GildaVM: a Non-Blocking I/O Architecture for the Cog VM

Guillermo Polito, Pablo Tesone, Eliot Miranda, David Simmons

GildaVM: a Non-Blocking I/O Architecture for the Cog VM

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

Pablo Tesone
Pharo Consortium, France
pablo.tesone@inria.fr

Eliot Miranda
Stellect Systems Inc, USA
eliot.miranda@gmail.com

David Simmons
The Light Phone, USA
smallscript1@gmail.com

Abstract

The OpenSmalltalk virtual machine (VM) was historically designed as a single-threaded VM. All VM code including the Smalltalk interpreter, the garbage collector and the just-in-time compiler run in the same single native thread. While this VM provides concurrency through green threads, it cannot take advantage of multi-core processors. This architecture performs really well in practice until the VM accesses external resources such as *e.g.,* FFI callouts, which block the single VM thread and prevent green threads to benefit from the processor.

In this paper we present GildaVM, a multi-threaded VM architecture where one thread at a time executes the VM while allowing non-blocking I/O in parallel. The ownership of the VM is orchestrated by a Global Interpreter Lock (GIL) as in the standard implementations of Python and Ruby. However, within a single VM thread concurrency is still possible through green threads. We present a prototype implementation of this architecture running on top of the Stack flavour of the OpenSmalltalk VM. We finally evaluate several aspects of this architecture like FFI and thread-switch overhead. While current benchmarks show good results for long FFI calls, short FFI calls require more research to minimize the overhead of thread-switch.

Keywords virtual machines, multhreading, non-blocking I/O, ffi

1 Introduction

The OpenSmalltalk [1] virtual machine (VM) [7], main runtime platform of several Smalltalk dialects such as Pharo, Squeak and Newspeak, was historically designed as a single-threaded VM. All VM code including the Smalltalk interpreter, the garbage collector and the just-in-time compiler run in the same single native thread. Although the single-threaded nature prevents parallel execution, the VM provides concurrency through green threads [11, 12]: threads that are scheduled by the virtual machine itself instead of relying on the operating system for it. Green threads run in slices of time, running one at a time in the single VM thread, which is shared in the long run by all green threads.

This architecture performs really well in practice, until there is a need for accessing resources residing outside of the VM such as external libraries or input/output (I/O) *e.g.,* reading from a socket or writing into a large file. Indeed, in a single threaded VM I/O and FFI callouts are *blocking*: they take ownership of the native VM thread until they are finished, thus preventing green threads to benefit from the processor (Section 2).

Supporting safe parallel execution of both VM and I/O in a more traditional way, *i.e.,* using native threads, requires a complete rewrite of the runtime system and the libraries on top [13], considering that the VM and many language libraries are not thread-safe. As a compromise, existing programming language implementations such as CPython or Ruby's MRI introduce concurrency through native threads using a lock that protects concurrent access to the language interpreter, namely a global interpreter lock (GIL) [1]. While the GIL prevents parallel code execution in the interpreter, it is released during blocking I/O to them to run in parallel with the interpreter code (Section 10).

In this paper we present GildaVM, a hybrid multi-threaded architecture that combines green threads with native threads, originally designed by David Simmons (Section 3). The VM has not one but many native threads that share the same VM. Only one thread at a time executes VM code while other threads execute blocking I/O in parallel. When the VM performs blocking I/O, another thread takes the relay and continues executing the VM, making I/O *non-blocking* (Section 4). This architecture is tailored to handle blocking FFI callouts (Section 5) and minimize the overhead on them (Section 6). The ownership of the VM is orchestrated by a GIL implementation that allows the VM to selectively schedule native threads instead of relying on the operating system (Section 7). Moreover, this architecture keeps using green threads to allow concurrent Smalltalk programs and thus providing

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

IWST'19, August 27–29, 2019, Cologne, Germany
2019.

backwards compatibility with most of the existing programs that run on the VM.

We present a prototype implementation of GildaVM, originally started by Eliot Miranda and now built on top of Cog's StackVM flavour. Our prototype shows that no significant overhead is introduced when threading is not explicitly used, and that long blocking I/O operations are particularly benefited with an improvement of around to 80% of execution time. Yet, thread switching shows itself particularly expensive, producing significant performance degradations in scenarios performing many short blocking I/O operations (Section 8). In the future we will research the impact of this architecture with the JIT compiler, and the implementation of the vm lock in terms of a low-overhead mutex such as a CompareAndSwap operation.

2 Problem: Blocking I/O and FFI

I/O operations such as reading from a socket block the VM execution because they are not under the VM's control: they use system-calls from the underlying operating system. Blocking I/O is un-optimal for several reasons: (1) I/O operations spend most of their time in hardware-related system calls making the processor idle and (2) keep the VM thread locked preventing other green threads to benefit from the idle processor. Moreover, the same problem happens when performing external library calls (a.k.a., FFI callouts), which is one of the basic bricks of integration with external systems.

To illustrate why this is a problem, let's consider a web application performing a requests to a relational database using a SQL query. If the execution of this SQL query takes several milliseconds to finish, the web application will be blocked the same amount of time. In other words, no new requests will be handled during that time, producing slow responses and throughput.

