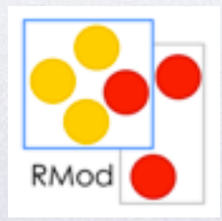



A low Overhead Per Object Write Barrier for the CogVM

Clément Béra



Introduction

- The Cog VM is the standard VM for:
  - Pharo
  - Squeak
  - Newspeak
  - Cuis


Introduction

- Working runtime optimizer for Cog's JIT
- Problem with literal mutability

Problem

- Is it possible to mark any object as read-only ?
- Smalltalk code to handle mutation failure
- Overhead


Terminology

- Discussion on VM mailing-list
- Immutable: state cannot change after object's initialization
- Write barrier or read-only object


Use-cases

- Modification tracker
- Read-only literals
  - Compiler optimizations
  - Inconsistent literal modifications
- Others...

This paper

- NOT about framework built using read-only objects
- Implementation details to limit the overhead

Feature

- Any object can be marked as read-only, except:
  - Immediate objects
  - Context instances
  - Objects related to Process scheduling
  - Objects internal to the runtime


APIs

- Object >> isReadOnlyObject
- Object >> setIsReadOnlyObject:
  - Object >> beWritableObject
  - Object >> beReadOnlyObject


Read-only object

- Instance variable store fail
- Primitives mutating a read-only object fail

IV store failure

```
| message |
message := Message selector: #foo.
message beReadOnlyObject.
message setSelector: #bar.
message
```

- Instance variable is not set.
- A call-back is sent:

```
message attemptToAssign: value withIndex: index
```

Primitive failure

```
| array |
array := Array with: 1.
array beReadOnlyObject.
array at: 1 put: 2.
array
```

- First value of array is not set

Primitive error code

```
instVarAt: index put: anObject
    <primitive: 174 error: ec>
    self primitiveFailed
```

- new error code: #'no modification'

Other details

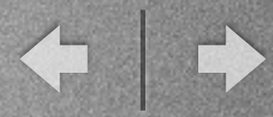
- Support flags

```
Smalltalk vm supportsWriteBarrier
```

- Mirror primitives
  - Object >> object:setIsReadOnlyObject:

VM compilation option

- VM C compiler flag
- The VM can be compiled with or without the feature.

Implementation

- Object representation
- Interpreter support
- JIT support

Implementation

- Most critical part:
  - How to keep IV store efficient ?
    - Machine code generated by the JIT

- Discussed in the paper...

IV Store details

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.

- Wanted
- to show it,

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.

Evaluation: Slow-down

- Binary trees
- IV Store intensive
- No significant difference


Evaluation: Slow-down

- Pathological case: setter

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

Evaluation: Slow-down

- At writing time, setter overhead was 17%

- Stack frame creation problem
  - Two path compilation
  - Now faster than before

Conclusion

- New feature: read-only object
- Overhead is very limited