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

{ 2011 03 01 }

Build me a JIT as fast as you can…

Pat-a-cake, pat-a-cake Baker’s man, Build me a JIT as fast as you can…

So how does one write a just-in-time compiler (or JIT for short)? First, *why* write a JIT? Essentially dynamic languages defer certain performance-critical computations to run-time. These languages are "late-bound". Type information is kept in objects themselves and this type information is used at run-time to bind operation names to operation implementations. In static languages such as C the programmer declares the type information to the compiler which is therefore able to bind operation names to operation implementations at compile and link time, but at the cost of writing the software within a potentially inflexible and constraining type system, and having to do without any modification of the executable at run-time. This is the essential trade-off and which side you choose depends on what works for you. Teleplace‘s Croquet-based system is written in Smalltalk so they’ve chose dynamism. If you’re reading this blog chances are you’re a Smalltalk user so it presumably works for you too. Writing a JIT for a dynamic language allows the implementation to generate code of much higher efficiency than an interpreter for invoking late-bound operations by deferring code generation until run-time, there-by regaining some of the advantages a static language compiler has. The primary optimization is applied to message sending, and one effective technique is inline cacheing.

So how *does* one write a JIT? There are two answers I like. One is "one step at a time" and the other is "the way you know how".

One Step at a Time

When I joined Teleplace, née Qwaq, I had a pretty good idea of the JIT I wanted to build. It would be architecturally similar to the VisualWorks VM’s execution engine, a medium performance JIT with good inline cacheing. But that would take quite a while and folks at Teleplace much prefer smaller steps over giant leaps; even if climbing the stairs takes longer there’s less chance of breaking one’s neck. So the project has been divided into smaller chunks.

I’ve described the first two chunks which were to add closures and build a VM that used a stack organization under the covers, the Cog Stack Interpreter. This was deployed towards the end of 2008, and provided modest performance improvements while proving one of the more complex parts of the VM, context-to-stack mapping. Mapping Smalltalk’s contexts to a stack organization enables the use of call instructions for message sends, an essential feature of inine cacheing. The next step has been to write a simple JIT with a naïve code generator but good send performance which I’ll describe in this and the next few posts.

Later steps are to write a more sophisticated code generator, to improve the garbage collector and object model, and to tackle adaptive-optimization/speculative inlining in an innovative way. I've completed the first of these, and will write it up subsequently. I've yet to start work on the garbage collector and object model, although I have a design in mind. For two weeks in February I've worked with Marcus Denker of INRIA on the infrastructure for adaptive-optimization/speculative inlining in Cog and Squeak and it's been fun!

The Way One Knows How

Knowing what one's doing in software engineering is a very privileged and advantageous position to be in. Not having to spend time exploring unknown parts of the solution space, being guided by past successes and failures, knowing valid overall architectures, writing the code cleanly from the start, all help increase quality and reduce development time. It also means less surprises for both developer and customer. In the past I've written a couple of bytecode-to-threaded code JITs and spent a decade working on HPS, the VisualWorks VM, so I'm in the lucky position of knowing how to write performant Smalltalk VMs. So while there are other ways to write a JIT, such as the in-vogue tracing JIT, I've taken a tried and tested approach, bytecode-to-machine-code translation with good inline cacheing. Better the devil you know than the devil you don't.

But Wait!

Except of course what I've been talking about is how to architect a VM, not actually how to develop it, what language to write it in and how to debug it. One thing I *have* learnt that's new is that writing a VM in a high-level language (the Squeak Interpreter and the Cog Stack Interpreter are written in Smalltalk) instead of a machine-oriented language (my BrouHaHa VMs and the HPS VM are written in C) is a great idea. At first I was really sceptical about the approach taken to write the original Squeak interpreter. This is to write the VM in a subset of Smalltalk, the source being derived from the blue-book VM specification in Smalltalk, and then to translate this into C via a Smalltalk program called Slang. As I developed the stack interpreter I soon grew to love being able to use the Smalltalk IDE to browse and debug the VM. It meant some compromise; one doesn't write pure Smalltalk and sometimes using Slang feels like being hit over the head by a Kensei without any resultant enlightenment, but it proved way better than C. Some terminology, when working with the Smalltalk form of the VM in the Smalltalk IDE we're working in the simulator.

