

Proxies All the Way Down

Bootstrapping a Userland Speculative Optimizer for the OpenSmalltalk VM

Christoph Thiede
christoph.thiede@hpi.de

Hasso Plattner Institute
University of Potsdam, Germany

Marius Dörbandt
marius.doerbandt@student.hpi.de

Hasso Plattner Institute
University of Potsdam, Germany

Eliot Miranda
eliot.miranda@gmail.com

Hasso Plattner Institute
University of Potsdam, Germany

Marcel Taeumel
marcel.taeumel@hpi.de
Hasso Plattner Institute
University of Potsdam, Germany

Robert Hirschfeld
robert.hirschfeld@uni-potsdam.de
Hasso Plattner Institute
University of Potsdam, Germany

Abstract

Self-supporting dynamic-language systems aim to lift core components otherwise implemented in the virtual machine up into userland to enable the expression and development of those core components at the same level and with the same tools as applications. Yet, bootstrapping components that are critical for userland stability poses a chicken-and-egg problem: bugs in such core components can destroy the running system. The OpenSmalltalk VM is implemented in Smalltalk as a nested simulation, and transpiled to C. The new speculative optimizer *Scorch* increases performance and is architected as a userland component to ease development. However, debugging *Scorch* into existence has proven impractical as faulty optimizations crash the system and make iterative development very challenging.

We present OOPSIE: an approach that allows components such as *Scorch* to live in userland during both development and production without alteration. It is a meta-architecture that decouples development from execution, meaning that only a simulation might be damaged. We employ proxies for critical paths, replacing inter-userland interfaces with proxy-based host-to-simulation interfaces. Using OOPSIE, we revived the development of *Scorch*, identifying and fixing multiple defects, and extended the OpenSmalltalk simulator with improved debugging capabilities by reusing existing userland tools across system boundaries. Our experience demonstrates that the OOPSIE architecture is effective in bootstrapping userland system capabilities by isolating bug effects to the simulation, supporting uninterrupted live-programming workflows even before a new component can sustain itself.

CCS Concepts: • Software and its engineering → Virtual machines; Development frameworks and environments; Software prototyping; Just-in-time compilers.

Keywords: meta-architectures, self-supporting systems, virtual machines, proxies, speculative optimization, Smalltalk

ACM Reference Format:

Christoph Thiede, Marius Dörbandt, Eliot Miranda, Marcel Taeumel, and Robert Hirschfeld. 2026. Proxies All the Way Down: Bootstrapping a Userland Speculative Optimizer for the OpenSmalltalk VM. In *Proceedings of the 2026 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! '26)*, October 4–9, 2026, Oakland, CA, USA. ACM, New York, NY, USA, 16 pages. <https://doi.org/10.1145/3840586.3843214>

1 Introduction

Self-supporting programming systems implement core components such as code editors, debuggers, and compilers in themselves [16, 22]. This allows programmers to develop such components in userland, reusing the high-level language and tools of the system, making changes at runtime without restarting the system, and thus benefitting from faster feedback cycles [41, 47, 56]. Once components are reliable, such an arrangement leads to a productive and flexible programming system, where programmers can easily extend programming tools such as the compiler incrementally. Still, brand-new components pose a problem: as soon as they become an integral part of this self-supporting arrangement, bugs can destroy the system’s reflexive integrity [41].

Squeak/Smalltalk [15, 20, 58] is a mature self-supporting system: all userland components are composed of objects, all objects are instances of classes, all code is expressed as methods for classes, all classes and methods are objects, and so on. *Squeak*’s compiler translates source text into bytecode method objects, which are then executed by a virtual machine (VM). Importantly, this compiler, composed of several classes, is implemented as a userland component in Smalltalk and can be used to (re)compile itself and all other userland components in the system at runtime.



This work is licensed under a Creative Commons Attribution 4.0 International License.

Onward! '26, Oakland, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2934-8/2026/10

<https://doi.org/10.1145/3840586.3843214>

The *OpenSmalltalk VM* (OSVM) [21] is the main VM implementation for Squeak/Smalltalk. Following the self-supporting philosophy, this VM is developed in userland. To this end, it is implemented in a subset of Smalltalk dubbed *Slang* that can be transpiled to C to build a production VM [7]. For development, on the other hand, the Slang implementation is executed in another (pre-compiled) VM instance. This mode of operation (called *simulation*) enables programmers to develop the VM interactively, inspect and debug it using Squeak’s extensive tool set, and modify it without restarts.

To improve execution performance, the OSVM uses a *just-in-time* (JIT) compiler that translates bytecodes to machine code [32]. Still, unlike many other dynamic object-oriented VMs, the OSVM currently does not include a *speculative optimizer*. Such optimizers detect hot code paths via branch profiling and dynamically inline many methods into large machine-code regions [1, 37]. While crucial for performance, they employ several optimization techniques that are particularly difficult to implement and maintain in a low-level language. Prior work has proposed the implementation of a userland speculative optimizer for the OSVM [3–5]. In this approach, called *Sista* (Speculative Inlining Smalltalk Architecture), the VM only implements primitive operations such as branch profiling, while most complexity of the inliner (called *Scorch*) is lifted to userland. Besides improving performance, the userland design would allow programmers to incrementally develop and refine optimizations at runtime, using the high-level language and tools of Squeak.

However, this userland design also poses a challenge for the initial bootstrapping of Scorch: a single bug in the new optimizer suffices to destabilize the entire development system, leading to frequent crashes and eventually preventing further development and productization of Sista/Scorch.¹ This motivates our research question:

How can we bootstrap userland components for self-supporting architectures such as Sista/Scorch in an expedient development workflow before those components are stable enough to sustain themselves?

We propose OOPSIE, a proxy-based meta-architecture that allows programmers to implement components such as Scorch in userland yet makes it possible to develop and debug them as first-class components in simulation. After establishing the challenges and opportunities of the OpenSmalltalk VM’s particular architecture and Sista/Scorch (section 2), this paper makes the following contributions:

1. We present the design of OOPSIE for decoupling development and execution of self-supporting system components (section 3).

2. We implement OOPSIE for the OpenSmalltalk simulator and use it to revive the development of Sista/Scorch (section 4).
3. We report on our experiences with the proposed meta-architecture and its impact on the development workflow of userland and VM components (sections 5 and 6).

Finally, we discuss the implications and shortcomings of OOPSIE for the VM development experience (section 7) and relate our approach to prior work (section 8) before we conclude with suggestions for future work (section 9).

2 Problem Context

In this section, we explain self-supporting systems as well as common concepts and practices associated with them. We introduce the relevant technologies *OpenSmalltalk VM*, *Sista*, and *Scorch*. Finally, we motivate our solution by outlining how bootstrapping Scorch is problematic, which impedes the realization of its underlying vision.

2.1 Self-Supporting Systems

Self-supporting systems [16, 22] implement significant parts of themselves using those same parts. For example, in Smalltalk [15] systems, the programming environment is implemented in Smalltalk and developed using itself. That includes tools such as the code editors, debuggers, object inspectors, and even the compiler. Building a system in a self-supporting way allows programmers to define core components using the same level of expressiveness as application code in the system. It also allows deeper introspection into and easier explorability of those core components because they are reachable by every tool in the system.

Self-supporting systems have an *environment*, in which everything they contain lives and can be modified from within. In a Smalltalk system, this environment is called the *image*, and it can be serialized to and restored from a file.

Self-supporting systems commonly support *live programming* [47, 56], where code can be changed while it is running. Live programming is helpful because it shortens feedback cycles as programmers do not have to wait for compilation, restart the program, or restore it to the previously explored state. In particular, this supports three programming practices: exploratory programming, programming in the debugger, and debugging into existence.

Exploratory programming [24, 47, 50, 55, 57] is an agile programming practice where programmers experiment with the problem and solution domains programmatically. The process is typically a gradual one, as programmers understand more and more about a system, problem, or solution while they interact with example programs, solution prototypes, and runtime artifacts.

¹Another reason for the stagnation of the Sista project was a lack of contributors or funding; yet the history of Squeak/Smalltalk has shown that an outstanding development experience could, to some degree, compensate for both of these factors [19].

Programming in the debugger [14, 15]² is a related practice that can be combined with exploratory programming: an experimental or faulty program is run until an error occurs, at which point a debugger shows that error and, crucially, allows the programmer to change the code in the debugger to attempt to fix the problem. Afterwards, they can use the debugger to resume the program from a previously executed point in the call stack, with the new version of the code being used from that point on. Smalltalk supports this practice by reifying stack frames as instances of the class `Context`.

A special case of programming-in-the-debugger is *debugging into existence* [48]. In this case, the error that is caught by the debugger is the absence of a required piece of code, such as a method. In Smalltalk, this error is called `MessageNotUnderstood`³ (the term *message send* refers to method calls) and is produced by the special method `#doesNotUnderstand`. The programmer can then create the missing method in the debugger and continue execution. By following this process, the program is iteratively constructed without needing to write any code upfront.

Self-supporting systems typically need some mechanism to end the infinite recursion of pure self-support. Thus, they implement fundamental operations such as arithmetic, indexing, and instantiation in the VM. In Smalltalk, these are provided by *primitives* [15, p. 52].⁴ Methods can be annotated with a primitive identifier (a number or name). When such a method is executed, the VM will attempt to execute the primitive. That implementation might fail, in which case the VM will execute the method's body (called the *primitive fallback code*).

2.2 OpenSmalltalk VM

The *OpenSmalltalk VM* [21] is the canonical virtual machine implementation for Squeak/Smalltalk [20, 58] and other Smalltalk dialects. Its development is based around a Smalltalk package called *VMMaker*. *VMMaker* includes the sources for the OSVM, which are written in Slang, a subset of Smalltalk. Another part of *VMMaker* can transpile these Slang sources to C for the purpose of building a fast VM [7].

