HAL
open science

Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8

Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, Pierre Misse-Chanabier, Carolina Hernandez Phillips

To cite this version:

Guillermo Polito, Pablo Tesone, Stéphane Ducasse, Luc Fabresse, Théo Rogliano, et al.. Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8. MPLR '21, Germany, Sep 2021, Münster, Germany. 10.1145/3475738.3480715 . hal-03332033

HAL Id: hal-03332033

https://inria.hal.science/hal-03332033v1

Submitted on 3 Sep 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cross-ISA Testing of the Pharo VM: Lessons Learned While Porting to ARMv8

Guillermo Polito
Univ. Lille, CNRS, Inria, Centrale Lille, UMR 9189 CRIStAL, F-59000
Lille, France
guillermo.polito@univ-lille.fr

Pablo Tesone
Stéphane Ducasse
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
{name}.{surname}@inria.fr

Luc Fabresse
IMT Lille Douai, Institut Mines-Télécom, Univ. Lille, Centre for Digital Systems, F-59000
Lille, France
luc.fabresse@imt-lille-douai.fr

Théo Rogliano
Pierre Misse-Chanabier
Carolina Hernandez Phillips
Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL
Lille, France
{name}.{surname}@inria.fr

Abstract

Testing and debugging a Virtual Machine is a laborious task without the proper tooling. This is particularly true for VMs with JIT compilation and dynamic code patching for techniques such as inline caching. In addition, this situation is getting worse when the VM builds and runs on multiple target architectures.

In this paper, we report on several lessons we learned while testing the Pharo VM, particularly during the port of its Cogit JIT compiler to the AArch64 architecture. The Pharo VM presented already a simulation environment that is very handy to simulate full executions and live-develop the VM. However, this full simulation environment makes it difficult to reproduce short and simple testing scenarios. We extended the pre-existing simulation environment with a testing infrastructure and a methodology that allow us to have fine-grained control of testing scenarios, making tests small, fast, reproducible, and cross-ISA.

We report on how this testing infrastructure allowed us to cope with two different development scenarios: (1) porting the Cogit JIT compiler to AArch64 without early access to real hardware and (2) debugging memory corruptions due to GC bugs.

CCS Concepts: • **Software and its engineering → Runtime environments**.

Keywords: just-in-time compilers, virtual machines, ARM, testing, ports, Cross-ISA

1 Introduction

Testing and debugging a Virtual Machine is a laborious task without the proper tooling. This is particularly true for VMs that support code generation for JIT compilation and dynamic code patching for techniques such as inline caching. This complexity is aggravated when the VM builds and runs on multiple target architectures [1].

Several solutions have been proposed to aid in VM debugging tasks. Traditionally, VM simulation environments have appeared in Self [19], Smalltalk [12, 17] and Metacircular VMs such as Maxine [20]. Complementary to simulation environments, multi-level debuggers [14, 21] aid VM developers to switch views between the program-level and the implementation (VM)-level. These solutions are indeed beneficial to identify and track problems once an issue has been spotted and reproduced. However, reproducing bugs still remains an expensive and long task because millions of instructions may need to be executed before hitting the actual problem. For example, it has been reported that debugging memory corruption bugs in a simulation could take several hours of execution[1]. Recently, the team of Maxine recently reported a test-based infrastructure for cross-ISA debugging [13]. They report that still debugging happens in gdb in a different abstraction level than the original source code and that they are not able to cover many parts of their codebase.

In this paper, we report on several lessons we learned while testing the Pharo VM, particularly while porting its Cogit JIT compiler to the AArch64 architecture. The Pharo VM presented already a high-level simulation environment that is very handy to simulate full executions and live program the VM [17] (See Section 2). We extended the pre-existing simulation environment with a testing infrastructure that allows us to have fine-grained control of testing scenarios, making tests small, fast, reproducible, and cross-ISA (See Section 3). We defined a hybrid testing methodology that takes advantage of the strengths of unit-testing, full-system simulation, and real-hardware execution where they perform better. In Section 5, we describe how this testing infrastructure allowed us to cope with two different development scenarios: (1) porting the Cogit JIT compiler

[1] http://forum.world.st/OpenSmalltalk-opensmalltalk-vm-Reproduceable-Segmentation-fault-while-saving-images-444-td5106898i20.html

to AArch64 without early access to real hardware and (2) debugging memory corruptions due to GC bugs.

2 Context: The Pharo VM

The Pharo Virtual Machine is an industrial level Virtual Machine written in Pharo itself and transpiled to C using a VM-specific translator called Slang [12]. The VM implements at the core of its execution engine a threaded bytecode interpreter, a linear non-optimising JIT compiler named Cogit [16] that includes polymorphic inline caches [10] and a generational scavenger garbage collector that uses a copy collector for young objects and a mark-compact collector for older objects [18]. The following numbers illustrate the complexity of this Virtual Machine:

- It implements 255 bytecodes, organized in a total of 77 different families [2].
- It implements about 340 primitive methods, with a number of them both duplicated in the interpreter and in the JIT compiler.
- The compiler defines about 150 different IR instructions.