In approaching writing the JIT, I had one big problem, how to execute the x86 machine code the JIT would generate in the simulator. One thought was to attempt to auto-generate a Smalltalk implementation of an x86 given some formal definition, but I couldn't find one. Instead I followed my colleague Josh Gargus' suggestion and used the Bochs pc-compatible emulator, making it a Squeak plugin, accessed through suitable primitives. As described previously, I took the core processor code, wrote a different memory interface for it and mated it to the byte array the simulator uses to hold the Smalltalk heap. Hence I am able to generate x86 machine code into the byte array and execute this with Bochs, with Bochs catching all errors and returning them as primitive failures which are translated into exceptions.

OK, let's go

Ok, so now I’m ready to dive into the VM. I could start with the overall architecture, but that’s going to delay showing you real code for a while. In subsequent posts I’ll discuss marrying the JIT and the interpreter, code generation and simulation and method management and method metadata and its uses. But let’s start with inline cacheing, the main rationale behind the JIT.

Inline cacheing is the technique of speeding up sends by cacheing the results of method lookups at send sites instead of in the first-level method lookup cache. Let’s remind ourselves of what a first-level method lookup cache probe looks like. We start with the interpreter’s implementation of a generic zero-argument send. In the below the green code between <>’s are "method tags", method metadata we use to communicate with the Smalltalk to C translator.

```
    StackInterpreter methods for send bytecodes
    sendLiteralSelector0ArgsBytecode
        "Can use any of the first 16 literals for the selector."
        | rcvr |
        messageSelector := self literal: (currentBytecode bitAnd: 16rF).
        argumentCount := 0.
        rcvr := self internalStackValue: 0.
        lkupClass := objectMemory fetchClassOf: rcvr.
        self commonSend

    StackInterpreter methods for message sending
    commonSend
        "Send a message, starting lookup with the receiver’s class."
        "Assume: messageSelector and argumentCount have been set, and that
        the receiver and arguments have been pushed onto the stack,"
        "Note: This method is inlined into the interpreter dispatch loop."
        <sharedCodeNamed: ‘commonSend’ inCase: 131>
        self internalFindNewMethod.
        self internalExecuteNewMethod.
        self fetchNextBytecode

    internalFindNewMethod
        "Find the compiled method to be run when the current messageSelector is sent
         to the class ‘lkupClass’, setting the values of ‘newMethod’ and ‘primitiveIndex’."
        | ok |
        <inline: true>
        ok := self lookupInMethodCacheSel: messageSelector class: lkupClass.
        ok ifFalse: "entry was not found in the cache; look it up the hard way"
            [self externalizeIPandSP.
            self lookupMethodInClass: lkupClass.
            self internalizeIPandSP.
            self addNewMethodToCache: lkupClass]
```

The confusing bits about the above are details to do with the translation of the code to C for the production interpreter. The internal prefix implies that such an interpreter method gets inlined into a single large interpreter function and that it uses local stack frame and instruction pointers that we expect the C compiler to place in registers (that's what externalizeIPandSP and internalizeIPandSP do, export the local to the global pointers and import the global to the local pointers respectively). The **sharedCodeNamed:** pragma causes one copy of the commonSend/internalFindNewMethod/lookupInMethodCacheSel:class: code to be generated in the interpreter loop (at bytecode 131, singleExtendedSendBytecode) and for all other users, such as sendLiteralSelector0ArgsBytecode to jump to that code. So here's the actual lookup. It does up to three probes to avoid collisions:

```
    StackInterpreter methods for message sending
    lookupInMethodCacheSel: selector class: class
        "This method implements a simple method lookup cache. If
an entry for the given selector and
         class is found in the cache, set the values of 'newMethod'
and 'primitiveFunctionPointer' and
         return true. Otherwise, return false."
        "About the re-probe scheme: The hash is the low bits of the
XOR of two large addresses, minus their
         useless lowest two bits. If a probe doesn't get a hit, the hash
is shifted right one bit to compute the
         next probe, introducing a new randomish bit. The cache is
probed 3 times before giving up."
        "WARNING: Since the hash computation is based on the
object addresses of the class and selector, we
         must rehash or flush when compacting storage. We've
chosen to flush, since that also saves the trouble
         of updating the addresses of the objects in the cache."

        | hash probe |
        <inline: true>
        hash := selector bitXor: class. "shift drops two low-order
zeros from addresses"

        probe := hash bitAnd: MethodCacheMask. "first probe"
        (((methodCache at: probe + MethodCacheSelector) =
selector) and:
             [(methodCache at: probe + MethodCacheClass) =
class]) ifTrue:
                  [newMethod := methodCache at: probe +
MethodCacheMethod.
                  primitiveFunctionPointer := self cCoerceSimple:
(methodCache at: probe + MethodCachePrimFunction)
                                                            to:
#'void (*)()'.
                  ^ true        "found entry in cache; done"].

        probe := (hash >> 1) bitAnd: MethodCacheMask. "second
probe"
        (((methodCache at: probe + MethodCacheSelector) =
selector) and:
             [(methodCache at: probe + MethodCacheClass) =
class]) ifTrue:
                  [newMethod := methodCache at: probe +
MethodCacheMethod.
                  primitiveFunctionPointer := self cCoerceSimple:
```

```
(methodCache at: probe + MethodCachePrimFunction)
                                                                    to:
#'void (*)()'.
                        ^ true        "found entry in cache; done"].

            probe := (hash >> 2) bitAnd: MethodCacheMask.
            (((methodCache at: probe + MethodCacheSelector) =
selector) and:
                   [(methodCache at: probe + MethodCacheClass) =
class]) ifTrue:
                        [newMethod := methodCache at: probe +
MethodCacheMethod.
                        primitiveFunctionPointer := self cCoerceSimple:
(methodCache at: probe + MethodCachePrimFunction)
                                                                    to:
#'void (*)()'.
                        ^ true        "found entry in cache; done"].

            ^ false
```