Inspired by the Self 3 system [17], the *Cog VM* [32] is a version of the OSVM that includes a JIT compiler. It compiles Smalltalk methods, represented as bytecode, to native machine code. The produced machine code makes heavy use of *polymorphic inline caches* [17]: small caches for the dynamic method lookup in Smalltalk call sites, placed at each machine code location that corresponds to such a call site.

In addition to transpilation, Slang being a subset of Smalltalk allows the VM to be *simulated* [21, 33] (see fig. 7). During simulation, a release build of the VM running a host image is used to interpret the Slang source code as Smalltalk code. That way, a nested image can be used to develop and debug the OSVM, employing live programming techniques on the Slang code. The infrastructure in *VMMaker* around simulating the VM is called the *simulator* and the nested image is called the *simulated image*. The simulator has a user interface that renders the simulated image's display and provides tools to inspect parts of the simulated VM and simulated image. These tools allow programmers to inspect (parts of) the call stack, JIT artifacts, simulated object slots and the object memory, as well as executed instructions and bytecodes. To simulate the VM in all its aspects, the simulator uses two facilities: first, a large `ByteArray` that houses the VM simulation's heap and machine code generated by the JIT compiler; and second, an architecture-specific CPU simulator to execute the machine code.

The OSVM uses an integer called *oop* (ordinary object pointer) to refer to a particular Smalltalk object. For most objects, oops are pointers to locations in the VM's heap.⁵

Unlike other state-of-the-art object-oriented VM implementations, the *Cog VM* does not include a *speculative optimizer* [18]. Such a component typically uses performance counters to detect hot code paths and subsequently inlines small methods into other methods, guided by type information in inline caches. Inlining reduces method lookup and branching overheads and causes the resulting methods to be large enough to profit from conventional static optimization techniques based on static single-assignment form.

Inlining of methods is speculative because it relies on type information collected in the past. Due to the liveness of self-supporting systems, this type information can be invalidated in the future through changes of code or data. This potential for an invalidation requires speculative optimizers to add guards for their assumptions and to deoptimize their generated code in the case of guard failures. Such guards are typically called *trap instructions*, and the event of triggering one is called a *trap*.

2.3 Sista and Scorch

The *Cog VM*'s JIT compiler already improves performance of Smalltalk code by compiling to native code and reducing the number of required dynamic method lookups. On a small benchmark, it can execute around 3x more bytecodes per second and 7x more method calls than the OSVM without the JIT compiler.⁶

²Gilad Bracha. 2012-11-17. *Debug Mode is the Only Mode*. <https://gbracha.blogspot.com/2012/11/debug-mode-is-only-mode.html> Accessed on 2026-05-14.

³This error is similar to an `AttributeError` in Python, a `TypeError` in JavaScript, or a `RuntimeBinderException` in .NET.

⁴Some primitives are also used to interact with the surrounding environment, provide a foreign function interface, etc.

⁵The exception is *immediate objects* such as integers and characters, whose values are directly encoded in the oop.

⁶The benchmark consists of two sub-benchmarks, one measuring bytecodes per second (sieve of Eratosthenes) and the other method calls per second (naïve recursive Fibonacci number computation). Testing environment:

However, the machine code still has to perform runtime type checks, function calls at method call sites, and primitive argument validation, such as bounds checking. In general, the JIT compiler can only perform method-local optimizations, since it only ever processes one method at a time and does not perform cross-method analysis.

Sista [5] (Speculative Inlining Smalltalk Architecture) attempts to address these issues. It does so in three steps: instrumenting machine code, providing an enhanced bytecode set, and enabling extraction of profiling data.

Instrumenting machine code Sista adjusts the JIT compiler to instrument the generated machine code with performance counters. When a part of the machine code was executed a certain number of times, a *counter trip* event is raised, which notifies the optimizer of a hot code path.

Providing an enhanced bytecode set Sista also provides an improved bytecode set, including a set of *unsafe bytecodes* that are specialized to specific types and perform no argument validation. For example, the Sista bytecode set includes an unsafe bytecode that returns an element of a byte array without checking the array’s type, the index’s type, or for out-of-bounds access. Given a hot loop over a byte array, such a bytecode could allow an optimizer to eliminate redundant bounds checks that are already performed in the loop condition.

Enabling extraction of profiling data Finally, Sista adds a primitive to extract profiling data from JIT compiled machine code. This data includes type information for method calls inside a profiled method, extracted from its inline caches.

Scorch [3] is a speculative optimizer that uses Sista to recompile and optimize hot JIT compiled Smalltalk methods. Scorch is written in Smalltalk and is architected as a userland component (fig. 1) to benefit from the usual advantages of self-supporting systems and allow for the persistence of optimized bytecode across image restarts. When the VM detects a counter trip, it notifies Scorch of the hot method. Scorch then uses the primitive to extract profiling data from it. Based on that data, it speculatively inlines method calls and uses Sista’s unsafe bytecodes to eliminate runtime checks where it can. Scorch still only emits bytecode—the VM’s JIT compiler then compiles it to machine code.

2.4 A Bootstrapping Problem

At the time of writing, Scorch should be considered an early prototype that already optimizes some benchmarks. Still, the development workflow for Scorch as shown in fig. 1 is currently highly impractical: because Scorch resides inside the image whose methods it optimizes and because it uses unsafe bytecodes, defects in its implementation can cause the VM to crash when a bytecode’s preconditions are not

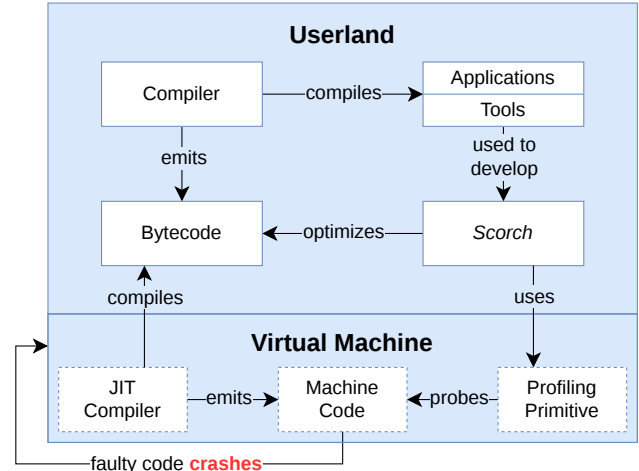


Figure 1. Integration of the speculative optimizer Scorch into a production OpenSmalltalk VM. Scorch lives in userland and optimizes the bytecode within it. The userland also contains the tools used to develop Scorch. Critically, if Scorch produces faulty bytecode that is subsequently JIT-compiled, execution of it can crash the VM, tearing down the userland including Scorch and its development environment.

upheld. As a consequence of this impracticality, development of Scorch has largely stopped.⁷ A crash of the VM is hard to debug since it not only kills the application that is being optimized, but also Scorch itself and any in-image debugger that might help with finding the defect.

One potential solution for handling crashes is shown in fig. 2: the entire VM, with Scorch in userland, is simulated. This technically works and handles crashes more gracefully, since only the simulated VM crashes in case of a defect in Scorch and tools in the host can be used to debug the crash in the VM. However, this solution has several drawbacks:

- P1 A crash in the simulated VM causes the programmer to lose the changes they made to Scorch. They have to manually recover them to continue development.
- P2 While the host’s tools can be used to debug the simulated VM, they cannot be applied to Scorch in the same way. This is because Scorch lives in the simulated userland, which can only be inspected via the VMMaker’s tools.
- P3 Simulating the entire VM is far too slow for productive development: Scorch would have to be developed inside the simulator, where the development tools are subject to the simulator’s significant performance overhead (see section 7.2).

⁷Historically, the original implementation of block closures at the time of Scorch’s inception was another obstacle for the project. A closure’s bytecode was tied to the bytecode of its lexically enclosing method, which prevented inlining closures without affecting those enclosing methods. However, the implementation of closures has since been revised to make their bytecode independent of their enclosing methods.

Squeak6.1alpha #23716 64-bit, VMMaker.oscog-eem.3759 (for both Stack and Cog VM), Arch Linux, 11th Gen Intel Core i7-11800H (16) @ 4.60 GHz.

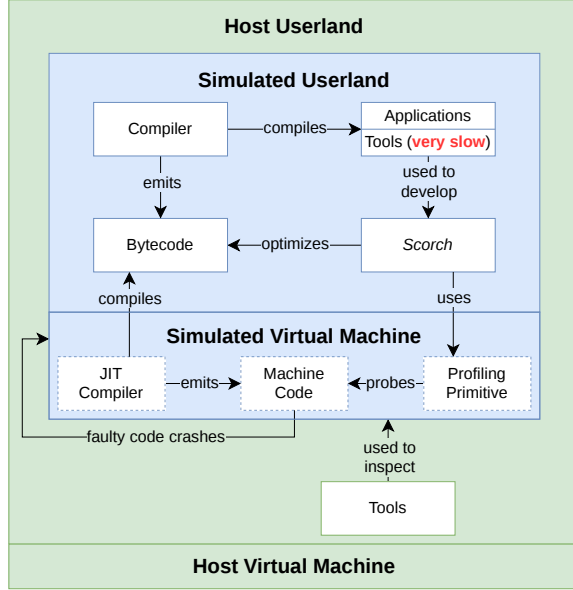


Figure 2. Developing the speculative optimizer Scorch by simulating the entire OpenSmalltalk VM. This solution handles crashes more gracefully, as only the simulated VM crashes and the host’s tools can be used to inspect and debug the (crashed) VM. However, the tools used to develop Scorch are unusably slow due to the simulator’s overhead; and because these tools live in the userland, a crash will cause the programmer to lose the changes they made to Scorch.

P4 Interacting with the development environment in the simulation triggers changes of profiling data. This can interfere with the inputs to the optimizer and make reproducible debugging sessions almost impossible.

So far, the only other explored approach was to do *post-mortem debugging* [13, 36]: using information produced by the crashed program (in this case Scorch and the VM) during their final moments of execution to attempt to reconstruct the chain of events that led to the crash. This approach has proven difficult in practice.

