A low Overhead Per Object Write Barrier for Smalltalk

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

Abstract

In several commercial Smalltalk, 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, typically induces overhead, 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, Immutability.

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) [10] is becoming the most popular open source Smalltalk virtual machine, with multiple widely used Smalltalk clients: Pharo [2], Squeak [7] 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. Pharo sounds reasonable as it seems to be the most popular Smalltalk client.

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 [8, 9]. 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 object 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 in Smalltalk.

2.2 Terminology

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

[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.

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 important one.

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 (logging, etc.). 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 use in Gem Builder for Smalltalk, a framework to deeply integrate a Smalltalk application with the gemstone persistence layer.

Core read-only objects. Another interesting read-only object use-case, though less popular, 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 and allows more aggressive optimisations.

2.4 Problem: limiting the overhead

The problem statement is as follow:

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 do not catch all the reflective APIs. For example, in Pharo, it is possible using reflective APIs to activate any primitive operation on any object in the system. Some primitives operations, such as the at:put: primitive, mutate objects. Catching this kind of cases is really difficult, maybe even impossible, without VM support.

3. Solution: Low-level implementation

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 ? A bit seems appropriate as there are only two possible cases. I detail later in the paper the position of the bit.

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. In fact, 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, all the discussions related to instance variable store apply to literal variable stores. I don't duplicate the discussions to make the paper as it does not bring any interesting concepts.

In the execution engine, instance variable store code was rewritten 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 store is not performed by default even if the call-back returns. By design, the VM assumes that temp vectors (data structure used to store closure enclosing context information), are never read-only.

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

Limitations. While implementing our 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 object mutations around process scheduling objects, 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 to inform the programs about the mutations.

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.

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

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

4. 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 to use the write barrier and to build features such as an object modification tracker.

4.1 Core write barrier primitives

Two main primitives were added into the Object class: Object»isReadOnlyObject and Object»setIsReadOnlyObject:.

Object»isReadOnlyObject. The primitive answers a boolean, depending if the object is marked as read-only or not. It should never fail on a VM supporting the write barrier. The primitive method code is available in Figure 1. The Pharo 6 production version is available with additional comments, omitted in the paper.

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

Figure 1. Object»isReadOnlyObject primitive

setIsReadOnlyObject: This second primitive marks the receiver as being read-only or writable, depending on the boolean parameter.

The design of this method 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 for simplicity, hence sharing the same primitive number for these two operations seemed the right thing to do. However, for convenience, I added two other (non primitives) methods, Object»beWritableObject and Object»beReadOnlyObject, as shown in Figure 2, that only calls Object»setIsReadOnlyObject: with the corresponding boolean parameter.

The primitive method code is available in Figure 2. The Pharo 6 production version is available with additional comments, omitted in the paper.

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

Object>>beReadOnlyObject
    ^ self setIsReadOnlyObject: true

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

Figure 2. Object»setIsReadOnlyObject: primitive

4.2 Primitive fall-back

As stated in the Section 3, primitive operations mutating objects fail if they attempt to mutate a read-only object. Hence, each primitive failure code needs to be edited to raise an appropriate error if it failed because of a read-only object. 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, raise an appropriate error instead of 'Instances of Objects are not indexable'.

Unfortunately, this part has not, at the moment where I write the paper, been integrated in Pharo 6.

4.3 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 hacking the active process. The cannotAssign:withIndex: call-back was designed using this hack. After handling the mutation failure, the call-back does not return any value as the code on Figure 3 shows. The comment "CAN'T REACH", as in the VM

Slang code, 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 hack 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 3. Pharo call-back implementation

4.4 Other in-image features

Support flags. The Cog VM provides to the Smalltalk clients a set of parameters. In Pharo, the VirtualMachine instance allows to do requests on the vm parameters. I added a method, VirtualMachine»supportsWriteBarrier, which answers if the VM currently used supports the write barrier feature.

Mirror primitives. The Cog VM 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 problems 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 does not understand the messages required for debugging.