VM Plugins are not enough. Currently in the OpenSmalltalk VM, developers provide asynchronous access to external resources through VM extensions/plugins. Using plugins, developers avoid blocking the VM thread by combining two different mechanisms:

- Do not block the VM thread using non-blocking system calls such as UNIX's select, and/or start dedicated native threads to notify the VM upon completion;
- Suspend the calling green thread on a VM managed semaphore letting the VM to schedule ready green threads.

Although this solution has proven useful for plugins such as the SocketPlugin, these plugins are often handcrafted and require a significant effort to write, build and deploy. Also, using this strategy for integration with external libraries (*i.e.,* libgit, Cairo, SDL) requires the implementation of an ad-hoc solution for each of the libraries that perform long or asynchronous calls.

GildaVM in a Nutshell. In the GildaVM architecture, the VM has no longer one but many threads that share the same VM. To guard from the fact that the VM and many language libraries are not thread-safe, only one VM thread at a time *owns* the vm. At the core of this architecture, there is a *Global Interpreter Lock (GIL)*, a lock creating a critical section around VM code. The GIL limits the VM to run in a single VM thread while other VM threads execute code from outside the VM in parallel *e.g.,* external libraries and wait for I/O.

3 Many Threads for a Single-Threaded VM

Instead of a single native thread executing the VM, the GildaVM architecture introduces a pool of native threads, namely VM threads. All VM threads have the same life-cycle, where they alternate the execution inside the vm-executive and outside of it. The vm-executive is all code related to VM tasks such as interpretting bytecode, executing JIT'ed code, running the garbage collector. All tasks unrelated to VM execution are considered outside the vm-executive, typically system calls performing I/O and FFI callouts.

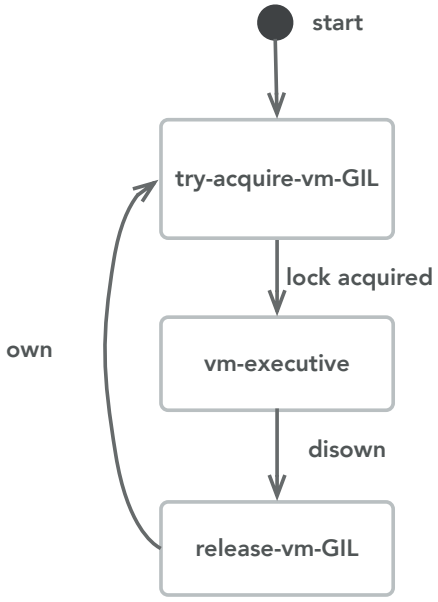

Figure 1. Illustrated life-cycle of a VM thread in the presence of I/O. The GIL is acquired before entering the vm-executive, and released on I/O operations. When the I/O operation finishes the VM thread attempts to re-acquire the GIL.

Figure 1 illustrates the life-cycle of a VM thread. When a VM thread starts it first tries to acquire the GIL. If the VM thread fails to acquire the GIL it blocks until the lock becomes free. Otherwise, the VM thread *owns* the VM: it acquires the GIL and enters the vm-executive. When the thread performs a blocking I/O operation it *disowns* the vm: it releases the GIL to let some other VM thread acquire it. Eventually, when

the blocking I/O operation finishes its execution and returns, the VM thread goes back to step 1 and tries to *re-own* the GIL.

Implementation wise, these boundaries are explicitly set in those VM plugins performing I/O operations. It is the VM plugin developers responsibility to explicitly *own* and *disown* the VM to allow parallel execution.

4 VM Thread Scheduling

In the GildaVM architecture only one thread is in the vm-executive code in a given moment. Initially, the vm-executive is owned and executed by a single thread: the main thread. A separate VM thread enters the vm-executive in one of two cases: (1) the developer explicitly states so by using a thread affinity schema (Section 4.1) or (2) the VM thread owning the VM releases it to execute an FFI call (Section 4.2).

Moreover, in the presence of multiple threads, the VM needs to manage the scheduling of both native VM threads and the already existing green threads. We call green thread scheduling to the process deciding which green thread is executed, and VM thread scheduling to the process deciding which VM thread gains access to the GIL and enters the vm-executive. These two mechanisms collaborate to keep the vm-executive busy as much as possible (Section 4.3).

Figure 2 updates the conceptual thread life-cycle considering the entire scheduling schema. A new transition represents a VM thread switch occurring because of a green thread switch. In this case, vm-executive transfers explicitly the GIL to the switched VM thread, wakes that VM thread up, and finally goes back to step 1 trying to re-acquire the GIL.

4.1 Green Thread Scheduling and Affinity

To explicitly assign code to run in a separate VM thread, the GildaVM architecture proposes an affinity schema in which a green thread is affined or bound to only-one native thread. A green thread *affined* to a VM thread will only execute in that VM thread. *Non-affined* green threads have affinity with no VM thread, so they may run on any VM thread.

