Clément Béra ~ Smalltalk, Tips 'n Tricks

Arithmetic, inlined and special selectors

12 - *Tuesday* - *Aug 2014*

Posted by Clement Bera in Cog

≈ Leave a comment

Today we are going to discuss how message sends are dispatched in the VM and/or compiled in the image for arithmetics, inlined and special selectors.

The "myth"

In Pharo, everything is an object (even integers, compiled methods or stack activations) and objects can communicate with each other only by sending messages.

Back to reality

A programming language with only message sends as a way to communicate between objects has performance issues. These problem can be solved, as in Self, by having an efficient Just-in-time (JIT) compiler that adaptively recompiles portion of code frequently used to portion of code faster to run. This approach has however two main problems:

- an efficient JIT compiler requires a lot of engineering work to be implemented and maintained for all the common hardware and operating systems.
- portion of code not frequently used are slow, which can be a problem for certain applications, such as large interactive graphical applications.

In our case, for the Cog VM, a baseline JIT compiler is available and we are now working on improving the JIT capabilities to speed up code execution. However, the JIT speeds up code execution only on processors it supports (right now x86, and within a year or so it will support ARM 32 bits and x86_64 bits). For other hardwares with other processors (as supercomputers with MIPS processors) or forbidding JIT compilation (as the iPhone), Pharo and Squeak rely on the stack interpreter speed.

However, performance should be decent in any cases, including:

- large interactive graphical application
- programs running on processors unsupported by the JIT or on OS that forbids JIT compilers

In the first case, the virtual machine usually relies on the interpreter speed for performance. In the latter case, Pharo and Squeak rely on the interpreter based version of the VM (aka the Stack VM), which is portable and JIT-free, and therefore relies on the interpreter speed for performance. Due to these cases, the interpreter performance is critical.

A few months ago, a guy asked me why the compiled method had a specific format in Pharo. One reason is for compactness, but I believe the main reason is that the interpreter performs better with our compiled method format where everything (literals, bytecodes, metadata) is encoded in the same object instead of fetching multiple objects to run one method. In the case of a JIT compiler, the compiled method would be fetched once to be compiled to machine code, so the literals and the bytecodes may be in other objects in memory, these memory access would most probably not slow down the VM as they are rarely accessed. The machine code version of the method, present with all the information at a single memory location, is accessed in most cases.

Static optimizations

To improve the interpreter performance, the bytecode compiler and the Cog VM implement some static tricks. Most of these tricks (the ones we are talking about in this post) consists in handling specifically some message sends based on a specific selectors list. We'll discuss in this blog post what are those selectors, what the tricks consist in for each of them and how they limit or do not limit the language features.

There are three main kinds of specific selectors:

- *arithmetic selectors:* #(#+ #- # #= #= #== #~= #* #/ #\\ #@ #bitShift: #// #bitAnd: #bitOr:)
- *special selectors:* #(#at: #at:put: #size #next #nextPut: #atEnd #== #class #blockCopy: #value #value: #do: #new #new: #x #y)
- *inlined selectors:*#(#and: #or: #caseOf: #caseOf:otherwise: #ifFalse: #ifFalse:ifTrue: #ifTrue: #ifTrue:ifFalse: #ifNil: #ifNil:ifNotNil: #ifNotNil: #ifNotNil:ifNil: #to:by:do: #to:do: #whileFalse #whileFalse: #whileTrue #whileTrue:)

Arithmetic selectors

#(#+ #- #< #> #<= #>= #= #~= # #/ #\\ #@ #bitShift: #// #bitAnd: #bitOr:)*

The arithmetic selectors are messages leading to arithmetic operations if they are sent on integers or floating pointers.

In the StackInterpreter, there are all optimized for SmallIntegers (integers in this range: [-1073741824 ; 1073741823]) and a subset of them (#(#+ #- #< #> #<= #>= #= #~= #* #/ #@) are also optimized for floating pointers.

In the case where a message with the arithmetic selector is sent, and that the receiver and the argument are both SmallInteger or both Float or a SmallInteger and a Float, the virtual machine will not performed any look up and run directly the primitive operation. If the primitive operation fails or if the receiver and the argument does not match the requirements, a regular message send is sent.