2.1 Slang

The Pharo Virtual Machine is written in a subset of Pharo that is easily transpilable to efficient C. As Pharo is a Smalltalk inspired language, the Pharo to C translation is done using Slang, a Smalltalk-to-C VM-specific transpiler [12]. Slang operates by translating a group of classes into a single C file. Methods are translated into functions, message-sends are translated as function calls. While the Pharo source program presents dynamic behaviour such as polymorphism, exceptions, or runtime reflection, Slang does not allow many of those: it either forbids them at translation-time or generates invalid C code.

Using Slang to develop the Pharo VM has two key advantages. First, Slang automatically introduces interpreter optimisations such as (a) the localisation of critical variables (frame pointer, instruction pointer) [15], (b) the inlining of bytecode cases inside the interpretation loop, or (c) threaded code [6]. Second, it allows us to simulate the Pharo VM just by executing it as normal Pharo code, avoiding expensive change-compile-test development cycles [17].

2.2 The Cogit JIT Compiler

The Cogit JIT compiler is a non-optimising method-based linear JIT compiler originally implemented by Miranda [16]. It uses a linear 2-address-code intermediate representation that does not explicitly model a control flow graph (CFG). The compiler implements monomorphic inline caches as linked sends, and polymorphic inline caches as stubs. The compiler's entry point for compilation is a single method where bytecodes are translated in linear fashion into machine code: the code layout of the generated machine code almost entirely mirrors the code layout of the bytecode. Compiling a method includes three main phases:

- **1. Bytecode scan phase** iterates the bytecodes of a method to extract meta-data from them. For example, it decides whether the method requires to be frame-less or not based on the presence of interruption points (message sends and backjumps).
- **2. Bytecode parsing phase** does an abstract interpretation of the bytecodes performing a stack bytecode to register IR transformation.
- **3. Code generation phase** computes IR instruction offsets and assembles the final machine code for the current platform.

2.3 CogRTL Intermediate Representation

The Cogit JIT compiler uses a linear 2-address-code intermediate representation named CogRTL. An interesting design point of the CogRTL IR is that it uses a fixed number of virtual registers with names such as ReceiverRegister or ClassRegister. Such registers have concrete roles at some point during the compilation (*e.g.,* they refer to the current receiver or the receiver's class at method entry point), and when those roles are already fulfilled or not required they are used as general purpose registers. An example of such behaviour are some compiler intrinsics and machine code versions of primitive methods.[2]

The fixed virtual register design avoids the need of a complex register allocator. Instead, virtual registers are allocated ahead of time to physical registers as a compiler configuration for each supported backend/platform. Notice that in the current architecture, a register allocator would have very short windows of code to improve because of three reasons: (1) the compiler does not implement inline substitution, (2) it spills all registers before any message-send, and (3) message sends are omni-present in Pharo's code [22].

As a final point on CogRTL's design: it aims to be as machine-independent as possible. There is a clear separation between the compiler's front end that parses bytecode and generates an IR, and the compiler's backend that generates machine code from the IR. Machine specific decisions such as the ahead-of-time register allocation are backend-specific. However, there are some machine-specific decisions that cripple into the frontend making that a single method can produce different IRs in different machines. For example, when the backend is a RISC machine the frontend produces IR instructions to push the link register on a method's preamble.

[2]Primitives are native methods in Smalltalk/Pharo's jargon.

2.4 Code Patching and PICs

Besides method compilation, the Cogit JIT compiler makes use of machine code patching without using any of the support explained above. Instead, machine code patching is implemented in the compiler's backend, reusing most of the compiler's assembler, but it requires in addition to implement a partial disassembler to identify the instructions to patch. Machine code patching happens in two main cases during the execution: updating/linking of mono/poly/mega-morphic inline caches and updating object references in machine code when objects are moved by the garbage collector.

The Cogit JIT compiler implements inline caches [5] and polymorphic inline caches [9, 11]. Briefly, code patching of the inline caches works as follows. Call-sites are initially compiled as calls to send trampoline routines. Send trampolines eventually perform the method lookup and link the call-site making it a monomorphic call-site: they rewrite the call to the trampoline to a call to a type-checked entry point of the found method. A monomorphic cache miss happens when the receiver is of a different type than what the linked method expects, thus upgrading the monomorphic inline cache to a polymorphic inline cache (PIC). Instead of recompiling a method with a PIC in place, PICs are implemented as machine code stubs. The so far monomorphic call-site is re-linked to the PIC stub in the same way it was done for the monomorphic case. PICs are patched with new cases at runtime when a (*type*, *method*) pairs cause PIC misses, up to a certain threshold. Finally, when a PIC surpasses the threshold it is upgraded to a megamorphic inline cache. Megamorphic caches are implemented as stubs shared by many call-sites and linked in the same way monomorphic and polymorphic caches were.

