HOME

PAGES
About Cog
About this blog
Building a Cog
Development Image
Cog Projects
Collaborators
Compiling the VM
Downloads
Eliot Miranda
On-line Papers and Presentations

CATEGORIES
Cog
Spur

SEARCH
Find

{ 2008 06 17 }

BlueBook CompiledMethods – Having Our Cake and Eating It Too

Apologies for this appearing out of order. I should be finishing the Closures posts but I can't for the moment. I crave your indulgence.

As I've said in the first post, both TSTTCPW and the Principles imply I'm keeping the existing CompiledMethod format. In Squeak, as in the Blue Book CompiledMethod is an anomalous hybrid , its first part being object references used mainly for literals, and its second part being raw bytes used mainly for bytecodes. The first word of a CompiledMethod is a SmallInteger that encodes amongst other things the size of the first part, the literal frame. This format is a good choice for an interpreter since both literals and bytecodes can be fetched from the same object, although it causes minor complications in the garbage collector since the GC must not misinterpret the bytecode as pointers.

In VisualWorks, which is a JIT on all platforms, CompiledMethod is a normal object and the bytecodes are held in a ByteArray referred to by a "bytes" instance variable. To reduce the footprint overhead of adding a separate object for bytecodes, short methods of 6 bytes of bytecode or less get their bytecodes encoded in a pair of SmallIntegers, one in the bytecode inst var and one in an additional literal. Because VisualWorks CompiledMethods are ordinary objects adding instance variables to CompiledMethod and its subclasses is directly supported by the system.

These are fine design decisions for a JIT but hopeless for an interpreter. Both interpreting the 6 bytes of bytecode in two SmallIntegers and skipping over the named instance variables on each literal access would slow an interpreter down significantly.

So the hybrid CompiledMethod format stays, but the pressure to add instance variables to CompiledMethod is high. In fact Squeak has had a per-CompiledMethod holder for extra instance variables for a while. Its called MethodProperties and while its useful it is IMO a tragic hack. *I've always taken tragedy to mean drama driven by a character flaw – see Hamartia or Tragic Flaw. Hamlet and Othello are tragedies because events are driven by both characters' insecurities. Titus Andronicus, on the other hand, is merely a bloodbath.*

I'm willing to bet that reason for CompiledMethod having a hybrid format was to save space in the original 16-bit Xerox implementations. This hybrid format is exactly what complicates adding named instance variables and subclasses to CompiledMethod. So it is tragic to see every CompiledMethod in the system get another object (at least another 20 bytes per CompiledMethod, 4 bytes for the literal slot, 16 bytes for the MethodProperties instance) that in most cases merely adds a selector instance variable (4 bytes). So how to we square the circle?

On starting at Qwaq I immediately wanted to get to work on restructuring the compiler to allow for easy migration of bytecode sets. This would give me free rein in when I wanted to redefine the bytecode set later on and permit implementing closures in a cleaned-up compiler. But I soon found out there were two compilers, one for base Smalltalk and one for Andreas Raab's Tweak. The Tweak compiler differs in allowing a class to define certain "instance varables" to be implemented as dynamically added properties. This is a similar idea to accessing instance variables through messages, something Gilad Bracha is quite rightly pushing in Newspeak.

In the Tweak compiler a class communicates what properties it uses by supplying a set of Field nodes, one for each property. The compiler compiles Field nodes as message sends. In the base compiler instance variables are always accessed directly using bytecodes containing the offset of a given instance varable. Other than that the differences between the compilers are minor. So the first order of business was to merge the Tweak compiler into the base compiler and restructure one compiler instead of two. In doing so I was given a working implementation of compiling accessors within the Tweak compiler. These field definitions can easily be adapted to implementing instance variables in CompiledMethod and subclasses.

A CompiledMethod's literals occupy the first N slots following the header word. The last literal is used by the super send bytecodes to fetch the class in which the current method is defined, whose superclass is the class to begin a super send lookup. The last literal is located by the VM extracting the literal count from the header word. All other literals used by the bytecode are accessed by encoding their literal index in each bytecode as appropriate, the slot immediately following the header word being literal 0. All bytecodes are position-independent, jumps being relative. So one can add literals immediately before the last literal without invalidating the method. The system needs to update the literal count in the header to reflect the extra literal slot but otherwise a method is unaffected. So the scheme is to store instance variables of CompiledMethod and subclasses at the end of the literal frame and access them by messages.

