Accepted to IWST2016

A low Overhead Per Object Write Barrier for the Cog VM

Clément Béra
RMOD - INRIA Lille Nord Europe
clement.bera@inria.fr

Abstract

In several Smalltalk implementations, a program can mark any object as read-only (unfortunately incorrectly sometimes miscalled immutable). Such read-only objects cannot be mutated unless the program explicitly revert them to a writable state. This feature, called write barrier, may induce noticeable overhead if not implemented carefully, both in memory footprint and execution time. In this paper I discuss the recent addition of the write barrier in the Cog virtual machine and the support introduced in the Pharo 6 image. I detail specific aspects of the implementation that allows, according to multiple evaluations presented in the paper, to have such a feature with little to no overhead.

Keywords Language Virtual Machine, Just-in-Time Compilation, Interpreter, Write Barrier, Store Check.

1. Introduction

Read-only objects are frequently used in several Smalltalk dialects to ensure the unchangeable state of runtime objects such as compiled methods' literals and in the context of object modification tracker frameworks such as Gem Builder for Smalltalk[1] (GBS). The Cog virtual machine (VM) [11] is the most widely used open source Smalltalk virtual machine, with multiple Smalltalk clients: Pharo [2], Squeak [8] and Cuis [3]. Unfortunately, the Cog VM did not support read-only objects. I decided to introduce such a feature, with the help and advises of the lead Cog VM architect, Eliot Miranda.

In this paper, I discuss the design decisions behind the write barrier and the implementation in both the Cog VM and the Pharo 6 implementation. Other Smalltalk clients running on the Cog VM have or can have a similar implementation, but each Smalltalk dialect has some specificities so I needed to pick a specific one to show the production implementation.

Conceptually, having read-only objects requires each store into an object to have an extra check to fail the store if the object mutated is read-only. An extra check induces extra memory and execution time overhead as additional machine instructions are required to perform the check. In addition, the memory representation of the object needs to be adapted to encode the read-only property of the object. The main challenge in the write barrier implementation is to reduce the overhead, both in term of memory footprint and execution time, as much as possible.

In most VMs for high-level object-oriented languages, each store into an object has already multiple checks for the garbage collector (GC) write barrier [9, 10]. In the implementation sections, I detail the most critical part: how the machine code generated by the JIT shares portion of machine code between the read-only check and the existing GC write barrier to limit the overhead.

2. Problem

In this section I specify what I mean by read-only object write barrier, discuss the terminology used, then describe briefly some use-cases and precise the problem statement.

2.1 Specification

The feature wanted, the write barrier, allows a Smalltalk program to mark or unmark any non-immediate object[2] as read-only at any time. Any write into a read-only object is intercepted **before** the object is mutated and it should be possible to handle the mutation failure at the language level.

2.2 Terminology

This feature is called in some other Smalltalk, especially VisualWorks, *immutability*. Using the term immutability was

[1] GBS is a tool maintained and evolve by GemTalk™ Systems allowing applications written in any Smalltalk dialects to communicate with the Gemstone persistence layer.

[2] Non-immediate objects have a memory zone holding the object's state and references to the object are implemented through pointers to that memory zone. On the contrary, immediate objects are not present in memory and their state is directly encoded in the pointer referencing the immediate object. The best example of immediate objects are 31-bit signed integers, also called SmallIntegers.

contested by the Smalltalk community. Indeed, in object-oriented and functional programming, for example in Racket [4], an immutable object is an object which state cannot be modified after it is created. Therefore, in our case, as the programmer can revert the read-only state of an object to writable state at any time, the immutability definition does not apply. This is why in Pharo and in this paper the feature is called write barrier and not immutability.

2.3 Use-cases

There are multiple use-cases for read-only objects. I detail here the two most common ones.

Modification tracker. The most popular use-case is the ability to track the mutations done to a specific object. In this case, the tracked object is marked as read-only. Each mutation of the tracked object triggers Smalltalk code specified by the programmer to do something about the mutation, for example, logging. Then, the modification tracker framework temporarily makes the object writable, performs the mutation, and mark back the object as read-only to resume the execution while still tracking the object's mutations. This modification tracking ability is for example used in GBS, a framework to deeply integrate a Smalltalk application with the gemstone persistence layer.

Core read-only objects. Another interesting read-only object use-case is the ability to mark runtime objects such as compiled methods' literals as read-only. Having the literals read-only allows compilers to make stronger assumptions to allows more aggressive optimisations and forbid inconsistent modification of literals avoiding hard-to-debug issues.

2.4 Problem: limiting the overhead

The problem statement is as follows:

Is it possible to mark object as read-only, forbidding any mutations and letting Smalltalk code handle the mutation failures, with little to no overhead in term of memory footprint and execution time ?