What this boils down to is that on every send the interpreter must
– fetch a selector from the literal frame of the method
– fetch the receiver argumentCount items down the stack (beneath any arguments)
– fetch the class of the receiver, expensive since the receiver may be a tagged SmallInteger,
or have one of three different header formats, all of which require a different form to extract the
class, one of which, compact classes, involves an expensive double-indirection into a table of classes
– form a hash function from the selector and class and use this to probe the first-level method lookup
cache, a set of class,selector,method,primitive tuples
– compare the current class and selector against those in the tuple and if there's a match evaluate the result
(i.e. if there's no primitive then activate the new method, otherwise evaluate the primitive and activate the
new method only if the primitive fails)

Ouch! Note of course that this is still way cheaper than searching the class hierarchy for the method matching the selector. For a typical 1024 entry cache about 3% of sends miss the cache (we could go into detail here but it's not the focus).

So let's JIT

First I need to present two keys of the overall architecture, when and when not to JIT and how to relate bytecoded (interpreted) methods to machine code methods. JIT compilation is akin to interpetation. We make a linear pass through the bytecodes for a method, decoding each bytecode and generating equivalent machine code for it. If a method is only executed once in any run it is not worth compiling; we may as well interpret it and save time and space (the space occupied by the machine code). If a method is very large we may not have the room to produce the bytecode (there's about an order-of-magnitude expansion from byte to machine code). The first time a message is sent that binds to a particular method, that method won't be in the lookup cache and we'll do a class traversal lookup via lookupMethodInClass: above. So a simple way to implement the interpret on single-use policy is to only compile to machine code when finding a method in the first-level method lookup cache. We avoid compiling large methods, which are typically initializers, and rarely performance-critical inner loops, by refusing to compile methods whose number of literals exceeds a limit, settable via the command line that defaults to 60 literals, which excludes a mere 0.3% of methods in my image.

```
| c n |
c := n := 0.
SystemNavigation new browseAllSelect:
        [:m|
        m numLiterals <= 60
                ifTrue: [c := c + 1]
                ifFalse: [n := n + 1].
        false].
{ c. n. n asFloat / (c + n) } #(92918 243 0.002608387630016853)
```

