Two Decades of Smalltalk VM Development: Live VM development through Simulation Tools

Eliot Miranda, Clément Bera, Elisa Gonzalez Boix, Dan Ingalls

Two Decades of Smalltalk VM Development

Live VM development through Simulation Tools

Eliot Miranda
Feenk
San Francisco, California
eliot.miranda@gmail.com

Clément Béra
Software Languages Lab
Vrije Universiteit Brussel
Brussel, Belgium
clement.bera@vub.be

Elisa Gonzalez Boix
Software Languages Lab
Vrije Universiteit Brussel
Brussel, Belgium
egonzale@vub.be

Dan Ingalls
ARCOS
Aptos, California
danhhingalls@gmail.com

Abstract

OpenSmalltalk-VM is a virtual machine (VM) for languages in the Smalltalk family (e.g. Squeak, Pharo) which is itself written in a subset of Smalltalk that can easily be translated to C. Development is done in Smalltalk, an activity we call "Simulation". The production VM is derived by translating the core VM code to C. As a result, two execution models coexist: simulation, where the Smalltalk code is executed on top of a Smalltalk VM, and production, where the same code is compiled to an executable through a C compiler.

In this paper, we detail the VM simulation infrastructure and we report our experience developing and debugging the garbage collector and the just-in-time compiler (JIT) within it. Then, we discuss how we use the simulation infrastructure to perform analysis on the runtime, directing some design decisions we have made to tune VM performance.

CCS Concepts • **Software and its engineering → Runtime environments**; *Just-in-time compilers*; *Interpreters*;

Keywords Just-in-Time compiler, garbage collector, virtual machine, managed runtime, tools, live development

ACM Reference Format:
Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two Decades of Smalltalk VM Development: Live VM development through Simulation Tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '18), November 4, 2018, Boston, MA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3281287.3281295

1 Introduction

To specify their virtual machine (VM), the Smalltalk-80 team at Xerox PARC wrote a Smalltalk VM entirely in Smalltalk [GR83]. In 1995, members of the same team built Squeak [BDN+07], an open-source Smalltalk dialect, and its VM [IKM+97], written in Smalltalk using the code from Smalltalk-80 [GR83] as a starting point. Part of the code base was, however, narrowed down to a subset of Smalltalk, called *Slang*, to allow Smalltalk to C compilation. Development was done in Smalltalk, an activity we call "Simulation", and the support code for which is "The Simulator". The production VM is derived by translating the core VM code to C, combining this generated C code with a set of platform-specific support files, and compiling with the platform's C compiler.

Two execution models were effectively available, simulation, where the Smalltalk code is executed on top of a Smalltalk VM, and production, where the same code is compiled to executable code through the C compiler. Simulation is used to develop and debug the VM. Production is used to release the VM. Dummy Smalltalk message sends[1] were used to embed meta information in the code, such as *self var: 'foo' type: 'char *'* which have no effect during simulation but guide the translation process of the executable Smalltalk.

When the Squeak VM was released it consisted mainly in:

- an interpreter with a spaghetti stack,
- a memory manager with a compact but complex object representation: a pointer-reversing tracing garbage collector and a heap divided into two generations,
- WarpBlt, a rotation and scaling extension for the bit-based BitBlt graphics engine,
- external C code and makefiles to support running the VM on popular platforms.

The first three components of the VM were written entirely in Slang. A few extra features, such as file management, were written both in Smalltalk for simulation purposes and in C for the production VM.

VMIL '18, November 4, 2018, Boston, MA, USA
2018. ACM ISBN 978-1-4503-6071-5/18/11…$15.00
https://doi.org/10.1145/3281287.3281295

[1] We use the Smalltalk terminology, send, to discuss virtual calls since we are talking about Smalltalk.

Over the years, the Squeak VM evolved to give birth recently to OpenSmalltalk-VM[2], the default VM for various Smalltalk-like systems such as Pharo [BDN+09], Squeak [BDN+07], Cuis, Croquet and Newspeak [BvdAB+10]. As the VM evolved, the original simulator co-evolved as a tool to develop and debug the VM. The metadata used to guide the translation process was replaced by *pragmas* [DMP16]. But the most significant evolution of the simulator came with the introduction of the Just-In-Time compiler (JIT) [Mir11]. The spaghetti stack was mapped to a more conventional stack frame organisation in a stack zone of a few hundred kilobytes organised into small pages, a scheme called context-to-stack mapping [DS84, Mir99]. A JIT was written. Machine code generated by the JIT was executed by binding multiple processor simulators (Bochs [Law96] for x86 and x64, SkyEye for ARMv6, Smalltalk code for 32-bit MIPS). Support for multiple bytecode sets was added, initially to support Newspeak, and more recently to support adaptive optimization. Finally, a new object representation and garbage collector was added to improve performance and also to support 64 bits [MB15]. This required refactoring the interpreter to allow the object representation to be chosen at start-up[3]. Slang was also extended with type inference to allow Smalltalk code that is independent of word size by virtue of Smalltalk's infinite precision arithmetic to be used in both 32 and 64 bit contexts.

One of the key design decisions in OpenSmalltalk-VM was to keep the core components (i.e. the interpreter, the JIT and the memory manager) written in Slang and not in C/C++. By interpreting the Slang code as Smalltalk code, emulating native code using an external processor simulator and simulating the memory using a large byte array, it is possible to simulate the whole VM execution. This allows development and debugging of the VM with the Smalltalk development tools resulting in a live programming experience for VM development[4].