To solve this problem, I chose to extent the virtual machine. Indeed, I believe the solutions provided at image level either induce an important overhead or are not thorough enough. For example, it is possible using reflective APIs to activate any primitive operation on any object in the system. Some primitive operations, such as the at:put: primitive, mutate objects. Detecting such mutations is very difficult, likely even impossible, without VM support.

The solution was implemented in three steps:

- Enhancing the memory representation of objects to be able to encode their read-only state.
- Adding support in the execution engine to forbid read-only objects mutations.
- Adding support in the Pharo image to be able to use the new feature.

Memory representation of objects. To support read-only object, the first thing is to change the memory representation of objects to be able to mark them as read-only. To do so, each object needs a specific memory location to encode the state: is the object read-only or not ? As other Smalltalk dialects, a bit seems appropriate as there are only two possible cases. I detail later in the paper the position of this bit. As the bit is in the object's header and immediate objects have no header, immediate objects cannot be read-only.

The VM can directly access the object's state, but the Smalltalk code can't. So I added two convenient primitives in Pharo to access the bit state. One primitive tells if an object is read-only or not, the other sets the object as read-only or writable.

Execution support. The objects are mutated in two main ways in the current virtual machine:

- By storing into one of their instance variable field (bytecode instruction).
- By performing a primitive operation that mutates object, such as at:put:.

In the paper I omit explicitly another case, the literal variable stores. For the execution engine, a literal variable store is an instance variable store mutating the second field of an object specified in the literal frame of the method. Hence, the discussions related to instance variable store also apply to literal variable stores.

In the execution engine, the instance variable store code was changed to fail if the mutated object is read-only. If that happens, a callback is triggered in the image to inform the program that an attempt to assign a value to a read-only object was made, and once the call-back returns, the execution resumes after the store. The callback is triggered instead of the store, hence if one wants the store to be performed one needs to do it explicitly in the callback.

The code of the primitives mutating objects was rewritten to fail the primitive if they mutate a read-only object.

Limitations. While implementing the solution, I realized it is really difficult to have a few specific objects read-only.

The first problem is related to process scheduling. At each interrupt point, the execution may switch to another process. Switching from a process to another process implies multiple process scheduling objects mutations, whereas the execution state (in the middle of a process switch) is not in a state where a call-back can be safely triggered in the image.

The second issue lies with context objects. Contexts represent method and closure activations. They are handled very specifically in the virtual machine for performance and they are mutated all the time during normal execution: any bytecode operation requires at least to mutate the active context program counter.

Lastly, by design, the VM assumes that temp vectors (data structure used to store closure enclosing context information), are never read-only.

To solve these problems, I specify here a list of objects that cannot be marked as read-only. Any attempt to mark those objects as read-only from Smalltalk will fail. These objects are:

- Context instances
- All objects related to process scheduling:
  - the global variable Processor
  - the array of linked lists of processes (Processor instance variable)
  - ProcessLinkedList instances
  - Process instances
  - Semaphore instances

In addition to those objects, specific objects used directly by the runtime cannot be marked as read-only. One example is temp vectors, which are used to hold block closures remote variable values, but also objects internal to the VM as for example the class table.

I discuss in future work how one may be able to bypass some of those limitations.

3. Image API design and implementation

In this section I introduce the APIs added in the image to support read-only objects. I do not discuss the in-image implementation of features using the write barrier such as an object modification tracker. I discuss only the interface between the virtual machine and the image allowing one to use the write barrier and to build features such as an object modification tracker.

3.1 Core write barrier primitives

Two main primitives were added into the Object class:

- Object»isReadOnlyObject
- Object»setIsReadOnlyObject:

Object»isReadOnlyObject. The primitive answers if the receiver read-only. The primitive cannot fail on a VM supporting the write barrier. The primitive method code is available in Figure 1. The Pharo 6 alpha version is available with additional comments, omitted in the paper.

```
Object>>isReadOnlyObject
    <primitive: 163>
    ^self primitiveFailed
```

Figure 1. Object»isReadOnlyObject primitive

Object»setIsReadOnlyObject: This second primitive marks the receiver as being read-only or writable, depending on the boolean parameter. The primitive method code is available in Figure 2.