This affinity schema produces a NxM mapping between green threads and VM threads where N green threads run on M native VM threads, with the edge case of M=1 for affined green threads. When the VM schedules a green thread that is affined to a VM thread different from the current one, VM ownership is transferred to the affined VM thread. In this scenario a *VM thread switch* happens: the running VM thread releases the vm-executive and awakes the VM thread affined with the awoken green thread. Table 1 presents the possible scenarios when a green thread is scheduled. Section 7 further describes how our implementation of a GIL allows us to perform this selective VM thread scheduling.

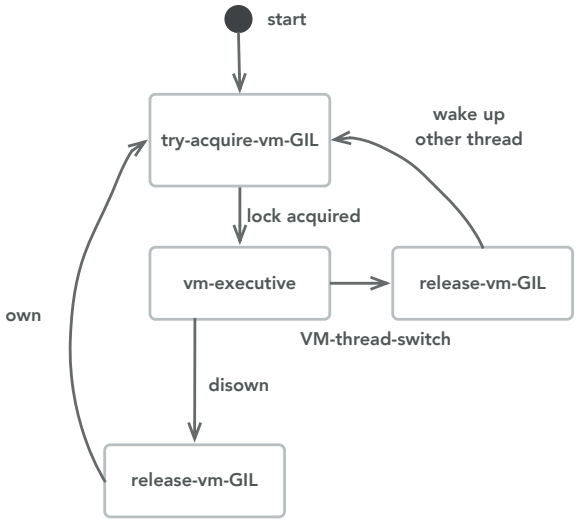

Figure 2. Illustrated life-cycle of a VM thread in the presence of VM thread-switch. When a green thread affined to a VM thread different than the current one is scheduled to run, the current VM thread releases the GIL and the affined VM thread is awoken.

Green Thread	Action
Not affined	Schedule green thread in current VM thread. No VM thread switch takes place
Affined to *current* VM thread	Schedule green thread in current VM thread. No VM thread switch takes place.
Affined to *different* VM thread	VM thread switch to the corresponding VM thread. Schedule green thread in the switched VM thread.

Table 1. VM scheduling rules in case of green thread affinity.

4.2 Scheduling on GIL Release

A VM thread owning the GIL releases it before performing an I/O operation, and thus it leaves the VM unowned. The VM must then ensure that another VM thread becomes active to avoid VM starvation (*i.e.,* avoid making the VM inactive for long periods of time). For this the VM performs automatically an *implicit VM thread switch* on a disown. The implicit VM thread switch consists in a green thread scheduling and a VM thread scheduling. In other words, it awakes a new VM thread on a new green thread. This is required because both the current green thread and VM thread are blocked waiting for the I/O operation to finish.

Several *VM upkeep* considerations have to be taken into account in this scenario to leave the vm-executive in a coherent

state. The green thread performing the I/O operation must be suspended in a safe suspension point and thus the VM must preserve its execution state. Also, all the thread-local VM state should be transferred to the global-state of the VM to allow other VM threads to enter the vm-executive in the same state. Then, the owning VM thread must restore the scheduled green-thread's state, restore all of the VM's state and enter the vm-executive. Only then, the VM resumes a ready green thread in the point it was suspended earlier.

Intuitively, performing this upkeep on each I/O operation introduces a performance overhead that may not be negligible. Section 5 presents a smarter way to outsource thread scheduling from the disowning VM thread to the owning VM thread to minimise this overhead.

4.3 Green Thread Scheduling

Green thread scheduling remains conceptually unchanged in comparison with the historical VM. Green threads are cooperative between threads with of same priority *i.e.,* a green thread has to explicitly *yield* to let other green threads with the same priority run. Green threads are preemptive with threads of lower priorities *i.e.,* if a green thread with higher priority is ready to run the current green thread is suspended to let the higher priority one to run. Green thread scheduling is triggered by two main reasons: explicit synchronisation operations and external events.

On the one hand, the VM exposes synchronisation operations to *yield* the processor, and *suspend* and *resume* a green thread either explicitly or implicitly through the usage of VM-managed semaphores (using *wait* and *signal* respectively). Each of these operations determine if the current green thread needs to be put to sleep and then schedule a new green thread to launch.

On the other hand, green thread scheduling happens also implicitly due to events outside of the vm-executive signaling VM-managed semaphores. When external events require green thread scheduling, the VM waits until the vm-executive reaches a safe suspension point before doing it. A separate thread, namely the heartbeat thread periodically signals an event to force the VM to check if a green thread scheduling is necessary. For example, sockets are implemented through a combination of these mechanisms. Asynchronous reading from a socket waits on a VM-managed semaphore producing the suspension of the waiting green thread. Eventually, when the socket VM plugin finished reading it signals that same semaphore to make the suspended green thread ready to run again.

The GildaVM architecture does not modify the rules for green thread scheduling. Instead, it augments them with VM thread switch semantics. That is, if at any time the green thread scheduling activates a green thread affined to another VM thread, the VM performs a VM thread switch to the second thread.