3 An Iterative Test-Based Methodology

The Pharo VM presents a high-level simulation environment that is very handy to simulate full executions and live-program the VM [17]. The simulation environment is a hybrid execution environment. The interpreter, JIT compiler and memory manager are written in a Pharo subset that is transpilable to C using Slang, and are executable as normal Pharo code. The generated runtime, *i.e.,* the machine code compiled methods generated by the JIT compiler, is executed using a machine code simulator. In Pharo, we have extended the original machine code simulation infrastructure to use Unicorn[3], a QEMU-based machine code simulation library. Unicorn provides a small yet powerful API to simulate machine code on multiple architectures.

We extended the pre-existing simulation environment with a testing infrastructure that allows us to have fine-grained control of testing scenarios, making tests small, fast, reproducible, and cross-ISA. The VM development environment is illustrated in Figure 1. During simulation, the VM has a heap and a native code cache memory regions instantiated as Pharo ByteArrays and is configured to work on a particular ISA/architecture. When JIT'ted code is simulated, the simulation environment gives control to the Unicorn machine simulator and when this latter hits a trampoline, it gives control back to the Pharo-side of the simulation. Our testing infrastructure lies within the simulation environment and has access to the same infrastructure as the full-simulation. Eventually, the VM is transpiled to C and compiled for a given ISA/architecture.

Based on this infrastructure we defined a hybrid testing methodology that mixes three different execution modes: unit-testing, full-system simulation, and full-system real-hardware execution. Our methodology takes advantage of the strengths of unit-testing, full-system simulation, and real-hardware execution where they perform better.

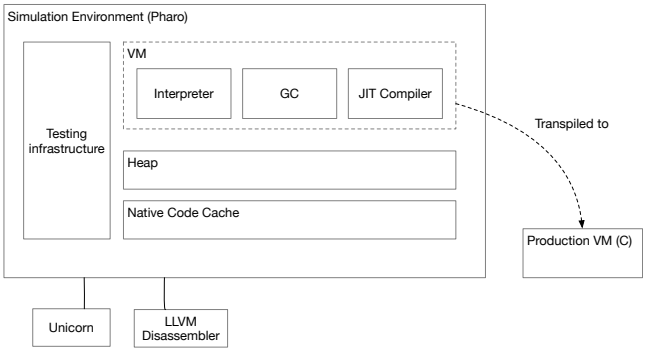

Figure 1. Development environment of the Pharo VM. The VM is executed as Pharo code in the simulation environment and transpiled to C to produce the production artefact. The simulation environment has its own heap and native code cache used during the simulated execution. The production VM will allocate similar regions on the real execution. This new testing infrastructure extends and makes use of the existing simulation environment.

3.1 VM Testing: An Agile Perspective

Each of these execution modes has different benefits and constraints, as shown in Table 1. Real-hardware is not always readily *available* or *easy to debug*, but its execution is the most *precise* in comparison with simulated hardware. At the same time, testing changes in real hardware is *expensive to compile and run*, leading to slow develop-compile-test feedback cycles. On the other side of the spectrum, unit tests are a handy way to express *reproducible and representative scenarios* and specially are capable of capturing regressions. However, since they are based on a simulation environment, they suffer from execution *imprecisions* because the simulation machinery is not 100% representative of real executions.

[3]https://www.unicorn-engine.org/

Table 1. Characterisation of different execution modes.

	Unit Testing	Full-System Simulation	Real Hardware Execution
Feedback-cycle Speed	High	Low	Very Low
Availability	High	High	Low
Reproducibility	High	Low	Low
Precision	Low	Low	High
Debuggability	High	High	Low

Based on these observations, we characterise the three different execution modes as per Table 1, using the following criterion:

- **Feedback-Cycle Speed.** How fast is the develop-compile-test cycle for a single test scenario? Unit tests support fast development cycles because they work on small and precise scenarios and do not require a full system initialization. Full-System simulations have slower development cycles because they require a full-system initialization plus the time to arrive to the interesting execution spot. Full-system real hardware execution has even slower feedback cycles than the full-system simulation because they require also a full VM compilation, which in our case includes transpilation, interpreter post-processing and C compilation.
- **Availability.** Is the testing scenario readily available to execute? On the one hand, simulation-based solutions (Unit tests, full-system simulations) are highly available although tied to the availability of a machine code simulator, which are in general software-based portable solutions. On the other hand, hardware-based solutions (real hardware) are much less available, as they require target hardware access.
- **Reproducibility.** What are the chances of reproducing a single test scenario? Full-System executions (simulation, real hardware) have low reproducibility by default because millions of instructions may need to be executed before hitting an actual problem, and non-determinism may worsen the problem. Contrastingly, unit-tests are by construction repeatable.
- **Precision.** How precise is the scenario execution? Simulation-based solutions (Unit tests, full-system simulations) have generally lower precision than real hardware, because a machine code simulator introduces a distance between both executions. For example, the versions of QEMU and the Unicorn machine simulator we use do not perform stack pointer alignment checks on AArch64 because of performance reasons.
- **Debuggability.** Is the testing scenario easy to debug? Simulation-based solutions (Unit tests, full-system simulations) are often built with debugging support in mind. On the other hand, real hardware execution requires abandoning the abstractions of the simulation environment and often deal with abstractions closer to the machine.

3.2 A Single Methodology to Rule Them All

Based on these constraints, we developed the methodology illustrated in Figure 2:

1. Spend as much time as possible working on unit tests: they are cheap to write and execute.
2. Introduce full-system simulations when we consider coverage is good enough: it represents a full-system execution without access to real hardware.
3. Introduce testing on real hardware when we consider full-system simulation reliable enough: to obtain final precise and reliable feedback.

The key of this methodology is to augment the team's velocity and to apply Test-Driven-Development (TDD) techniques from Software Engineering to JIT compiler development. In this schema, unit tests are the main development unit because of their fast feedback. In our experience, unit-tests have caught the vast majority of bugs without the need to validate it in real hardware. Really few problems seemed to remain *untestable* at first because of simulation imprecisions, but most of them became testable after introducing fixes in the simulation environment.

We then use full-system executions (either simulated or on real hardware) to get feedback on failing untested functionalities. As soon as a full-system execution fails, we use that as feedback for unit testing: we build one or more failing tests for the scenario, we manually do test reduction on them, and finally fix the actual problem to make them pass. Such a mixed mode methodology allowed us to port most of the JIT compiler to AArch64 before we had access to real hardware. As a nice side effect, we created a large set of tests that are small, fast to run, reproducible and cross-ISA, and even unveiled old bugs in the pre-existing backends.

3.3 The Pharo VM Testing Infrastructure

The most complex part of writing a VM-specific unit test is defining the test fixture/setUp. Indeed, executing a short method doing `push a, push b, send +, return` requires the initialization of the heap, the execution stack, internal VM data structures, among others... Because of these, the largest effort of our infrastructure was put on extending SUnit[4] with VM-specific initializations and testing primitives. The core of our infrastructure lies within four abstract Test classes that developers extend to define their new testing scenarios:

- **Heap Initialized Test Case.** The root of our testing hierarchy. All tests that inherit from this class run configured with a heap. This class also provides facilities to create/load classes and instantiate objects.

[4]SUnit is the Pharo Unit Testing Framework

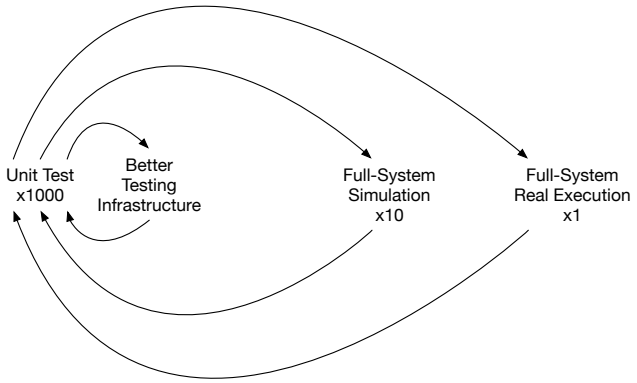

Figure 2. Illustrating the Methodology of the AArch64 Port. Our methodology was based mainly on developing mostly black-box unit tests. When the number of running unit tests gave us enough confidence we introduced full-system simulations, and when we were confident enough about the full system simulations we spent time on real hardware. At each step, the problems we found were reduced to be bare minimum reproducible, translated as unit tests and introduced as regression tests.

- **Interpreter Test Case.** All tests that inherit from this class run configured with a heap and an execution stack. They have access to a call-stack builder to create well-formed stack frames.
- **Compiler Test Case.** All tests that inherit from this class run configured with a heap, an execution stack, and a code cache. They have access to a simplified compiler interface that compiles either entire methods, compiler intrinsics and individual bytecodes. Moreover, all tests extending this class are by default executed on all supported ISAs.

4 VM Testing Guidelines

In our journey of writing tests for the Pharo VM, we developed a large test suite that covers different VM concerns such as the interpreter, the compiler, the object format, and the garbage collector. In this section, we report on our experience designing VM-specific unit tests in the form of several guidelines. These VM-specific guidelines emerged from applying general well-known testing principles such as:

Fast. A test should be as instantaneous as possible to ensure we have a fast feedback cycle.