The design of the primitive in Pharo can be questionable: why having a single method with a boolean argument instead of two methods ? The answer is simple, the number of primitives has to be kept as small as possible to keep the VM implementation simple, hence sharing the same primitive number for these two operations seemed the right thing to do. However, for convenience, two helper methods were added, Object»beWritableObject and Object»beReadOnlyObject, only calling the primitive with the corresponding boolean parameter, as shown in Figure 3.

```
Object>>setIsReadOnlyObject: aBoolean
    <primitive: 164 error: ec>
    ^ self primitiveFailed
```

Figure 2. Object»setIsReadOnlyObject: primitive

```
Object>>beWritableObject
    ^ self setIsReadOnlyObject: false

Object>>beReadOnlyObject
    ^ self setIsReadOnlyObject: true
```

Figure 3. Helper methods

3.2 Primitive failure

As stated in the Section 2.4, primitive operations mutating objects should fail if they attempt to mutate a read-only object. Two modifications are required to support the write barrier.

Image-side. Each primitive failure fall-back code needs to be edited to raise an appropriate error if it failed due to the write barrier. For example, in the case of the primitive for at:put:, the in-image fall-back code should check if the receiver is read-only, raising an appropriate error (for example 'object cannot be modified') instead of 'Instances of Objects are not indexable'. Unfortunately, this part has not, at the moment where I write the paper, been integrated in the Pharo 6 alpha.

VM-side. To help the programmer understanding why a primitive fails, the virtual machine is able to provide error messages. This is done by adding the keyword error: in the primitive pragma. For example, in Figure 2, the primitive pragma has the error: keyword, hence if the primitive fails the temporary variable ec is going to hold an error message. The special objects array defines a list of error messages that can be answered by the VM. This list now defines the message 'no modification' which is raised when a primitive fails due to the write barrier.

3.3 Instance variable store

As instance variable stores are encoded directly in the bytecode and not through message sends as primitives, they can't simply just fail or the VM state would be inconsistent. The easiest way to handle this case was to add a VM call-back to

be performed when a store fails. An infrastructure for such call-backs is already available and is used for example for doesNotUnderstand:.

However, this VM-call back is more difficult to implement. Our specification requires the read-only failure to resume execution, once the call-back is done, after the variable store. The problem is that the VM does not expect any value to be pushed on stack after a variable store.

If we take the example of doesNotUnderstand:, the call-back is triggered during a message send. In Smalltalk, each message send is expected to return a value, hence the value returned by the doesNotUnderstand: method activation is pushed on stack instead of the regular message send returned value.

In the read-only call-back case, the VM does not expects any value to be pushed on stack after a variable store. Therefore, I needed to design a call-back that does not answer any value. This is currently possible in Pharo by modifying the active process. The cannotAssign:withIndex: call-back was designed this way. After handling the mutation failure, the call-back does not return any value as the Smalltalk code on Figure 4 shows. The comment "CAN'T REACH", indicates that the execution flow cannot reach that part of the code.

```
attemptToAssign: value withIndex: index
    | process |

    "Handle here the mutation failure. Code omitted."

    "Process modified to return no value"
    process := Processor activeProcess.
    [  | sender |
      sender := process suspendedContext sender.
      process suspendedContext: sender.
    ] forkAt: Processor activePriority + 1.
    Processor yield.
    "CAN'T REACH"
```

Figure 4. Pharo call-back implementation

This implementation is a temporary solution as it cannot work with processes already at max priority. I am considering alternatives, such as a new primitive or a bytecode instruction performing returns non returning any value.

3.4 Other in-image features

Support flags. The Cog VM provides to the Smalltalk clients a set of parameters. A new parameter was added, answering if the VM currently used supports read-only objects. In the case of Pharo, it is now possible to run Smalltalk vm supportsWriteBarrier to know if the feature is enabled.

Mirror primitives. The Cog VM, as well as several other Smalltalk VMs, supports having objects with a class not inheriting from Object. Such objects are typically used for proxies. Sending messages to this kind of objects can be a problem: the object may not be able to answer the message nor to answer the doesNotUnderstand: message, leading to a VM crash. This kind of problem usually happens when the programmer attempts to debug a program with proxy objects: in this case, the proxies understand all the messages required for the application, but do not understand the messages required for debugging.

To avoid VM crashes, proxies are debugged through mirror primitives. For example, the primitive instVarAt: answers the value of an instance variable of an object. This primitive exists in two variants:

- instVarAt:: Answers an instance variable of the receiver.
- object:instVarAt:: Answers an instance variable of the object passed as first argument.

The second version, ignoring the receiver entirely, is called a mirror primitive. It is able to perform a primitive operation on an object (in this case, the first argument), without requiring the object to be able to understand a message. In the context of the write barrier, the two primitives isReadOnlyObject and setIsReadOnlyObject: are also available as mirror primitives (the primitive number is shared), in the form of object:isReadOnlyObject and object:setIsReadOnlyObject:. This way, it is possible to modify and read the read-only property of proxy objects.

4. VM implementation

The VM implementation is split in three subsections, the object representation, the interpreter and the JIT compiler changes.

