Multi-threaded OpenSmalltalk VM: Choosing a Strategy for Parallelization

Leon Matthes
leon.matthes@student.hpi.uni-potsdam.de
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

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

Eliot Miranda
eliot.miranda@gmail.com
Hasso Plattner Institute
University of Potsdam
Potsdam, Germany

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

ABSTRACT

Dynamic, object-oriented programming languages are widely regarded as enjoyable and easy to use. These languages lend themselves well to exploration and very short iteration cycles and feedback loops. However, many of them have no or limited support for multithreading. Squeak, a modern Smalltalk programming environment that focuses on interactivity and programming experience, doesn't support multithreading. We discuss multiple high-level strategies employed by similar languages and runtime environments to support parallel execution. Existing research and implementations using the presented strategies are analyzed to find a good fit for the Squeak/Smalltalk ecosystem. Due to Squeak's strong focus on interactivity and programming experience, we decided for an approach with limited support for parallelization. Our focus on a straight-forward implementation is based on our observation that reduction of pause times is more important for the programming experience than a model for fully parallel execution.

CCS CONCEPTS

• **Software and its engineering** → **Interpreters**; *Runtime environments*; *Object oriented frameworks*; • **Computing methodologies** → **Concurrent programming languages**.

KEYWORDS

object-oriented programming, virtual machine, concurrency

ACM Reference Format:
Leon Matthes, Marcel Taeumel, Eliot Miranda, and Robert Hirschfeld. 2024. Multi-threaded OpenSmalltalk VM: Choosing a Strategy for Parallelization. In *Companion Proceedings of the 8th International Conference on the Art, Science, and Engineering of Programming (‹Programming›Companion '24), March 11–15, 2024, Lund, Sweden.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3660829.3660846

‹Programming›Companion '24, March 11–15, 2024, Lund, Sweden

ACM ISBN 979-8-4007-0634-9/24/03
https://doi.org/10.1145/3660829.3660846

1 INTRODUCTION

Dynamic, object-oriented programming languages are widely popular, with JavaScript and Python as some of the most popular programming languages of the last years[1].

These languages are often praised as easy to use and beginner-friendly, but have also held up to the demands of many professional large-scale deployments. Therefore, it may sound surprising, that these languages still often have very limited support for parallel execution. Python is infamous for its "Global Interpreter Lock", which limits the VM to a single thread that executes Python code at any one time.

With such a large user base, it is surprising that the standard CPython VM never gained support for true parallelism. Today, multicore systems are the norm in personal computing, and by default Python cannot make full use of the available hardware. Similarly, other dynamic object-oriented languages such as Ruby, and even JavaScript, also only support limited parallelization.

We explore the design decisions that dynamic programming languages face when adding support for parallel execution. Through the lens of the Squeak/Smalltalk community, multiple strategies for parallelization are discussed and compared. The comparison will not only focus on best multithreaded performance, but also consider trade-offs like required maintenance effort and the effects on single-threaded programs.

Even though the requirements of the Squeak community depend on values in their specific ecosystem, this comparison should also help to understand why other similar languages have chosen their respective strategies.

2 BACKGROUND

The Squeak[2] system is an interactive programming environment that allows programmers great flexibility in editing and creating their own tools.

Squeak is based on the Smalltalk programming language [7]. The language is a dynamic, object-oriented programming language with extensive support for run-time reflection. In Smalltalk, everything is an object, including the Smalltalk code itself.

[1] According to the last years of the StackOverflow developer surveys: https://insights.stackoverflow.com/survey
[2] https://squeak.org/

This high level of reflection, combined with access to all development tools from within the Squeak environment itself, allows programmer free rein over their own programming experience. The freely modifiable tools include the Smalltalk source-to-bytecode compiler, the debugger, various tools for object inspection, the code browser, editor, and many more. Due to this flexible access, Squeak makes prototyping and development of new language concepts very easy.

Like many other dynamic languages, Smalltalk relies on a runtime to achieve this high level of reflection. In the case of Squeak, this runtime is the OpenSmalltalk Virtual Machine [14][3]. This virtual machine (VM) abstracts platform differences and therefore provides a stable target that the dynamic language can rely on.