While there are useful tricks one can play in mapping bytecoded methods to machine code, such as "copy down" where we produce a distinct machine code copy of a bytecoded method for each class of receiver that uses it, which can help in making sends more specific in machine code, we can also keep things simple and have a one-to-one correspondence from bytecoded to machine- code methods, and that keeps with the measurable improvements as soon as possible requirement and so is the approach taken in Cog. In Squeak, the first slot in a bytecoded method is a SmallInteger that encodes things like number of arguments, number of literals, primitive index – if any, and so on. This header is only accessed from Squeak via the CompiledMethod>>objectAt: primitive, and in the VM is only accessed via the StackInterpreter>>headerOf: method. So one straight-forward way to implement the map from bytecode to machine code is by keeping a copy of the header word in the machine code method and replacing the header word SmallInteger with a direct pointer to the start of the machine code method. If a bytecoded method's first word is a SmallInteger it has no machine code dual; if it isn't, then the header points directly to the machine code. By making the first word of the machine code form of a method look like the object header of an empty object we prevent the garbage collector from ever examining the insides of a machine code method when it follows the header word in scanning a bytecoded method. We include a back pointer in the machine code method to get to the bytecoded method from a machine code frame.

```
*CoInterpreter methods for compiled methods*
cogMethodOf: aMethodOop
        <api>
        <returnTypeC: #'CogMethod *'>
        | methodHeader |
        methodHeader := self rawHeaderOf: aMethodOop.
        ^self cCoerceSimple: methodHeader to: #'CogMethod *'

headerOf: methodPointer
        | methodHeader |
        methodHeader := self rawHeaderOf: methodPointer.
        ^(self isCogMethodReference: methodHeader)
                ifTrue: [(self cCoerceSimple: methodHeader to:
#'CogMethod *') methodHeader]
                ifFalse: [methodHeader]
```

```
    rawHeaderOf: methodPointer
        <api>
        ^objectMemory fetchPointer: HeaderIndex ofObject:
methodPointer
```

```
typedef struct {
    sqInt objectHeader;         makes this appear to be an empty
object to the garbage collector
    unsigned cmNumArgs : 8;
    unsigned cmType : 3;
    unsigned cmRefersToYoung : 1;
    unsigned cmIsUnlinked : 1;
    unsigned cmUsageCount : 3;
    unsigned stackCheckOffset : 16;
    unsigned short blockSize;
    unsigned short blockEntryOffset;
    sqInt methodObject;         back-pointer to the bytecoded method
    sqInt methodHeader;          the bytecoded method's real header
word, whose header is a pointer to this CogMethod
    sqInt selector;
 } CogMethod;
```

Inline Cacheing

Inline cacheing depends on the fact that in most programs at most send sites there is no polymorphism and that most sends bind to exactly one class of receiver over some usefully long period of time. Since the selector is implicit at a particular send site if we can cache information about the expected class of receiver at the site we can avoid checking the selector at all and merely compare the receiver class against what's expected, avoiding hash function and probe altogether. We still need some class-related cache tag, but that can also be a lot cheaper than the full class fetch above as we'll see.

In Cog we implement inline caches in three forms:

– a monomorphic inline cache; a load of a cache tag and a call of a target method whose prolog checks that the current receiver's class matches the cache tag

– a closed polymorphic inline cache; a growable but finite jump table of class checks and jumps to method bodies, (closed because it deals with a closed set of classes).

– an open polymorphic cache; a first-level method lookup probe specialized for a particular selector, (open since it deals with an open set of classes).

Respectively, these correspond to sites with precisely one receiver class, sites with some limited degree of polymorphism and so-called megamorphic sites with a large degree of polymorphism. What's nice is that while these sends are increasingly expensive moving from monomorphic to megamorphic they are also decreasingly common in roughly a 90%, 9% 0.9% ratio, at least for typical Smalltalk programs.

When a bytecoded method is first jit-compiled to machine-code each send is compiled as a load of a register with the send's selector followed by a call to a run-time routine that will look-up the selector in the current receiver's class and update the send site. Here's the JIT code for the same bytecode above:

```
    SimpleStackBasedCogit methods for bytecode generators
    genSendLiteralSelector0ArgsBytecode
        ^self genSend: (coInterpreter literal: (byte0 bitAnd: 15)
ofMethod: methodObj) numArgs: 0

    genSend: selector numArgs: numArgs
        <inline: false>
        (objectMemory isYoung: selector) ifTrue:
            [hasYoungReferent := true].
        self MoveMw: numArgs * BytesPerWord r: SPReg R:
ReceiverResultReg.
        numArgs > 2 ifTrue:
            [self MoveCq: numArgs R: SendNumArgsReg].
        self MoveCw: selector R: ClassReg.
        self CallSend: (sendTrampolines at: (numArgs min:
NumSendTrampolines – 1)).
        self PushR: ReceiverResultReg.
        ^0
```

The "(objectMemory isYoung: selector)" phrase is necessary to add the generated method to the list of methods that the garbage collector must examine when doing a scavenge (an incremental GC in the Squeak VM's parlance). Since few methods have references to young objects doing this cuts down on the amount of work the GC has to do updating object references in machine code. self MoveMw: numArgs * BytesPerWord r: SPReg R: ReceiverResultReg. loads a register with the receiver from the stack. The "numArgs > 2" phrase is there because we generate four run-time entry-points for sends, one each for 0, 1 and 2 arguments sends, and one for N > 2 argument sends. In typical Smalltalk code arities above 2 have low dynamic frequency. The CallSend: phrase, instead of just a simple Call: adds metadata to the method to allow the VM to locate send sites for various uses such as voiding inline caches when bytecoded methods are redefined in the IDE.