In this paper, we present key details of the Simulation infrastructure, give examples that demonstrate its productivity advantages, and discuss some of its limitations. The paper is structured as follows. Section 2 introduces the simulation infrastructure used to develop and debug the VM. Section 3 reports our experience developing the VM with the simulation infrastructure. Section 4 discusses some of the simulation infrastructure limitations, how we work around them and related works.

[2]https://github.com/OpenSmalltalk/opensmalltalk-vm/

[3]The JIT was written with this eventuality in mind.

[4]Arriving at the startup that first funded the OpenSmalltalk-VM, the first author had considerable experience maintaining the 2nd generation Deutsch Schiffmann VM written in C, and was sceptical that the Squeak Simulator would be viable for implementing a JIT. The startup wanted incremental development so as to reap value early and reduce risk, so it asked first for an interpreter using context-to-stack mapping. Writing this initial deliverable within the simulator was so much more productive and pleasurable than his previous work in C that he happily decided to stay with the simulator and invest in the necessary machine code simulation infrastructure.

2 Virtual Machine Simulation

The key idea of the Simulation is to allow developers to reuse the whole Smalltalk IDE including the browser, inspectors and debugger to develop the VM. Most new features can be developed interactively, adding code to the VM at runtime, in the simulation environment, like Smalltalk programming.

In this section we first briefly introduce the compilation pipeline to generate the production VM and Smalltalk snapshots, key elements to understand the design of the simulation infrastructure. Then we describe the memory layout of both the production and simulation runtimes. Next we detail how VM execution is simulated and the interactions with the simulated memory. In the following subsection, we explain specific aspects of the simulation infrastructure such as the simulation of the machine code generated by the JIT through the processor simulator. The last subsection discusses simulation features related to the development of the JIT.

2.1 Context

VM compilation. The VM executable is generated in a two step process. Firstly, the Slang-to-C compiler translates the Slang code (interpreter, JIT and GC code) to C code, generating a few C files. This first step usually takes several seconds. Secondly, the C compiler (depending on the platform LLVM, GCC or MSVC) translates all the C files (generated C files and platform-specific C files) into an executable. This second step can take up to a few minutes the first time. Subsequent compilations usually only take a few seconds, since object files for unchanged C files do not need to be recreated.

The VM can be configured in two main flavours, interpreter-only or interpreter+JIT [5]. Although the version with the JIT is the most widely used in production, the interpreter version is convenient for development purposes and essential on devices that outlaw JITs. For example, debugging the garbage collector or evaluating new language features can be done in the interpreter-only VM, avoiding JIT complexity.

Snapshots. Smalltalk is an object *system*, rather than a language. The entire system, including its development tools and application code is stored in a snapshot file, which is essentially a memory dump of the entire heap. When programming with Smalltalk, the programmer usually starts from a snapshot which contains the core libraries, the development environment and the application under development. More precisely, the snapshot includes objects (such as the classes), the compiled methods in the form of bytecodes and the running processes. Developing applications consists essentially in writing and editing code, which installs, modifies and removes classes and compiled methods to and from the

[5]Combining an interpreter and a JIT has performance and complexity advantages, as well as complexity disadvantages. But discussion is beyond the scope of this paper.

class hierarchy. Programming may be done live, as the application under development is running. For example, objects may have their shape changed on the fly as instance variables are added and removed. A new snapshot can be made during or at the end of the development session.

Snapshots can also be used to avoid long start-up times when fixing specific bugs. The VM can be run to a point where the bug is about to manifest and a snapshot taken. Then multiple analyses of the bug can be undertaken by loading the snapshot and resuming execution, either in the normal VM or in the simulator, short cutting the time to reach the bug.

2.2 Memory architectures

Production memory layout. Figure 1 describes, from a high-level perspective, the memory used by OpenSmalltalk-VM in the production VM and its simulated counter-part. Let us detail briefly the memory used by the production VM, on the top of the figure. On the left, low address, side, we can first find Text, the section holding the native code of the VM (the compiled code of the interpreter, the memory manager and the JIT itself, but not the code compiled by the JIT). Then, Data holds initialized and uninitialized data, including the VM's global variables.

At higher addresses we find the beginning of the memory managed by the VM. At start-up the VM mmaps a memory region used for the executable code generated by the JIT (this optional section exists only if the JIT is enabled), for new space (initially empty) and for the old objects present in the snapshot plus a little more space for the first *tenures*[6] to be performed without triggering the full garbage collector. Later during execution, as old space grows, new memory regions are mmap'ed at higher addresses to store other old objects.

Usually at a very high address we find the C stack. Room in the C stack is allocated at VM start-up to hold the Smalltalk stack zone. Both stacks are disjoint and managed differently, the stack zone being broken up into small stack pages [Mir99].

Simulation memory layout. In the simulator, the heap is stored as a large contiguous byte array. References between objects are indices into the byte array instead of pointers. All the Slang variables, normally translated to C variables, are instance variables of Smalltalk objects. When required, they use specific wrapper classes, such as *CArrayAccessor*, over normal Smalltalk classes, to emulate the C behavior (only array accesses are available in C, not high level iterator APIs, etc.). The Slang code is executed as Smalltalk code. The Smalltalk stack zone is represented in OpenSmalltalk-VM as a doubly linked cycle of stack pages which are maintained by the VM. In the interpreter-only simulator, the stack zone is a Smalltalk *Array* object. In the full simulator, the stack zone is in the byte array, to allow access to the stack from generated machine code[7].

[6] A tenure is the process of promoting young objects to old objects in a generational GC.