5 Non-Blocking FFI

The GildaVM architecture is particularly suited for non-blocking FFI callouts. For example, thread affinity allows developers to control what FFI callouts are executed in what VM thread. However, native library callbacks add the possibility of reentrant FFI callouts*i.e.,* FFI callouts in the dynamic context of other FFI callouts. In this section we present additional considerations to perform safe callbacks and re-entrant callouts.

5.1 Re-entrant Callouts and Callbacks

The safe execution of FFI callouts depends often on how the called library is designed. Indeed, some external libraries are not (sometimes by design) thread-safe, meaning they do not perform well if different callouts are executed in parallel from different threads. It is the developer's responsibility to ensure that all callouts to that library are performed from one or more green threads affined to the same native VM thread.

Callbacks from external libraries invoking VM code must be taken into consideration in this design too. Callbacks are pieces of code (*i.e.,* block closures in the current implementation) sent as argument to a callout and eventually invoked from the external library to execute some smalltalk code. When an external library invokes a callback, the thread where it happens waits until it acquires back the GIL and then executes the callback within the vm-executive. When the callback finishes its execution and returns, the VM is disowned and the external library that performed the callback takes the control back.

Callbacks return the control to the vm-executive, leading to the case of reentrant or nested callouts *i.e.,* the vm-executive performs a new callout within the dynamic execution context of a previous callout. Figure 3 shows an example of a reentrant callout: callout2 is invoked before callout1 has finished.

5.2 Dynamic Affinity

A different scenario presents itself in the absence of thread affinity. Let's suppose the case of a green thread A without thread affinity performing a callout. This callout and further callbacks coming from it will be executed in current VM thread. A problem appears if during the callback execution a thread switch transfers the vm-executive to a separate VM thread, since the green thread executing the callback can execute in a different thread. Moreover, reentrant callouts coming from this green thread will now run in a different VM thread creating potential race conditions and failures.

To solve this problem, the OpenSmalltalk VM's GIL introduces the idea of dynamic or temporary affinity within the scope of a callout. That is, when a callout happens, if the current green thread has no affinity to a VM thread the VM affines it with the current VM thread. Callbacks and

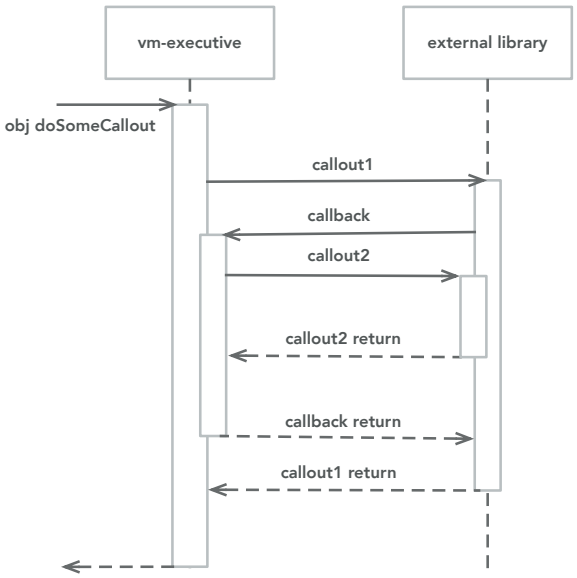

Figure 3. Reentrant callouts in the presence of callbacks. Callout2 is a reentrant callout, executed during the dynamic scope of callout1's execution. This happens because a callback gives back the control to the vm-executive, allowing new callouts before callout1 finishes.

reentrant callouts within the scope of this execution then benefit from thread affinity as if it was explicitly defined by the developer. Finally, when the initial callout returns, the VM removes the affinity of the current green thread, leaving it back in an unaffined state.

6 Lowering the Overhead of I/O Operations

One of the objectives of the GildaVM architecture is to keep the VM running as much as possible when one or more green threads are blocked in I/O. When a VM thread disowns the VM on I/O, if there are threads waiting to acquire the vm-executive, the VM selects and activates one of the waiting VM threads. Safely transferring the VM's ownership between VM threads requires that on each VM thread-switch the running green thread is *preempted* to let other green threads run *i.e.,* it is suspended and the execution stack is left in a coherent state.

While a safe VM thread-switch is indeed capital for the correct execution of the VM, this requires forcing green thread suspension and a VM thread-switch when a VM thread disowns the VM. All this VM upkeep represent a significative overhead in I/O operations, specially in short ones. In this section we describe how the responsibility for VM thread preemption and activation are removed from a disowning VM thread to minimize the overhead on I/O.

6.1 Green Thread Preemption is the Acquirer's Responsibility

To minimize the overhead on I/O operations, the responsibility of performing the green thread scheduling is transferred from the disowning thread to the owning thread. In other words, it is the VM thread that enters the vm-executive that performs the VM upkeep. This allows the disowning VM thread to not perform any stack or green thread upkeep and keep on executing the I/O operation. When the I/O operation finishes, the VM thread makes an attempt to re-own the VM. If no other VM thread was activated while the VM was free, the VM thread finds the execution stack in the same shape it was before the callout. Thus, the VM thread directly re-enters the vm-executive avoiding all the extra-work of VM upkeep.