Figure 1 shows a conceptual block diagram of the OpenSmalltalk-VM. It includes an execution engine (interpreter or just-in-time compiler) that takes care of Smalltalk bytecode execution. In the following we will use the term "interpreter" to cover both alternatives In a fully reflective language like Smalltalk, the interpreter reads the bytecode from the object memory and executes it. As Smalltalk code may freely modify the object space, it can therefore modify its own code.

Another important system provided by the OpenSmalltalk-VM is automatic garbage collection. As the VM prescribes the layout of the object memory, it can automatically collect objects that are no longer reachable. This removes a difficult task from the programmers and allows them to focus more on their programs desired behavior. The Garbage Collector is a system independent from the Interpreter. In the single-threaded OpenSmalltalk-VM, the Garbage Collector and the Interpreter take turns. Each system therefore has independent, exclusive access to the object memory, which is not thread-safe.

In the multicore era, this presents some challenges, however. As the Interpreter and Garbage Collector expect exclusive access to the Object Memory, the VM is entirely unable to use multiple threads for these primary functions. Squeak in turn is not able to take full advantage of modern multicore systems, as the VM does not support running multiple interpreters at once.

We explore different possibilities to achieve parallel execution of multiple *Interpreter* instances. Parallelization of Garbage Collection and Interpretation at the same time is out of scope for this paper. For this reason any further diagrams don't include the Garbage Collector. The reader should keep in mind that the Garbage Collector may still need exclusive access to the object space in all the presented scenarios, which can limit the potential for parallelism. To address this concern, existing literature describes multiple approaches that can be used to achieve concurrent interpretation and garbage collection [11][18].

3 OVERVIEW

In the following sections, different high-level approaches to parallelization of object-oriented language interpreters are discussed. Whilst the primary motivation is to decide on an approach to use in the Smalltalk ecosystem, many of the points made here can apply to other language ecosystems as well. Therefore, the following sections will remain rather general and refer to languages that are hosted by a VM in general, rather than mentioning Smalltalk specifically. Many of the presented approaches have already been implemented in other language ecosystems and existing implementations will be discussed.

[3]See also: https://github.com/opensmalltalk/

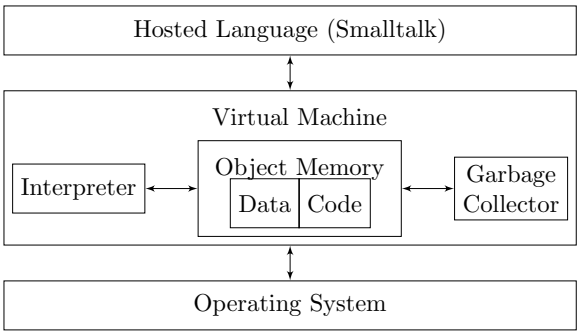

Figure 1: Block diagram of the OpenSmalltalk-VM

The sections are ordered by the amount of parallelization within the virtual machine itself, from least to most. This roughly corresponds to the amount of work required to support the implementation within the virtual machine itself. It does not necessarily correlate to the amount of parallelism possible in the hosted language.

The sections are:

- Parallel Operating System Processes (section 4)
- Multiple interpreters with a Global lock (section 5)
- Parallel interpreters with Shared Immutable State (section 6)
- Parallel interpreters with Shared Mutable State (section 7)

For the following discussion in section 8, the focus will be on comparing which implementation approach fits best for the needs of the OpenSmalltalk-VM, as well as the Squeak community at large. This includes the requirements of the virtual machine and its implementors, as well as the needs of the users (Smalltalk programmers). For this reason, the focus is not entirely on performance, but includes other trade-offs, like the required implementation and maintenance effort, which cannot be ignored in a relatively small community.

4 MULTIPLE OPERATING SYSTEM PROCESSES

All relevant operating systems provide support for multiple processes with isolated memory.

These operating system processes can be used to run multiple instances of the language virtual machine in isolation. Every interpreter will therefore have an isolated object space in which no race conditions can occur. This strategy can be often be employed without changes to the virtual machine. Often it can be implemented entirely in the hosted language itself. As long as the virtual machine exposes the required operating system functions, it is already capable of supporting this use case. Because this approach starts at a higher level of abstraction than the VM itself, it is also entirely compatible with all other approaches presented in this chapter. Figure 2 depicts a conceptual overview of this approach.