4.1 Object representation

Each non-immediate object is represented in memory with an object header, describing the object, and multiple fields, depending on the object's layout. Several bits in the object header are unused and a single bit was reserved by design in the Spur Memory Manager [12] for the write barrier. I used this bit to mark the read-only state of an object, as shown on Figure 5.

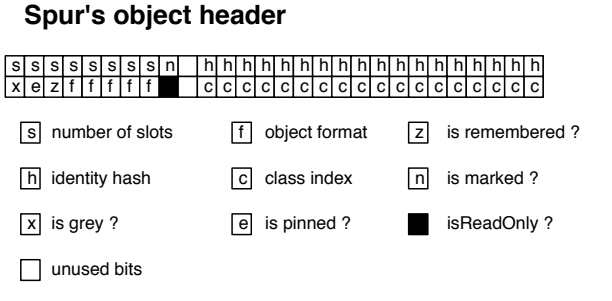

Figure 5. Object header memory representation in Spur

4.2 Interpreter implementation

4.2.1 Primitives.

I needed to add support for primitives to fail if they attempt to mutate a read-only object.

Many primitives can already fail. For example, <primitive:1>, the addition between two small integers, fails if the argument is not a small integer. Hence, I needed to edit all the primitives mutating objects to first check if the object mutated is read-only, and fail the primitive if this happens. This was quite tedious as I had to go through all the primitive table and check manually for each primitive if the code mutates an object. This task was simplified by the limitations: as stated in Section 2.4, several objects can't be read-only, so the primitives related to process scheduling and context accessing didn't need to be changed.

4.2.2 Instance variable stores.

I needed to update the interpretation of instance variable stores to fail and trigger the cannotAssign:withIndex: callback if the object mutated is read-only. Some aspects are challenging.

Interpreter compilation and emulation. The interpreter code is written in Slang, a DSL to write virtual machines written using the Smalltalk syntax to be able to emulate the execution on top of the Smalltalk VM. For the production VM, Slang is compiled to C with the GNU extensions, which is then compiled to machine code. The C-language extensions are critical for performance as an interpreter has a very different behavior than mainstream C application.

C extension constraints. Most of the interpreter code is compiled in a single C function. That function uses the C-extensions to fix specific values to registers, such as the Smalltalk stack pointer, frame pointer and instruction pointer. The execution jumps quickly from the interpretation of a bytecode to the next one using threaded jumps the the new bytecode execution code address. If the interpreter needs to call another function, it needs to save the fixed registers manually and restore them upon function return if they are going to be used.

Challenges met. This specificity is sometimes difficult to handle because the execution flow in the extended C code is non trivial to reproduce on the simulation engine which runs on top of the Smalltalk VM. In addition, one has to be very careful if the interpreter calls a function non-inlined during Slang to C compilation in the main interpreter function to correctly maintain the registers state.

Conclusion. To implement the read-only write barrier, both the simulation engine used for debugging and the extended C code needs to have the same behavior according to the new specifications.

4.3 JIT compiler support

4.3.1 Primitives.

As for the interpreter, I needed to update the JIT to compile primitive operations according to the new specification.

Primitives redefined in the JIT. The interpreter primitives are normally written in Slang and are compiled to machine code as the rest of the VM. As the compilation is done through the optimizing C compiler, the primitives performance is usually very good. However, calling C code from a machine code Smalltalk method has a cost: the runtime needs to switch from the Smalltalk machine code runtime to the C runtime, execute the primitive, then switch back to the Smalltalk machine code runtime. This cost can be significant on very frequently used primitives, as for example the addition between two small integers. For this purpose, a set of primitives are redefined in the JIT register transfer language (RTL)[3] and are compiled to machine code with the methods holding the corresponding primitive number.

For the purpose of this paper, we will consider that there are two kinds of primitives:

- Frequently called primitives: They are redefined in the JIT's RTL.
- Rarely called primitives: When a method with such primitive is compiled to machine code, the machine code switches to the C runtime and then calls the interpreter primitive code.

All the existing interpreter primitive code was updated to fail for read-only objects. However, primitives redefined in the JIT's RTL needs to be updated to correctly fail if they mutate a read-only object.

Updating at:put:. Fortunately, only two primitives considered as frequently called and therefore defined in the JIT's RTL mutate objects. One of them is the primitive at:put: while the other one is a specific version of at:put: for strings. I updated these two primitives to generate machine code failing if the receiver is read-only.

4.3.2 Instance variable stores.

With the write barrier, the machine code generated for instance variable stores require an extra check to fail if the object mutated is read-only.