If another VM thread acquires the GIL in the middle of an I/O operation, it *preempts* the first VM thread. It suspends the current green thread and sets it in a *blocked in callout* state. Then, it performs a green thread scheduling and enters the vm-executive. Now, when the former VM thread re-owns the VM, it detects it had been preempted and restores the stack of the *blocked in callout* process. This makes the slow case of execution to happen only if a callout returns and the current VM thread had been preempted by another thread.

This strategy particularly benefits short I/O operations and FFI callouts. Indeed, if they are fast enough to disown and own the interpreter without being preempted, they do not pay the price of the VM upkeep.

6.2 The Thread Watchdog and Thread Activation

Disowning the VM on I/O and FFI callouts triggers an implicit VM thread switch to keep the vm-executive busy and avoid starvation (Section 4.2). However, doing so when disowning incurs into additional (and undesired) overhead. To avoid this overhead we adopted a solution similar to the one above: move the responsibility of VM thread scheduling from the disowning thread to a separate thread, namely *a watchdog* thread.

The watchdog is a separate thread that actively checks if the VM is free with a fixed frequency of 200 micro-seconds. If the watchdog finds that the VM is free, it acquires the GIL and performs a VM thread scheduling, transfering the VM ownership to a VM thread waiting for ownership or to a free VM thread in the thread pool.

This strategy removes the extra scheduling overhead from I/O operations and FFI callouts while keeping the vm-executive busy with a best effort. Still, the impact of the watchdog frequency on the behaviour of the VM requires more research. On the one hand, during the window of time in which the watchdog is not active I/O operations and FFI callouts will not be preempted. On the other hand, during this window of time the vm-executive is idle if all active threads are waiting on I/O and FFI callouts.

7 A Mutex is not enough: a GIL for VM Scheduling

To allow the VM to select which VM thread to awake during VM thread scheduling, the GIL is not implemented as a mutex lock. A mutex lock would relegate the entire VM thread scheduling to the operating system, as discussed in Section 9.2. Instead, the GIL is implemented with:

A vmOwnerId global variable.
Storing the current thread id owning the VM. Its value determines whether a green thread has affinity with the current VM owner or not. The access to this variable is synchronized through the vmOwnerLock to avoid race conditions.

A vmOwnerLock global variable.
Storing the global lock to protect concurrent access to vmOwner.

A semaphore thread-local variable.
One native semaphore per VM thread allowing the VM to selectively wake up the desired process.

Acquiring the GIL boils down to first check if the VM is unowned or the current VM thread is the VM owner, in which case the thread can enter the vm-executive code safely. If the current VM thread does not own the VM, it suspends itself waiting in the thread-local semaphore. VM Thread scheduling transfers the vm-executive to another thread by (a) setting that thread as vmOwner and (b) signaling its semaphore to wake it up.

Releasing the GIL unsets the vmOwnerId variable: since releasing the GIL only happens from the single thread that owns the VM, doing it does not require thread synchronization. Figure 4 illustrates how the GIL acquisition works in pseudo-code.

```
VM >> threadSchedulingLoop
  [ self tryAcquireVM
      ifTrue: [ self enterSmalltalkExecutive ].
    currentThread semaphore wait.
    true ] whileTrue.

VM >> tryAcquireVM
    | vmAcquired |
    "Tries to set the vmOwner variable safely.
      Returns true if the VM has been acquired or the current
       thread already owned it.
      Returns false if the VM is being owned by other thread
       or the vmOwner was locked"
    vmOwnerLock tryLock ifFalse: [ ^ false ].
    self isVmUnowned ifTrue: [
      vmOwnerId := currentThread threadId ].
    vmAcquired := vmOwnerId = currentThread threadId.
    vmOwnerLock release.
    ^ vmAcquired

VM >> releaseVM
    vmOwnerId := 0.

VM >> isVmUnowned
    ^ vmOwnerId = 0.

VM >> wakeUpVMThread: vmThread
    vmOwnerId := vmThread threadId.
    vmThread semaphore signal.
```

Figure 4. Implementing the GIL with Semaphores. Pseudo code illustrating the GIL implementation with a semaphore-per-thread instead of a single mutex.

8 Evaluation

In this section we evaluate a prototype of the GildaVM architecture implemented on top of the StackVM flavour of the OpenSmalltalk's VM. Our validation consists in several benchmarks measuring several aspects of the architecture in the presence of FFI callouts as an example of blocking operations. Our initial measurements show that this architecture presents a faster execution of long parallel FFI callouts. However, there is an important penalty in the execution of parallel short FFI callouts. Moreover, we also show that the execution of parallel green threads that do not execute FFI callouts is neither improved nor penalized by the solution.

8.1 Setup

Our benchmarks compare the stock Stack 64 bits implementation of the OpenSmalltalk-VM[2] and our modified version[3] both running the same version of Pharo 8 64 bits. These versions do not include the support for Cog's just-in-time compiler. We have performed our benchmarks on Mac OS Sierra (10.12.6) running on a 2,8 GHz Intel Core i5 processor and 8Gb of RAM.

All our benchmarks are included in the Pharo benchmark project[4]. Each benchmark was run 50 times. For the sake of clarity, the results shown in our analyses include only the average of the runs.