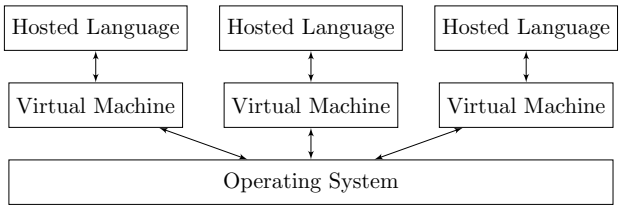

Figure 2: Parallelization using multiple OS Processes

Apart from the creation and management of OS processes, the operating system must also provide facilities for communication between processes (i.e. interpreter instances) to allow for synchronization. The communication protocols provided by the OS usually support inter-process communication via binary data streams (e.g. pipes, etc.), but not objects. As the synchronization is managed by the host language, the host language itself must be able to convert objects to and from binary streams. As any given object may represent an arbitrarily large directed graph, this is unfortunately not a trivial problem to solve for the general case. Grochowski et al. list and discuss some of the challenges involved in (de-)serialization, including platform differences, polymorphism, and multiple references to the same object. [8]

The "Distributed Smalltalk" [4] implementation explores a very similar approach. Bennet describes a system consisting of multiple independent Squeak/Smalltalk instances that are networked together and can therefore work in parallel. The system relies primarily on message passing as a means of communication between interpreter instances. Bennet shows that messages can be converted to a binary representation efficiently and can be relied upon for communication.

The individual operating system processes can be seen as objects, which send messages to each other, which fits well with the object-oriented nature of Squeak. Synchronizing code changes unfortunately remains a difficult problem that isn't entirely solved, as class incompatibilities can lead to data transfers being rejected. Virtual machine tasks like Garbage Collection also need to be reimplemented in the host language for shared data.

To summarize, the approach of using multiple isolated virtual machine instances requires minimal, if any, changes to the virtual machine. Some implementations like "Distributed Smalltalk" even implement this entirely in the hosted language. This approach can therefore be an option for languages where there is no parallel VM available, and even used in addition to any of the other approaches presented. The downside is a considerable amount of management needed in the hosted language. There is also a performance overhead, as all communication between interpreter instances must change format and pass through the OS.

5 MULTIPLE INTERPRETERS WITH A GLOBAL LOCK

For an existing single-threaded VM, an easy first step is to instantiate multiple interpreter instances, but guard the object space with one global lock that only allows a single interpreter to run at a time (see Figure 3).

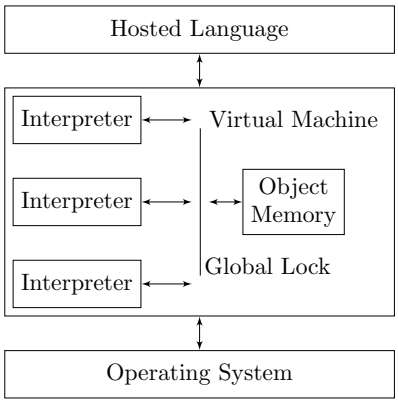

Figure 3: Multiple interpreters with a Global Lock

At first glance this may sound like it is no different to a single-threaded VM, and no parallel execution can take place. This only changes when callouts to external code are considered. In case an interpreter calls out to an operating system function or a library that does not modify the object space, it is possible to release ownership of the object space. During that time, another interpreter can acquire the lock and execute code in the hosted language.

This approach is still used by the standard Python and Ruby VMs, despite ample demand in the community for a more parallel VM. The rest of this section explores these implementations and their characteristics.

5.1 Python Global Interpreter Lock

The CPython virtual machine includes a "Global Interpreter Lock" (GIL). An interpreter that wants to execute Python code must acquire the GIL. The lock ensures that only a single interpreter may execute Python code or modify the Python object space at any given time. External code (typically extensions written in C) may unlock the CPython GIL and therefore allow another interpreter to run. In Python this is most often used in I/O operations, whilst waiting for or sending data.