We are thus motivated to create a development environment that is more robust against and helpful in the case of crashes by preventing loss of code changes and VM execution state. It should allow programmers to continue using their usual tools both for developing components like Scorch and for debugging crashes caused by these components.

3 Proxies to the Rescue

Approach. To enable a continuous workflow for bootstrapping Sista/Scorch, we propose a variation of the OpenSmalltalk VM’s simulation architecture that decouples the development of self-supporting system components from their execution. In our proposed meta-architecture, we develop and run the speculative optimizer in a regular Smalltalk

image. However, instead of having it optimize the methods of that same development image, we start a simulated VM in the development image via VMMaker and connect our prototype of Scorch to that VM (fig. 3).

To connect the optimizer to the simulated VM across the simulation boundary between the two environments, we design the proxy-based meta-architecture OOPSIE (*Ordinary Object PointerS Interoperation Engine*), which handles two responsibilities:

Event redirection Forwards Sista counter trips and traps from the simulated VM to an instance of Scorch in the host image.

Transparent proxy membrane Provides access from Scorch in the host image to methods, context frames, and profiling data in the simulated image (i.e., in the object space of the simulated VM).

Our proxy membrane draws inspiration from the *Membrane pattern*, where all requests to a proxy are forwarded dynamically to its target and all responses are wrapped in further proxies, ensuring a strict separation between objects on both sides of the membrane [30, 62]. However, we execute all requested method calls on a proxy in the host image and only forward single state accesses to the target object in the simulated VM.

For example, if the optimizer sends the message `#isDead` to a proxy for a `Context` instance, the membrane will execute the implementation of `Context»isDead` on the host side:

Context»isDead

`^ pc == nil`

During the execution, only the value of the instance variable `pc` will be retrieved from the simulated object memory.

Implications. This design minimizes unintended side effects between the development environment and the “environment under optimization”: if the optimizer were to run in the production VM, and it were to generate a bytecode instruction that is not interpreted correctly by the VM, the entire VM including the development environment could crash (problem P1). By instead running the optimizer against a simulated image using OOPSIE, only the simulated VM is suspended in the case of a crash, while the development environment in the host remains intact and shows an interactive debugger on the suspended simulated VM.

In the other direction, neither user interactions with the development environment nor the execution of the optimizer itself will interfere with the environment under optimization, keeping the inputs to the optimizer (such as performance counters) deterministic and reproducible (problem P4). Even when the optimizer invokes behavior from target objects (such as computing a property from the header of a compiled method or traversing the call stack of a context), the requested methods will be executed in the host image and not interfere with the current state of the simulated VM.

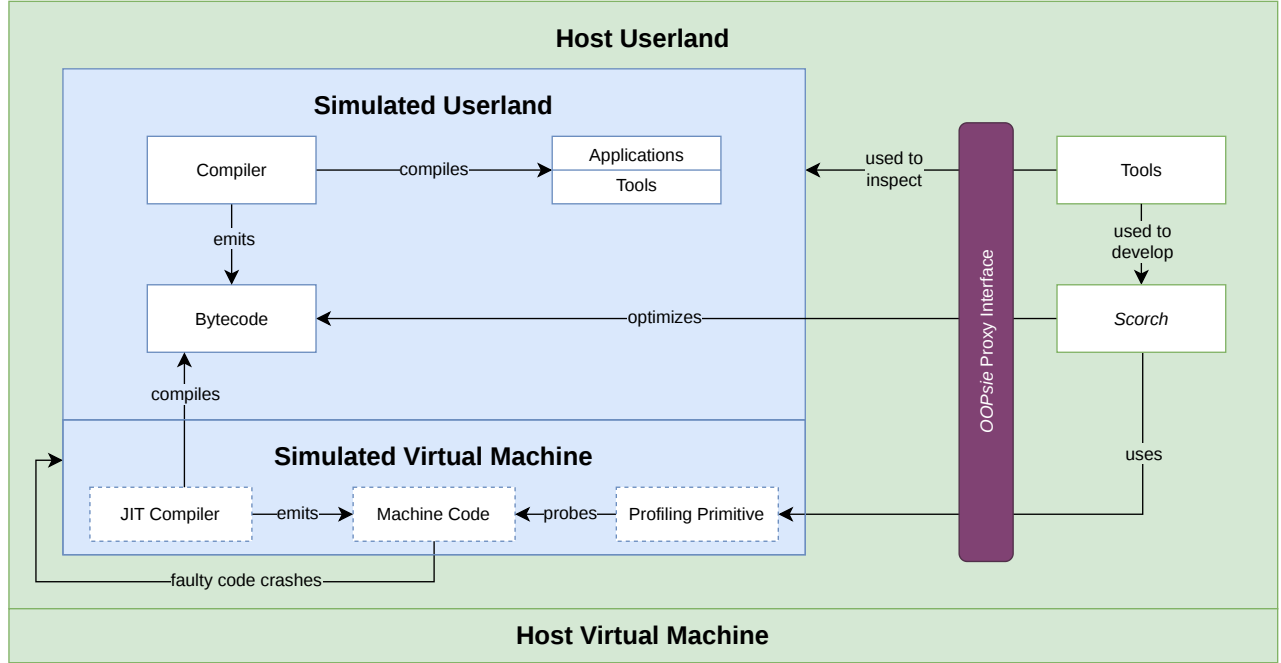


Figure 3. Developing the speculative optimizer Scorch using OOPSIE. Scorch now lives in the host userland, but optimizes the bytecode inside the simulated userland. The Scorch code base remains unchanged because OOPSIE transparently redirects all accesses into the simulation using proxies. This way, programmers can develop Scorch interactively using the tools in the host and isolate crashes to the simulator.

4 Implementing the Proxy System

In the following, we describe our implementation of OOPSIE for the OpenSmalltalk VM in Squeak. We make the source code of our solution, complemented by usage examples and benchmarks, available in a public repository.⁸

Event redirection. When the Sista VM encounters a counter trip or a deoptimization trap, it normally reifies the current call frame as a new Context object [31] and sends a special event message⁹ to this object. To redirect such event messages from the simulation to the host VM, we override the relevant event methods in the simulator subclass of the VM’s interpreter. In the override, we wrap the reified Context object into a proxy on the host side and send the event message to this proxy instead of simulating a message send to the Context object inside the simulated VM.

Transparent proxy membrane. The purpose of our membrane is to provide a transparent connection between objects from the host image and oops (ordinary object pointers) from the simulated image. To this end, we implement a proxy that wraps an oop and redirects all accesses to the simulated object memory.

To handle any message sends dynamically, our proxy class inherits from ProtoObject (a quasi-empty supermost class

in Squeak) and overrides the method #doesNotUnderstand:. When receiving a message, the proxy looks up the required method in the target class and executes it in the host system. During the execution, it redirects all state accesses from the method (such as instance variable reads and writes) by accessing the relevant words in the object memory of the simulated VM.

For the method lookup, we assume that all target classes in the simulated image have a corresponding class in the host image and look up methods there. While not technically necessary, this eases the development of the optimizer as changes to methods in classes such as Context used by the optimizer can be made directly in the host image without needing to manipulate the simulated image.

Intercepting state accesses. To redirect state accesses during the execution of methods, we run those methods in a metacircular evaluator in the host image and modify this evaluator to intercept state-referring instructions. Squeak’s metacircular evaluator, also referred to as *context simulator*,¹⁰ consists of a set of methods on the class Context¹¹ that

¹⁰Despite their similar names, the context simulator is technically unrelated to the OSVM simulator, which is predominantly not metacircular but a heavyweight executor with its own implementation of object memory, primitive functions, garbage collector, JIT, etc.

¹¹In the case of context simulation, these Context objects do not represent frames from an actual call stack in the VM but model a purely userland call

⁸<https://github.com/hpi-swa-lab/osvm-oopsie/>

⁹In Smalltalk terminology, these are referred to as *abort messages*.

implement the bytecode instruction set. To modify the evaluator, we use the SimulationStudio framework for streamlined overriding of particular instruction types.^{12,13}

For our proxies, we override three types of instructions in the evaluator:

Instance variable reads and writes (`pushReceiverVariable:`, `popIntoReceiverVariable:`) to redirect state accesses;

Message sends (`send:`, `directedSuperSend:`) to look up methods in the equivalent target class of the host image;

Primitive calls (`doPrimitive:`, `tryNamedPrimitive:`) to redirect further state accesses, allocations, and other VM operations.

Newly activated methods remain under control of the proxy evaluator, but we only redirect instructions if the current receiver is a proxy object. Once the first activated method returns, control is returned to the original evaluator (usually the host VM). When an event message is sent to a Context proxy, we run the corresponding entry method for Scorch outside the proxy evaluator (but still point it to the Context proxy) because a significant portion of it is not directly involved with simulated objects and otherwise would be unnecessarily slowed down by the metacircular evaluation.¹⁴

When fetching an oop from the simulated object memory (e.g., by reading an instance variable), we wrap it into a further proxy to perpetuate the membrane. If the oop represents an immediate object (such as a `SmallInteger` or a `Character`) or `nil`, we usually do not rewrap it for performance reasons but translate it to the equivalent immediate object in the host VM. However, for special inspection purposes (see section 6), immediate objects can be manually wrapped in proxies, too. To make those proxies interoperable with actual immediate objects in the host system (e.g., when sending `2 + aProxyOnASimulatedInteger`), we modify the host image’s fallback code of all primitive methods that expect immediate objects to automatically resolve any proxy arguments and then retry the primitive.

When storing an object from the host image in the simulated memory, we create a key for this object and encode it as an oop with a special object-header tag. If the modified evaluator encounters this oop during a later read operation, it can map it back to the original object. Note that such references to host objects would become invalid when the simulated image is later run in a production VM; however, this encoding is crucial for bootstrapping and debugging

stack. Notably, the VM can convert between userland and physical stacks to swap the executor mid-execution, which is used to power Squeak’s debugging machinery [19, 59].