Reproducible. Test executions should be deterministic.

Repeatable. Several test runs should be independent from each other.

Unitary. Each test should test a single concern. Particularly, if a functionality has many aspects to test such as several border-cases, a test exists for each of them.

Validating. A test must have at least one assertion, and ideally only one.

4.1 Black Box Testing

We use black-box testing in the vast majority of our tests: we test externally observable behaviour. For example, most of our memory management tests are word-size independent. Another example are compiler tests: we avoid as much as possible exposing internals of the compiler such as the IR or the generated machine code within the test code. Indeed, most of our compiler related tests work with the granularity of a single bytecode. This design gives our tests two main properties: (1) resistance to changes in the VM and compiler implementation, and (2) architecture/ISA independence.

Listing 1 shows one of the simpler tests in our compiler test suite. The test uses our compiler interface to compile a single bytecode, execute the generated machine code and test that the constant zero was pushed to the operand stack. This test is written once and runs automatically in all supported compiler backends (AArch64, AArch32, x86, and x86-64), both in 32bits and 64bits machines.

```
1 testPushConstantZeroBytecodePushesASmallIntegerZero
2   self compile: [ compiler genPushConstantZeroBytecode ].
3   self runGeneratedCode.
4   self assert: self popAddress equals: (memory
      integerObjectOf: 0)
```

Listing 1. Test compilation of push constant zero.

4.2 Cross-ISA Testing Using Test Parameterisation

Our test infrastructure makes use of parameterized tests to automatically run a single test on many configurations. We express such configurations as a matrix. Listing 2 shows an example configuration of our compilation testing matrix. Our root test classes already provide pre-configured configuration matrices for the most common cases. For example, the interpreter and memory manager tests automatically run in 32bits and 64bits configurations. Compiler tests are configured to run in all supported compiler backends. Listing 2 illustrates the compiler configuration matrix which at the moment of writing this paper includes cases for AArch64, AArch32, x86 and x86-64.

```
1 testParameters
2   ^ ParametrizedTestMatrix new
3       addCase: { #ISA -> #'aarch64'. #wordSize -> 8};
4       addCase: { #ISA -> #'x86'. #wordSize -> 4};
5       addCase: { #ISA -> #'X64'. #wordSize -> 8};
6       addCase: { #ISA -> #'ARMv5'. #wordSize -> 4};
7       yourself
```

Listing 2. Testing matrix for compiler tests.

4.3 Grow Slowly in Complexity

A simple heuristic we use to develop tests is to always start by the simplest test we can write. If a test cannot be easily written with the testing infrastructure as-is this is an indicator that either the infrastructure is missing support for some feature, or that a simpler test needs to be developed before.

A second heuristic is to treat test code as any other code. In other words, test code can be subject to refactorings, cleaning, and the extraction of other reusable components.

4.4 Dealing with Platform Specific Constraints

The guidelines above make tests that are generic by default, but generic tests do not deal correctly with platform specific constraints. Platform specific constraints arise from different VM configurations, target operating systems or target processors. For example, AArch64 differs from Intel based processors in many ways: multiplication overflow does not set the overflow flag, subtraction present an inverted carry flag, the stack pointer has alignment restrictions. Some operating systems impose exclusive write-executable permissions. VM specific differences appear between 32bits and 64bits memory models.

In cases like the ones exemplified above, our testing infrastructure does not forbid testing, but requires more fine-grained control and *narrowing* the scope of test. We achieve fine-grained control in platform specific tests by allowing tests to manually feed Intermediate Representation instructions to the compiler. Also, when narrowing test scenarios for a particular platform, we observed the appearance of two main cases in our test suite:

Generic tests with exceptions. Some tests are valid for all but one matrix configuration. In such cases, the specific configuration is explicitly skipped for that test.

Platform Specific Tests. Some tests are valid only for a single platform. In such cases, a separate test class with a specialized test matrix is written to host them. An example of this are the ARM stack alignment tests, in class VMARMStackAlignmentTest, with a test matrix using a single case, as shown in Listing 3.

```
1 testParameters
2   ^ ParametrizedTestMatrix new
3       addCase: { #ISA -> #'aarch64'. #wordSize -> 8};
4       yourself
```

Listing 3. Testing matrix for a platform specific test.

5 Case Studies

All unit tests and the testing infrastructure reported in this article are available in the package VMMakerTests, in the branch headless of our Git public repository[5]. Our testing suite defines as-per-today 1367 written unit tests[6], representing a total of 3603 run tests when the different matrix configurations are taken into account. A full run of our unit test suite amounts to a total time of ~2.5 minutes in our continuous integration server to the day we wrote this article. Table 2 details the processor independent test cases, and Table 3 details the processor dependent test cases.