8.2 Overhead Executing Smalltalk code

In this scenario we analyse the overhead produced during the execution of Smalltalk code that does not perform any FFI call. All green threads are unaffined and are able to run on any of the VM threads. We executed three different benchmarks:

Same priority without yielding:
We spawn three green threads doing processor intensive operations in a close loop (calculating 100 times the factorial of 600). All threads have the same priority to make them run serialized. The benchmark measures how much time it takes for all green threads to end.

[2]We used the built version corresponding to the commit 4a3c9 in http://github.com/OpenSmalltalk/opensmalltalk-vm

[3]We used the commit 69e50d5 available in http://github.com/tesonep/opensmalltalk-vm

[4]http://github.com/tesonep/pharo-benchmarks

Same priority with yielding:
We spawn three green threads doing processor intensive operations loop (calculating 100 times the factorial of 600). All threads have the same priority. After each iteration of the loop, the thread performs an explicit *yield* to make them run interlaced. The benchmark measures how much time it takes for all green threads to end.

Different priorities:
We spawn three green threads doing processor intensive operations in a close loop (calculating 100 times the factorial of 600). The threads have different priority to make them run serialized establishing the order through the use of priorities. The benchmark measures how much time it takes for all green threads to end.

Each iteration of the benchmark repeats the benchmark 20 times (problem size).

Analysis. The results shown in Table 2 show that the execution of processor intensive Smalltalk code in the interpreter is not affected by multi threaded architecture. The results show no noticeable impact in the execution of the benchmarks. These benchmarks validates that the proposed solution preserves the same execution qualities of the stock solution.

Benchmark	Stock VM (Avg.)	Modified VM (Avg.)
Same Priority without yielding	1784 ms	1784 ms
Same Priority with yielding	1786 ms	1800 ms
Different priorities	1783 ms	1784 ms

Table 2. Comparison of the Execution of Smalltalk code

8.3 Long callouts Impact

In this scenario we analyse the impact of performing long FFI calls. For this analysis we implemented a function in C that blocks the calling native thread during a second using a sleep. We built the following benchmarks:

Non Parallel callouts.
We used this benchmark as a baseline for the comparison. In this benchmark we execute two long callouts in sequence. There is no possibility of parallel execution of these calls. The benchmark measures how much time it takes for it to end.

Parallel callouts.
We spawn two green-threads, each of it is performing a long callout. In the GildaVM, each green-thread is affined to a different VM thread to force the native thread switching and the parallel execution. The benchmark measures how much time it takes for all green threads to end.

Analysis. The results shown in Table 3 expose the main benefit of the proposed solution. The execution time of parallel long blocking FFI calls is improved. The VM is able to execute other green threads while a blocking FFI call is executed. Depending on the load of green threads, the GildaVM executes faster than the stock VM. In our benchmarks, the improvement is around 80%. Moreover, there is a result that is not shown in the benchmark. The UI green thread is not blocked in the modified VM but blocked in the stock VM.

Benchmark	Stock VM (Avg.)	Modified VM (Avg.)
Non Parallel	2001 ms	2003 ms
Parallel	2006 ms	1216 ms

Table 3. Comparison of the Execution of Long callouts

8.4 Short Callouts Impact

In this scenario, we analyse the impact of performing short FFI callouts. For this analysis we implemented a function in C that returns immediately an integer constant. We built the following benchmarks:

Non Parallel callout.
We used this benchmark as a comparison baseline. In this benchmark we execute 200,000 short callouts in sequence. There is no possibility of parallel execution of these calls. The benchmark measures how much time it takes for it to end.

Parallel callout.
We spawn two green-threads, each of it performing 100,000 short callouts. In the GildaVM, each green-thread is affined to a different VM thread to force the native thread switching and the parallel execution. The benchmark measures how much time it takes for all green threads to end.

Analysis. The results shown in Table 4 present the main disadvantage of the proposed solution. The solution is unable to execute multiple parallel short FFI calls without introducing a noticeable performance penalty. In our results, we can observe that the execution of such parallel short FFI calls is 10 times slower that executing them in sequence. We believe this overhead is due to a VM thread switch that is forced very often because of the number of FFI callouts.

Benchmark	Stock VM (Avg.)	Modified VM (Avg.)
Non Parallel	63	88 ms
Parallel	60	995 ms

Table 4. Comparison of the Execution of Short callouts

8.5 VM Thread Switch Overhead

This benchmark measures the impact of having different green-threads running in different VM threads. Thus, this scenario is only valid for our proposed implementation and cannot be compared with the stock StackVM.

We executed three green-threads doing processor intensive operations loop (calculating 100 times the factorial of 600). All threads have the same priority. After each iteration of the loop, the thread performs an explicit *yield* producing the green-thread switch. Each green thread is affined with a different VM thread to make the yield operation force a VM thread switch too. Thus, the expected behaviour of this benchmark is that green threads run interlaced but each of them in a different VM thread. The benchmark measures how much time it takes for all green threads to end.