¹²<https://github.com/LinqLover/SimulationStudio>

¹³We also considered more conventional alternatives to evaluator modification including source code and bytecode rewriting but did not pursue them for our current prototypical implementation due to available frameworks and conceptual clarity of evaluator modification.

¹⁴Appendix A lists an excerpt of our evaluator’s implementation.

purposes, since “uplifted” components such as the speculative optimizer would often attach (temporary) information to objects from the simulated VM.

Other membrane properties. To preserve identity semantics across the simulation boundary, proxies forward their identity to the simulated oop by overriding `#==` and `#identityHash`. When the simulated VM garbage-collects an oop or terminates, all affected proxies are revoked and will raise an error when they receive new messages.

5 Bootstrapping Scorch across the Simulation Boundary

In this section, we illustrate how a programmer uses OOPSIE to bootstrap Scorch in a less interrupted workflow. To debug the optimizer, our programmer starts by opening a VM simulator and instructs it to execute the following code snippet in the simulated image: `30 fibonacci`. The image then compiles the snippet to unoptimized bytecode, which the VM starts executing and soon JIT-compiles to machine code. Because the JIT-compiled method `Integer»fibonacci` contains a hot code path, its execution will trigger a counter trip at some point.

The VM handles the counter trip by sending an event message to invoke Scorch. OOPSIE redirects this invocation and runs Scorch in the host image. Scorch now analyzes the hot code path, unaware that it is doing so through the proxy membrane, generates optimized bytecode for it, and installs that bytecode in the simulated image. The VM then continues executing the code snippet, but uses the optimized bytecode instead.

However, our prototype of Scorch has not yet optimized the snippet correctly: shortly after the counter trip, a debugger pops up, which notifies the programmer of an `AssertionFailure` in the VM. By reading the failed assertion statement shown in the debugger, the programmer learns that the VM just caught an error in the optimized bytecode: it has an imbalanced stack. Thanks to OOPSIE, this error did not crash the host image, but was contained in a debugger on the VM simulator.

By using the VMMaker tools, the programmer prints a textual representation of the head frame, finds the oop of the executed method, and produces a bytecode listing of it. They read through the optimized bytecode and spot a push instruction without a corresponding pop instruction. Now, the programmer wants to find the faulty code in Scorch that generated the incorrect bytecode. Since Scorch runs in the host image with all standard tools available, they place a breakpoint in Scorch’s bytecode emitter. Because Scorch has already completed its generation of the optimized bytecode, the programmer now manually retriggers optimization of the hot code path to activate the breakpoint. They search for the problem by alternating between using the debugger to step towards the faulty code and opening object inspectors on

parts of Scorch’s intermediate code representation to know where to step next. Finally, they find the missing statement to emit a pop instruction and insert it directly in the debugger.

With the fix applied, the programmer lets execution of Scorch continue. Scorch thus generates and installs a corrected version of the optimized bytecode. The VM then starts executing the new bytecode, but again, a debugger appears. This time, it is not an assertion failure, but a case error: the JIT compiler has not yet implemented all 53 variants of the `callInlinePrimitive:` instruction.¹⁵ The programmer quickly looks up the intended behavior for the missing case and implements it in the debugger. Because the VM is still suspended at the JIT compilation of the bytecode, they can now let it proceed with the newly implemented case. At last, the VM successfully executes the full bytecode and shows the correct result of the supplied code snippet in the terminal. Thus, our programmer used OOPSIE to iteratively develop Scorch and the VM with Squeak’s interactive debugging tools.

6 Debugging across the Simulation Boundary

Beyond bootstrapping userland components such as Scorch from a host image, our proposed meta-architecture also makes it possible to improve the debugging experience across the simulation boundary.

Status Quo: Limited Tool Support for the OSVM. In Squeak, typical development workflows for prototyping or debugging systems are supported by a set of tools such as object inspectors and explorers, debuggers and process browsers, system browsers and method finders [14, 58]. These tools promote *exploration* through close integration between different tools, *experimentation* through means for interactivity such as ubiquitous code execution (“do-its”), and *multitasking* through free juxtaposition of fine-grained windows on the screen [46, 55, 57]. Through *domain-specific representations*, programmers can conveniently understand and modify objects with different internal structures such as dictionaries, compiled methods, and graphical widgets [10].

In contrast, developers of the OpenSmalltalk VM only have access to rather primitive tool support when working with objects from the simulated image: the simulator currently mainly offers text-based, stateless terminals for dumping the fields of an object, variables of a stack frame, frames of a call stack, etc. Programmers can request these dumps by invoking a command from a menu and entering an oop or a memory address in a pop-up menu. Thus, they are restricted to a user interface with limited interactive affordances, higher navigation costs, and poorer support for domain objects.

Proxies to the Rescue, Again. Notably, Squeak’s tools are implemented as self-supporting components that exist

¹⁵This is the entry point for all unsafe instructions of the bytecode set.

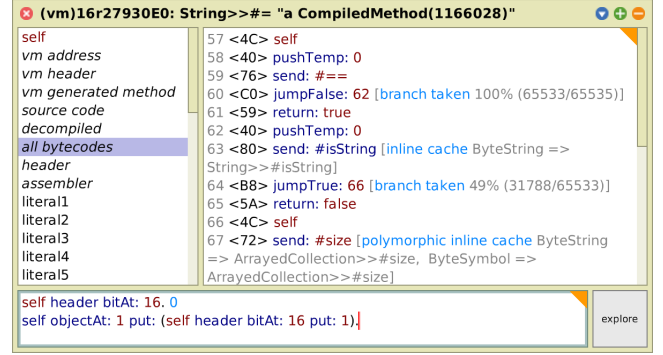


Figure 4. Interacting with a proxy for a CompiledMethod from a simulated image in an inspector from the host image. Through the proxy membrane, the VM programmer can reuse existing abstractions and tools from the self-supporting environment to inspect its properties (here: its bytecodes and branch counters) and manipulate them (here: changing a bit in the method header). The inspector is unaware that it is displaying a proxy instead of a regular CompiledMethod.

in the same object space as their objects under inspection. To retrieve or modify an object, the tool typically sends regular messages such as `#size`, `#instVarNamed:put:`, or `#sourceCode` to it. This opens up an opportunity for improving the tooling during simulation: analogously to connecting a Scorch instance in the host environment to context frames in the simulated environment, we can also connect host instances of those tools to proxies on the simulated object memory. Thus, we can reuse the proven tools from regular Squeak development for debugging VM components and inspecting oops across the simulation boundary.

6.1 Example: Exploring the Inputs to Scorch

For illustration, imagine a programmer who is bootstrapping Scorch in the simulator. During an invocation of the optimizer from a counter trip, they are investigating a runtime error in a debugger.

From this debugger, they can now inspect a proxy for the CompiledMethod that was currently being analyzed by Scorch in a regular inspector, as if it were a regular object in the object space of the host environment. In this inspector, they can examine the CompiledMethod’s object header, its (still unoptimized) bytecodes, or its current performance counters (fig. 4). They can also modify the proxied object from here, for example, by toggling its optimization bit in the header through the UI. All messages that the inspector sends to the object to read or change the displayed information are transparently forwarded through the proxy membrane to the object memory of the simulated VM. The inspector also integrates VM-internal information provided by the proxy such as the JIT-compiled code of the method and its current garbage-collection state.

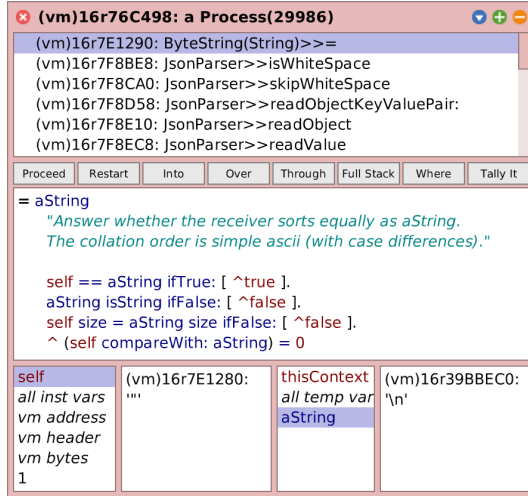


Figure 5. Exploring a call stack from the simulated image in a regular debugger in the host image using proxies on Context objects (here: running the Are We Fast Yet JSON Benchmark [27]). By selecting frames and variables, the programmer can view their source code and spawn further tools to navigate through the simulated object graph.

In the debugger, the programmer also finds a reference to the Context frame from where the optimization was originally triggered. To understand where this frame has originated, they open a second regular debugger on a proxy for this Context object (fig. 5). In this new debugger, they can select and view all stack frames of the current call stack from the simulated VM, read their source code, and interact with the receivers and temporary variables of these frames. Analogously, they can also open the class of one receiver in a regular system browser to navigate through the source code of all its methods in the simulated VM (fig. 6).

At some point, they realize that the runtime error was caused by a certain bytecode instruction in a method that their current prototype of Scorch does not yet support. So, they decide to recompile that method with an alternative compiler to temporarily avoid the emission of the yet-un-supported instruction. Finally, they restart the invocation of Scorch from the original debugger.

6.2 Example: Building Custom Tools for Memory Introspection

Besides bootstrapping VM components, we can also use OOPSIE to build impromptu tools for analyzing the simulated object space without interfering with VM execution.