2.3 Runtime simulation

Modularity. The Slang code is implemented in multiple Smalltalk classes, to organise the code, to add modularity through polymorphism, and reuse through inheritance. For example, the *AbstractCompactor* class has two subclasses, one implementing a sweep algorithm and the other a compaction one. For production, during Slang-to-C compilation time, all the code is compiled into a single C file[8]. All the modularity is removed, using the same example, the VM developer chooses at this moment if he wants to compile a VM with a sweep or compact algorithm. No polymorphism is available at runtime. However, since polymorphism is available in the simulator, it can be reused for debugging purposes. Still in our example, the *AbstractCompactor* class has also simulation specific subclasses. Such versions typically express additional constraints in the form of assertions which can be written in plain Smalltalk without restrictions to easily express complex constraints. They also keep specific values live so they can be accessed at debugging time.

Deterministic simulation. To be able to reproduce the same bug exactly multiple times in a row, we designed the simulator to be as deterministic as possible. The most important features are:

- Simulated memory is not subject to Address Space Layout Randomisation,
- A synthetic clock is used so that time advances in lock step with code execution.

JIT simulation. In addition to the interpreter simulator, simulating the JIT requires simulating execution of the machine code it generates. The JIT itself is written in Slang and simulated with the Smalltalk execution model. When the JIT is enabled, the start of the byte array representing the memory is used to hold the machine code it generates at runtime.

Bindings to processor simulator libraries (Bochs for x86 and x64, Skyeye for ARMv6, Smalltalk code for 32-bit MIPSLE) were implemented so that machine code can be executed safely. All but the MIPSLE simulators are accessed through FFI calls and surrounding glue code. Each processor simulator is invoked via calls that supply the byte array as a parameter, and effectively the byte array is the processor simulator's memory, while the processor simulator contains its register state which is accessed via Smalltalk's support

[7] This difference is historical, the interpreter-only version was implemented first in the easiest way possible, but with the JIT the Smalltalk stack must be accessible to the processor simulator.

[8] One C file contains the interpreter and memory manager, with one JIT file per back end

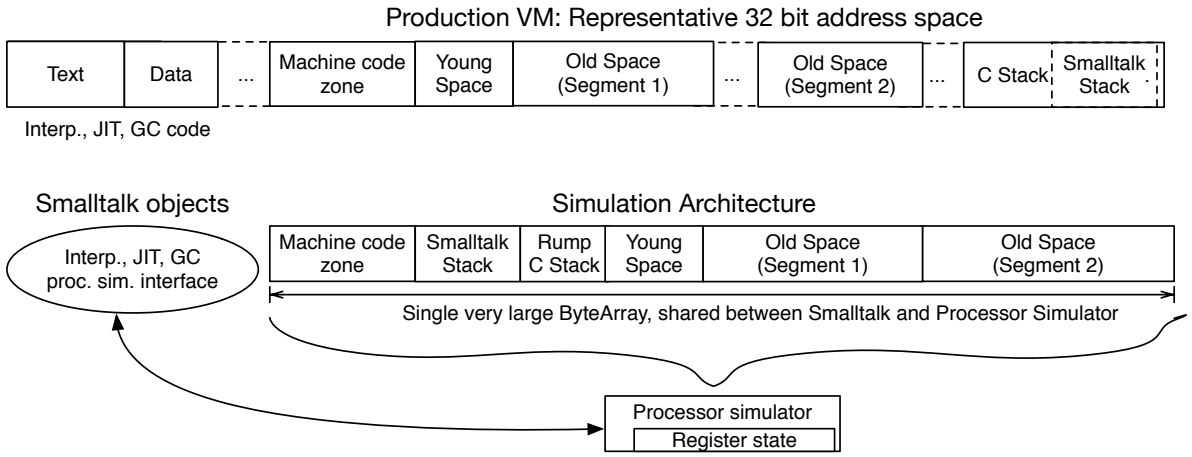

Figure 1. Runtime Memory and Simulated Counter-Part

for external memory access. [9] The next paragraphs describe in detail how this scheme is used to ensure that simulation and production machine code are as similar as possible and how machine code is interfaced to the rest of the simulation.

Interfacing machine code with simulation objects. A key requirement to enable effective development of the JIT is that the code generated during simulation be as close as possible to the code generated by the production JIT. The code is of course not bit-identical; the address space is laid out very differently in memory to the simulation's byte array and the simulator is not subject to Address Space Layout Randomisation. But we benefit from the code generated under simulation being otherwise identical to that generated in the production VM and we go to some effort to achieve this. The technique we use, involving illegal addresses, also allows us to interface generated machine code with the Smalltalk objects that comprise the Simulation, the interpreter, memory manager and JIT, and variables within them. To explain this let's consider how the interpreter references a stack frame.

The interpreter has *framePointer*, *stackPointer* and *instructionPointer* variables used to interpret code. When machine code wants to enter the runtime, for example to have the runtime look up a send on a cache miss, or to invoke the scavenger when young space is full, it needs to record the current stack frame, writing the native stack and frame pointers into the *stackPointer* and *framePointer* variables, and then to call the relevant routine. The runtime can then access the current machine code stack frame though these variables. In the production VM, the *stackPointer* and *framePointer* are C variables in Data, and routines in the runtime have been compiled from C and exist in Text; all have fixed addresses. Referencing these variables and invoking these runtime routines from machine code is therefore straight-forward. But in the simulation they are inaccessible. First of all they are instance variables inside Smalltalk objects, objects that have strong encapsulation, accessible only through messages, and second of all they are alongside the simulation's byte array, not within it, therefore the processor simulator has no access to these variables.