The results of this benchmarks are comparable with the results of the *Same priority with yielding*. This comparison illustrates the overhead of performing a VM thread switch when no FFI calls are involved, since one of the benchmarks performs a VM thread switch and the other does not.

Analysis. The results shown in Table 5 show a problem when performing VM thread switching. The proposed solution presents an overhead of around 5% when the interpreter is forced to execute shared between two VM Threads. These results show that the introduction of thread affinity in the execution of pure Smalltalk green-threads does not improve the execution, but also there exists a performance penalty on it.

Benchmark	Modified VM (Avg.)
Without Thread Switching	1800 ms
With Thread Switching	1881 ms

Table 5. Comparison of VM Thread Switch Overhead

8.6 Overview of the Results

The preliminary set of benchmarks and results in this section present an overview of the benefits and disadvantages of the proposed solution. Long callouts before blocking the vm-executive in the stock VM, can be cheaply run in parallel to the vm-executive in GildaVM. There is however a place for improvement to make the VM thread switch cheaper both in the absence of FFI calls and in their omnipresence.

9 Discussion and Future Perspectives

9.1 Non-Blocking I/O and Non-Blocking FFI

An interesting open question for further work is if non-blocking FFI support is enough to implement all non-blocking I/O such as sockets and files.

In the current implementation of the OpenSmalltalk-VM, non-blocking I/O is achieved through the use of ad-hoc written plugins. Implementing non-blocking I/O with non-Blocking FFI would represent a vast simplification of the VM implementation and allow the developers to implement the I/O operations at the image level.

9.2 VM vs OS Thread Scheduling

The correct scheduling of the threads has an impact on the performance of the execution. In GildaVM the scheduling of the VM threads is performed by the VM, thus the operating system does not take part in the scheduling. As future work we plan to explore the impact of operating system scheduling in this architecture.

9.3 Growing the ThreadPool on Demand

The appropriate size of the pool of VM threads is a matter of future analysis. A first approach could be to have a fixed number of threads. This number of threads could be a configuration in the image or a calculated value (*e.g.,* depending of the number of available cores in the machine) during the start of the VM. The fixed approach eases the predictability of the resources used by the VM. Although, it also limits the level of parallelism and produce starvation if all VM threads are busy but the VM has work to do.

An alternative approach is that the number of threads is dynamically calculated during the execution of the program depending on the level of multiprogramming. For example, a dynamic approach could spawn a new thread if all VM threads in the pool are busy while the VM has work to do. The dynamic approach improves the use of the processor as it guarantees that there is always a VM thread to run the VM. However, this approach requires a careful handling of the thread pool. The size of the pool should still be limited to not overwhelm the operating system with too many native threads. Moreover, this approach could also require a policy to purge the unused VM threads.

9.4 FFI Considerations: libraries are not simple nor similar

The main use of FFI is to integrate the language with existing libraries. Different libraries impose different restrictions in the way they should be used.

For example, SQLite[5] is not designed to accepts calls from different native threads in parallel. All callouts should be performed from the same native thread. The library is not designed to be used in multi-threading applications.

Another example is LibGit2[6]. This library support access from multiple threads but the resources accessed in the operations should be exclusive.

Finally UI Frameworks such as Cocoa[7] not only require that all callouts are performed in the same thread but also that they are performed from a specific one. In the case of Cocoa, this is the main UI thread. Moreover, Cocoa does not only require it, but this requirement is enforced in all the functions aborting the calling thread in such a case.

Although GildaVM allows to control what FFI callout executes in what VM thread thanks to thread affinity, it is still a responsibility of the developer to respect the requisites of the external library. We believe several programming models that provide support for these different scenarios can be built on top of this architecture, though they are matter of further work and analysis.

10 Related Work

Evolving Existing VMs. The standard implementations of Ruby and Python support green threads. They rely on interpreter locks (GIL). Each of the green threads is assigned to a single native thread sharing the same instance of the interpreter. And each native thread only runs a single green thread producing a one-to-one mapping. The green thread schedulling is performed by the native thread support library or operating system [1]. They provide non-blocking FFI through the execution of the program in different green threads (resulting in different native threads running). A FFI call releases the GIL and allows other thread to run the interpreter. This model presents the same advantages and disadvantages than GildaVM. However, the execution of process intensive green threads is penalized as they require thread switching.

Swaine et al. present a solution to modify an existing VM to support incremental parallelization [13]. They also starts from a existing VM without parallel execution and with existing programs. They implement parallel execution through the analysis and rewriting of the operations that does not allow them to have a full parallel solution. They implement their solution incrementally without performing a whole rewrite of the VM nor the programs. Their solution does not use a global interpreter lock.

Daloze et al. present a compressive study of the changes required to modify an existing language to run in a multi-threaded VM [2]. They show alternatives to these changes and the impact of rewriting the programs and libraries running on top of the existing VM.

Approaches Requiring Full Rewrites. Java VMs [5] and the .NET Command Language Runtime (CLR) [8] provide support for multithreading through the use of multiple instances of the interpreter. They implement a N-to-M mapping between green threads and native threads. Each of the FFI calls is executed in a different native thread allowing to have concurrent non-blocking FFI calls. This design requires the construction of a VM designed to support multithreading.