In one situation while debugging the invocation of a primitive function, we discovered an unexpected oop for a Morph (a graphical object in Squeak) and wondered to which part of the UI this morph would belong. Normally, we could simply send this morph a message to highlight it in the UI to identify it (the simulator renders the simulated image’s display into

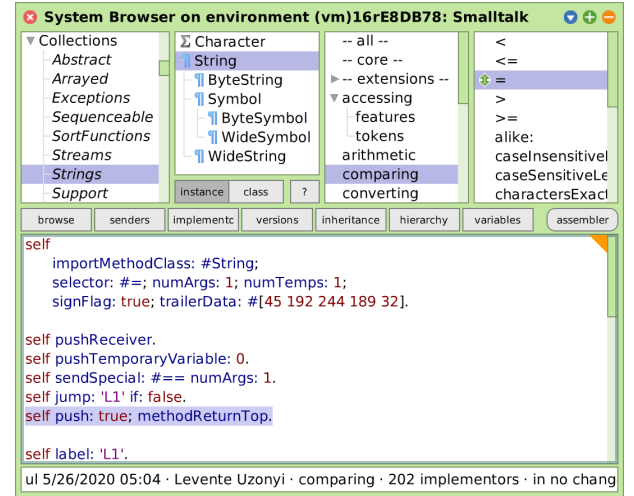


Figure 6. Browsing the source code packages in the simulated image by using a regular system browser from the host image. Using Squeak’s integrated MethodMessage compiler (denoted as “assembler” in the UI), which is invoked in the host image as well, the programmer can manipulate the byte-codes of a CompiledMethod in the host image to trigger or avoid certain paths in the speculative optimizer.

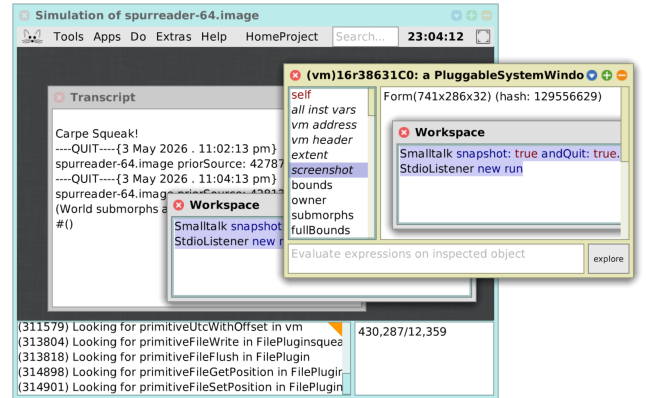


Figure 7. Rendering a Morph (a graphical object in Squeak) from the simulated UI tree using the Morphic renderer from the host image (here, embedded as a custom inspector field). This can be helpful to visually inspect or identify widgets while the simulated VM is unresponsive.

the host image), but because the VM was currently halted in the debugger, its (self-supporting) rendering process was blocked, too. Instead, we used the Morphic renderer of the host environment to display the simulated morph so we could conveniently recognize it (fig. 7).

In another situation, we created a simple interactive visualization of the object memory to understand the roles and layouts of the first oops in the image (fig. 8). While this is

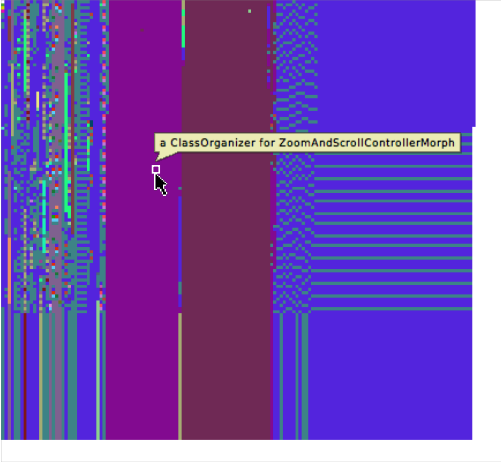


Figure 8. Visualizing the first 1000 oops in the simulated object memory as morphs in the host image. Colors represent different classes. Each morph can be hovered or clicked to retrieve a textual representation or a graphical inspector that reuses the existing facilities of the host image (e.g., see fig. 4).

a task that could also be done from within the image under inspection, this approach would reveal a paradox: every graphical object that visualizes an oop would require another oop, causing infinite allocation and potentially destabilizing the visualization when the garbage collector is triggered and rearranges oops. By instead creating the visualization from the outside, we could circumvent that probe effect and augment the visualization with VM-internal information. Unlike previous approaches,^{16,17} we can directly interact with any object from the visualization by using the host tools or sending it arbitrary messages through the proxy membrane.

7 Discussion

In the following, we evaluate how OOPSIE changes the development experience for bootstrapping Scorch and discuss possible limitations of our approach.

7.1 A More Immediate Development Workflow

In comparison to the production architecture of Sista/Scorch, our meta-architecture for development in simulation greatly improves the *flow* of programmers [11]. When an invalid optimization or a bug in the VM breaks execution, the development environment neither crashes nor needs to be restored but remains stable and responsive (solving *problem P1*). At the same time, programmers receive richer and faster feedback: instead of reading a shallow crash dump, they can

investigate the runtime error in an interactive debugger and use additional tools to inspect runtime data such as variable stacks or optimized bytecodes (solving *problem P2*). Since the environment under optimization is isolated from the development environment, programmers can inspect or modify any part of the running system without accidentally manipulating the object memory of the target environment (solving *problem P4*).

By directly changing methods of Scorch or Sista without restarting the VM in this way, the resulting development can feel almost *immediate* [60]. This enables previously unavailable practices such as exploratory programming [24, 50, 55] and debugging into existence [48]. Even if the simulated VM was already invalidated and cannot be resumed, programmers can still interactively debug the VM *post mortem* [36] by exploring simulated objects and sending them messages that are executed in the host.

Still, Scorch remains a self-supporting component that can be run, introspected, and debugged from within the production VM after initial development. Only for bootstrapping a stable version of the speculative optimizer do we run it against a simulation to develop it together with the VM. This arrangement also avoids observer effects from tool interactions as all tools run in the host image and their execution does not affect the performance counters in the simulated image, keeping optimizer invocations deterministic.

Beyond bootstrapping, OOPSIE remains available when programmers wish to make critical changes to or debug destabilizing bugs in the optimizer at a later point in time. Thus, our meta-architecture provides a tool for flexible re-composition of system components that is orthogonal to their behavior and can be based on the task context in which programmers wish to develop and explore these components.

7.2 Performance: Lower Execution Speed, Yet Steadier Feedback

Naturally, code execution in simulation is slower than production by orders of magnitude. In our experiments, the same Are We Fast Yet benchmarks [27] ran slower by a factor of approximately 2000x–3000x when running in a simulated versus a production Sista VM (each without speculative optimization).¹⁸ In absolute numbers, a single JSON benchmark completes in about 6 seconds instead of 2.1 milliseconds. Similarly, interfacing Scorch to OOPSIE proxies instead of regular objects in the host image involves an overhead for redirecting state accesses. For typical invocations of Scorch through a counter trip while running the JSON benchmark, we measured cycle times of 1–7 seconds using OOPSIE compared to 10–200 milliseconds in production, corresponding to an overhead of up to 200x (fig. 9).

¹⁶Craig Latta. 2013-02-07. *A detailed description of Context’s object memory visualization tools*. <https://thiscontext.com/a-detailed-description-of-spoons-object-memory-visualization-tools/> Accessed on 2026-08-21.

¹⁷Jens Lincke, Tom Beckmann, and Robert Hirschfeld. 2023-11-04. *Mini Image for a Cloud Squeak*. <https://github.com/hpi-swa-lab/cloud-squeak/blob/main/mini-squeak/index.md> Accessed on 2026-08-21.

¹⁸Testing environment: Squeak6.1alpha #23308 64-bit, SistaV1; OSVM 202507201654/VMaker.oscog-eem.3580; Ubuntu 24.04; Intel Core Ultra 7 255H (power profile: performance).

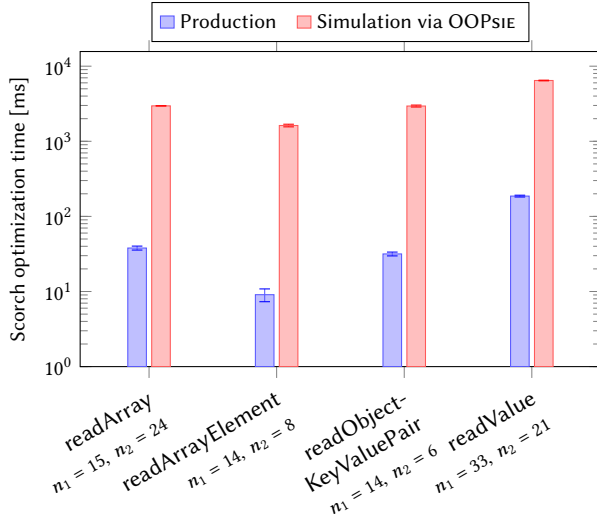


Figure 9. Scorch cycle times in production versus simulation via OOPSIE, optimizing different methods from the Are We Fast Yet JSON benchmark. Error bars indicate standard error of the mean (all errors < 90 ms).

At the moment, these delays partially exceed a range of 2–4 seconds that is often perceived as acceptable for common operations [52, chap. 11]. Still, they substantially outperform the cost of running tools within the simulation (solving [problem P3](#)) or restarting a crashed production VM and recovering the state of the development environment. If a programmer needed to rebuild the production VM, this would even slow down a single feedback cycle by more than one minute.¹⁹ Finally, moderate but reliable delays can even create a more convenient experience than typically shorter but less predictable delays [52, sec. 11.5].

In the future, we plan to replace our current naïve implementation of state redirection via evaluator modification with a faster approach such as bytecode rewriting or virtualization support in the OSVM [43]. In early experiments with bytecode rewriting using Squeak’s new `MethodMessage` package,²⁰ we have already seen promising speed-ups of up to 50x over our current implementation. This would reduce Scorch cycle times to under 250 milliseconds and could significantly improve the perceived liveness of development using OOPSIE. In addition, we expect that the intended productionization of Scorch itself will moderately speed up the OSVM simulator as well.

7.3 Implications for Implementation

Our implementation of the OOPSIE system itself is relatively simple, comprising 81 methods with 475 SLOC²¹ in its essential core. For the speculative optimizer, we were generally

¹⁹4.4 seconds for an incremental transpilation of Slang plus 59.5 seconds for recompiling the generated C code (same testing environment, clang 18.1.3).