The JIT maintains a set of Dictionaries (hash maps) that map integers representing addresses to closures that get or set variables in the simulation objects, or invoke runtime routines, in both cases by sending messages to the simulation objects. As code is JITted and specific variables and runtime routines are referenced, the JIT manufactures a unique illegal address to reference each variable or routine, and uses this as the key in the relevant map, creating and storing a suitable closure accessor as the value. When the processor simulator attempts to execute an instruction containing one of these illegal addresses, which will be some variety of read, write, call, jump or return instruction[10], it will return an error from the FFI call. Support code analyses the failure and raises a *ProcessorException* which contains all necessary details, the type, pc, next pc and so on. The processor simulator is always invoked in the context of an exception handler for such exceptions which invokes the relevant closure. Once the closure has completed, the processor simulator is advanced to the next pc and simulation resumes. In this way, machine code generated in simulation can access arbitrary simulation objects while remaining essentially identical to the generated machine code in the production VM. Further, *ProcessorException*, and a handful of other methods implemented by a processor simulator makes the interfacing of machine code to the Smalltalk part of the simulation processor independent.

[9] http://www.squeaksource.com/Alien

[10] Activations of JITted machine code methods return to calling interpreter frames by returning to a trampoline that transfers control to the interpreter; hence the return address of a machine code activation above an interpreter frame has this trampoline as its return address.

2.4 Machine code debugging

Instruction recording. The JIT maintains a simulation-only variable that determines whether the processor simulator is invoked in single-stepping mode, or run-until-exception-or-interrupt mode. When in single stepping mode a circular buffer remembers N previous instructions and associated register state, allowing one to examine an arbitrary number of instructions (by default 160) preceding some error. The breakpoint facility is intelligent enough to only enable single stepping once code has been JITted at that address, using run mode until that point.

Conditional breakpoints. There is a small polymorphic scheme for breakpoints. Booleans, integers, and arrays understand *isBreakpointFor: pc*. So if the *breakPC* variable is *false* there is no breakpoint. If *breakPC* is *true* then any pc is a potential breakpoint. If it is an integer then that pc is a potential breakpoint. If the pc is an *Array* then the pc is a breakpoint if any value in the array answers true to *isBreakpointFor:*. The scheme could trivially be extended to include intervals. If *breakPC isBreakpointFor: pc* is *true* then the *breakBlock* is sent *shouldStopIfAtPC: pc*. booleans and closures understand *shouldStopIfAtPC:*. *true* and *false* make the breakpoint unconditional or disabled respectively, and a closure evaluates itself, allowing one to specify arbitrarily complex breakpoints. For example, one can specify that execution should stop at a particular pc if the top two elements on the stack satisfy some criterion. It is much more powerful and much simpler to use than typical machine-level debuggers, closures can be created at any point in any tool with a text interface, such as the debugger, and the scheme can be extended as one is debugging (for example, adding interval support during the middle of a debugging session).

In-image compilation. Simulating the whole VM requires going through the entire Smalltalk system start-up sequence: loading the snapshot, running code registered in the start-up sequence and resuming the user interface. In the simulator, start-up takes around 15 seconds on a recent MacBook Pro. While developing the JIT, this start-up time may still be too long and degrade the live programming experience to an edit-compile-run cycle, which we prefer to avoid. In addition, when developing a specific part of the JIT, it is convenient to compile a well-chosen simple method exercising that specific new part. This is difficult to do with the full simulator since that new part of the JIT might be called by the start-up logic, leading to erratic behavior earlier than expected.

To work around these problems, we implemented a tool called *In-image compilation*. In-image compilation allows the JIT to be invoked as a Smalltalk library on any bytecode compiled method in the host Smalltalk system (i.e. *not* in the simulated heap) to generate the corresponding machine code. In-image compilation invokes the JIT on the method, calls the bound processor simulator to disassemble the code, decorates the disassembly and then dumps the output in a text window.

To generate the machine code, the JIT has to access specific objects (the compiled method, the literals, known objects such as *true*, *false* or *nil*) as if they were in the simulated memory. To implement this we built a facade, which masquerades as the simulation's memory manager, and translates the state of the bytecode compiled method in the current simulation into state that makes it appear as if it were resident in a simulation byte array. This includes mock addresses for all the objects the JIT may require to generate the machine code of a given method. Figure 2 summarizes the in-image compilation process. Note that this technique applies for the baseline JIT, which translate a single bytecode method into machine code; adaptive optimizations and speculative optimizations are debugged differently.

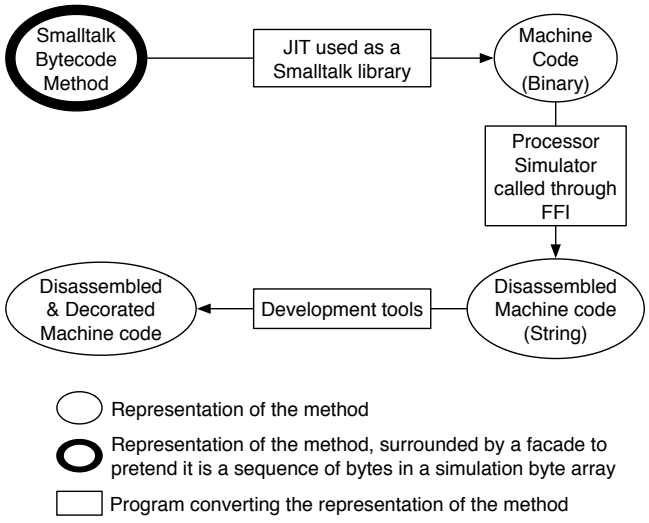

Figure 2. In-image compilation

We also go in the opposite direction to the facade. Smalltalk provides a pretty printer for bytecoded methods. The class *VMCompiledMethodProxy* is polymorphic with *CompiledMethod* and makes methods in the simulation's heap appear to be ordinary *CompiledMethods* in the Smalltalk system, allowing the pretty printer, and other tools, to be applied to methods in the simulation. This scheme can be extended as required; potentially any object in the simulation can be reified to be inspected using the Smalltalk tool suite.