For an example of the inline cache's life cycle I'll use the basicNew send in the following method. Since this is used to instantiate most classes its two sends soon become megamorphic:

```
    Behavior methods for instance creation
    new
        "Answer a new initialized instance of the receiver (which is a
class) with no indexable variables. Fail if the class is indexable."

        ^ self basicNew initialize
```

whose bytecodes are

```
    21 <70> self
    22 <D1> send: basicNew
    23 <D0> send: initialize
    24 <7C> returnTop
```

When compiled on x86 this produces the following code sequence for the basicNew send:

```
        0000440f: movl %ss:(%esp), %edx : 8B 14 24
        00004412: movl $0x003f199c=#basicNew, %ecx : B9 9C 19
3F 00
        00004417: call .+0xffffc014 (0x00000430=ceSend0Args) : E8
```

```
14 C0 FF FF
    IsSendCall:
        0000441c: pushl %edx : 52
```

This is called an "unlinked send" since it hasn't been resolved to any particular method target. It merely calls a run-time routine that will do that lookup. You can see the effect of the CallSend: metadata in labelling the address following the call of ceSend0Args as the return address for a send call. %edx is ReceiverResultReg, holding both the receiver for inline cache checking and the result of all sends. %ecx is the ClassReg which holds the selector for unlinked sends but will get updated to hold the class tag for the current receiver if and when it is executed. The machine-code is nicely decorated; the Bochs x86 emulator includes a disassembler that I scan for addresses and look them up using the VM simulator.

The run-time routine ceSend0Args, generated at start-up, is glue code between native code and the rest of the VM, written in C and hence callable via the platform ABI. The routine writes the native stack and frame pointers, in the x86's case %esp and %ebp, into the interpreter's stackPointer and framePointer, sets the native stack and frame pointers to the C stack (more of this in a subsequent post, just suspend disbelief for now), and then calls into the interpreter with the relevant arguments. This is very like a system call in the OS, switching from the user stack to kernel space. Here we're jumping from the Smalltalk stack zone to the C stack. I call such a jump a trampoline, and use enilopmart to name jumps from C into machine code.