²⁰https://squeak.org/release_notes/6.1/#:-:text=MethodMessage,-

²¹Source lines of code, excluding blank lines and comments.

able to reuse the existing implementation of Scorch without changes by invoking it with proxies instead of regular Context objects due to our transparent proxy membrane. However, we observed two groups of code patterns that required manual adaptation for OOPSIE:

Compiler and bytecode optimizations A small set of *special selectors* such as `#class` and `#==` are compiled to special message send instructions that the VM interprets without performing a regular method lookup. Thus, we cannot override those selectors in our proxy to forward them to the simulated image. Similarly, certain selectors are *inlined* unconditionally by the compiler (e.g., `aProxy ifNotNil: [...]` would become `aProxy == nil ifTrue: [...]`), making it impossible to forward them as well.

As a remedy, we can either avoid these selectors in the Scorch source (e.g., by rewriting `aContext receiver class` to `aContext receiver perform: #class`) or disable such optimizations in the compiler globally.

Hard-coded class references In some cases, Scorch hard-coded names for certain classes of Squeak’s execution model, such as when creating a new `CompiledMethod` or comparing the receiver class of a frame against the Context class. In the first case, this would store new objects in the simulated image yet allocate them in the host image, requiring frequent reverse proxy redirections from the simulation to the VM and preventing efficient execution. In the second case, a comparison against the class of a proxy frame would never succeed, as the class in the simulation is not identical to the class in the host.

Thus, we replaced these references with dynamic class lookups such as `oldMethod class`²² and `oldMethod class environment classNamed: #Context`. While it would be possible to avoid these changes by dynamically resolving class references [12], this refactoring also improved the modularity of the optimizer, avoiding global state and easing future testing.

Using OOPSIE, we could revive Scorch, port it to the latest version of Squeak/Smalltalk and the OSVM, and run some benchmarks for the speculative optimization across the simulation boundary. As described in [section 5](#), we already fixed a number of errors in Scorch’s bytecode generator and in the Sista interpreter of the VM. Beyond this, we are planning to use OOPSIE to extend Sista/Scorch by implementing further optimization rules and unsafe bytecodes. Finally, we intend to complete support in OOPSIE for lazily constructed Context objects (see [column 1 on page 6](#)), which would make it possible to use the debugger of the host image to step through stack frames of the simulated VM and thus further improve support for interactive prototyping.

²²Or `oldMethod perform: #class` if compiler inlining is still enabled (see previous paragraph).

8 Related Work

Previous work has repeatedly aimed to refine self-supporting systems by lifting traditional VM components into userland [8, 9, 42], leading to developments such as the Bee/Smalltalk runtime [39, 41] and the Klein/Self VM [61]. While such attempts generally can improve the liveness and development experience of these systems, developers have often suffered from stability issues or debugging interference when bootstrapping components. For instance, when a userland garbage collector is interrupted half-way through a breakpoint or a runtime error, objects may have been partially moved or temporarily overwritten, and subsequent inspections by a debugger could read garbage [41]. Similarly, incorrect machine code that is generated at runtime to modify a primitive or optimize a method could corrupt the object space or crash the system [8].

To address these issues, many approaches choose to bootstrap such components *offline*, that is, not applying changes from within the running system but from a separate environment [41, 61]. To make such offline development feel more live and immediate, often a *remote debugger* is employed that provides direct access to the source code and objects in the target system from the distant development environment [28, 38].

While it is possible to support remote debugging through VM support [64], several projects such as Bee/Smalltalk, Smalltalk-25, and Maxine/Java employ OS-level debugging facilities such as ptrace instead and disassemble or reify high-level information [40, 51, 61, 63]. This can support a *multi-level debugging* workflow [23, 25, 35] where programmers inspect and modify high-level artifacts such as source code and runtime objects together with low-level information and VM internals such as generated machine code and raw memory contents through unified tooling [26, 51, 61, 63]. To reify high-level objects, a recurring pattern is the use of *mirrors* [6] and *transparent proxies* [2, 53, 62]. For instance, in Klein/Self, a debug server exposes mirrors to objects in a remote world, which can be used to inspect and change these objects using the regular inspection and debugging tools in a client world [61].

Additionally, several methods have been proposed to support the reuse of specific object protocols (such as for sending messages to remote objects in a debugging console or for reusing domain-specific tools [10]) while minimizing debugging interference with the target systems (for example, when the target VM is currently halted or corrupted, or simply to avoid applying unintended side effects to the target system): *out-of-place debugging* operates on a copied slice of the object graph that was transferred from the target environment to a client image [28]. The *metaphysics* framework used for Smalltalk/Bee development introduces *direct* and *mirage* [34] *gates* that execute code locally but interface it to reifying

mirrors on a remote object space [40]. This method of behavioral intercession has also inspired the design of our proxy membrane. Similarly, *holographic objects* are proxies on objects preserved in the heap dump of a crashed VM that can be interacted with through ordinary tools *post mortem* [49].

In contrast to offline development, fewer approaches choose *online* development of self-supporting components and rely on safety mechanisms such as fallback routines or upfront validation [3, 8, 54]. *Transaction mechanisms* such as interchangeable source code layers [29] or hypervisors [44] can help perform “brain surgeries” to the running system by applying sets of mutually dependent changes atomically. Similarly, *lemming debugging* in the OSVM simulator allows developers to roll back the latest garbage collection pass from a backup if a validation fails, enabling safe modifications to the garbage collector at runtime [21].

For Squeak/Smalltalk, self-supporting components are usually developed online, occasionally with the help of migration scripts or code loading infrastructure to install changes atomically.²³ Using OOPSIE to reuse host tools for proxies (see section 6), we enabled out-of-place development for such components in the OSVM simulator. More importantly, we have demonstrated that our proxy membrane can be used to even *execute* entire components such as Scorch out-of-place, developing and running them locally in the host image but interfacing them to objects and methods in the simulated image, thus minimizing unintended interferences during execution and development. To our knowledge, such *out-of-place execution* has not been described before. At the same time, the simulation makes it possible to develop remaining VM components at runtime together with uplifted image components in a shared object space and environment while restarts of the simulated VM are minimized.

9 Conclusion

“Shooting yourself in the foot” is a common phenomenon when changing critical infrastructure in a live self-supporting system to bootstrap new components. Developers need to find ways to separate the doctor from the patient to succeed, or they will endure endless crashes and restarts of their development environment before they can make new components sufficiently stable and robust. This is even more true for event-driven components such as speculative optimizers, whose mere execution in the target system can trigger further optimizations, threatening not only stable development environments but also reproducible debugging sessions.

With OOPSIE, we have proposed a new meta-architecture for extracting nascent components from their surrounding userland without their knowledge to address these stability issues and interferences. By running components in a host

²³Historically, the SystemTracer was used to perform fundamental migrations of Smalltalk images offline, such as when changing the object memory layout [19].

environment but interfacing them through a transparent proxy membrane to objects within the target environment, programmers can decouple development from execution and isolate unsafe side effects from their working environment. In a case study for the OpenSmalltalk VM we have demonstrated that this setup effectively supports resumed development of the speculative optimizer Scorch. By co-locating the optimizer with traditional VM components such as interpreter and JIT in the OSVM simulation environment, programmers can conveniently develop and debug both of them at the same time in a tight feedback loop. Moreover, OOPSIE has also shown promise as a means for multi-level debugging the VM and inspecting or manipulating high-level representations of simulated objects using conventional tools from the host.

With these experiences in mind, we remain cognizant of the remaining work that has to be done to realize the practical benefits of the combination of OOPSIE and Scorch. To solve performance limitations and compiler inlining impairments of our current proxy evaluator, we are planning to move to a bytecode-rewriting or VM-supported implementation for behavioral intercession in the future. Sista/Scorch itself is still in its infancy and requires further development—that can now be more productive—by implementing state-of-the-art optimization passes, adding a broader set of inline primitives, and improving overall stability [3].

Beyond the case of Sista/Scorch, we want to explore how we could apply our meta-architecture to further VM components such as JIT or garbage collector in order to possibly reify them within userland. Finally, we think that the unusual simulation context of OOPSIE has the potential to support additional VM development activities: for instance, it could help us to prepare very small images from the outside without loading fundamental tools such as a compiler into them [45]. By expanding Squeak’s regular debugger into a symbolic multi-level debugger using OOPSIE introspection, we hope to turn bytecode, machine code, and their respective executors into more tangible, explorable, and “live” artifacts.

Acknowledgments

We thank the anonymous reviewers for their detailed feedback. We are grateful to the HPI Research School on Systems Design,²⁴ LabWare,²⁵ and the HPI-MIT program “Designing for Sustainability”²⁶ for supporting this work.

