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 to generate the production VM through the C compiler. Two execution models are 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.

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 infrastructure 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 mention some of the limitations and how we worked around them. We discuss specifically how we use the VM simulator to develop and debug two core VM components, the garbage collector and the just-in-time compiler. Then, we discuss how we use the simulation infrastructure 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, 7 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. Part of the code base was however narrowed down to a subset of Smalltalk, called *Slang*, to allow Smalltalk to C compilation [IKM+97] and generate an efficient VM through the C compiler. Effectively, the VM could be both simulated by executing the Slang code on top of the Smalltalk VM or compiled through the C compiler to native code to generate the production VM. At this point, the VM consisted mainly in:

- an interpreter with a spaghetti stack,
- a memory manager with a generational garbage collector (GC) and
- BitBlt, an extension for the user interface bit-based engine.

These three 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 such as 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). The existing simulator was not able to simulate the machine code generated by the JIT and had to be extended by binding multiple 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 and Section 4 reports our experience developing respectively the full GC and the JIT with our infrastructure. Section 5 shows how we abuse the simulator to analyse the runtime and direct design decisions to improve performance. 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 (the interpreter, the JIT

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/

and the memory manager), is written in Slang. The platform code, *i.e.,* Operating System dependent code such as file management or I/O is written directly in C.

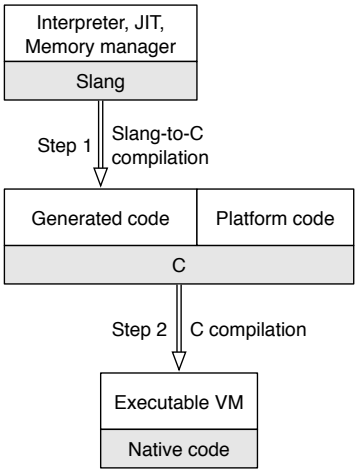

Figure 1. Cog VM compilation

The VM executable is generated in a two step process. Firstly, the Slang-to-C compiler translates the Slang code to C code, generating a few files. This first step takes several seconds. Secondly, the C compiler (depending on the platform, LLVM or GCC), translates the C code into an executable. This second step may take several dozens of seconds the first time, then, depending on what the programmer changed and what the C compiler have cached, it usually only takes a few seconds to recompile the C code.

The VM can be compiled in two main flavours, 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, debugging the garbage collector or evaluating new language features without dealing can be done in the interpreter-only VM, avoiding the JIT complexity.

There are multiple reasons why we keep writing the core components in Slang and not in C/C++. Most of them are small details. For instance, the Slang-to-C compiler generates C code slightly different from the Slang code using different annotations. It can for example duplicate the implementation of specific methods with specific constant operands to generate more efficient code in the interpreter. The main reason why we keep using Slang over C/C++ 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, allowing to develop and debug it with the Smalltalk development tools and a live programming experience.

2.1 Simulating the Virtual Machine

In this section, we firstly describe briefly the start-up sequence of Open-Smalltalk-VM using snapshots, a key aspect of the VM simulator success. 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 compiled 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 process active at snapshot time.

When programming with Smalltalk, the programmers usually starts 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 set of live objects. When the changes are done, the developer takes a new snapshot, which includes its changes. Deployment is performed using a snapshot containing the deployed application and the required core librairies (unused librairies and development tools may be removed from the deployed snapshot to decrease memory footprint and for security purposes).

Starting up Open-Smalltalk-VM from 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.

Figure 2 describes, from a high-level perspective, the memory used by Open-Smalltalk-VM at runtime and the simulated counter-part. Let's detail briefly the memory used by the production VM, on the top of the figure. On the left side, which is usually the low addresses, we can find Text, the memory section holding the native code of the VM (the interpreter, the memory manager and the JIT itself, but not the code compiled by the JIT). Then, Data holds initialized and uninitialized data, mainly the C variables used globally in the VM. On higher addresses, we can 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 (optional executable section, exists only if the JIT is enabled), for the young objects and for the old objects present in the snapshot started plus a little more space for the first tenures[2] to be performed without triggering the full garbage

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

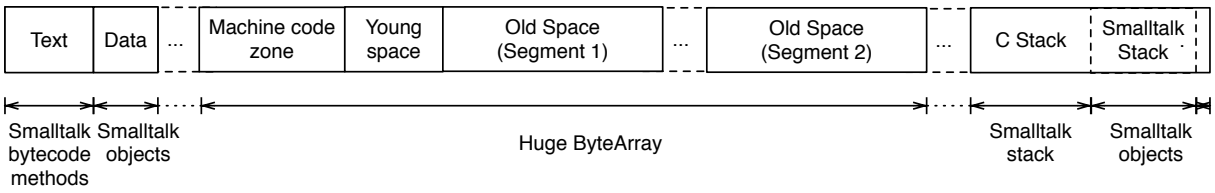

Figure 2. Runtime Memory and Simulated Counter-Part

collector. Later during execution, as old space grows, new memory regions are mmap'ed on higher addresses to store other old objects. Usually at very high addresses, we can find the C stack. Room in the C stack is alloca'ed at VM start-up to hold the Smalltalk stack. Both stacks are disjointed and managed differently.

In the simulator, 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 Slang variables, normally translated to C variabled, 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, etc.). The Slang code is executed as Smalltalk code. The Smalltalk 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 in the simulator.

