Over Twenty Years of Virtual Machine Development and Debugging Through Simulation

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

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

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

Abstract

Open-Smalltalk-VM was originally written in Smalltalk to specify and example the virtual machine (VM) implementation. The same code base was changed a little bit to allow Smalltalk-to-C compilation, generating effectively the production VM. Two execution models are effectively available, the simulation mode, executing the Smalltalk code on top of a Smalltalk VM, and the production mode, compiling the code to executable through C. Simulation is used to develop and debug the VM. Production is used to release the VM.

As the VM evolved, by introducing better garbage collector algorithms or a just-in-time compiler, both execution models co-evolved. For example, the simulation mode was extended with a processor simulator to simulate the code generated by the just-in-time compiler.

In this paper, we detail the VM simulation infrastructure and we report our experience developing and debugging the VM within it. We discuss some of the limitations and how we dealt with it. Then, we focus on two specific use-cases, a bug in the just-in-time compiler and the development of a new full garbage collector compaction algorithm, and we show how the simulation infrastructure helped us. Lastly, we discuss how we use the simulation mode to perform analysis on the runtime, directing some design decisions we make to tune the VM performance.

Keywords Just-in-Time compiler, Virtual machine, Managed runtime, Tools

ACM Reference Format:
Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. 2018. Over Twenty Years of Virtual Machine Development and Debugging Through Simulation. In *Proceedings of ACM Conference (VMIL'18).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

To specify the language and explain how to write the virtual machine (VM) for it, the Smalltalk-80 crew wrote a Smalltalk VM entirely in Smalltalk [GR83]. In 1995, the same team built an open-source Smalltalk and VM, Squeak [BDN+07]. The VM was ported from the original specifications however part of the code base was narrowed down to a subset of Smalltalk, called *Slang*, to allow Smalltalk to C compilation [IKM+97]. Effectively, the VM could be both simulated by executing the Slang code on top of the Smalltalk VM or compiled to native code to produce the production VM. At this point, the VM consisted mainly in an interpreter, a memory manager with a generational garbage collector (GC) and BitBlt, an extension for the user interface bit-based engine. These tree parts 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.

Over the years, the Squeak VM evolved to give birth recently to Open-Smalltalk-VM[1], the default VM for different Smalltalk and Smalltalk-like runtimes (Pharo [BDN+09], Squeak [BDN+07], Cuis, Croquet and NewSpeak [BvdAB+10]). As the VM evolved, the simulator co-evolved as a tool to develop and debug the VM. The most significant evolution of the simulator came with the introduction of the Just-In-Time compiler (JIT), *Cogit*, a template-based JIT. The existing simulator was not able to interpreter the machine code generated by the JIT and had to be extended by binding various processor simulators (Bochs for x86 & x64, SkyEye for ARMv6).

In the following section, we explain the VM infrastructure with both the compilation pipeline to generate the production VM and the simulation infrastructure used to develop and debug the VM. Section 3 reports our experience developing the full GC with our infrastructure. Section 4 explains how we fixed a bug in the Just-in-Time compiler using single stepping in machine code in the simulator. Section 5 shows how we abuse the simulator to analyse the runtime and direct our performance decisions. Lastly, we discuss some related work and conclude.

2 Virtual Machine Infrastructure

As shown in Figure 1, the VM code base is written both in Slang and in C. The Core VM code, also known as the object engine, is written in Slang and includes mainly the interpreter, the template-based JIT and the memory manager. The platform code, *i.e.,* Operating System dependent code such as file management or I/O is written directly in C.

VMIL'18, November 2018, Boston, Massachusetts
2018. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

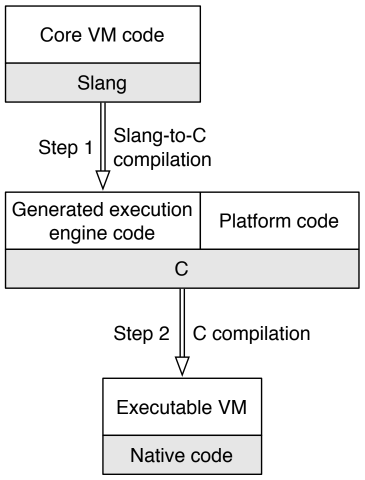

Figure 1. Cog VM compilation

The VM executable is generated in a two step process. Firstly, the Slang-to-C compiler translates the Slang code to C code in a few files. This first compilation pass takes a few seconds. Secondly, the C compiler (depending on the platform, LLVM or GCC), translates the C code into an executable. This second compilation pass may take several dozens of seconds the first time, then, it depends on what the programmer changed and what the C compiler can cache or not, but it usually also takes a few seconds to recompile the C code. The VM can be compiled in two main flavors, interpreter-only or interpreter+JIT. Although the version with the JIT is used in production, the interpreter version is convenient for special purposes, for example to debug the garbage collector or to evaluate new language features without dealing with the JIT complexity.