Whitewash alert. There are some details.

Either CompiledMethod class>>newMethod:header: or its callers need to be redefined to add in the relevant number of literals for the named instance variables. Methods such as CompiledMethod>>numLiterals need to be redefined to subtract the number of named instance variables from the literal count. For example

```
CompiledMethod methods for accessing
numNonHeaderPointerFields
    "Answer the number of pointer objects in the receiver."
    ^(self header bitShift: -9) bitAnd: 16rFF
numLiterals
    "Answer the number of literals used by the receiver."
    ^self numNonHeaderPointerFields – self class instSize
```

The compiler needs to keep these accessors hidden and prevent their accidental redefinition. Something we did in Newspeak was to keep accessors out of the class's organization. With a little polish one can easily ensure that the system does not file-out unorganized methods. Additionally the accessors should be name-mangled so their message names are not legal Smalltalk message selectors or variable names so

they can't be accidentally redefined, for example prepend an underscore. So each instance variable needs a pair of messages that look like the following. Let's say that methodClassAssociation is the first instance variable, selector is the second, and pragmas is the third (and in a subclass of CompiledMethod). Then the accessors would be equivalent to

```
_methodClassAssociation
	^self objectAt: self numNonHeaderPointerFields + 1
_methodClassAssociation: anObject
	^self objectAt: self numNonHeaderPointerFields + 1 put: anObject
_selector
	^self objectAt: self numNonHeaderPointerFields
_selector: anObject
	^self objectAt: self numNonHeaderPointerFields put: anObject
_pragmas
	^self objectAt: self numNonHeaderPointerFields – 1
_pragmas: anObject
	^self objectAt: self numNonHeaderPointerFields – 1 put: anObject
```

etc…

The setters must answer the value assigned rather than self to simplify compiling

```
	instVar1 := instVar2 := expr
```

The accessors need to be created as a side-effect of the ClassBuilder redefining CompiledMethod or a subclass. setInstVarNames: seems to be the right hook here.

The ClassBuilder needs to allow the creation of subclasses of CompiledMethod, giving them an instSpec of 12. The instSpec of 12 is the magic number that the garbage collector uses to identify objects that are part pointers, part bytes. See Behavior>>instSpec and Interpreter>>formatOf:. The ClassBuilder also needs special instance mutation code for CompiledMethod and subclasses that would use objectAt: and objectAt:put: to copy state between mutated instances and keep the header up-to-date accurately reflecting the number of literals in the header word. IMO, the bulk of this code belongs on the class side CompiledMethod.
Once we have this done we can say bye bye to MethodProperties, gaining about a megabyte in a 20 megabyte image and make adding extensions such as pragmas cheap and localised.

So who wants the brush? Don't everyone step forward at once…. Anyone? Why is everyone backing away mumbling to themselves? I have always depended on the comfort of strangers…

P.S. Anyone who does want to work on this can either contact me in email or wait a few days until I can start publishing code to a Croquet repostory near you.

2008/06/23
P.P.S. As Joshua Gargus, one of my colleagues at Qwaq, astutely pointed out today one also needs to deal with the changes to the pc in existing contexts when one shape-changes compiled methods. So one would need to enumerate all context objects to fix up their pcs and this gets tricky since those contexts could be in-use by the ClassBuilder. So some care is required to pull this off. Thanks Josh!

Posted by admin on Tuesday, June 17th, 2008, at 8:49 pm, and filed under Cog.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment.

{ 13 }

Comments

1. **André Wendt** | 23-Jun-08 at 12:34 pm | Permalink

   Hi Eliot,

   funny you are about to do what I have planned for my thesis. I was about to contact you by e-mail but couldn't find your address anywhere…

   I got stuck on the part where the literal count has to be redefined. Please drop me a line,

   André

   Will do!