For the CPython VM this implementation results in very good single-threaded performance and good parallelism of I/O bound threads. Larry Hastings lists multiple alternative implementations that would remove the GIL, but all the options explored thus far have seen degraded single-threaded performance [9]. Due to this single-threaded performance degradation, no attempt to remove the GIL has been approved by the CPython maintainers yet.[4]

On the other hand, research by Beazley [3] has explored the limitations of the GIL. CPython relies on operating system scheduling of threads, which historically led to a single thread repeatedly acquiring the GIL which starved other threads.

A redesign by Antoine Pitrou for Python 3.2 has changed the behavior to enforce a switch of the GIL owner. However, this implementation causes increased latency of I/O threads, if they are competing with compute threads that rarely release the GIL.

[4]See this statement about potential removals of the GIL: https://www.artima.com/weblogs/viewpost.jsp?thread=214235

Beazley proposes that the GIL behavior could become controllable and predictable by introducing thread priorities and preemption within Python.

5.2 Ruby Global VM Lock

"La concorrenza in Ruby" by Moro details the behavior of different versions of CRuby [15]. Support for multiple threads was added in CRuby 1.9, which also introduced a global lock. The Ruby community refers to this lock as the "Giant VM Lock", or "Global VM lock" (GVL). For simplicity, we will use the term "Global Interpreter Lock" (GIL) instead, as introduced above.

Ruby's GIL behaves very similarly to that of Python, and suffers from similar issues. It has been analyzed by Anjo, who has found increased latency when threading, as well as unfair scheduling of I/O threads when competing with compute threads [1].

For true parallelism, Moro recommends to use multiple OS processes, as described in section 4. Ruby and Python both provide libraries for this, the Process and multiprocessing modules respectively.

Summary. Introducing a global lock is an easy way to parallelize a VM without requiring the VM itself to be thread-safe. This can take advantage of times during which no execution of the hosted language would otherwise occur. The behavior and performance of single-threaded applications remains largely unchanged, which can be a desirable feature. If implemented naively, this strategy can lead to unpredictable program behavior and lock contention. Adding priorities to the threads and a preemption mechanism could ensure the predictability of execution.

6 ISOLATED INTERPRETERS WITH SHARED IMMUTABLE STATE

In this approach, multiple interpreters work in isolation. However, immutable data may be shared between interpreters (see Figure 4). Interpreters can therefore reside in the same operating system process/virtual machine and can be implemented with one thread per interpreter.

This approach lends itself well to implementing the Actor model for concurrency/parallelization in the hosted language. Different actors work independently of each other, but can communicate through message passing. These messages can be implemented efficiently by using shared but immutable data.

Actors and message passing have successfully served as the parallel execution model of the Erlang [2] and Racket [20] programming languages. More recently, Sasada and Matsumoto have proposed an adaptation of this model for use in the Ruby programming language [17].[5] This has been implemented in CRuby in the form of Ractors.[6] Like the approach presented in section 4, this model maps well to object-oriented languages. Actors are comparable to objects that send messages to each other and encapsulate their own state.

Compared to multiple OS processes, sharing immutable memory means removing the need for serialization of messages. This is especially useful for object-oriented languages, as (de-)serialization

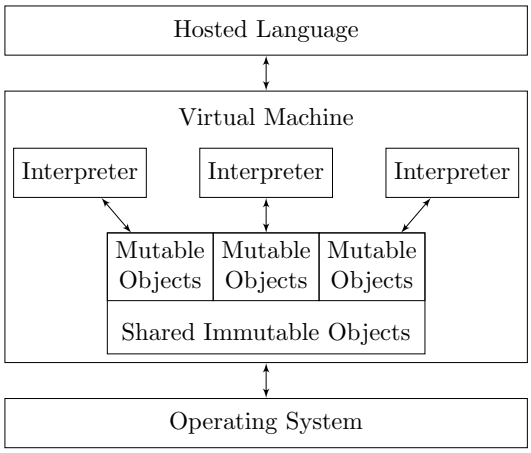

Figure 4: Multiple interpreters with shared immutable state

of objects is non-trivial in complex object graphs. The Ractor implementation has shown that creating shared immutable state can be achieved by simpler means, by either:

- Marking objects as read-only
- Moving object ownership between interpreters
- Recursively copying unshareable objects

Especially the first two approaches are very efficient, as they can be implemented by setting flags on the objects, without the need to copy any data.