Decorated Disassembly. Each processor simulator is expected to be able to disassemble its machine code. The Smalltalk code for each processor simulator implements a simple parser that identifies constants and field offsets embedded in that disassembly and passes these to routines in the JIT and the interpreter that lookup addresses, matching them to objects, in particular message selectors and classes, and to method local variable names, rendering the disassembly much more readable than that displayed in a typical machine oriented

```
00002063: movq 0x800(%rbx), %rax = 'stackLimit' : 48 8B 83 00 08 00 00
0000206a: cmpq %rax, %rsp : 48 39 C4
0000206d: jb .−0x47 (0x2028=indexOf:startingAt:ifAbsent:@28) : 72 B9
    HasBytecodePC bc 40/41:
0000206f: movq start@20(%rbp), %rsi : 48 8B 75 14
00002073: movq anElement@28(%rbp), %rdi : 48 8B 7D 1C
00002077: movq self@−12(%rbp), %rdx : 48 8B 55 F4
0000207b: movq $0x0, %rcx : 48 C7 C1 00 00 00 00
00002082: call .−0x1B2F (0x558=ceSend2Args) : E8 D1 E4 FF FF
    IsSendCall indexOf:startingAt: bc 44/45:
00002087: movq %rdx, index@−20(%rbp) : 48 89 55 EC
```

Figure 3. Decorated disassembled code

debugger. For example, Figure 3 is a snippet of decorated disassembly for x64, in this case from an in-image compilation.

This code is the end of the entry sequence that checks the stack pointer against the stack limit for the end of the current stack page, a check which is also used to break out of machine code to service interrupts or events such as invoking the scavenger, etc. *%rbx* is used as a base register pointing at the variables in interpreter. Following that is the code for *index := self indexOf: anElement startingAt: start* where *anElement* and *start* are method parameters and *index* is a local variable. HasBytecodePC and IsSendCall are decorations that identify important locations in machine code, including object references, runtime calls, etc. These points are specified using metadata implemented as a simple bytecoded language, a stream of such bytes being added to the end of each machine code method. This metadata is parsed when the garbage collector needs to locate object references in machine code, when linked sends must be relocated as the machine code zone is compacted to make space for new code by throwing away some number of least recently used methods, and when machine code pcs must be mapped to their corresponding bytecode pcs since at the Smalltalk level only bytecode pcs are seen.

Templates. The JIT generates a sequence of machine instructions for a given pattern of bytecodes. We use the term template to describe the sequence of machine instructions generated for a pattern of bytecodes, even though we do not like to reduce the JIT to a simple template-based JIT since each bytecode generates slightly different machine instructions based on register pressure, a simulated stack and a few heuristics. Since the JIT is template-based, in-image compilation is very convenient to develop and optimize each of the JIT templates.

Since we also have a bytecode assembler that allows us to manufacture bytecoded methods we can in practice exercise any part of the JIT by creating a suitable bytecode method. This is hardly pleasurable for methods of the complexity produced by the adaptive optimiser though.

3 Experience reports

In this section we describe some memorable and representative personal experiences using the simulation infrastructure to develop the VM. The different subsections describe various tasks we undertook and in each case we emphasise how we used the simulator and how it helped us perform each given task.

3.1 Initial JIT development and new back ends

When the JIT was first being developed and was at a state where bytecoded methods could actually be translated to machine code, development fell into a very productive and enjoyable rhythm. As a bytecode was encountered for which the template had yet to be written execution would stop in the debugger, which would create a skeleton implementation for the template method from the *MessageNotUnderstood* exception. The programmer would implement the body of the method and execution would continue. Were the new template definition to be incorrect and cause a bug during JIT generation, rather than when the generated machine code was run, then the debugger could be used to wind execution back to the beginning of the JITting of the current method and resume execution, repeating the cycle until disassembly looked good (somewhat like lemming debugging below). As a new method was encountered and JITted yet more templates would be encountered. Similarly as the abstract machine instruction to concrete machine instruction mappings were added these could be implemented in the debugger, an experience repeated when new back ends were added. Much of the JIT was written in this interactive live style. This style encourages tool creation because one can create, extend and polish tools in the middle of debugging sessions, enhancing them to make sense of ones current predicament. Immediate feedback makes tool investment cheap and cheerful.

3.2 Debugging new GC algorithms and Lemming Debugging

Debugging graph rewriting garbage collectors can be hard; bugs leave the heap in a scrambled state, and figuring out what happened when one is left with a corrupted graph, often too broken to traverse and make sense of, or apply tools to, is often effectively impossible. Therefore one is often stuck trying to create a reproducible case in a system that is essentially chaotic: the slightest change in the heap might change the bug; any variability in timing or user input can result in a different heap and hence in the bug morphing or going into hiding.

Here, simulation comes to the rescue. Because the simulation is simply an object graph it is easy to copy the entire simulation. Therefore the simulator has been extended to by default take a copy of itself on every scavenge or old space reclamation, and to first run the collection in the copy, performing a leak check afterwards (if enabled). Hence if a

bug occurs or validation fails it will fail in the copy while the original remains as is, yet to be corrupted. We then take a fresh copy and use it to debug, repeating the process as many times as we like, an approach we call Lemming Debugging. One throws the copy off the cliff and if it gets to the bottom without incident it is descarded and the simulation continues in the original. There is no need to create a reproducible case for bugs encountered during simulation[11]; the reproducible case is at hand. This approach was key in rapid development of the new memory manager that supports both 32 and 64 bits [MB15].