Jython [4], JRuby [9] and IronPython [3] executes on top of existing multithreading VMs (Java Virtual Machine and .NET CLR). They are able to execute multiples instances of the interpreter, each of them running in an independent host-language thread. These solutions do not use a GIL and are able to run green-threads in parallel. Although, it requires to implement a multithreading interpreter. Also, they serialize the access to the object memory and other data structures used by the interpreters.

Rubinius [10] is an implementation of Ruby that runs the interpreter in several native threads. This solution does not implement a GIL, but it requires the reimplementation of the existing VM to safely run in as a multithreading interpreter.

PyPy-STM [6] emulates the GIL semantics and enables scaling; through the use of multiple interpreters synchronized through transactional memory. However, this solution incurs a significant overhead on single-threaded performance as single-threaded execution is still using the transactional memory.

11 Conclusion

This paper presents GildaVM, a multi-threaded VM architecture to provide non-blocking I/O on top of the CogVM. This work is a description and analysis of the original design of David Simmons and the initial implementation of Eliot Miranda; without their work this architecture would have not existed.

This architecture is intended to be applied in an existing implementation of a single threaded VM. In GildaVM a single thread is executing the VM at a time while allowing other threads to execute non-blocking I/O in parallel. This is done through the use of a Global Interpreter Lock and a specific strategy to handle the scheduling of VM threads.

We present not only the described solution, but also a prototype implementation of this solution on top of the OpenSmalltalk-VM. To evaluate the architecture we performed a series of benchmarks showing different scenarios where the solution over performs the stock VM and some others where it under performs it. Preliminary results show that it is possible to take advantage of the non-blocking I/O without requiring a major rewrite of the VM nor existing applications. We showed that the time spent on an FFI callout affects directly the performance of the solution. Long

[5] https://www.sqlite.org/index.html
[6] https://libgit2.org/
[7] https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/OSX_Technology_Overview/CocoaApplicationLayer/CocoaApplicationLayer.html

running FFI callouts are heavily improved, whereas short callout are negatively affected. This results open the door to further analysis and improvements.

Acknowledgments

This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council, CPER Nord-Pas de Calais/FEDER DATA Advanced data science and technologies 2015-2020. The work is supported by I-Site ERC-Generator Multi project 2018-2022. We gratefully acknowledge the financial support of the Métropole Européenne de Lille.

References

[1] Shannon Behrens. 2008. Concurrency and Python. http://www.drdobbs.com/open-source/concurrency-and-python/206103078. http://www.drdobbs.com/open-source/concurrency-and-python/206103078

[2] Benoit Daloze, Arie Tal, Stefan Marr, Hanspeter Mössenböck, and Erez Petrank. 2018. Parallelization of Dynamic Languages: Synchronizing Built-in Collections. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 108 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276478

[3] Jim Hugunin et al. 2004. IronPython: A fast Python implementation for .NET and Mono. In *PyCON 2004 International Python Conference*, Vol. 8.

[4] Josh Juneau, Jim Baker, Frank Wierzbicki, Leo Soto, and Victor Ng. 2010. *The Definitive Guide to Jython: Python for the Java Platform.* Apress.

[5] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification.* Pearson Education.

[6] Remigius Meier, Armin Rigo, and Thomas R. Gross. 2016. Parallel Virtual Machines with RPython. In *Proceedings of the 12th Symposium on Dynamic Languages (DLS 2016).* ACM, New York, NY, USA, 48–59. https://doi.org/10.1145/2989225.2989233

[7] Eliot Miranda, Clément Béra, Elisa Gonzalez Boix, and Dan Ingalls. 2018. Two decades of smalltalk VM development: live VM development through simulation tools. In *Proceedings of the 10th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages.* ACM, 57–66.

[8] netCLR [n.d.]. .NET Common Language Runtime (CLR) overview. https://docs.microsoft.com/en-us/dotnet/standard/clr. https://docs.microsoft.com/en-us/dotnet/standard/clr

[9] Charles Nutter, Thomas Enebo, Ola Bini, Nick Sieger, et al. 2018. JRuby–The Ruby Programming Language on the JVM.

[10] Evan Phoenix, Brian Shirai, Ryan Davis, Dirkjan Bussink, et al. 2018. Rubinius–An Implementation of Ruby Using the Smalltalk-80 VM Design.

[11] Minyoung Sung, Soyoung Kim, Sangsoo Park, Naehyuck Chang, and Heonshik Shin. 2002. Comparative performance evaluation of Java threads for embedded applications: Linux Thread vs. Green Thread. *Information processing letters* 84, 4 (2002), 221–225.

[12] Java On Solaris SunSoft. [n.d.]. 2.6-A White Paper, September 1997.

[13] James Swaine, Kevin Tew, Peter Dinda, Robert Bruce Findler, and Matthew Flatt. 2010. Back to the Futures: Incremental Parallelization of Existing Sequential Runtime Systems. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10).* ACM, New York, NY, USA, 583–597. https://doi.org/10.1145/1869459.1869507