Because shared objects are immutable, parallel access to these objects is possible. State Modifications however only affect objects that are accessible by a single interpreter, therefore no synchronization is required when objects are mutated. Only global mutable operations like garbage collection need to be synchronized. For this reason, the actors model can provide a good trade-off between implementation cost in the virtual machine and potential for parallel execution in the host language.

Depending on the required level of isolation, some synchronized mutable state may still be required. Ruby, like Smalltalk, allows for runtime reflection and modification (i.e. access to classes and other parts of the code). These changes to the program itself affect all interpreters. The virtual machine implementation must therefore ensure that these changes can be done safely.

In CRuby, shared state that must be synchronized includes:

- Global variables
- Objects representing code (i.e. Class, Module, etc.)
- Garbage collection state
- The Ractor object itself

The virtual machine synchronizes most of this state by only allowing the main Ractor to mutate (and in some cases even read) the shared state. However, the documentation warns that this can still lead to some race conditions in Ruby code.

For the hosted language, the actor model provides a system for parallel execution with little management required in the hosted language itself. The immutability of shared state prevents a large

[5] The original proposal used the name "Guild" to refer to different isolated interpreters. The was later changed to "Ractor" (Ruby Actor).
[6] https://docs.ruby-lang.org/en/master/ractor_md.html

class of issues that may occur in shared mutable state (e.g. race conditions). In traditional message passing, deadlocks are still possible, if actors are circularly waiting on messages [5].

The Storm language platform presents an interesting solution to this problem.[7] Every OS thread includes cooperatively scheduled user threads on top of the OS threads. When a message is passed from one thread to another, a new user thread is created to execute the corresponding function with the provided arguments. This removes opportunities for deadlocks, as messages don't have to be received manually. New messages can always be processed. As Smalltalk includes processes, which are analogous to Storms user threads, this solution could be a reasonable fit for Smalltalk as well. At the time of writing, the implications of this approach are not yet well understood however, and require further research.

While the actor model with message passing allows true parallelization, it still limits the hosted language by not allowing shared mutable state. This means that certain parallel programming patterns are not possible.

As Smalltalks object model and reflection capabilities are very similar to those of Ruby, an implementation of this approach for use in Squeak could be implemented by closely following CRubys Ractor design. Shared state could be implemented by using read-only objects, which Smalltalk already supports. An important difference is that Smalltalk developers can currently freely change the read-only flag by calling Object>>#setIsReadOnlyObject:. An additional flag would have to be added to ensure shared objects cannot be made writable again.

7 PARALLEL INTERPRETERS WITH SHARED MUTABLE STATE

The implementation strategy that provides the most flexibility to the hosted language is to provide a fully thread-safe virtual machine. For such an implementation, the virtual machine must ensure that it can keep all contracts even when multiple interpreters are modifying the object space and executing byte-code in parallel (see Figure 5).

Meier and Rigo compare multiple options that can be used to achieve parallel execution [12]. Options include fine-grained locking, as well as hardware and software transactional memory (HTM/STM). Which methodology to use depends entirely on the needs of the virtual machine and the host language. As a virtual machine consists of many systems that must all be thread-safe, implementing a virtual machine in a thread-safe manner is no trivial task.

For the hosted language on the other hand, having a fully thread-safe VM provides the most flexibility, as it has full access to parallelization. The potential for parallel performance scaling is therefore not limited by the virtual machine. However, programmers of the hosted language must be aware of the safety and correctness implications of parallel execution and handle synchronization of data structures correctly. This may require additional locking and management of data structures inside the hosted language itself, further increasing the amount of effort required for this approach.

