ISSN: 0362-1340. DOI: 10.1145/38807.38828.

[38] C. G. Yarvin. *Urbit: functional programming from scratch*. 2010.

[39] M. Zaleski, A. D. Brown, and K. Stoodley. "YETI: a graduallY extensible trace interpreter". In: *Proceedings of the 3rd international conference on Virtual execution environments*. VEE '07. San Diego, California, USA: ACM, 2007, pages 83–93. ISBN: 978-1-59593-630-1. DOI: 10.1145/1254810.1254823.

ISBN 978-3-86956-355-8
ISSN 1613-5652