²⁴<https://hpi.de/en/research/research-schools/systems-design/>

²⁵<https://www.labware.com/>

²⁶<https://hpi.de/en/research/cooperations-partners/research-program-designing-for-sustainability.html>

A Proxy Evaluator

The following is a simplified excerpt of our modified evaluator, which redirects state accesses when handling messages sent to a proxy in OOPSIE, as described in section 4. The full source code is provided in footnote 8.

```
ProtoObject variableSubclass: #Proxy
  instanceVariableNames: 'objectMemory oop'
  classVariableNames: "
  poolDictionaries: "
  category: 'OOPSIE'.
```

```
Proxy»doesNotUnderstand: aMessage
```

```
  ^ ProxyContext evaluate:
    [aMessage sendTo: self]
```

```
Proxy»vmObjectClass
```

```
  ^ (Smalltalk classNamed: (objectMemory nameOfClass:
    (objectMemory fetchClassOf: oop)))
    ifNil: [Object]
```

```
SimulationContext variableSubclass: #ProxyContext
```

```
  instanceVariableNames: "
  classVariableNames: 'VMPrimitives'
  poolDictionaries: "
  category: 'OOPSIE'.
```

```
ProxyContext»send: selector to: receiver with: arguments
```

```
  (self objectClass: receiver) = Proxy ifFalse:
    [ ^ super send: selector to: receiver with: arguments ].
  ^ self send: selector to: receiver with: arguments lookupIn: re-
    ceiver vmObjectClass
```

```
ProxyContext»pushReceiverVariable: offset
```

```
  | objectMemory |
  (self objectClass: self receiver) = Proxy ifFalse:
    [ ^ super pushReceiverVariable: offset ].
  objectMemory := self receiver vmObjectMemory.
  ^ self push:
    (Proxy forObjectMemory: objectMemory oop:
      (objectMemory interpreter fetchPointer: offset ofObject:
        self receiver vmOop))
```

```
ProxyContext»simulatedObject: anObject instVarAt: index
  put: value
```

```
  | objectMemory |
  (self objectClass: anObject) = Proxy ifFalse:
    [ ^ super simulatedObject: anObject instVarAt: index put: value ].
  objectMemory := anObject xxxObjectMemory.
  objectMemory storePointer: index - 1 ofObject: anObject xxxOop
  withValue: (value xxxOopIn: objectMemory).
  ^ value
```

```
ProxyContext»doPrimitive: primitiveIndex method: aCom-
  piledMethod receiver: receiver args: arguments
```

```
  | objectMemory oop |
  (self objectClass: receiver) = Proxy ifFalse:
    [ ^ super doPrimitive: primitiveIndex method:
```

```

aCompiledMethod receiver: receiver args: arguments].
objectMemory := receiver xxxObjectMemory.
oop := receiver xxxOop.
self ifQuickReturnInstVarPrimitive: primitiveIndex do: [:offset |
    ^ self push:
        (self proxyClass forObjectMemory: objectMemory oop:
            (objectMemory fetchPointer: offset ofObject: oop))].
(VMPPrimitives at: primitiveIndex) ifFalse:
    [ ^ super doPrimitive: primitiveIndex method: aCompiledMethod
        receiver: receiver args: arguments].
^ self method: aCompiledMethod receiver: receiver arguments:
arguments simulatePrimitive: [:interpreter |
    interpreter
        primitiveFunctionPointer: (interpreter functionPointerFor:
primitiveIndex inClass: objectMemory nilObject);
    initPrimCall;
    dispatchFunctionPointer: interpreter primitiveFunctionPointer ]

```

References

- [1] Matthew Arnold, Stephen J. Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2005. A Survey of Adaptive Optimization in Virtual Machines. *Proc. IEEE* 93, 2 (2005), 449–466. doi:10.1109/JPROC.2004.840305
- [2] John K. Bennett. 1987. The Design and Implementation of Distributed Smalltalk. *SIGPLAN Notices* 22, 12 (Dec. 1987), 318–330. doi:10.1145/38807.38836
- [3] Clément Béra. 2017. *Sista: A Metacircular Architecture for Runtime Optimisation Persistence*. Doctoral dissertation. Université de Lille 1. <https://inria.hal.science/tel-01634137>
- [4] Clément Béra and Eliot Miranda. 2014. A Bytecode Set for Adaptive Optimizations. In *Proceedings of the 6th Edition of the International Workshop on Smalltalk Technologies (IWST 2014)*. European Smalltalk User Group (ESUG), Cambridge, UK, 12 pages. <https://inria.hal.science/hal-01088801>
- [5] Clément Béra, Eliot Miranda, Tim Felgentreff, Marcus Denker, and Stéphane Ducasse. 2017. Sista: Saving Optimized Code in Snapshots for Fast Start-Up. In *Proceedings of the 14th International Conference on Managed Languages and Runtimes (Prague, Czech Republic) (ManLang 2017)*. ACM, New York, NY, USA, 1–11. doi:10.1145/3132190.3132201
- [6] Gilad Bracha and David Ungar. 2004. Mirrors: Design Principles for Meta-Level Facilities of Object-Oriented Programming Languages. *SIGPLAN Notices* 39, 10 (Oct. 2004), 331–344. doi:10.1145/1035292.1029004
- [7] Tom Braun, Marcel Taeumel, Eliot Miranda, and Robert Hirschfeld. 2023. Transpiling Slang Methods to C Functions: An Example of Static Polymorphism for Smalltalk VM Objects. In *Proceedings of the 15th International Workshop on Virtual Machines and Intermediate Languages (Cascais, Portugal) (VMIL 2023)*. ACM, New York, NY, USA, 88–93. doi:10.1145/3623507.3623548
- [8] Camillo Bruni, Stéphane Ducasse, Igor Stasenko, and Guido Chari. 2014. Benzo: Reflective Glue for Low-Level Programming. In *Proceedings of the 6th Edition of the International Workshop on Smalltalk Technologies (IWST 2014)*. Cambridge, UK, 12 pages. <https://inria.hal.science/hal-01060551>
- [9] Guido Chari, Diego Garbervetsky, Stefan Marr, and Stéphane Ducasse. 2015. Towards Fully Reflective Environments. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Pittsburgh, PA, USA) (Onward!)*. ACM, New York, NY, USA, 240–253. doi:10.1145/2814228.2814241
- [10] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Girba. 2015. The Moldable Inspector. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Pittsburgh, PA, USA) (Onward!)*. ACM, New York, NY, USA, 44–60. doi:10.1145/2814228.2814234
- [11] Mihaly Csikszentmihalyi. 2008. *Flow: The Psychology of Optimal Experience* (1 ed.). Harper Perennial, New York, USA. ISBN 978-0061339202.
- [12] Marcus Denker. 2022. First Class Variables as AST Annotations. Lecture at the 30th International Conference on Software Language Engineering (ESUG 2022), Novi Sad, Serbia. <https://www.youtube.com/watch?v=embn4kX0O3k> Accessed on 2026-05-08.
- [13] Stanley Gill. 1951. The Diagnosis of Mistakes in Programmes on the EDSAC. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 206, 1087 (may 1951), 538–554. doi:10.1098/rspa.1951.0087
- [14] Adele Goldberg. 1984. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Reading, Mass.
- [15] Adele Goldberg and David Robson. 1983. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, USA.
- [16] Robert Hirschfeld and Kim Rose (Eds.). 2008. *Self-Sustaining Systems: First Workshop, S3 2008* (Potsdam, Germany). Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-89275-5
- [17] 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 Proceedings (Geneva, Switzerland) (ECOOP '91, Vol. 512)*, Pierre America (Ed.). Springer, Berlin, Heidelberg, 21–38.
- [18] Urs Hölzle and David Ungar. 1996. Reconciling Responsiveness with Performance in Pure Object-Oriented Languages. *Transactions on Programming Languages and Systems* 18, 4 (July 1996), 355–400. doi:10.1145/233561.233562
- [19] Daniel Ingalls. 2020. The Evolution of Smalltalk: From Smalltalk-72 through Squeak. *Proceedings of the ACM on Programming Languages* 4, HOPL, Article 85 (June 2020), 101 pages. doi:10.1145/3386335
- [20] 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 Conference on Object-Oriented Programming, Systems, Languages, and Applications (Atlanta, GA, USA)*. *SIGPLAN Notices* 32, 10, 318–326. doi:10.1145/263700.263754
- [21] Daniel Ingalls, Eliot Miranda, Clément Béra, and Elisa Gonzalez Boix. 2020. Two Decades of Live Coding and Debugging of Virtual Machines through Simulation. *Software: Practice and Experience* 50, 9 (2020), 1629–1650. doi:10.1002/spe.2841
- [22] Joel Jakobovic, Jonathan Edwards, and Tomas Petricek. 2023. Technical Dimensions of Programming Systems. *The Art, Science, and Engineering of Programming* 7, 3 (Feb. 2023), 59 pages. doi:10.22152/programming-journal.org/2023/7/13
- [23] David Johnson, Mike Hibler, and Eric Eide. 2014. Composable Multi-Level Debugging with Stackdb. *SIGPLAN Notices* 49, 7 (March 2014), 213–226. doi:10.1145/2674025.2576212
- [24] Mary Beth Kery and Brad A. Myers. 2017. Exploring Exploratory Programming. In *Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, Raleigh, NC, USA, 25–29. doi:10.1109/VLHCC.2017.8103446
- [25] 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 (Málaga, Spain) (MODULARITY Companion 2016)*. ACM, New York, NY, USA, 91–93. doi:10.1145/2892664.2892679
- [26] Dmitri Makarov and Matthias Hauswirth. 2013. Jikes RDB: A Debugger for the Jikes RVM. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (Stuttgart, Germany) (PPPJ '13)*. ACM, New York, NY, USA, 169–172. doi:10.1145/2500828.2500847