It should also be noted that simply allowing parallel execution in the virtual machine may not necessarily improve performance of the hosted language. The RSqueak/VM by Felgentreff et al. [6] has

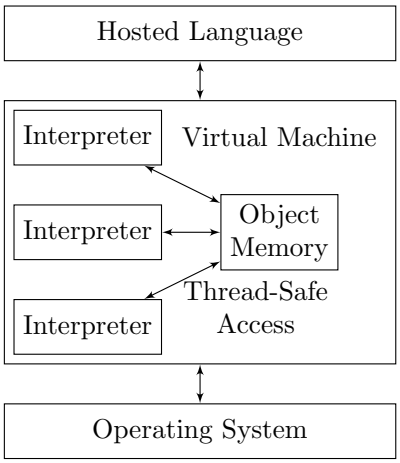

Figure 5: Multiple interpreters with shared mutable state

shown that a naive implementation may even degrade performance considerably. Their implementation uses STM to implement thread-safe interpreters in Squeak/Smalltalk. This reduced single-threaded performance considerably and only improved multithreaded performance in a few cases. Some multithreaded workloads like the Mandelbrot set even exhibited worse performance with multiple threads than without threading support.

On the other hand, works by Meier et al. show that it *is* possible to use STM for parallelization with success. They modify the PyPy[8] VM using a sophisticated STM approach that yields speedups in the range of 1.5x to 6.9x with 8 threads [13].

Implementors should keep in mind that fully parallel VMs tend to produce overhead in single-threaded workloads. This has been observed in the implementation by Meier et al., Felgentreff et al, and by Pallas and Ungar in their "Multiprocessor Smalltalk" implementation [16].

8 DISCUSSION

At first glance, a fully thread-safe virtual machine may seem like the no-compromises solution to parallelization.

For the Squeak/Smalltalk community, implementing such a VM may still not be the ideal. The implementation cost and complexity of such a VM is comparatively high. Depending on the number of VM maintainers available, implementing full thread-safety may tie up a significant percentage of development time, and may even lead to an unstable final product.

Two of the authors already explored how the values of openness, liveness, directness, malleability and feedback in the Squeak community favor Repairability over Reuse [19]. For this reason, a lot of features have been rewritten and are maintained in Smalltalk. This includes the VM itself, which is written in (a subset of) Smalltalk [10][14]. For Smalltalk developers this offers the better experience, as they can use familiar and powerful tools to modify all parts of their system, at the cost of having to maintain more code themselves.

[7] See: https://storm-lang.org/Language_Reference/Storm/Threading_Model.html (Last Access: 22.04.2024)

[8] https://doc.pypy.org/en/latest/index.html

To keep this large code base manageable, the Squeak community prefers covering 80% of use cases over supporting all edge-cases. Any missing cases can be easily implemented when needed. With this culture in mind, the community benefits more from a less parallel VM implementation and investing the time saved into other VM features. Especially due the increased maintenance efforts, the minimal implementation that still meets the requirements of Squeak can be considered ideal. Other small language communities may equally benefit more from a less parallel implementation.

In the Squeak system, interactivity and feedback to the user are of utmost importance. Longer computations are usually considered acceptable, as Squeak includes concurrency primitives that can keep the interactive environment running and provide feedback to the user during wait times. Interruptions of the interactive environment can however be caused by an interpreter that needs to execute external code written in another language (usually C) or interact with the operating system. The primary motivation for parallelization in Squeak is therefore to avoid times when no Smalltalk code is running and the interactive system comes to a halt. This can be solved by using multiple interpreters that can take over in case one blocks.

Therefore, the minimal parallelization strategy that supports this requirement is a GIL-based implementation (see section 5). Interpreters can unlock the object space when they are executing external code and another interpreter can continue execution of the interactive environment. It is also a good fit for the Squeak/Smalltalk community due to a multitude of other factors:

- The Squeak community is small, with few VM maintainers available.
- Extensive runtime reflection capabilities make synchronization of the object space difficult.
- The main GUI framework used in Squeak (called Morphic) is currently entirely single-threaded, reducing its performance should be avoided.
- Smalltalk Process instances are scheduled with priorities, which could avoid issues with GIL-contention.
- An actor-based implementation similar to Rubys Ractors provides a reasonable path forward, should a more parallel implementation be desired in the future.

With a simpler parallelization strategy, the VM maintainers can focus on other features. For example, adding incremental garbage collection could be more beneficial to the goal of interactivity than a fully parallel VM. It would remove noticeable stutters caused by the current fully synchronous Garbage Collection.