To avoid VmM crashes, proxies are debugged through mirror primitives. For example, the primitive instVarAt:, which answers the value of an instance variable of an object, exist 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, I made sure the two primitives isReadOnlyObject and setIsReadOnlyObject: were 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 without any problem.

5. VM implementation

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

5.1 Object representation

Each 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 [11] for the write barrier. I used this bit to mark the read-only state of an object, as shown on Figure 4.

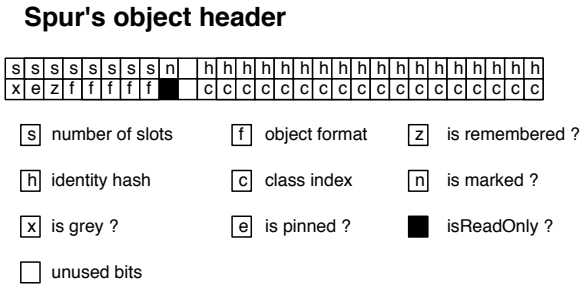

Figure 4. Object header memory representation in Spur

5.2 Interpreter implementation

5.2.1 Primitives.

I needed to add the support here 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 3, several objects can't be read-only, so the primitives related to process scheduling and context accessing don't need to be changed.

5.2.2 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 correctly the read-only write barrier, both the simulation engine used for debugging and the extended C code have to be correct and to have the same behavior.

5.3 JIT compiler support

5.3.1 Primitives.

As for the interpreter, I needed to update the JIT to correctly compile primitive operations.

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 smallIntegers. For this purpose, a set of primitives is redefined in the JIT register transfer language (RTL)[2] and is compiled to machine code when the method with the corresponding primitive number is.

For the purpose of this paper, we will consider 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 calls the interpreter primitive code.

All the existing interpreter primitive code was updated correctly when I patched the interpreter, so they correctly 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. The two primitives are the two versions of at:put:, the generic one and the one for strings. I updated these two primitives to generate machine code failing if the receiver is read-only.

5.3.2 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 if the value assigned is a literal or what register is live or not at this point in the code. In this subsection, I will only discuss the most common case, 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.

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. In the future, there are plans to implement an incremental garbage collector with tri-color marking [1], adding new constraints to the write barrier.

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[3] 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 lambada store is shown on Figure 5.

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 6, 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

[2] A register transfer language (RTL) is a kind of intermediate representation that is very close to assembly language, such as that which is used in a compiler. It is used to describe data flow at the register-transfer level of an architecture

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

Disassembly	Meaning
movl -12(%ebp), %edx	Load the receiver in %edx.
popl %edi	Load the value to store in another register.
movl %edi, %ds:0x8(%edx)	Perform the store in the first instance variable using the registers.
testl $0x00000003, %edi	If the value to store is immediate, jump after the store check.
jnz .+0x0000001b	
movl $0x00040088, %eax	If the receiver is a young object, jump after the store check.
cmpl %eax, %edx	
jb .+0x00000012	
cmpl %eax, %edi	If the value to store is an old object, jump after the store check.
jnb .+0x0000000e	
movzbl %ds:0x3(%edx), %eax	If the receiver is already in the remembered table, jump after the store check.
testb $0x20, %al	
jnz .+0x00000005	
call .+0xfffff34f	Calls the store check trampoline.

Figure 5. Vanilla lambda store

not the case. This solution implied the creation of a single new trampoline that calls the correct interpreter function when the receiver is read-only to call in the language the cannotAssign:withIndex: call-back.

Disassembly	Meaning
movl %ds:(%edx), %eax	If the receiver is writable, jump to the store.
testl $0x00800000, %eax	
jz .+0x0000000a	
call .+0xfffff371	Calls the read-only failure trampoline.
movl -12(%ebp), %edx	Restore the receiver (to keep its register live) and jump after the store.
jmp .+0x00000024	

Figure 6. 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. With the advises of Eliot Miranda, I 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 7. 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. For this implementation, I added 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 call the variable index needs to be passed as a parameter, requiring extra machine instructions per-store. I decided to duplicate the trampoline instead: a fixed number of trampolines based on a VM setting are created, currently 6. This way, each of the most common variable indexes (0 to 4) can call the corresponding trampoline version specialized 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.