2. **tim@rowledge.org** | 23-Jun-08 at 4:55 pm | Permalink

   "These are fine design decisions for a JIT but hopeless for an interpreter. Both interpreting the 6 bytes of bytecode in two SmallIntegers and skipping over the named instance variables on each literal access would slow an interpreter down significantly." I hope you can offer a solid argument for bothering with that concept in any Squeak implementation; how often is it of any benefit in VW? In the current Squeak OM one could just as easily have those 6 (actually up to 8) bytecodes in a Byte Array and make sure the it was valid for a compact 4byte header for a total of 12 bytes instead of 8. And of course, using a ByteArray handles any size of bytecodes list, savingthe code to check for special cases.

   So yes, a pair of SmalInts might be nasty for an interpreter but since they would be pretty pointless in Sqeak it is a moot point.

   I'm sorry Tim, I don't get your point. I was comparing and contrasting VW and Squeak methods. I wasn't proposing using the VM format in Squeak. I was trying to illustrate one of the problems of using normal objects for compiled methods, which is an increase in footprint. So pressing was the increase in footprint in VW that the up-to-6-bytecodes-in-2-smallintegers scheme was concocted. Now that's a complicated scheme.

   Eliot

3. **tim@rowledge.org** | 23-Jun-08 at 5:09 pm | Permalink

   Go Clean, young man. Go Clean.

   Split CMs into at least a ByteArray for bytecodes and a variable subclass for named instvars and lists of literals. The cost – assuming a compact class is used for ByteArray (or hell, a ByteCodeArray if one wants to be cautiously specific) is 4 bytes per method. And as you've pointed out, cleaning things up to remove the hackiness of methodproperties saves 20 bytes anyway, so spending 4 on cleanliness is not a major issue.
   It would also simplify the GC code and, logically, speed it up a little by avoiding the tests in, for example, ObjectMemory>lastPointerOf:

   Since you've already accepted that image format compatability has to go (something we've been saying for years now without actually getting off our arses and dealing with) why stick with a bad design for CMs?

   Hi Tim,

   A few points. First the overhead is 8 bytes per method, not 4. There are 4 bytes for the slot in the CompiledMethod object that refers to the bytecode ByteArray, and 4 bytes for the header of the bytecode ByteArray.

   Second, the "clean" format incurs a time cost. An extra object for bytecodes incur an extra object table indirection on every send and return. One might poo poo this as being insignificant. But as my great grand mother was (so I'm told) fond of saying "many a mickle makes a muckle".

   Third, whether CompiledMethod is a bad design or not on whether the design meets its requirements than on its internal elegance or lack there-of. The scheme I've illustrated keeps all the footprint and speed advantages of the hybrid format and eliminates its short-comings. So it looks to me like having one's cake and eating it too. Who cares what CompiledMethod looks like internally? If it walks like a duck and quacks like a duck it's a duck, right?

   *Lastly no one is stopping **you** from doing what you' suggest. I'm only doing what seems right to me. I'm certainly not trying to dictate. Go ahead and have a go yourself.*

   Best
   Eliot

4. **tim@rowledge.org** | 23-Jun-08 at 8:55 pm | Permalink

   "So pressing was the increase in footprint in VW that the up-to-6-bytecodes-in-2-smallintegers scheme was concocted. Now that's a complicated scheme"

Well, yes, exactly. So we wouldn't do that again. My point was pretty much that complicated ways of saving a few bytes aren't really worth it; they were essential in 1980, marginal in 1995 and pretty pointless by 2000. In 2008 I suspect they're seriously contraindicated by virtue of wasting development time.

Sorry to keep on being so contradictory but I still think its very important to keep footprint to a minimum on small machines. The iPhone/iPod Touch class of machine is going to be really, really important over the next few years, probably eclipsing the desktop as the computing device we use most often. The OLPC is not a large machine. RAM costs power (even dynamic ram). So I'm going to keep a downward pressure on footprint.

A few bytes here and there for CMs (and good catch on my arithmetic mistake in forgetting the pointer to a bytecodearray but it still leaves a win of 12 bytes per) really isn't important in the grand scale of things. The real space waster in a typical image these days is scads of crap pointless code. Sadly similar to the space explosion of pretty much all software…

Yes! I would love to see work on modularising the Squeak image so that it is really easy to strip down to the kind of kernel Craig Latta has produced and really easy to build up to an image of the kind I love to work in (a big fat Croquet image with lots of goodies in it). Its all very well producing small images and large images but if there isn't an easy way to get from one to the other then it is more difficult to test the other end of the spectrum.