Larger ecosystems can also benefit from the use of separate VMs to optimally support single threaded and multithreaded programs. Both the Ruby and Python communities have chosen this route which has resulted in a great number of VM options for both languages, with some focused on ease of use and single-threaded performance, and others focused on multithreaded performance. Smaller communities should be careful though, as they can risk fragmenting maintenance efforts in the implementation of multiple VMs. Libraries of the hosted language must then also be checked for compatibility with multiple VMs, which can further increase the workload.

9 CURRENT PROTOTYPE

We are currently working on a prototype of the GIL-based approach in the OpenSmalltalk-VM. The prototype is already merged into the main Squeak source tree,[9] but disabled by default.

As expected, the implementation has proven reasonably simple. The simple semantic of unlocking and re-acquiring the GIL is easy and ergonomic to use, which provides a good experience to the developers that want to add threading to external code. Retaining liveness and interactivity is therefore easy to do, which solved the most pressing issues around long-running external code. Compared to CRuby and CPython, Squeak includes priorities and a scheduler. Preliminary results support the theory that this is beneficial, as using this scheduler could fix latency issues and avoid contention on the GIL.

We have however found two new issues that are specific to the Squeak-based implementation. The first issue is around state persistence. The Squeak system state is saved when quitting the VM. Unfortunately, this doesn't apply to threads that are currently executing external code, which lose their state. We are currently exploring different solutions to this problem.

Additionally, the current implementation automatically manages a thread-pool, instead of assigning a single thread for each Smalltalk process. This allows us to avoid some threads switches, but leads to incompatibilities with certain pieces of external code. Some low-level functions need to be called from a specific thread.[10] Adding thread affinity to Smalltalk processes gives the user the required tools to solve this issue.

Squeaks built-in tooling allows us to easily support Smalltalk developers with this new feature. We have adapted the Process Browser, such that it can display which Processes are currently executing in another thread. To inspect which threads are in use at a given time, we have added a custom inspector that graphs this information. The VM profiler has also been adapted to show data from individual threads. We hope that these changes provide Smalltalk developers with a good experience when using parallelism within Squeak.

All in all the current prototype is promising. It allows us to solve the problem of keeping interactivity during long-running external code, without an overly complicated implementation. For the future we aim to stabilize the prototype so that we can enable it by default.

10 CONCLUSION

There are several high-level strategies to parallelize virtual machines of dynamically-typed programming languages. Unless multi-threaded scaling is chosen as the only goal, none of those strategies seemed an ideal fit. Because of that, many of the presented approaches remain in use in real world systems despite their often limited support for parallel execution.

Squeak/Smalltalk's primary motivation for supporting parallel execution is to reduce or avoid pauses in interpretation of Smalltalk when executing external code. From the approaches discussed, a

[9] http://source.squeak.org/VMMaker.html

[10] This includes event handling on macOS, which must happen on the main thread, as well as OpenGL calls, which operate on the context of the current thread. (See: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/Multithreading/ThreadSafetySummary/ThreadSafetySummary.html and https://www.khronos.org/opengl/wiki/OpenGL_and_multithreading).

limited parallelization is proposed as the best fit. For this requirement, the approach discussed in section 5 (Multiple Interpreters with a Global Lock) is sufficient and requires little effort compared to the other approaches.

REFERENCES

[1] Ivo Anjo. 2023. Understanding the Ruby Global VM Lock (GVL) by observing it. RubyKaigi Ruby Conference. Nagano, Japan. https://ivoanjo.me/blog/2023/07/23/understanding-the-ruby-global-vm-lock-by-observing-it/

[2] Joe Armstrong. 2010. Erlang. *Commun. ACM* 53, 9 (sep 2010), 68–75. https://doi.org/10.1145/1810891.1810910

[3] David Beazley. 2010. Understanding the Python GIL. PyCON Python Conference. Atlanta, Georgia. https://dabeaz.com/python/UnderstandingGIL.pdf

[4] John K. Bennett. 1987. The design and implementation of distributed Smalltalk. *SIGPLAN Not.* 22, 12 (dec 1987), 318–330. https://doi.org/10.1145/38807.38836