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

Disassembly	Meaning
movl -12(%ebp), %edx	Load the receiver in %edx.
popl %ecx	Load the value to store in another register.
movl %ds:(%edx), %eax	If the receiver is read-only, jump to the store trampoline.
testl $0x00800000, %eax	
jnz .+0x0000001e	
movl %ecx, %ds:0x8(%edx)	Perform the store in the first instance variable using the registers.
testb $0x03, %cl	If the value to store is immediate, jump after the store check.
jnz .+0x0000001e	
movl $0x00040088, %eax	If the receiver is a young object, jump after the store check.
cmpl %eax, %edx	
jb .+0x00000015	
cmpl %eax, %ecx	If the value to store is an old object, jump after the store check.
jnb .+0x00000011	
movzbl %ds:0x3(%edx), %eax	If the receiver is already in the remembered table, jump after the store check.
testb $0x20, %al	
jnz .+0x00000008	
call .+0xfffff34f	Calls the store check trampoline.
movl -12(%ebp), %edx	Restore the receiver (to keep its register live).

Figure 7. Production lambda store with the write barrier

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. Eliot Miranda then changed the write barrier 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.

6. Evaluation

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

6.1 Memory overhead

Object representation. As described in Section 3, the overhead per object is a single bit. 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 very small executable zone holding all the machine code versions 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 know which objects are referenced. It has a cost as for each machine code method it needs to read the metadata associated to locate the object referenced, and to avoid getting it too slow, the machine code zone has to be limited.
- 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 8, 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 8. Machine code zone size

Locally: trampolines. When comparing the first available address between the VM with and without the read-barrier, one notices a difference of 400 bytes, which corresponds to the size of the new trampolines (plus the size of the alignment Nops they require). Pharo normally uses a machine code zone size of 1 or 2MB, hence the memory overhead

is between 0.02 and 0.04% of the total machine code zone size.

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%.

6.2 Execution time

Benchmarks. I evaluated the difference in performance using the Games benchmarks [6] 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 do not have an infrastructure precise enough 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 ±3 nanoseconds per run
Write Barrier	13.6 ±2 nanoseconds per run

Figure 9. Pathological benchmark code and results

In this pathological case, as shown on Figure 9, one notices a 18.2% performance overhead. However, the binary tree benchmark, which was larger, calls extensively a similar method (see Figure 10) 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 10. 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.

7. 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 Pharo 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 [8, 9]. In our approach, there is also have a garbage collector write barrier and part of the machine code is shared with the read-only write barrier.

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, 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).

8. Future Work

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

8.1 Performance improvement

Stack frame mapping and trampolines. While profiling code in the VM to look for methods getting slower with the write barrier, it was possible to see that one could optimize multiple trampolines in the JIT (related and unrelated to read-only objects). Indeed, trampolines such as cannotAssign:withIndex: or mustBeBoolean are taken very rarely, while their presence forces no register to be live across the trampoline call due to the creation of a new stack frame for Smalltalk to be able to handle the errors. It would be possible to convert stack frames triggering those trampolines from machine code frame to bytecode interpreter frames. This way, each time one of those trampolines is taken, the execution would fall back on the bytecode interpreter to resume the method and all registers will be able to be live across the trampoline call.

Stack frame creation for setter. As discussed in Section 6.2, 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.

8.2 Features

Read-only contexts. For simplicity, I enforced all contexts to be writable. It would be interesting to allow context to be read-only. In this case, the VM would not be able to execute the method (method execution includes at least the mutation of the program counter of the context), and the execution would need to fall-back to a user-defined in-image interpreter.

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 handled store failures of both primitives such as at:put and instance variable store.

In-image primitive fall-back. As stated in Section 4.2, 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.

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

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

[8] 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.

[9] 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.

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

[11] 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.