In the JIT, there are 3 cases:

- *#(#+ #- #bitAnd: #bitOr:)*: if the JIT can infer the type of at least one of the 2 operands (the operand is a constant and a smallInteger), then a fast path is inlined in the machine code instead of the send site for the arithmetic operation. If one of the 2 operand is not necessarily a smallInteger, a fallback code, sending the message, is also generated, and the fastest path is taken at runtime depending on the types of the operands.
- *#(#< #> #<= #>= #= #~=)*: for #= and #~=, if the JIT can infer that one of the operand is a constant and a smallInteger, and in any case for the 4 other selectors, and if the next instruction of the comparison is a conditional jump, the JIT computes 2 paths for the jump, a fast path using cpu instruction jumpBelow:, JumpGreaterOrEqual: and co that is used at runtime if the 2 operands are SmallInteger, and a regular path that sends the message and compares the resulting value to the object true and false if one of the 2 operands at least is not a smallInteger.
- *the others*: they're not optimized by the JIT at the send site. (common message send that activates the primitive)

Restrictions
The Pharo/Squeak user cannot change the default behavior of the optimized instructions. One cannot for example remove the primitive pragma from `SmallInteger>>#+`. This constraint is not very important as it is uncommon to want to change the execution of integers and floating-pointers arithmetics.

Special selectors
#(#at: #at:put: #size #next #nextPut: #atEnd #== #class #blockCopy: #value #value: #do: #new #new: #x #y)

Firstly, Pharo does not use any more #class and #blockCopy: as special selectors. Squeak uses #class but not #blockCopy: anymore. I will not discuss about these two.

In the JIT, only #== is optimized specifically. This operation checks for identity of two objects, and is performed without any lookup. In addition, the JIT generates faster conditional jump machine code if the result of this operation is used for a conditional jump (in a similar fashion to #(#< #> #<= #>= #= #~=), but no fall back code is needed). All the other special selectors are compiled normally to machine code (message send).

#(at: at:put:)
These two operations are handled specifically in the StackInterpreter, using a specific cache to improve performance. Basically if the method lookup for #at: or #at:put: for an object ends up with a method with the primitive for #at: or #at:put:, and that this object is not a context, the receiver, its variable size, its number of fixed fields and its format are stored in the cache. Next executions of #at: and #at:put: will use the cached values instead of computing the needed data from the object header.

The cache has currently 8 entries (but Eliot will blame me if I do not precise that the number of entries is a settings that can be changed very easily).

at: and at:put: cache	
entry 1	receiver
	receiver format
	receiver fixed fields number
	receiver length - receiver fixed fields number
entry 2	receiver
	receiver format
	receiver fixed fields number
	receiver length - receiver fixed fields number
...	...
	...
	...
	...
entry 8	receiver
	receiver format
	receiver fixed fields number
	receiver length - receiver fixed fields number

The values in the cache are sorted by hash, the hash used being 3 bits from the receiver's address (the last but 2 bits). If the entry is already occupied by another object, the new receiver replaces the previous object.

I don't go into details but this cache is also correctly flushed when needed, so if I create a subclass of array and run this kind of code:

```
a := MyArray new: 1.
a at: 1 put: #myValue.
(a at: 1) logCr.
MyArray compile: 'at: index ^ #newResult' classified: 'access'.
(a at: 1) logCr.
Smalltalk garbageCollect.
(a at: 1) logCr.
```

The results on Transcript are always correct.

#size

For Array and ByteString, the operation *size* is performed without lookup and directly returns the object variable size. The message is sent normally on other objects.

#(#next #nextPut: #atEnd #do: #new #new:)

These selectors are special only to reduce their memory footprint (by encoding them in the bytecode instead of the literal frame of a method). They are executed as regular message sends.

#==
In addition to being the only special selector optimized by the JIT, #== is optimized by the StackInterpreter. It always answers a boolean, true if the 2 objects have the same address, else false. This operation is performed without any look up.

#value #value:
If the receiver of a message with one of these selectors is a closure, the VM directly activate the closure without any lookup. Else a regular message send is sent.

#x #y
If the receiver is a point, directly answers the correct instance variable of the point without any lookup. Else a regular message send is sent.

Restrictions

Most of these optimizations are restricted to given classes and/or primitive methods. So the restrictions for these optimizations are low: you cannot remove the primitive pragma from the optimized methods and replace it with something else or the virtual machine will not be aware.

In the case of #at: and #at:put:, there is no restriction at all (a look up is performed each time to check that a method with a primitive for #at: and #at:put: is found each time).

The only exception is #==. This selector is implemented in ProtoObject, and the corresponding primitive is performed on all objects without any lookup. Therefore, no object can override #== in the system and the implementation in ProtoObject cannot be changed.

Inlined selectors
#(#and: #or: #caseOf: #caseOf:otherwise: #ifFalse: #ifFalse:ifTrue: #ifTrue: #ifTrue:ifFalse: #ifNil: #ifNil:ifNotNil: #ifNotNil: #ifNotNil:ifNil: #to:by:do: #to:do: #whileFalse #whileFalse: #whileTrue #whileTrue:)

All the inlined messages are control flow oriented messages. In other languages, such as Javascript, some keywords are reserved for loops and conditions (if, for, foreach, …). No keywords are reserved in smalltalk. The problem of this approach, using messages instead of dedicated keywords, is that message sends are slower for the interpreter. Therefore, it was decided that control flow messages would be inlined statically to jumps in order to improve the overall performance (by a factor 2.5x-10x).