All the Slang code is implemented in multiple Smalltalk classes, to organise the code and to add flexibility through polymorphism. For example, the AbstractCompactor class has two subclasses, one implements a sweep algorithm and the other a compact one. For production, at Slang-to-C compilation time, all the code is compiled in a single C file. All the flexibility 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 abused for debugging purposes. Still re-using the same 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.

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 holds, when the JIT is enabled, the machine code generated at runtime.

Bindings to processor simulator libraries (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. Such traps can be caught is the VM simulator, which then resumes Slang simulation by calling the correct method based on the invalid processor instruction. Calling machine code from Slang requires to start the machine code simulation but also to raise an exception to stop Slang simulation. Indeed, in the production VM, the processor can execute the C code or the native code generated by the JIT, but not both at the same time.

Machine code simulation can be performed in two different ways. The processor simulator can start simulating code until it meets an invalid instruction. That version is convenient because it is the fastest to execute, while still catching errors such as invalid memory accesses. Alternatively, it can simulate one instruction at a time. This second version is slower, but it 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 JIT, this start-up time may still be too long and move the live programming experience to an edit-compile-run cycle, which we want to avoid.

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, calls the bound processor simulator to disassemble the code and decorate the disassembly with Smalltalk specific information. 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 includes mock addresses for all the object the JIT may require to generate the machine code of a given method. Figure 3 summarizes the 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.

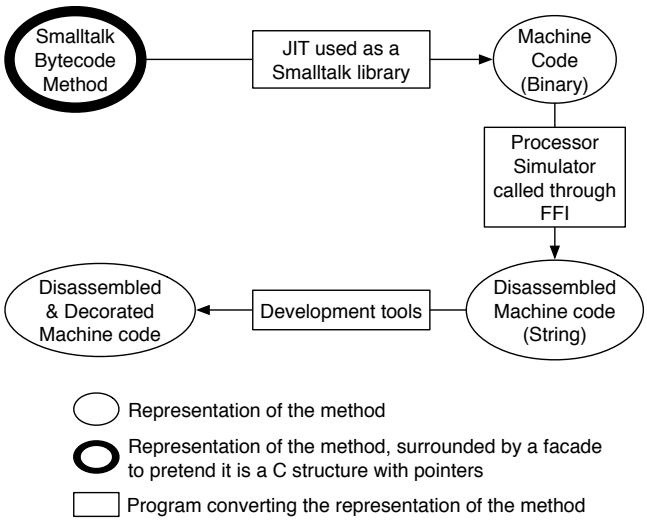

Figure 3. In-image compilation

Templates. The JIT generates a sequence of machine instructions for a given pattern of bytecodes. We will 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.

2.3 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 this kind of bugs are really 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 be simulated.

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. We usually enabled conditional breakpoints in-between machine instructions only when we reach a point in the simulator where the bug is about to happen since it is way slower, so the overall performance in this context is not really relevant.

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 worse. 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 also 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 debugging can be done in the simulator, 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, 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.).

The main limitation we have right now is with Foreign Function Interfaces (FFI). We have a significant amount of bugs in 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.

3 Garbage Collection Development

Recently, to evaluate new old space garbage collection algorithms we designed against standard algorithms, we implemented a Mark-Sweep in addition to the existing Mark-Compact collector. 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 was kind of 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 assertions. Each assertion ensures a specific state is as the VM developer would expect it to be and stops or log the incorrect result if not. We have a multi-level assertion system. Assertions written in plain Smalltalk are convenient since more complex constraints can be expressed easily using high level structures (Sets, etc.), but they can be performed only in the simulator. Assertions written in Slang are executed in the simulator but they are also compiled to C. The C compiler, based on a compilation flag, chooses to compile the VM with or without assertions. The VM with assertions is used for debugging and to recreate snapshots just before a VM 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 had to write the missing pieces inside the debugger, installing the new code at runtime, as one can do it in any Smalltalk application. 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, and likely an assertion would fail or an error such as incorrect memory access would be raised, the erratic behavior would have happened in a copy of the heap. The simulator can then reproduce the exact same erratic behavior as many times as the VM developer wants duplicating again and again the same original heap. This is very convenient to debug specific GC bugs. Indeed, 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, which is usually very difficult or impossible.

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

In the 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 bytecodes in the bytecode to bytecode optimizer, we had to introduce new unsafe bytecodes allowing, for example, accesses in arrays without any checks (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 comes 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 would generate that method to machine code. Then, when the simulator effectively stopped, 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 previous ones wrote on stack the invalid address. 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 (See next subsection).

The debugging process discussed here is exampled in a youtube video[3].

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

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. 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 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), we've just had to evaluate performance and correctness through benchmarks and tests to validate the implementation.

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

5.1 Analysis example

One of the first analysis 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. As show in Table 1, 1752 methods were compiled to machine code by the JIT, 6352 sends[4] 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 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 % 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 used for these analysis is 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"[5].

5.2 Directing the VM behavior

The results of the analysis are sometimes 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) [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) and fall back into a standard look-up routine if nothing is found after three attempts.

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 implemented different tools to help them working more efficiently on their VM, but they rarely publish about it. We focus in this section on two related work.

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

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

Maxine Inspectors. The Maxine inspectors [Mat08] were demonstrated at OOPSLA'18. They allow to inspect the running state of the Maxine VM while it runs for debugging purposes. One of the main difference with out 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 artefact 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 are 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, unlike Slang which feels like C and is as easy to write as C. 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 such a thing, mostly because executing code in this 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 our JIT). This time may be quite 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

[Bí16] 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.

[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.