Recently, to evaluate some new old space garbage collection algorithms, we decided to implement standard algorithms to get reference benchmark results. Since we had already a Mark-Compact collector, we mainly introduced a Mark-Sweep non moving GC as an alternative. The whole implementation was done in the simulator, and only when it was working there was it compiled to C. Using this process, the compiled C code worked first time.

We started by sketching the new Sweep algorithm in the code base. Once it was partially written, we started the simulator. Since the algorithm was not complete, we added the missing pieces inside the debugger, installing the new code at runtime and checking against the current values of the function parameters if the function behaved as expected, as one can do in any Smalltalk application.

3.3 Debugging a crash in machine code with conditional breakpoints

In recent years, we added support for a more aggressive JIT with speculative optimizations through a bytecode to bytecode optimizer [BMF+17], re-using the existing template JIT as a back-end. To be able to generate efficient code in the bytecode to bytecode optimizer, we introduced new unsafe bytecodes allowing, for example, accessing of arrays without type and bounds checks [BM14]. For each new bytecode, we introduced new templates in the existing JIT to generate efficient machine code for the optimized methods using the new bytecodes. Once all the basic unit tests worked, we ran the VM with the speculative optimizer, which executed optimized code, and got a crash. We could figure out which bytecode method was triggering the crash, but we had no idea from which template the crash came from. In addition, optimized methods include many inlined methods, making them very large, so it was difficult to figure out where the issue came from just by looking at thousands of bytecodes.

To understand the crash, we created a snapshot where the faulty method was executed right after start-up. We started the VM simulator, and set-up a conditional breakpoint so that simulator would stop when the JIT generated machine code for that method. Then, when the simulator reached the breakpoint, we changed the conditional breakpoint to stop execution when the address corresponding to the faulty method entry in machine code would be used, either by a call from the interpreter or through inline cache relinking. The simulator stopped again, about to execute the machine code corresponding to the faulty method. We then cloned the simulator to be able to reproduce the crash again and again.

Executing the faulty method led to an assertion failure. However, that assertion failed in a GC store check, telling us that the object to store into looked suspicious (address outside of the heap). It happens that this object was read from a field on stack, and that this field held an incorrect address. We could not tell anymore at this point in the execution what instruction among the thousands of previous ones wrote the invalid address to the stack. So we discarded the cloned simulation, and cloned a fresh simulator again, just before the execution of the faulty method in machine code to reproduce again the crash. This time we enabled single stepping (*i.e.,* the processor simulator simulates one instruction at a time) and we added a breakpoint stopping execution when the specific field on stack would be written to the incorrect value found before. In this case, the conditional breakpoint is checked in between each machine instruction, and execution stopped right after the machine instruction which wrote the incorrect value on stack. From the machine instruction address, we could figure out which bytecode pattern generated the incorrect machine code (it was the new bytecode template for inlined allocations). From there, we built a simpler method crashing the runtime and fixed the template using in-image compilation.

The debugging process discussed here is illustrated in a video[12].

3.4 Optimizing the store templates with in-image compilation

A few years ago, we added support in the VM for read-only objects [Bí16]. Read-only objects were critical performance-wise for specific customers using the system in the context of object databases. To maximize their performance, we changed the templates in the JIT compiler for the different memory stores. To optimize each template, we use the in-image compilation framework. We selected a method with a single store to make it simple. We requested the JIT to generate the machine code and changed the template to optimize until the machine code generated was the exact instructions we wanted. It is possible, in in-image compilation, to use the Smalltalk debugger on the JIT code itself to inspect the JIT state and fix the JIT code on-the-fly without any major compilation pause. Once we went through the few store

[11] Alas not for bugs in the production VM. But Lemming Debugging *could* be implemented in C using fork to create the copy in a child process, continuing only if the child successfully completes its GC. Our leak checker works both in Simulation and the production VM.

[12] https://www.youtube.com/watch?v=hctMBGAXVSs

templates (there are a few different templates for optimization purposes, for example, storing a constant integer does not require a garbage collector write barrier check), we've just had to evaluate performance and correctness through benchmarks and tests to validate the implementation.

3.5 Analyzing the machine code zone

The simulator can be stopped at any given point and arbitrary Smalltalk code can be written and evaluated similarly to the *eval* Javascript construct. We used this technique to analyse the simulated memory, including the machine code zone, the heap or any Smalltalk object representing the VM state.

One of the first analyses we ran was on the machine code zone. We stopped the simulation when the machine code zone reached 1Mb. We then iterated over it and investigated what was in it. As show in Table 1, 1752 methods were compiled to machine code by the JIT, 6352 sends are present but 2409 of them are not linked (they have never been executed).

Number of methods	1752
Number of sends	6352
Average number of sends per method	3.63
Number of unlinked sends	2409
Percentage of unlinked sends	37.9%

Table 1. General Machine Code Zone Analysis

Further analysis, in Table 2, confirms Urs Hölzle's statement [HCU91]: around 90% of used send sites are monomorphic, around 9% are polymorphic (up to 6 different cases in our implementation) and the remaining % are megamorphic.

	Number of sends	% of linked sends
Monomorphic	3566	90.4 %
Polymorphic	307	07.8 %
Megamorphic	70	01.8 %

Table 2. Polymorphism Inline Cache Analysis

The code used for these analyses is detailed in the Cog blog, Section "Let Me Tell You All About It, Let Me Quantify" of the blog post "Build me a JIT as fast as you can"[13].

3.6 Directing VM optimizations through analysis

The results of specific analyses are sometimes used to direct performance design decisions on the VM. In this section we describe how the analysis on polymorphic inline caches impacted a hack called "Early polymorphic inline cache promotion".