Studied case. The JIT compiles to machine code the stores differently depending on multiple constraints. For example, the compilation is different depending on which register is live or not when compiling the store. In this subsection, I will only discuss the most common cases, a generic instance variable store of the first instance variable of an object that we will call a *lambda store*. Other cases are handled in a similar way.

[3] A register transfer language (RTL) is a kind of intermediate representation that is very close to assembly language, similar to those used by compilers. RTLs describe data flow at the register-transfer level of an architecture.

GC store check. Before the write barrier implementation, a lambda store needs to change in memory the value of the instance variable and to deal with the GC write barrier. Currently, the GC requires each object from old space referencing a young object to be in the remembered table. Hence, each store can require the VM to add an entry in the remembered table.

Each store generates machine code to check if the object needs to be added in the remembered table. If this is the case, the VM calls a trampoline[4] which saves the registers state, call the interpreter function adding the object in the remembered table, restores the registers and resumes execution. The existing machine code generated for a lambda store is shown on Figure 6.

x86 Assembly	Meaning
movl -12(%ebp), %edx	Load the receiver in %edx.
popl %edi	Load the value to store in %edi.
movl %edi, %ds:0x8(%edx)	Perform the store in the first instance variable using both registers (%edx and %edi)
testl 0x00000003, %edi	If the value to store is immediate, jump after the store check.
jnz after_store	
movl 0x00040088, %eax	Jump after the store check if the receiver is young: compare the young object space limit with receiver address
cmpl %eax, %edx	
jb after_store	
cmpl %eax, %edi	If the value to store is an old object, jump after the store check.
jnb after_store	
movzbl %ds:0x3(%edx), %eax	If the receiver is already in the remembered table, jump after the store check.
testb 0x20, %al	
jnz after_store	
call store_check_trampoline	Calls the store check trampoline.
after_store:	Code following the store.

Figure 6. Vanilla lambda store

Naive read-only check implementation. I needed to add the read-only check. My first idea was to add it at the beginning of the store, once the receiver and the value to store are loaded in register. As shown on Figure 7, I added a branch which ensures that the receiver is writable and calls a trampoline to trigger the cannotAssign:withIndex: call-back if it's not the case. This solution implied the creation of a single new trampoline that calls an interpreter function when the receiver is read-only to call in the language the cannotAssign:withIndex: call-back.

[4] A trampoline is a specific machine code routine switching from the assembly code runtime to the C runtime.

x86 Assembly	Meaning
movl %ds:(%edx), %eax	If the receiver is writable, jump to the store.
testl 0x00800000, %eax	
jz begin_store	
call cannot_assign_trampoline	Calls the read-only failure trampoline.
movl -12(%ebp), %edx	Restore the receiver (to keep its register live) and jump after the store.
jmp after_store	
begin_store:	Next instruction is the first store instruction.

Figure 7. Considered read-only check

This solution implied quite some overhead because the machine code needed to take an extra branch on the common path and because many new machine instructions were added per instance variable store.

Efficient read-only check. I then built a second solution, where a single per-store trampoline is shared between the GC and the read-only write barrier, as shown on Figure 8. As the instruction to call the trampoline is the one that takes the more bytes, the general idea was to avoid most of the overhead by having single call. I created new trampolines that are able to deal with both the case of the GC and the read-only write barrier. In this new version, the machine code tests first if the object is read-only, and if so, directly jumps to the shared trampoline.

New trampolines. To be able to share the trampoline without adding too many instructions, as the trampoline is rarely taken, the trampoline duplicates the read-only check. The normal execution flow checks if the object is read-only and jumps to the trampoline if it is the case. In the trampoline, the VM does not know any more if the trampoline was reached for a read-only mutation failure or the GC write barrier. Hence, the trampoline tests again if the object mutated is read-only and calls the correct interpreter method to handle either case.

Specialized trampolines for common indexes. In the case of a read-only mutation failure, to perform the call-back, the VM has to know what is the variable index of the object. In the case of a lambda store, we said the instance variable was the first instance variable, so in a 0-based array, the variable index is 0. The problem is that to perform the trampoline call the variable index needs to be passed as a parameter, requiring extra machine instructions per-store (In the Cog VM all trampoline parameters are passed by registers). To avoid the extra instructions, the trampoline is duplicated. A fixed number of trampolines based on a VM setting are created, currently 6. Each of the most common variable indexes (0 to 4) can call a specialized version of the trampoline for the

given index (so it is not required to pass the variable index by parameter in those common cases), and other variable indexes, less common, call the generic trampoline passing by parameter the variable index.