"Second, the "clean" format incurs a time cost. An extra object for bytecodes incur an extra object table indirection on every send and return. One might poo poo this as being insignificant."

Well, one might, but I'm sure you remember how anal retentive I've always been about such stuff. I think it was you that taught me how indirect costs that simply don't appear in 'normal' profiles can actually add up to very large problems. I don't know – because they're indirect – how expensive the unclean CM format is but I doubt it's free.

But, yes, there isn't much point in harping on about such things. Circumstances mean that you have the opportunity to do something – and I know that you will make huge improvements as always – and I don't right now. It would, however, be remiss of me not to make my points and give you something to think about. Besides – what would be the fun in us always agreeing?

Right 🙂 Good!

"If it walks like a duck and quacks like a duck it's a duck, right?"

What! It could be a zombie mecha-duck. Or a Sony aibo-duck.

tim
—
tim Rowledge; tim@rowledge.org; http://www.rowledge.org/tim
It's a damn shame that the irrational orthography of English has lost its rightful place in modern pedagogy.

Right on. Cheers!
Eliot

5. **Craig Latta** | 25-Jun-08 at 11:31 pm | Permalink

Great to see this conversation happening! For what it's worth, for my modularity work I'm just wondering where I can get one bit on each compiled method that gets set when the method is run. 🙂 I'd lazily used a bit in the "trailer" before, but of course trailers are entirely superfluous with the way I'm storing source code now.

Hi Craig!

sorry for the late reply. There are a couple of unused bits in the header.

You could use the sign bit in the method header. It is currently used for the "isClosureCompiled" bit which you can probably do without for your purposes (e.g. compile all methods to be either closure compiled or not and then change the "isClosureCompiled" method to return the appropriate answer as a constant, then steal the bit).

You could use the flag bit (bit 29 in the image, bit 30 in the VM, the bit immediately beneath the sign bit). See CompiledMethod>>flag.

e.g. in Squeak 3.9 final:
(SystemNavigation default allSelect: [:m| m isClosureCompiled]) size 0
(SystemNavigation default allSelect: [:m| m flag]) size 0

Which ever bit you use, you could set the bit when adding a method to the lookup cache.

HTH
Eliot

6. **tim@rowledge.org** | 30-Jun-08 at 7:18 pm | Permalink

I just noticed this
"2008/06/23
P.P.S. As Joshua Gargus, one of my colleagues at Qwaq, astutely pointed out today one also needs to deal with the changes to the pc in existing contexts when one shape-changes compiled methods. So one would need to enumerate all context objects to fix up their pcs and this gets tricky since those contexts

could be in-use by the ClassBuilder. So some care is required to pull this off. Thanks Josh!”

Err, so another bit of not really needed and quite tricky complexity just to allow mixed format CMs? Are sure all the pros and cons have been lined up against a wall and the right ones shot?

7. **David** | 01-Jul-08 at 8:28 pm | Permalink

   OK, I’ll try my hand at the whitewash.

   http://rdrvr.blogspot.com/2008/06/hacking-compiledmethod-instance.html

   David

8. **Paolo Bonzini** | 14-Jul-08 at 1:40 am | Permalink

   Back from holidays — the conversation between Tim and Eliot is really interesting!

   In particular I’m on Tim’s side regarding the cost of the hybrid CompiledMethod format. The problem is that you need to look at the method header to compute the origin of the bytecodes and, with some luck (cache hits). Since the method header is something that you should only fetch on every send, *but not on every return*, you’re already wasting time on every return. Since returns are cheaper than sends, that’s almost as expensive as storing the bytecodes in a separate object.

   Good point, but this applies only in a pure context interpreter. But in a context-to-stack-mapping VM the instruction pointer can be kept as a raw pointer so there is no determination of the number of literals at return time. More on the context-to-stack-mapping VM in a different thread. Currently it is very green but functional.