The generic idea, if I write pseudo code, is that when you write:

```
MyClass>>#foo: argument
| temp |
argument ifTrue: [temp := 1] ifFalse: [temp := 0].
^ temp
```

The code is compiled to:

```
MyClass>>#foo: argument
label 0:
argument jumpFalse: label 1.
temp := 1.
jumpTo: label 2.

label 1:
temp := 0.

label: 2
^ temp
```

And something similar is compiled for loops, with a conditional jump marking if the loop continues or if the loop exits and an unconditional back jump.

These inlined selectors have different constraints.

The loop inlined selectors have very low constraints. Basically, one cannot change the implementation of (SmallInteger>>#to:by:do: SmallInteger>>#to:do: BlockClosure>>#whileFalse BlockClosure>>#whileFalse: BlockClosure>>#whileTrue BlockClosure>>#whileTrue:). One usually does not really care, because it is very rare to want to override these selectors.

The conditions inlined selectors (#and: #or: #caseOf: #caseOf:otherwise: #ifFalse: #ifFalse:ifTrue: #ifTrue: #ifTrue:ifFalse: #ifNil: #ifNil:ifNotNil: #ifNotNil: #ifNotNil:ifNil:) constraint more the system. One cannot change the implementation of any of those selectors, but cannot either override one of these selectors in any classes of the system. This is a very annoying constraint when you are building a DSL where you want to use condition selectors or for Boolean proxies.

Discussion
Some control flow messages are not inlined and are currently missing in Pharo. These two messages are SmallInteger>>#timesRepeat: and BlockClosure>>#repeat:. Right now, in the kernel, when one wants to use a loop to optimize some code, he uses #to:do: or #to:by:do:. However, these 2 selectors requires a 1-argument block. Therefore, if we would inline #timesRepeat:, which requires a 0-argument block, we would remove the overhead of pushing the block argument at each iteration, which in certain micro benchmarks is noticeable. In addition, compiling #to:do:,#to:by:do: and #timesRepeat: requires to compile a conditional jump at the beginning, to check if the loop has reached its maximum number of iteration. Compiling and inlining #repeat would allow to compile a loop without a conditional jump at the beginning, and again will be faster in certain micro benchmarks.

Inlining these two messages add very low constraints and may be interesting performance wise. I'm looking forward to someone wanting to do that. I made a first attempt once but it broke the Xtreams library (this library use timesRepeat: on non integer objects).

How to avoid constraints due to these three kinds of selectors

If one needs to avoid the constraints detailed before to experiment with something exotic, there is a simple solution. All these selectors are compiled specifically by the bytecode compiler to tip the VM on how to run them specifically. One can simply remove the compiler specifications for these selectors.

For arithmetics and special selectors, in OpalCompiler, the solution consists in editing the special selectors array, replacing the problematic selectors by nil (`IRBytecodeGenerator>>#specialSelectorsArray`), then reinitializing the bytecode generator (`IRBytecodeGenerator initialize`), and lastly recompiling the whole image (`OpalCompile recompileAll`). The arithmetic and/or special selectors you have removed will now be compiled as any other selector and will be executed as such in the virtual machine.

For inlined selector, this is trickier. One cannot remove them globally from the compiler, as some kernel code rely on the inlined selectors, making the image crash if you remove these optimizations.

However, one can remove the optimization in a given scope through a compilation option. There are 2 ways of setting a compilation option:

- per method, with the pragma #<compilerOptions: #(- optionInlineIf)#>
- per hierarchy of classes, by overriding class side the compiler method:
  ```
  MyClass class>>#compiler
  ^ super compiler options: #(- optionInlineIf)
  ```

The hierarchy of classes way is more convenient, however, it has a major drawback, it is not compatible with monticello/metacello, meaning that the methods will be loaded but miscompiled. Therefore, if you are using this compiler hack, you need to recompile the whole image after loading your code.

Here are the available options:

```
+ optionInlineIf
+ optionInlineIfNil
+ optionInlineAndOr
+ optionInlineWhile
+ optionInlineToDo
+ optionInlineCase
```

Conclusion

It is difficult to combine high performance and high flexibility. With a lot of engineering work, it is possible and has been proven with the Self VM and the Self language. The Self language however still misses a fast interpreter for code used infrequently, which can lead to slow large interactive graphical application.

In our case, the Cog VM chose to limit as little as possible the capability of the system while reaching high performance. However, a few constraints are left. Most of these constraints are easy to manage though, as explained in this post.

Create a free website or blog at WordPress.com.