Register liveness. As the read-only failure trampoline creates a new stack frame for the cannotAssign:withIndex: call-back, the registers cannot remain live across the trampoline. I decided to keep the receiver live if it was already live by injecting the corresponding machine code after the store if the receiver was live before, as a live receiver is the most critical for performance. Hence, only the receiver can remain live in a register across the read-only write barrier trampoline call.

x86 Assembly	Meaning
movl -12(%ebp), %edx	Load the receiver in %edx.
popl %ecx	Load the value to store in %ecx.
movl %ds:(%edx), %eax	If the receiver is read-only, jump to the store trampoline.
testl 0x00800000, %eax	
jnz store_trampoline	
movl %ecx, %ds:0x8(%edx)	Perform the store in the first instance variable using both registers (%edx and %ecx)
testb 0x03, %cl	If the value to store is immediate, jump after the store check.
jnz after_store	
movl 0x00040088, %eax	If the receiver is a young object, jump after the store check.
cmpl %eax, %edx	
jb after_store	
cmpl %eax, %ecx	If the value to store is an old object, jump after the store check.
jnb after_store	
movzbl %ds:0x3(%edx), %eax	If the receiver is already in the remembered table, jump after the store check.
testb 0x20, %al	
jnz after_store	
store_trampoline:	Calls the store check trampoline.
call store_trampoline	
movl -12(%ebp), %edx	Restore the receiver (to keep its register live).
after_store:	Code following the store.

Figure 8. Production lambda store with the write barrier

Debugging support. Without the write barrier, literal and instance variable stores are not interrupt points. The debugger cannot be opened at this program counter and processes can't switch on variable stores. With the write barrier, the cannotAssign:withIndex: call-back can create new stack frame. If one of the method called opens a debugger, the programmer needs to be able to debug the context with the cannotAssign:withIndex: call-back and the sender of this context. I therefore needed to extend the machine code method metadata to be able to debug methods interrupted on stores.

Compilation. The write barrier was introduced as a compilation setting in the Cog virtual machine. By design, two choices were at hand, having the write barrier as a Slang to C compiler setting or as a C to machine code compiler setting. I firstly made it as a Slang compiler setting, but it was inconvenient as the build repository hierarchy needed to be duplicated by two to support the write barrier in all the builds. The write barrier was lastly changed to be a C compiler setting. The C compilation has now an extra setting, the (misleading) -DIMMUTABILITY=1 flag, to compile the VM with the write barrier.

5. Evaluation

I evaluate firstly the memory overhead of the feature, then the execution time overhead.

5.1 Memory overhead

Object representation. As described in Section 2.4, each object requires a single bit to mark their read-only state. As all the objects need to be 64bits aligned in the spur memory manager and one bit had already been reserved for the write barrier, in practice there is no memory overhead at all.

Machine code memory footprint evaluation. The size of the machine code representation of methods matters a lot in the Cog VM. In fact, the VM keeps a fixed-sized executable zone holding all the machine code of methods. This zone is allocated at start-up depending on an in-image setting, which is usually between 1 and 2 Mb, but can be any value.

The size of the machine code matters because:

- When installing a new method, the VM needs to scan all the machine code zone and flush all the caches related to the new method selector. The machine code zone has to have a limited size to avoid for this scan to be too long.
- Internally, the processor maps the frequently executed machine code to the cpu instruction cache. Having a limited machine code zone allows the cpu to have more instruction cache hits and improve the VM performance.
- As machine code versions of methods directly refer to objects (the literals are compiled inlined in the machine code), the GC needs to scan the machine code zone to mark referenced objects. The size of the machine code zone matters as the GC needs to read the metadata associated with each machine code method to locate the objects referenced, so the bigger the zone is, the longer the GC takes.
- As the machine code zone has a fixed size, if the methods are compiled in a smaller amount of machine code, the VM can fit more methods in the machine code zone before requiring a machine code zone garbage collection.

I evaluate the machine code size growth firstly globally, then locally.

Machine code zone (globally). As shown on Figure 9, just after start-up, the machine code zone occupied is 1.52% bigger with the write barrier that without. The overhead is there for multiple reasons:

- Each instance and literal variable store is compiled in more machine instructions for the read-only write barrier.
- The at:put: primitives are compiled with more instructions.
- Additional trampolines are required at the beginning of the machine code zone for the write barriers failure.

	Machine code zone size after start-up (hex)
Vanilla	91C00
Write Barrier	93F80

Figure 9. Machine code zone size

Locally: trampolines. When comparing the first available address between the VM with and without the read-barrier, one notices an overhead of 400 bytes, which corresponds to the size of the new trampolines.

Locally: per-store overhead. In the case of a lambda store, the most common, the store needs 12 extra bytes per store to encode the extra machine instructions for the read-only check. The overhead may vary slightly as the number of Nops required for alignment between methods may change if the number of bytes of the method changes.

Locally: at:put:. Each at:put: primitive is 16 bytes bigger with the write barrier.