There are multiple reasons why the object engine is written in Slang and not in C. One of the reason is that the Slang-to-C compiler, using différents annotations, generates C code slightly different from the Slang code: for example, it duplicates the implementation of specific methods with specific constant operands in Slang to generate more efficient code in the interpreter. The main reason is VM simulation. 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, we are able to simulate the whole VM execution.

2.1 Simulating the Virtual Machine

in this section, we first describe briefly the start-up sequence of Open-Smalltalk-VM using snapshots. Then, we detail the simulation of the interpreter-only VM. The simulation of the full VM is a superset and is explained in the last subsection.

Start-up from snapshots. Smalltalk is a snapshot-based language. A Smalltalk program is started from a snapshot, *i.e.,* a memory dump of all live objects at a given point in the execution of a program. The snapshot includes objects such as the classes, the complied methods in the form of bytecodes and the running processes. At start-up, the VM restores the state of all objects in memory and resumes execution in the active process at snapshot time.

When programming with Smalltalk, the programmers usually start ups from a snapshot which contains the core libraries, the development environment and the application developed. Development of the application consists essentially in writing and editing code, which effectively installs, modifies and removes compiled methods and classes from the snapshots. When the changes are done, the developer takes a new snapshot, which includes its changes. Deployment is done from a snapshot containing the deployed application and required core librairies (unused librairies and development tools may be removed from the deployed snapshot).

Starting up the VM for source files has never been possible. However, recent work [PDF$^+$14] allowed one of the Smalltalk runtime to recreate a snapshot from sources, indirectly allowing to start the VM from source files.

Interpreter-only simulation. As the VM can be compiled with and without the JIT, the VM can also be simulated with or without the JIT.

The heap is simulated as a large contiguous byte array. References between objects are effectively indexes inside the byte array instead of pointers. All the C variables are simulated as Smalltalk objects. They used specific 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 for example). One of the most complex component is the Stack. The Stack is represented in Open-Smalltalk-VM as a double linked list of stack pages which are maintained by the VM. Each stack page is represented as a Smalltalk object.

All the Slang code is implemented in multiple Smalltalk classes, to organise the code. For production, at Slang-to-C compilation time, all the code is compiled in a single C file. No polymorphism is available at runtime. However, polymorphism can be abused for debugging purposes. For example, a class is available holding all the garbage collection compaction logic. This class might be subclassed with a simulation only version: this version can use normal Smalltalk code over Slang, allowing to express easily specific constraints which leads to assertion failures if not met, such constraints are convenient for debugging.

In Simulation, one of the core feature is to be able to re-use the whole Smalltalk IDE, including the browser, the inspectors and the debugger to develop and debug the VM. Most new features can be develop and debug interactively, adding code to the VM at runtime, in the simulation environment, as for normal Smalltalk programs.

JIT simulation. In addition to the interpreter simulator, simulating the JIT requires to simulate the execution of native code it generates. The JIT itself is written in Slang and simulated with the Smalltalk execution model. To simulate the machine code, the start of the byte array representing the memory now holds the machine code generated and installed by the JIT. Bindings to processor simulator libraires (Bochs for x86 and x64, Skyeye for ARMv6) were implemented so that the machine code can be simulated. Calls in-between slang code and direct machine code are a little bit trickier to simulate. Calls from machine code to slang code are implemented by using multiple invalid processor instructions, leading to a trap in the processor simulator. This trap is caught is the VM simulator, which then resumes Slang simulation by calling the method corresponding to the invalid processor instruction. Calling machine code from Slang requires to start the simulation and then launch an exception to stop Slang simulation. The processor simulator can be start in two different ways. It can either start simulating code until it meets an invalid instruction or simulate one instruction at a time. The second version is slower, but allows to implement specific debugging features, such as conditional breakpoints in-between each machine instruction.