We designed the polymorphic inline caches (PICs) [HCU91] with two implementations:

- *Closed PICs:* Such caches can deal with up to 6 cases, and are basically implemented as a jump table.
- *Open PICs:* Such caches can deal with any number of cases, they consist of three probes searching the global look-up cache (a hash map shared with the interpreter), falling back to a standard look-up routine on miss.

One idea we had was to promote a monomorphic inline cache directly to an open PIC if available, and create the closed PIC only if no open PIC is available for the given selector. The benefit is avoiding lots of code space modifications and an allocation. The downside is replacing faster closed PIC dispatch with slower open PIC dispatch. The question was how many send sites would prematurely be promoted to megamorphic, or how many closed PICs have selectors for which there are open PICs?

The analysis showed that 17% of polymorphic send sites would get prematurely promoted. So we implemented a simple sharing scheme; the JIT maintains a linked list of open PICs, and before it creates a closed PIC for a send site it will patch it to an open PIC if the list contains one for the send's selector.

Analysing the question was easy in our context. This analysis took about an hour to perform, including writing an iterator over send sites. Somewhat similar analyses performed on the second generation Deutsch Schiffman VM [DS84], which is written entirely in C, when adding the same PIC scheme, took several days, having to be written statically and tested with a traditional edit-compile-debug cycle. The productivity difference is extreme.

4 Discussion and Related Work

In this section we first discuss some VM simulator limitations and how we work around them. Then we cover the interaction between the simulator and bootstrapping processes. Lastly we mention some relevant related work.

4.1 Virtual Machine simulation limitations

The simulator has several limitations. Due to the simulation infrastructure being different from the actual hardware, code is run differently and one could think it leads to strange bugs happening only in simulation or in production. However, we have been using this infrastructure since 1995, and these kinds of bugs are rare and usually easy to fix. We have however two main limitations: simulation performance which is quite slow and calls to external C/C++/machine code which cannot (yet) be simulated.

Performance. The first limitation is due to simulation performance. The interpreter-only simulator is between a hundred and a thousand times slower to execute code than the normal VM. With the JIT and processor simulation enabled, without specific debugging options such as conditional breakpoints in between machine instructions, simulation performance usually drops by a factor of two, although in specific cases it is actually faster. We usually enable conditional breakpoints in-between machine instructions only when we

[13] http://www.mirandabanda.org/cogblog/2011/03/01/build-me-a-jit-as-fast-as-you-can/

reach a point in the simulator where the bug is about to happen since single stepping is much slower, so the overall performance in this context is not really relevant.

In practice, if a GC bug happens in an application 15 minutes after start-up, it takes around 50 hours to reproduce in the simulator!! Fortunately, we work around this problem by using snapshots. Once we are able to reproduce a bug in the production VM, we try to snapshot the runtime just before it crashes. The VM simulator can then be started just before the crash and the debugging tools can be used in far less than a minute. In addition, since the interpreter-only simulator is usually quicker at executing code, and garbage collection simpler because machine code is not scanned for object references, if the bug is unrelated to the JIT (typically, a GC bug), the interpreter-only simulator can be used.

Calls to external code. Although most of the GC and JIT development and debugging can be done in the simulator, some specific tasks cannot be done this way. Basically, any calls outside of the machine code generated by the JIT and the Slang code cannot be simulated. For specific small parts of the VM, such as file management and socket support [14], we extended the simulator, effectively duplicating the code base with the C code, to support those features in simulation. However, there is no solution in the general case: we cannot afford to simulate both the compiled C code and the jitted code on the processor simulator; that would be horribly slow, and specific behaviors in the machine code not present in the code generated by the JIT can hardly be simulated (Access to C variables, OS variables, etc.).

We have a significant amount of bugs in Foreign Function Interfaces (FFI), often due to specific interaction between callbacks, low-level assembly FFI specific glue code and moving objects. Such bugs cannot be debugged with our simulation infrastructure so far and we have to rely on gdb/lldb.

4.2 VM Simulator and Bootstrapping facilities

In general, the simulator provides a toolkit for manipulating and inspecting snapshot files. Snapshots can be used to avoid long start-up times for specific bugs. A simulation can be run to a point where the bug is about to be injected, the system snapshotted, and then multiple analyses of the bug undertaken by loading the snapshot and resuming the simulation, short cutting the set up time for the simulator.

Besides being a key tool in VM development, the simulator can be used to indirectly allow the VM to be started from source files [PDF+14]. The simulator is also used to create 64-bit images from 32-bit images, a transformation that involves replacing certain instances of classes by certain others (e.g. in the 32-bit system all floats are 8 byte boxed objects while 64-bit systems support an immediate floating point type that represents a subset of 64-bit IEEE 754 floating point numbers).

[14] Adding socket simulation support was as meta as it gets. The Socket primitive support in the Simulator was written above the socket primitives in the production VM. An application that used the socket support, the Smalltalk system updating itself from a remote repository, was used to stress the socket primitives. Errors in the simulated primitives under development caused primitive failures in the simulation, causing a debugger to open up in the simulation. We interacted with this debugger to debug the socket primitives we were writing for the simulator!

4.3 Related Work

Many VM developers have implemented different tools to help them work more efficiently on their VM, but they rarely publish about it. For example, the teams working on the Truffle/Graal projects [WWW+13] have very convenient development tools for their VMs such as a tool to visualize the intermediate representation of their JIT Graal at different points in the compilation process. But, despite an impressive number of publications on the projects, including publications on development tools for the guest languages [dVSH+18] or on specific parts of the VM [DWS+13], no publication seems to be directly related to their VM development tools. This section discusses some related work: the Maxine inspector and the RPython toolchain infrastructure.