Comments. The main concern in our case is the number of literal and instance variable stores. The number of trampolines is fixed during execution and there are at most two at:put: primitives. Hence, only the number of stores can seriously impact the memory foot print. As the global evaluation shown, stores seems to be pretty rare as the overall memory overhead is evaluated at 1.52%.

5.2 Execution time

Benchmarks. I evaluated the difference in performance using the Games benchmarks [7] that is normally used for VM performance evaluation. Even in benchmarks with intensive instance variable stores, such as the binary tree benchmark, the execution overhead was within the cpu noise (so little that it could not be evaluated). I believe there is some overhead in such benchmarks, but the overhead is under 1% of execution time and I did not achieve to measure it.

Building a pathological case. To see the performance difference, I built a micro-benchmark around a pathological case doing almost only instance variable store.

```
MicroBench>>#setImmediate: imm nonImmediate: nonImm
    "Immediate constant store"
    iv1 := 1.
    "Non Immediate constant store"
    iv2 := #foo.
    "Immediate store"
    iv3 := imm.
    "Non Immediate store"
    iv4 := nonImm.

DoIt
    | guineaPig |
    guineaPig := MicroBench new.
    [guineaPig setImmediate: 2 nonImmediate: #bar ] bench
```

	Time to run pathological bench
Vanilla	11.5 $\pm$3 nanoseconds per run
Write Barrier	13.6 $\pm$2 nanoseconds per run

Figure 10. Pathological benchmark code and results

In this pathological case, as shown on Figure 10, one notices a 18.2% performance overhead. However, the binary tree benchmark, which was larger, calls extensively a similar method (see Figure 11) and does not show any significant overhead. It is therefore unclear if this result means anything on real applications.

```
ShootoutTreeNode>>left: leftChild right: rightChild item: anItem
    left := leftChild.
    right := rightChild.
    item := anItem.
```

Figure 11. Binary tree setter method

I profiled the pathological case and realized the performance overhead was mostly due to the stack frame creation. Indeed, instance variable stores do not require a stack frame without the write barrier, but they do with the write barrier to be able to perform the cannotAssign:withIndex: call-back. Different solutions are considered for this problem, as discussed in the future work section.

6. Related work

Immutability. Other programming languages such as Ada, C++, Java, Perl, Python, Javascript, Racket or Scala support immutable objects. In those cases, an immutable object is an object whose state cannot be modified after it is created. It differs from our approach where at any time, the program can mark or unmark an object as read-only. In the context of Smalltalk where most features are reflexive, it seems the

right thing to allow an object to be able to change from immutable to mutable state, and the other way around, using reflexive APIs.

Garbage collector write barrier. Other people have implemented write barriers in the machine code for efficient garbage collection [9, 10]. Tracing generational GCs require the runtime to maintain a specific invariant: objects referencing other objects from a younger generation need to be remembered. This way, the runtime can mark a generation of objects without scanning older generations, leading to better performance. In this kind of GCs, when an object is stored into an older object, some actions may be taken by the runtime to remember the old object thanks to a write barrier.

In addition, many modern GCs are also incremental to limit the impact of garbage collection pauses for the application. Incremental GCs may require additional invariants. For example, in a tri-color marking garbage collector [1], a new invariant is that black and white objects cannot reference each others.

In the case of the Cog VM, the runtime now provides both a write barrier for the generational GC and for read-only objects. As discussed in Section 4.3.2, part of the machine code is shared between both write barriers to limit the overhead. As of today, the Cog VM does not feature an incremental GC hence no write barrier is required for this purpose.

High level modification tracker tools. The main use-case of the write barrier is the implementation of object modification trackers. Others implementation of objects modification trackers are available. The most popular nowadays are the ones made with the Reflectivity framework [5]. On the contrary to our approach where the overhead is close to zero, the other approaches available have a significant overhead as they need to execute additional bytecodes.

Other Smalltalks. Other Smalltalk dialects, such as VisualWorks Smalltalk and the HPS VM (High Performance Smalltalk Virtual Machine) [6], have a similar features. In the case of VisualWorks, as the VM is a pure-JIT VM (there is no interpreter), the implementation does not require the cannotAssign:withIndex: call-back to return no value (the machine code generated has a specific execution path to take care of it).

7. Future Work

I discuss in this section multiple performance improvement and features that would be nice in the future release of the Cog VM.

7.1 Performance improvements

Stack frame mapping and trampolines. While profiling code in the VM to look for methods getting slower with the write barrier, it came to light that one could optimize multiple trampolines in the JIT (related and unrelated to read-only objects). Indeed, trampolines such as cannotAssign:withIndex: or mustBeBoolean add strong pressure on register allocation[5] while they are taken infrequently.