- [27] Stefan Marr, Benoit Daloze, and Hanspeter Mössenböck. 2016. Cross-Language Compiler Benchmarking—Are We Fast Yet?. In *Proceedings of the 12th Symposium on Dynamic Languages* (Amsterdam, Netherlands) (DLS '16). ACM, New York, NY, USA, 120–131. doi:10.1145/2989225.2989232
- [28] Matteo Marra, Guillermo Polito, and Elisa Gonzalez Boix. 2018. Out-of-Place Debugging: A Debugging Architecture to Reduce Debugging Interference. *The Art, Science, and Engineering of Programming* 3, 2 (Nov 2018), 28 pages. doi:10.22152/programming-journal.org/2019/3/3
- [29] Toni Mattis, Patrick Rein, and Robert Hirschfeld. 2017. Edit Transactions: Dynamically Scoped Change Sets for Controlled Updates in Live Programming. *The Art, Science, and Engineering of Programming* 1, 2 (1 April 2017), 32 pages. doi:10.22152/programming-journal.org/2017/1/13
- [30] Mark Samuel Miller. 2006. *Robust Composition: Towards a Unified Approach to Access Control and Concurrency Control*. Ph. D. Dissertation. Johns Hopkins University, Baltimore, Maryland, USA. <http://www.erights.org/talks/thesis/>
- [31] Eliot Miranda. 2009. *Under Cover Contexts and the Big Frame-Up*. <http://www.mirandabanda.org/cogblog/2009/01/14/under-cover-contexts-and-the-big-frame-up/> Cog Blog. Accessed on 2026-05-09.
- [32] Eliot Miranda. 2011. The Cog Smalltalk Virtual Machine—Writing a JIT in a High-Level Dynamic Language. In *Proceedings of the 5th Workshop on Virtual Machines and Intermediate Languages* (Portland, Oregon, USA) (VMIL 2011). 7 pages. doi:10.5281/zenodo.3700608 Presented at the workshop, but not archived in the ACM Digital Library (quoted after [19]).
- [33] 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* (Boston, MA, USA) (VMIL 2018). ACM, New York, NY, USA, 57–66. doi:10.1145/3281287.3281295
- [34] Stijn Mostinckx, Tom Van Cutsem, Stijn Timmermont, and Éric Tanter. 2007. Mirages: Behavioral Intercession in a Mirror-Based Architecture. In *Proceedings of the 2007 Symposium on Dynamic Languages* (Montreal, Quebec, Canada) (DLS '07). ACM, New York, NY, USA, 89–100. doi:10.1145/1297081.1297095
- [35] Fabio Niephaus, Tim Felgentreff, Tobias Pape, and Robert Hirschfeld. 2017. Squeak Makes a Good Python Debugger: Bringing Other Programming Languages Into Smalltalk's Tools. In *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming* (Brussels, Belgium) (Programming '17). ACM, New York, NY, USA, Article 27, 8 pages. doi:10.1145/3079368.3079402
- [36] David Pacheco. 2011. Postmortem Debugging in Dynamic Environments. *Commun. ACM* 54, 12 (Dec. 2011), 44–51. doi:10.1145/2043174.2043189
- [37] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot™ Server Compiler. In *Proceedings of the 2001 Symposium on Java™ Virtual Machine Research and Technology* (Monterey, California, USA) (JVM '01, Vol. 1). USENIX Association, USA, 21 pages. <https://www.usenix.org/conference/jvm-01/java-hotspot%E2%84%A2-server-compiler>
- [38] Nick Papoulias, Noury Bouraqadi, Luc Fabresse, Stéphane Ducasse, and Marcus Denker. 2015. Mercury: Properties and Design of a Remote Debugging Solution Using Reflection. *The Journal of Object Technology* 14, 2 (Sept. 2015), 36 pages. doi:10.5381/jot.2015.14.2.a1
- [39] Javier Pimás, Javier Burroni, and Gerardo Richarte. 2014. Design and Implementation of Bee Smalltalk Runtime. In *Proceedings of the 6th Edition of the International Workshop on Smalltalk Technologies* (Cambridge, UK) (IWST 2014), Alain Plantec and Jannik Laval (Eds.). 43–56. <https://esug.org/2014-Conference/iwst14-proceedings.pdf#page=43>
- [40] Javier Pimás and Stefan Marr. 2017. Metaphysics: Towards a Robust Framework for Remotely Working with Potentially Broken Objects and Runtimes. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on Meta-Programming Techniques and Reflection* (Vancouver, BC, Canada) (Meta '17), Shigeru Chiba, Elisa Gonzalez Boix, and Stefan Marr (Eds.). 5 pages. <https://pimas.ar/assets/images/2017-Meta-Metaphysics.pdf> Accessed on 2026-05-15.
- [41] Javier E. Pimás, Stefan Marr, and Diego Garbervetsky. 2023. Live Objects All the Way Down: Removing the Barriers between Applications and Virtual Machines. *The Art, Science, and Engineering of Programming* 8, 2 (Oct. 2023), 34 pages. doi:10.22152/programming-journal.org/2024/8/5
- [42] Ian Piumarta. 2005. *Accessible Language-Based Environments of Recursive Theories (a white paper advocating widespread unreasonable behavior)*. Technical Report. Viewpoints Research Institute. https://tinlizzie.org/VPRI Papers/rn2006001a_colaswp.pdf
- [43] Guillermo Polito. 2015. *Virtualization Support for Application Runtime Specialization and Extension*. Ph. D. Dissertation. Université des Sciences et Technologies de Lille. <https://inria.hal.science/tel-01251173>
- [44] Guillermo Polito, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse, and Max Mattone. 2015. Virtualization Support for Dynamic Core Library Update. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Pittsburgh, PA, USA) (Onward!). ACM, New York, NY, USA, 211–223. doi:10.1145/2814228.2814233
- [45] Guillermo Polito, Stéphane Ducasse, Luc Fabresse, Noury Bouraqadi, and Benjamin van Ryseghem. 2014. Bootstrapping Reflective Systems: The Case of Pharo. *Science of Computer Programming* 96 (2014), 141–155. doi:10.1016/j.scico.2013.10.008 Special Issue on Advances in Smalltalk-Based Systems.
- [46] Patrick Rein, Stefan Ramson, Tom Beckmann, and Robert Hirschfeld. 2025. An Information Foraging Interpretation of Liveness. In *Symposium on Visual Languages and Human-Centric Computing* (Raleigh, NC, USA) (VL/HCC). IEEE Computer Society, Los Alamitos, CA, USA, 128–138. doi:10.1109/VL-HCC65237.2025.00022
- [47] Patrick Rein, Stefan Ramson, Jens Lincke, Robert Hirschfeld, and Tobias Pape. 2019. Exploratory and Live, Programming and Coding: A Literature Study Comparing Perspectives on Liveness. *The Art, Science, and Engineering of Programming* 3, 1 (jul 2019), 33 pages. doi:10.22152/programming-journal.org/2019/3/1
- [48] Mary Beth Rosson and John M. Carroll. 1993. Active Programming Strategies in Reuse. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP '93)*, Oscar M. Nierstrasz (Ed.). Springer, Berlin, Heidelberg, 4–20. doi:10.1007/3-540-47910-4_2
- [49] Robin Salkeld and Gregor Kiczales. 2013. Interacting with Dead Objects. *SIGPLAN Notices* 48 (Oct. 2013), 203–216. doi:10.1145/2509136.2509543
- [50] David W. Sandberg. 1988. Smalltalk and Exploratory Programming. *SIGPLAN Notices* 23, 10 (1988), 85–92. doi:10.1145/51607.51614
- [51] Boris Shingarov and Jan Vraný. 2024. Towards a Dynabook for Verified VM Construction. *Journal of Computer Languages* 80, 101275 (2024), 14 pages. doi:10.1016/j.cola.2024.101275
- [52] Ben Shneiderman and Catherine Plaisant. 2005. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (4th ed.). Pearson Education, India.
- [53] David A. Smith, Alan Kay, Andreas Raab, and David P. Reed. 2003. Croquet - A Collaboration System Architecture. In *First Conference on Creating, Connecting and Collaborating Through Computing (C5 2003)*. 2–9. doi:10.1109/C5.2003.1222325
- [54] Marcel Taeumel and Robert Hirschfeld. 2016. Evolving User Interfaces from within Self-Supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs. In *Proceedings of the Programming Experience 2016 Workshop* (Rome, Italy) (PX/16). ACM, New York, NY, USA, 43–59. doi:10.1145/2984380.2984386

- [55] Marcel Taeumel, Jens Lincke, Patrick Rein, and Robert Hirschfeld. 2022. A Pattern Language of an Exploratory Programming Workspace. In *Design Thinking Research: Achieving Real Innovation*, Christoph Meinel and Larry Leifer (Eds.). Springer, Cham, 111–145. doi:10.1007/978-3-031-09297-8_7
- [56] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *1st International Workshop on Live Programming (LIVE)* (San Francisco, CA, USA). IEEE Computer Society, Piscataway, NJ, USA, 31–34. doi:10.1109/LIVE.2013.6617346
- [57] Christoph Thiede, Tom Beckmann, Marcel Taeumel, and Robert Hirschfeld. 2026. Literate Exploratory Programming for Asynchronous Collaboration. In *Companion Proceedings of the 10th International Conference on the Art, Science, and Engineering of Programming* (Munich, Germany) (*Programming Companion '26*). ACM, New York, NY, USA, 51–61. doi:10.1145/3801119.3801129
- [58] Christoph Thiede and Patrick Rein. 2023. *Squeak by Example* (6.0 ed.). Hasso Plattner Institute, Potsdam, Germany. ISBN 978-1-4476-2948-1.
- [59] Christoph Thiede, Marcel Taeumel, and Robert Hirschfeld. 2023. Object-Centric Time-Travel Debugging: Exploring Traces of Objects. In *Companion Proceedings of the 7th International Conference on the Art, Science, and Engineering of Programming* (Tokyo, Japan) (*Programming '23*), Shigeru Chiba, Youyou Cong, and Elisa Gonzalez Boix (Eds.). ACM, New York, NY, USA, 54–60. doi:10.1145/3594671.3594678
- [60] David Ungar, Henry Lieberman, and Christopher Fry. 1997. Debugging and the Experience of Immediacy. *Commun. ACM* 40, 4 (apr 1997), 38–43. doi:10.1145/248448.248457
- [61] David Ungar, Adam Spitz, and Alex Auch. 2005. Constructing a Metacircular Virtual Machine in an Exploratory Programming Environment. In *Companion to the 20th Annual Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) (*OOPSLA '05*). ACM, New York, NY, USA, 11–20. doi:10.1145/1094855.1094865
- [62] Tom Van Cutsem and Mark S. Miller. 2010. Proxies: Design Principles for Robust Object-Oriented Intercession APIs. *SIGPLAN Notices* 45, 12 (Oct. 2010), 59–72. doi:10.1145/1899661.1869638
- [63] 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. *Transactions on Architecture and Code Optimization (TACO)* 9, 4, Article 30 (Jan. 2013), 24 pages. doi:10.1145/2400682.2400689
- [64] Thomas Würthinger, Michael L. Van De Vanter, and Doug Simon. 2009. Multi-Level Virtual Machine Debugging Using the Java Platform Debugger Architecture. In *Proceedings of the 7th International Andrei Ershov Memorial Conference on Perspectives of Systems Informatics* (Novosibirsk, Russia) (*PSI '09*). Springer, Berlin, Heidelberg, 401–412. doi:10.1007/978-3-642-11486-1_34

Received 2026-05-16; accepted 2026-06-23