In the remaining of this section we report our experience with two concrete case studies where our testing infrastructure has been proven useful. We provide in appendix code examples of both scenarios.

5.1 Porting the Cogit JIT Compiler to AArch64

We used this approach to port the Cogit JIT compiler to AArch64. When we started the port, the JIT compiler had mainly assembler tests that did not follow the principles stated above. Following our guidelines above, we guided our port to AArch64 by writing unit tests in increasing complexity, specifically by translating and executing single bytecodes and working exclusively on our test suites: we performed no full-system execution of any kind. For example, the first tests we wrote covered the bytecodes for push and pop of constants, we later introduced tests for message-send bytecodes which required to add testing support for trampolines, and later we introduced tests for polymorphic inline caches and their patching. This approach helped us with introducing the assembler support gradually.

In addition, some testing scenarios required modifications in the compiler's intermediate representation *e.g.,* to test multiplication overflow we introduced a new JumpMultiplyOverflow instruction because AArch64's multiply instruction does not set the overflow flag. The JumpMultiplyOverflow is defined by default on all platforms as a JumpOverflow and redefined for AArch64. Having a cross-ISA and fast test suite allowed us to easily refactor and verify the change not only in AArch64 but in all our supported platforms in a couple of minutes, making sure we did not introduce any regression in the compiler.

Once our test suite was large enough and gave us enough confidence on our implementation, we switched to run full-system simulations. During a full-system simulation, we run the Pharo VM as if it was run from the command line. This made the simulation environment exercise parts of the code that were not necessarily covered by tests. When the simulation reached a bug or an error, we investigated the problem and rolled-back to design unit tests for the failing scenario, incrementing our test coverage with concrete cases.

When the simulation was able to execute for long periods of time without failing, we had a strong confidence in our implementation. We switched to real-hardware execution, knowing that we were going to face at least the stack pointer

[5] https://github.com/pharo-project/opensmalltalk-vm

[6] https://github.com/pharo-project/opensmalltalk-vm/commit/8f3028057c4f98afa38c5645223d19ecfbfb2bf3

Table 2. Processor Independent Tests (32bits / 64bits): These tests center in the memory representation. They are executed for 32 and 64 bits machines. They don't use Unicorn's machine simulator.

VM Component	Operation	Independent Tests	Variations	Total Executions
	Method Builder	10		20
Test Infrastructure	Stack Builder	18	32 bits / 64 bits	36
	Total	*28*		*56*
	Stack Reification	7		14
	Context / Stack Mapping	13		26
	GC Data Structures	13		26
	Unmovable Objects	9		18
	Old Object Garbage Collection	63		126
Object Memory	Young Objects Garbage Collection	38	32 bits / 64 bits	76
	Weak Object Garbage Collection	9		18
	Ephemeron Object Garbage Collection	19		38
	Old Objects FreeSpace Management	85		170
	Memory Structure Preconditions	30		60
	Total	*286*		*572*
	Bytecode Tests	43		86
	Method Lookup	15		30
Interpreter	Object Representation	17	32 bits / 64 bits	34
	Primitives	62		124
	Total	*137*		*274*
	Total	**451**		**902**

alignment restrictions that were not simulated properly. During real-hardware execution, we only found three missing features in our JIT port:

Stack alignment restrictions. We faced it right away in the first execution. This problem lead us to design platform specific tests to check the alignment, extend the unicorn machine simulation with support for stack alignment check, and refactor the JIT to use a general purpose register (x28) as stack pointer register for the Pharo execution stack. Since the Pharo VM uses as stack a stack allocated memory region, the x28 register is guaranteed to be always above the real stack pointer.

Marshalling of Single-precision Floats in FFI. A bug in our assembler has been found: when reading/writing single-precision floats, we used the 64bits instruction variant. Our tests were not covering all possible scenarios and this caused memory corruptions. This problem was not unveiled by the full-system simulation because it takes a huge amount of time to reach this point.

OSX W+X. When preparing the Pharo VM build for the new M1 Apple Silicon Machines, we had to adapt the entire JIT architecture to use the mmap JIT permissions suggested by Apple[7].

[7] https://developer.apple.com/documentation/apple-silicon/porting-just-in-time-compilers-to-apple-silicon

5.2 Debugging and Testing Memory Corruptions

We used this same approach recently when facing a bug in the prototype ephemeron [8] implementation reported by a member of the Pharo community[8]. We started by investigating the issue in real-hardware using gdb. This first exploratory step was to be able to identify more clearly the symptoms of the problem and making hypothesis about the potential causes. A quick analysis showed us a heap corruption where the crash manifested itself far away from its real cause. After some debugging iterations we could observe that the corrupted objects were effectively the ephemeron objects, and that the corruption happened during GC, and we proceeded to reproduce the same problem in a unit test and debug it in the simulation environment. During this debugging session in the simulation environment we found other related bugs. The fixes and associated tests were published in the following pull request: https://github.com/pharo-project/opensmalltalk-vm/pull/183.