2.2 In-image compilation

Simulating the whole VM requires going through the whole start-up sequence: loading the snapshot, running code registered in the start-up sequence and resuming the user interface. The whole start-up takes around 15 seconds on a recent Macbook pro. While developing the template JIT, this start-up time may still be too long and move the live programming experience to an edit-compile-run cycle. To work around this problem, we implemented a tool called *In-image compilation*. In-image compilation basically allows to call the JIT as a Smalltalk library on a given bytecode compiled method to generate the corresponding machine code and display the disassembly. Since the JIT is template-based, in-image compilation is very convenient to develop and optimize each of the JIT templates. 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 work around this, we built a facade, which is started on the compiled method and only includes the subset of objects required by the JIT to generate the machine code.

2.3 Virtual Machine Simulation Limitations

Performance. The first limitation is due to the simulation performance. The interpreter-only is simulator is around 200 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 drops to around 500 times slower than the normal VM. This means for example that if a GC bug happens in an application 15 minutes after start-up, it will take 50 hours to reproduce in the interpreter-only simulator. Bugs in the jitted code are worst. Fortunately, we work around this problem by using snapshots and the interpreter-only simulator for GC bugs. In general, 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 after only several dozens of seconds. If the bug is unrelated to the JIT, the interpreter-only simulator can be used and it is a little bit quicker to execute code.

Calls to external code. Although most of the GC and JIT development and bug resolution can be done in the simulator, specific developments and bugs cannot. 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, 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 cannot be simulated (Access to C variables, OS, etc.).

The main limitation we have is with Foreign Function Interfaces (FFI). Most bugs we have with FFI are due to specific interaction between call-backs, low-level assembly FFI specific glue code and moving objects. Such bugs cannot be debugged, so far, with our simulation infrastructure and we have to rely on gdb/lldb.

3 Garbage Collection Development

Recently, to evaluate new garbage collection algorithm against standard algorithm, we implemented a Mark-Sweep in addition to the existing Mark-Compact for old space garbage collection. The whole implementation was done in the simulator, and only when it was working there, we compiled it to C and it was working out of the box. To describe the implementation process, we need to discuss briefly first the assertion levels in the VM. Then we will show how we debugged the algorithm.

Assertions. To stabilise our code base and find easily production bugs, all the code based is annotated with assertion, ensuring a specific state is as the VM developer would expect it to be, and stopping the program execution if not. We have basically three levels of assertions. Assertions written in plain Smalltalk are convenient since more complex constraints can be expressed, but they can be performed only in the simulator. Assertions written in Slang are also verified in simulation mode, but they are compiled to C. The C compiler, based on a compilation flag, then picks if it wants to compile the VM with or without assertions. The VM with assertions is used for debugging and to recreate snapshots of runtime juste before they crash. The VM without assertions is the production VM.

Simulation. Once we had partially written the new Sweep algorithm, we started the simulator. Since the algorithm was partially written, we could just write the missing pieces inside the debugger, installing the new code at runtime, as one can do it in Smalltalk. In addition, the simulator has an interesting property: each time it performs a GC, either a scavenge or an old space collection, it first copies the simulated memory (*i.e.,* the heap), performs the GC there, and if no assertion fails, it then performs the GC on the original version. This means that if the algorithm was not working, an assertion would fail or an error such as incorrect memory access would happen in a copy of the heap. The simulator can then reproduce the exact same crash as many times as we want duplicating again and again the same original heap. This is very convenient to debug specific GC bugs because often in GC bugs, when the bug happens, the memory is already corrupted and it's quite difficult to track back where the bug comes from without rewinding the memory state.

4 Just-in-Time Compiler Development

We firstly discuss how we use the simulator to debug crashes in deployed applications, then how we use in-image compilation to develop the JIT itself.

4.1 Debugging crashes with conditional breakpoints in machine code

TOWRITE FROM MY YOUTUBE VIDEO. Back-in-time debugging of machine state Conditional stepping.