[5] Mats Cronqvist. 2004. Troubleshooting a large erlang system. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Erlang* (Snowbird, Utah, USA) *(ERLANG '04)*. Association for Computing Machinery, New York, NY, USA, 11–15. https://doi.org/10.1145/1022471.1022474

[6] Tim Felgentreff, Tobias Pape, Patrick Rein, and Robert Hirschfeld. 2016. How to Build a High-Performance VM for Squeak/Smalltalk in Your Spare Time: An Experience Report of Using the RPython Toolchain. In *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies* (Prague, Czech Republic) *(IWST'16)*. Association for Computing Machinery, New York, NY, USA, Article 21, 10 pages. https://doi.org/10.1145/2991041.2991062

[7] Adele Goldberg and David Robson. 1983. *Smalltalk-80: the language and its implementation.* Addison-Wesley Longman Publishing Co., Inc.

[8] Konrad Grochowski, Michał Breiter, and Robert Nowak. 2019. Serialization in Object-Oriented Programming Languages. In *Introduction to Data Science and Machine Learning*, Keshav Sud, Pakize Erdogmus, and Seifedine Kadry (Eds.). IntechOpen, Rijeka, Chapter 12. https://doi.org/10.5772/intechopen.86917

[9] Larry Hastings. 2015. Python's Infamous GIL. PyCon Python Conference. Montréal, Canada.

[10] 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* (Atlanta, Georgia, USA) *(OOPSLA '97)*. Association for Computing Machinery, New York, NY, USA, 318–326. https://doi.org/10.1145/263698.263754

[11] Richard Jones, Antony Hosking, and Eliot Moss. 2023. *The garbage collection handbook: the art of automatic memory management.* CRC Press. https://doi.org/10.1201/9781003276142

[12] Remigius Meier and Armin Rigo. 2014. A way forward in parallelising dynamic languages. In *Proceedings of the 9th International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems PLE* (Uppsala, Sweden) *(ICOOOLPS '14)*. Association for Computing Machinery, New York, NY, USA, Article 4, 4 pages. https://doi.org/10.1145/2633301.2633305

[13] Remigius Meier, Armin Rigo, and Thomas R. Gross. 2018. Virtual machine design for parallel dynamic programming languages. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 109 (oct 2018), 25 pages. https://doi.org/10.1145/3276479

[14] 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)*. Association for Computing Machinery, New York, NY, USA, 57–66. https://doi.org/10.1145/3281287.3281295

[15] Federica Moro. 2010. La concorrenza in Ruby. http://hdl.handle.net/20.500.12608/14024 Read in English and translated using DeepL Translate.

[16] J. Pallas and D. Ungar. 1988. Multiprocessor Smalltalk: a case study of a multiprocessor-based programming environment. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 268–277. https://doi.org/10.1145/53990.54017

[17] Koichi Sasada and Yukihiro Matsumoto. 2016. A proposal for a new concurrent execution model in Ruby 3. *Transactions of Information Processing Society of Japan, Programming (PRO)* (2016). Read in English and translated using DeepL Translate.

[18] KC Sivaramakrishnan, Stephen Dolan, Leo White, Sadiq Jaffer, Tom Kelly, Anmol Sahoo, Sudha Parimala, Atul Dhiman, and Anil Madhavapeddy. 2020. Retrofitting parallelism onto OCaml. *Proceedings of the ACM on Programming Languages* 4, ICFP (Aug. 2020), 1–30. https://doi.org/10.1145/3408995

[19] Marcel Taeumel and Robert Hirschfeld. 2022. Relentless Repairability or Reckless Reuse: Whether or Not to Rebuild a Concern with Your Familiar Tools and Materials. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software* (Auckland, New Zealand) *(Onward! 2022)*. Association for Computing Machinery, New York, NY, USA, 185–194. https://doi.org/10.1145/3563835.3568733

[20] Kevin Tew, James Swaine, Matthew Flatt, Robert Bruce Findler, and Peter Dinda pdinda@northwestern.edu. 2011. Places: adding message-passing parallelism to racket. In *Proceedings of the 7th Symposium on Dynamic Languages* (Portland, Oregon, USA) *(DLS '11)*. Association for Computing Machinery, New York, NY, USA, 85–96. https://doi.org/10.1145/2047849.2047860