6 Related Work

Several solutions have been proposed in the past to aid in VM debugging tasks and programming language implementation validation.

[8] https://github.com/pharo-project/pharo/issues/8153

Table 3. Processor Dependent Tests (x86 / x86-64 / AArch32 / AArch64)
These tests center in the execution of generated code. They depend on the target machine. The tests are executed for the 4 different platforms (x86 / x86-64 / AArch32 / AArch64). These tests run using Unicorn's Machine Simulator. We include here some platform specific tests for AArch64. These tests are used to validate AArch64 specific requirements. Also, Float immediate representations and selector dereferencing are only available in 64 bits systems (x86-64 / AArch64).

VM Component	Operation	Independent Tests	Variations	Total Executions
Platform Specific Tests	AArch64 Instruction Encoding	27	AArch64	27
	AArch64 Stack Alignment Simulation	2		2
	Total	*29*		*29*
Stack Manipulation	Stack Reification	15	All 4 Platforms	60
	Total	*15*		*60*
Primitives	Integer Division	8	All 4 Platforms	32
	Float Immediate Operations	21	x86-64 / AArch64	42
	FFI Marshalling	46	All 4 Platforms	184
	Integer Immediate Operations	36	All 4 Platforms	144
	Object Size Operations	20	All 4 Platforms	80
	Object Access Operations	32	All 4 Platforms	128
	Total	*136*		*610*
Test Infrastructure	Unicorn Validation	17	All 4 Platforms	68
	Total	*17*		*68*
JIT Compiler	Code Compaction	11	All 4 Platforms	44
	Method Header Generation	2	All 4 Platforms	8
	Primitive Inlining	33	All 4 Platforms	132
	Selector Dereferencing	4	x86-64 / AArch64	8
	Abort Routine Generation	3	All 4 Platforms	12
	Bytecode Compilation	144	All 4 Platforms	576
	Special Selectors Message Send	68	All 4 Platforms	272
	Monomorphic Message Send	3	All 4 Platforms	12
	Polymorphic Message Send	55	All 4 Platforms	220
	Megamorphic Message Send	14	All 4 Platforms	56
	VM Routine Invocation	28	All 4 Platforms	112
	Total	*365*		*1452*
	Total	**589**		**2219**

VM Simulation and Meta-circular VMs. VM frameworks and Meta-circular VMs have offered for a long time simulation environments that helped in testing and debugging virtual machines. Such is the case of Self [19], Smalltalk [12, 17] and Maxine [20]. These solutions allow full-system simulations to ease debugging at the cost of slower and less-precise executions. Our solution complements full-system simulations with a unit-testing infrastructure and methodology, that helped us in fixing complex bugs and porting the JIT compiler to AArch64.

Compiler and VM Testing. Several works exist on the area of compiler testing, particularly on the automatic generation of test programs and oracles for their validation [4]. Chen *et al.,* do not study test generation or validation: they study the infrastructure required to produce maintainable, fast, focused and cross-ISA tests. Moreover, our tests cover not only the JIT compiler but other VM concerns such as the memory manager, the interpreter and the GC. We plan in the future to work on VM-specific test generation.

Recently, the team of Maxine reported a QEMU test-based infrastructure for cross-ISA testing and debugging [13]. They reported that this infrastructure helped them in porting their VM to AArch32 (ARMv7), similarly that what ours helped us in porting to AArch64. Still, their debugging happens in gdb in an abstraction level far-away from the original source code. Moreover, they reported they still cannot cover

many parts of their codebase. We believe that many of these limitations come from the fact that Maxine is a meta-circular VM that uses its JIT compiler to compile itself.

Finally, some work report recently efforts to validate in an automatic or semi-automatic way optimising compilers [3, 7]. Although this is not the focus of this paper, we plan to extend our infrastructure in the future to test our prototype optimising JIT.

7 Conclusion

Testing and debugging a Virtual Machine is a laborious task without the proper tooling. This is particularly true for VMs with JIT compilation and dynamic code patching. In this paper we present a testing infrastructure and a methodology that allows us to take advantage of the strengths of unit-testing, full-system simulation, and real-hardware execution where they perform better, and have fine-grained control of testing scenarios, making tests small, fast, reproducible, and cross-ISA.

We report on how this testing infrastructure allowed us to cope with two different development scenarios: (1) porting the Cogit JIT compiler to AArch64 without early access to real hardware and (2) debugging memory corruptions due to GC bugs. In our experience, unit-tests have caught the vast majority of bugs without the need to validate them in real hardware. Really few problems seemed to remain *untestable* at first because of simulation imprecisions, but most of them became testable after introducing fixes in the simulation environment.