   By the way, in GNU Smalltalk I allow non-pointer classes to have named instance variables (this is anyway much less of a hack than the hybrid CompiledMethod format!), and so I went for “literals in a separate array, bytecodes in the CompiledMethod”. However, I think the performance would be more or less the same as separating the bytecodes.

   There is a space tradeoff too; the consensus being that it is at least 8 bytes per method, 4 for the slot to point to the bytecodes, 4 for a possibly 1 word header.

   Another idea for improving the speed of the interpreter is to make a more efficient usage of the blue book bytecodes “blockReturn” and “returnTop”. This can be implemented easily, as it should be strictly a compiler/decompiler change.

   Right now, “blockReturn” is only used for blocks, and methods use “returnTop”. However, “blockReturn”

can be used as a generic “return from context” bytecode (usable also to return from a method to the parent context), restricting the “return from method” meaning of “returnTop” to blocks only. The advantage is that returns from a method to the parent context cannot fail, which is why using the “blockReturn” code for them makes returns faster.

I’m not sure I agree entirely 🙂 Both returnTop and blockReturnTop can fail. Just do
thisContext swapSender: nil
and they’ll fail. So they both involve some checking. But I can see that blockReturnTop avoids a check for a block context having a home. So indeed in the closure scheme I’ve implemented blockReturn would be slightly faster. So thanks for this!

Cheers
Eliot

9. **Paolo Bonzini** | 14-Jul-08 at 1:53 am | Permalink

   Other small points, since you’re considering changing the bytecode set. Of course, the GNU Smalltalk bytecode set is open for you to use; it’s designed as a high-performance stack-based bytecode set. I can send you a whitepaper on its design.

   Yes, please do!

   Even if you go for your own set, the relevance of my bytecode set to this discussion is that in gst 28% of the methods have *no literals*. 13% of these are the ones that are simple “return self” or “return instance variable” methods; 3% are primitives that do not have any failure code (this number goes down as the size of the system increases, of course). The important ones are the other 12%, which achieve these through a “push integer” bytecode and a more powerful immediate send bytecode that supports 256 different selectors (extensible to 65536, but I didn’t need that).

10. **Paolo Bonzini** | 17-Jul-08 at 1:39 am | Permalink

    Done, it is at http://smalltalk.gnu.org/files/vmspec.pdf — start from sections 2.1 and 2.2.

    You may also want to peek at http://smalltalk.gnu.org/files/vmimpl.pdf and http://git.savannah.gnu.org/gitweb/?p=smalltalk.git;a=tree;f=superops;hb=HEAD for details about the implementation.

11. **Igor Stasenko** | 25-Jul-08 at 11:36 pm | Permalink

    I agree with Tim, that changes to allow CM having arbitrary number of ‘ivars’ through accessors not worth added complexity.

Why not redesign CM format from scratch then? I think this is the case where you sacrifice too much for backward compatibility.

Second: I doubt that CM needs subclassing. AFAIK, Squeak allows to put arbitrary object in method dictionary, and then if VM discovers such abnormal thing, its just encapsulates arguments into a MessageSend and sends #performMessage: (if i remember correctly) message to that object.
So, this is a nice and already present way to circumvent rigid CM format: for those who wants, they could use such scheme – just put own arbirtary object into a method dict and then do whatever is needed within a message handling method.
Its maybe not fast, but flexible and does not requires too much mumbo-jumbo at VM level.

My own preference: keep VM as stupid as possible, and let complex and clever things live at language side instead.

12. **David** | 05-Aug-08 at 2:20 pm | Permalink

    Whitewash pt. 2

    http://rdrvr.blogspot.com/2008/08/hacking-compiledmethod-allocating-space.html

    David

13. **Kent Beck** | 05-Sep-08 at 3:23 pm | Permalink

    Eliot,

    Regarding the original motivation for squooshed compiled methods, the story I heard that it was primarily to save OOPS, not bytes. A 16-bit object table VM let you allocate 32K objects total, of which ~15K were taken by the base image. Of those, something like 7K were CMs. Separate objects for bytecodes and literals would run application writers clean out of objects.

    Cheers,

    Kent

Post a Comment

Your email is *never* published nor shared. Required fields are marked *

Name *

Email *

Website

Message

Post

« CLOSURES PART I

CLOSURES PART II – THE BYTECODES »