Maxine Inspectors. The Maxine inspectors [Mat08] were demonstrated at OOPSLA'08. They allow one to inspect the running state of the Maxine VM while it runs for debugging purposes. One of the main differences with our design is that the Maxine VM is metacircular, hence it does not have a simulation and a production mode as we do but a single production debuggable mode. We believe having two different modes allows us to easily generate a production VM while still having nice debugging features. Having a full metacircular VM would be interesting, but it is unclear whether it is convenient to build a production VM in that way. So far, most production VMs (Java, Javascript, etc.) are still compiling through the C/C++ compiler and are not metacircular.

RPython toolchain. The RPython toolchain [RP06] was designed and implemented quite similarly to OpenSmalltalk-VM. Most of the VM code is written in RPython, a restricted Python, instead of Slang, and some leftovers are written in plain C. RPython is, however, much closer to Python than Slang is to Smalltalk. The key advantage of such a design choice is that the RPython code feels like Python code and is relatively quite easy to read write, unlike Slang which feels like C and is as easy to write as C. The main drawback is that RPython to C compilation takes much longer than the Slang to C compilation (up to 40 minutes in a recent Macbook pro for the RSqueak VM [FPRH16], instead of several seconds for Slang).

The RPython code can be executed as normal Python code. As for Slang simulation, it is however very slow compared to the production code. To work around poor simulation performance, we use snapshots in OpenSmalltalk-VM to execute the code to debug at start-up. Since snapshots do not

seem to be available in RPython, debugging specific bugs is in practice very difficult (days of simulation required).

Conclusion

This paper introduced and discussed the OpenSmalltalk-VM's simulation infrastructure, used to develop and debug the VM. In particular, we described our experiences when developing and debugging the garbage collector and the JIT compiler. We have found that Simulation is a powerful tool allowing us to reduce development time and to fix bugs quickly.

In the near future, we plan to extend the simulator with customizable development tools. Currently the tools built on top of the simulation infrastructure are mainly textual. More advanced tooling such as moldable inspectors and debuggers [CNSG15, CGN14] should enable a more interactive interface, making it easier to apprehend by new developers, and further improving productivity.

Acknowledgements

The authors would like to thank Ryan Macnak who wrote the MIPSLE processor simulator and back end, as well as much of the Newspeak send machinery.

References

[B16] Clément Béra. A low Overhead Per Object Write Barrier for the Cog VM. In *International Workshop on Smalltalk Technologies IWST'16*, 2016.

[BDN+07] Andrew Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Squeak by Example.* Square Bracket Associates, 2007.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example.* Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[BM14] Clément Béra and Eliot Miranda. A bytecode set for adaptive optimizations. In *International Workshop on Smalltalk Technologies 2014*, IWST '14, 2014.

[BMF+17] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. Sista: Saving optimized code in snapshots for fast start-up. In *Managed Languages and Runtimes*, ManLang 2017, 2017.

[BvdAB+10] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. Modules As Objects in Newspeak. In *European Conference on Object-oriented Programming*, ECOOP'10, 2010.

[CGN14] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. The moldable debugger: A framework for developing domain-specific debuggers. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering*, 2014.

[CNSG15] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. The moldable inspector. In *Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, Onward! 2015, 2015.

[DMP16] Stéphane Ducasse, Eliot Miranda, and Alain Plantec. Pragmas: Literal Messages as Powerful Method Annotations. In *International Workshop on Smalltalk Technologies*, IWST'16, 2016.

[DS84] L. Peter Deutsch and Allan M. Schiffman. Efficient Implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, POPL '84, 1984.

[dVSH+18] Michael L. Van de Vanter, Chris Seaton, Michael Haupt, Christian Humer, and Thomas Würthinger. Fast, flexible, polyglot instrumentation support for debuggers and other tools. *CoRR*, abs/1803.10201, 2018.

[DWS+13] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, 2013.

[FPRH16] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. How to build a high-performance vm for squeak/smalltalk in your spare time: An experience report of using the rpython toolchain. In *International Workshop on Smalltalk Technologies*, IWST'16, pages 21:1–21:10, New York, NY, USA, 2016. ACM.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[HCU91] Urs Hölzle, Craig Chambers, and David Ungar. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *European Conference on Object-Oriented Programming*, ECOOP '91, London, UK, UK, 1991.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '97, 1997.

[Law96] Kevin P. Lawton. Bochs: A portable pc emulator for unix/x. *Linux J.*, 1996.

[Mat08] Bernd Mathiske. The maxine virtual machine and inspector. In *Companion to the Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, 2008.

[MB15] Eliot Miranda and Clément Béra. A partial read barrier for efficient support of live object-oriented programming. In *International Symposium on Memory Management*, ISMM '15, 2015.

[Mir99] Eliot Miranda. Context Management in VisualWorks 5i. In *OOPSLA'99 Workshop on Simplicity, Performance and Portability in Virtual Machine Design*, Denver, CO, 1999.

[Mir11] Eliot Miranda. The Cog Smalltalk Virtual Machine - writing a jit in a high-level dynamic language. In *VMIL '11*, VMIL 2011, 2011.

[PDF+14] G. Polito, S. Ducasse, L. Fabresse, N. Bouraqadi, and B. van Ryseghem. Bootstrapping reflective systems. *Sci. Comput. Program.*, 96(P1), 2014.

[RP06] Armin Rigo and Samuele Pedroni. Pypy's approach to virtual machine construction. In *Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, pages 944–953, New York, NY, USA, 2006. ACM.

[WWW+13] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward 2013, 2013.