Ref Sista

4.2 Optimizing the templates with in-image compilation

A few years ago, we added support in the VM for read-only objects [B́16]. Read-only objects were critical performance-wise for specific customers using them in the context of object databases. To maximize the performance, we changed the templates in the JIT compiler for the different memory stores. Since the template JIT has to generate very quickly machine code, it goes through the code three times: (1) it scans the bytecode, (2) it generates abstract instructions from the bytecode and (3) it generates the native instructions from the abstract instructions. The abstract instructions are mapped almost one to one to machine instructions, their main purpose is to compact the generated code by removing Nops used for jump targets or finding out out the size of jumps.

To optimize the 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 store template 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 code on -the-fly without any recompilation process. Once we went through the few store templates (there are a few different templates for optimizations purposes, for example, storing a constant integer does not require a garbage collector write barrier check),

5 Virtual Machine Analysis: Directing performance decisions

A side-effect of VM simulation, and specifically to be able to interrupt the simulation and introspect the simulated memory and simulation specific objects, is to be able to analyse the runtime with scripts written on-the-fly.

5.1 Analysis example

One of the first analysis we run is to stop the simulation when the machine code zone reached 1Mb. We then iterated over the machine code zone and investigated what was in. As show in Table 1, 1752 were compiled to machine code by the template JIT, 6352 sends[2] are present but 2409 of them are not linked (basically, they have never been used).

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 analysis [HCU91]: around 90% of used send sites are monomorphic, 9% are polymorphic (up to 6 different cases in our implementation) and the remaining % is 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 for these analysis are detailed in the Section "Let Me Tell You All About It, Let Me Quantify" of the blog post "Build me a JIT as fast as you can"[3].

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

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

5.2 Directing the VM behavior

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

We designed the polymorphic inline caches (PICs) 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, and fall back into a standard look-up routine if there is a cache miss.

One idea we had was to promote a monomorphic inline cache straight 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 is how many send sites would be prematurely promoted to megamorphic, or how many closed PICs have selectors for which there are open PICs. Analysing the question is easy in our context.

The analysis result showed that 17% of polymorphic send sites would get prematurely promoted. So we have 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.

6 Related Work and Conclusion

Many VM developers developed different tools to help them being more efficient, but they rarely publish about it. We focus in this section on two related work.

Maxine Inspectors. The Maxine inspectors [Mat08] were demonstrated at OOPSLA'18. They allow to inspect the running state of the VM while it runs for debugging purposes. One of the main difference with out design is that the Maxine VM is metacircular, they do not have a simulation and a production mode as we do. We believe having two different modes allows us to easily generate a production artifact while still having nice debugging features. Having a full metacircular VM would be interesting. However, so far, most VMs used in production (Java, Javascript, etc.), even after the huge recent investments in the Javascript VMs by the four major web vendors, are still compiling through the C/C++ compiler and not metacircular. Hence, although a metacircular VM has interesting advantages, it is not clear it is that convenient to build a VM in such a way.

RPython toolchain. The RPython toolchain [RP06] was designed and implemented quite similarly to Open-Smalltalk-VM. Most of the VM code is written in RPython, a restricted Python, instead of Slang, and some leftovers are written in plain C. The main difference is that RPython is much closer to Python than Slang is to Smalltalk, RPython allows higher level structures such as dictionaries to be used. The design decision comes with its set of advantages and drawbacks. The key advantage is that the RPython code feels like Python code and is relatively quite easy to read write. The main drawback is that RPython to C compilation takes way 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).

Although the RPython code can be executed as normal Python code, for some reasons, the developers seem to think it is not worth to do it, mostly because executing code in such a way is very slow. The overall architecture of the RPython toolchain is different, which leads to a longer time to reach peak performance (though their peak performance is at least in theory better than with a template JIT as Open-Smalltalk-VM features). This time may be very significant in simulation mode. In addition, RPython was originally designed for Python, which does not feature snapshot by default, so they cannot abuse snapshots to work around the simulation slow performance.

Conclusion

We introduced and discussed the Open-Smalltalk-VM simulation infrastructure, used to develop and debug the VM. We believe it is a powerful tool allowing to reduce our development time and to allow to fix bugs quickly. In the near future, we plan to extend the simulator with customizable development tools, especially the moldable inspectors and debuggers [CNSG15, CGN14], to have a fancy user interface on top of the current simulation model, currently quite tricky to apprehend by new developers.

References

[B1́6] 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.

[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.

[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.

[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.

[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.