It would be possible to convert stack frames triggering those trampolines from machine code frame to bytecode interpreter frames lazily, only when one of the unfrequent trampoline is taken. This way, infrequent execution paths would be interpreted, leading to overhead only in rare cases, while the common execution path could be optimized better.

Stack frame creation for setter. As discussed in Section 5.2, the main remaining slow-down in the current implementation lies with setter methods, *i.e.,* methods only setting the value of one or multiple instance variables. It is possible to change the JIT to generate two paths for such methods. The method would start by testing if the receiver is read-only or not, if it is not the case, which is the most common, a quick path without stack frame creation nor read-only checks can be taken instead of the slow path with stack frame creation and read-only checks.

7.2 Features

Read-only contexts. For simplicity, I enforced all contexts to be writable. It would be interesting to allow context to be read-only, though it is not clear what would be the use-cases. Read-only contexts can't be executed by the existing VM as code execution requires at least the mutation of the program counter of the context. Hence, such contexts would not be mapped to stack frames and would only exist as normal objects. Execution returning to read-only contexts would fail and Smalltalk code would be able to handle the failure. The only way such contexts could be executed is through a separated runtime directly written by the programmer to correctly execute the code. Work in that direction is going to happen if someone provides a valid use-case.

Modification tracker. One of the main use-cases of the write barrier is to track object modifications. To do so, one has to implement an in-image framework on top of the write barrier APIs proposed in this paper. The framework has to correctly handle store failures of both primitives such as at:put and instance variable store.

In-image primitive fall-back. As stated in Section 3.2, all the primitive methods mutating an object need to have their fall-back code updated to raise the correct error. If such primitives fail because of a read-only object, the primitive failure error should be appropriate and not an unrelated error. This has still to be done.

8. Conclusion

In this paper I have described the implementation of the write barrier in the Cog VM and the Pharo image. According to the multiple evaluations, the feature was introduced

[5] These trampolines require specific registers to hold specific values to be called and forbid other registers to stay alive across the trampoline calls.

with little to no overhead in term of memory footprint and execution time in most applications.

Although the overhead is minimal, very uncommon pathological cases still show an execution time overhead of up to 18.2%. I believe the pathological cases overhead could be solved by compiling two paths for setter methods and by falling back to bytecode interpretation on uncommon machine code paths. Hopefully, once polished over months of production and customer feed-back, the write barrier will induce a negligible overhead even in uncommon cases.

Acknowledgements

I thank Eliot Miranda for helping me implementing the write barrier in the Cog VM and reviewing all my commits.

I thank Colin Putney for clarifying the term immutability against write barrier and discussing the implementation in general, as well as Tobias Pape, Jan Van de Sandt, Ryan Macnak, Tudor Girba, Chris Cunnigham, Tim Rowledge, Ben Coman, Bert Freudenberg and Denis Kudriashov on the Squeak virtual machine mailing list.

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.

References

[1] H. G. Baker. The Treadmill: Real-time Garbage Collection Without Motion Sickness. *SIGPLAN Not.*, 1992.

[2] A. P. Black, S. Ducasse, O. Nierstrasz, D. Pollet, D. Cassou, and M. Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009.

[3] Contributors. Cuis smalltalk website. http://www.cuis-smalltalk.org/.

[4] Contributors. Racket website. http://racket-lang.org/.

[5] M. Denker. Reflection in Pharo 5, European Smalltalk User Group talk, ESUG '15, 2015. http://www.slideshare.net/MarcusDenker/reflection-in-pharo5.

[6] L. P. Deutsch and A. M. Schiffman. Efficient Implementation of the Smalltalk-80 system. In *Principles of Programming Languages*, POPL '84, 1984.

[7] I. Gouy and F. Brent. The Computer Language Benchmarks Game, 2004. http://benchmarksgame.alioth.debian.org/.

[8] M. Guzdial and K. Rose. *Squeak — Open Personal Computing and Multimedia*. Prentice-Hall, 2001.

[9] U. Hölzle. A Fast Write Barrier for Generational Garbage Collectors. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA'93 Workshop on Garbage Collection, 1993.

[10] A. L. Hosking, J. E. B. Moss, and D. Stefanovic. A Comparative Performance Evaluation of Write Barrier Implementation. In *Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '92, 1992.

[11] E. Miranda. Cog Blog: Speeding Up Terf, Squeak, Pharo and Croquet with a fast open-source Smalltalk VM, 2008. http://www.mirandabanda.org/cogblog/.

[12] E. Miranda and C. Béra. A Partial Read Barrier for Efficient Support of Live Object-oriented Programming. In *International Symposium on Memory Management*, ISMM '15, New York, NY, USA, 2015.