References

[1] Bowen Alpern, Maria A Butrico, Anthony Cocchi, Julian Dolby, Stephen J Fink, David Grove, and Ton Ngo. 2002. Experiences Porting the Jikes RVM to Linux/IA32. In *Java Virtual Machine Research and Technology Symposium*. 51–64.

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

[3] Clément Béra, Eliot Miranda, Marcus Denker, and Stéphane Ducasse. 2016. Practical Validation of Bytecode to Bytecode JIT Compiler Dynamic Deoptimization. *Journal of Object Technology* 15, 2 (2016), 1:1–26. https://doi.org/10.5381/jot.2016.15.2.a1.

[4] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. 2020. A Survey of Compiler Testing. *Comput. Surveys* (May 2020), 1–36. https://dl.acm.org/doi/10.1145/3363562

[5] L. Peter Deutsch and Allan M. Schiffman. 1984. Efficient Implementation of the Smalltalk-80 system. In *Proceedings POPL '84*. Salt Lake City, Utah. https://doi.org/10.1145/800017.800542

[6] M. Ertl and David Gregg. 2003. The Structure and Performance of Efficient Interpreters. *J. Instruction-Level Parallelism* 5 (Nov. 2003).

[7] Olivier Flückiger, Gabriel Scherer, Ming-Ho Yee, Aviral Goel, Amal Ahmed, and Jan Vitek. 2017. Correctness of Speculative Optimizations with Dynamic Deoptimization. In *Principles of programming languages (POPL'17)*. https://doi.org/10.1145/3158137

[8] Barry Hayes. 1997. Ephemerons: A new finalization mechanism. In *Proceedings OOPSLA '97, ACM SIGPLAN Notices*. https://doi.org/10.1145/263700.263733

[9] Urs Holzle. 1994. *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. Ph.D. Dissertation. Stanford University, Stanford, CA, USA. Advisor(s) Ungar, David M. AAI9508373.

[10] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *European Conference on Object-Oriented Programming (ECOOP '91)*. https://doi.org/10.1007/BFb0057013

[11] Urs Hölzle, Craig Chambers, and David Ungar. 1991. Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings ECOOP '91 (LNCS)*, P. America (Ed.), Vol. 512. Springer-Verlag, Geneva, Switzerland, 21–38. https://doi.org/10.1007/BFb0057013

[12] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. 1997. Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*. ACM Press, 318–326. https://doi.org/10.1145/263700.263754

[13] Christos Kotselidis, Andy Nisbet, Foivos S Zakkak, and Nikos Foutris. 2017. Cross-ISA debugging in meta-circular VMs. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. 1–9. https://doi.org/10.1145/3141871.3141872

[14] Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. 2016. Multi-Level Debugging for Interpreter Developers. In *Companion Proceedings of the 15th International Conference on Modularity (MODULARITY Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 91–93. https://doi.org/10.1145/2892664.2892679

[15] Eliot Miranda. 1987. BrouHaHa — A Portable Smalltalk Interpreter. In *Proceedings OOPSLA '87, ACM SIGPLAN Notices*, Vol. 22. 354–365. https://doi.org/10.1145/38765.38839

[16] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine. In *Proceedings of VMIL 2011*.

[17] 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*. ACM, 57–66. https://doi.org/10.1145/3281287.3281295

[18] Dave Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices* 19, 5 (1984), 157–167. https://doi.org/10.1145/390011.808261

[19] David Ungar, Adam Spitz, and Alex Ausch. 2005. Constructing a metacircular Virtual machine in an exploratory programming environment. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1094855.1094865

[20] Christian Wimmer, Michael Haupt, Michael L. Van De Vanter, Mick Jordan, Laurent Daynès, and Douglas Simon. 2013. Maxine: An approachable virtual machine for, and in, java. *ACM Transaction Architecture Code Optimization* 9, 4 (Jan. 2013). https://doi.org/10.1145/2400682.2400689

[21] Thomas "Würthinger, Michael L. Van De Vanter, and Doug" Simon. 2010. Multi-level Virtual Machine Debugging Using the Java Platform Debugger Architecture. In *Perspectives of Systems Informatics*, Amir Pnueli, Irina Virbitskaite, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 401–412. https://doi.org/10.1007/978-3-642-11486-1_34

[22] Oleksandr Zaitsev, Stéphane Ducasse, and Nicolas Anquetil. 2020. *Characterizing Pharo Code: A Technical Report*. Technical Report. Inria Lille Nord Europe - Laboratoire CRIStAL - Université de Lille ; Arolla. https://hal.inria.fr/hal-02440055