The method called is the following, the core of the binder for unlinked sends. Some slang, "a method is cogged" means a bytecoded method has been compiled to machine code, and so a send site can be linked to the machine code form of the method.

```
    Cointerpreter methods for trampolines
    ceSend: selector super: superNormalBar to: rcvr numArgs: numArgs
        "Entry-point for an unlinked send in a CogMethod. Smalltalk stack looks like
                        receiver
                        args
            head sp ->  sender return pc

        If an MNU then defer to handleMNUInMachineCodeTo:… which will dispatch the MNU and
        may choose to allocate a closed PIC with a fast MNU dispatch for this send. Otherwise
        attempt to link the send site as efficiently as possible. All link attempts may fail; e.g.
        because we're out of code memory.

        Continue execution via either executeMethod or activateInterpreterMethodFromMachineCode:
        depending on whether the target method is cogged or not."
        <api>
        | class canLinkCacheTag errSelIdx cogMethod |
        <inline: false>
        <var: #cogMethod type: #'CogMethod *'>
        cogit assertCStackWellAligned.
        self assert: ((objectMemory isIntegerObject: rcvr) or:
```

```
[objectMemory addressCouldBeObj: rcvr]).
		self sendBreak: selector + BaseHeaderSize
			point: (objectMemory lengthOf: selector)
			receiver: rcvr.
		superNormalBar = 0
			ifTrue: [class := objectMemory fetchClassOf: rcvr]
			ifFalse: [class := self superclassOf: (self
methodClassOf: (self frameMethodObject: framePointer))].
		canLinkCacheTag := (objectMemory isYoungObject: class)
not or: [cogit canLinkToYoungClasses].
		"We set the messageSelector and lkupClass for
executeMethod below since things
		 like the at cache read messageSelector and lkupClass and
so they cannot be left stale."
		messageSelector := selector.
		lkupClass := class.
		argumentCount := numArgs.
		(self lookupInMethodCacheSel: selector class: class)
			ifTrue:"check for coggability because method is in the
cache"
				[self
					ifAppropriateCompileToNativeCode:
newMethod
					selector: selector]
			ifFalse:
				[(errSelIdx := self
lookupMethodNoMNUEtcInClass: class) ~= 0 ifTrue:
					[self handleMNU: errSelIdx
InMachineCodeTo: rcvr classForMessage: class mayLink:
canLinkCacheTag.
					"NOTREACHED"
					self assert: false]].
		"Method found and has a cog method. Attempt to link to it.
The receiver's class may be young.
		 If the Cogit can't store young classes in inline caches we
can link to an open PIC instead."
		(self maybeMethodHasCogMethod: newMethod) ifTrue:
			[cogMethod := self cogMethodOf: newMethod.
			 cogMethod selector = objectMemory nilObject ifTrue:
				[cogit setSelectorOf: cogMethod to: selector].
			 canLinkCacheTag
				ifTrue:
					[cogit
						linkSendAt: (stackPages longAt:
stackPointer)
						in: (self mframeHomeMethod:
framePointer)
						to: cogMethod
						checked: superNormalBar = 0
						receiver: rcvr]
				ifFalse: "If patchToOpenPICFor:.. returns we're
out of code memory"
					[cogit
						patchToOpenPICFor: selector
						numArgs: numArgs
						receiver: rcvr].
			 instructionPointer := self popStack.
			 self executeNewMethod.
```

```
                self assert: false
                "NOTREACHED"].
        instructionPointer := self popStack.
        ^self activateInterpreterMethodFromMachineCode
        "NOTREACHED"
```

The method handles both normal and super sends, hence the superNormalBar = 0 phrase in determining the lookup class. Early on in Cog's evolution I didn't maintain the subset of machine-code methods referring to young objects (the young referrers) and so allowed the Cogit to refuse to link a send if the receiver's class is young. Now the Cogit refuses only if there isn't room to add the method to the young referrers, hence the canLinkCacheTag phrase. Then the message is looked-up in the first-level method lookup cache and, if not already so translated and small enough, is compiled to machine code (ifAppropriateCompileToNativeCode:…). Otherwise there's a full lookup (lookupMethodNoMNUEtcInClass:…) that may have to invoke doesNotUnderstand: or cannotInterpret: (handleMNU:…), Smalltalk and Squeak's responses when messages are not understood or classes not fully present. If the target method has been compiled to machine code (maybeMethodHasCogMethod:) we can link the send site to it. When the target method was compiled to machine code it may not have been encountered via a send, but instead by block value (in blue-book Smalltalks, and current Squeak,, blocks are embedded within their outer method), in which case it wouldn't have had a valid selector, and so we ensure the selector is valid (cogMethod selector = objectMemory nilObject ifTrue:). Finally we can link the send site. For now we'll ignore patchToOpenPICFor:… and look at linkSendAt:…

```
    Cogit methods for inline cacheing
    linkSendAt: callSiteReturnAddress in: sendingMethod to:
targetMethod checked: checked receiver: receiver
        <api>
        <var: #sendingMethod type: #'CogMethod *'>
        <var: #targetMethod type: #'CogMethod *'>
        | inlineCacheTag address extent |
        self assert: (callSiteReturnAddress between:
methodZoneBase and: methodZone zoneLimit).
        inlineCacheTag := checked
                                ifTrue: [objectRepresentation
inlineCacheTagForInstance: receiver]
                                ifFalse: [targetMethod selector
"i.e. no change"].
        (sendingMethod cmRefersToYoung not
         and: [(objectRepresentation inlineCacheTagIsYoung:
inlineCacheTag)]) ifTrue:
            [sendingMethod cmRefersToYoung: true.
             methodZone addToYoungReferrers: sendingMethod].
        address := targetMethod asInteger
                        + (checked ifTrue: [cmEntryOffset] ifFalse:
[cmNoCheckEntryOffset]).
        extent := backEnd
                        rewriteInlineCacheAt:
callSiteReturnAddress
                        tag: inlineCacheTag
                        target: address.
        processor
```

```
                    flushICacheFrom: callSiteReturnAddress – 1 – extent
                    to: callSiteReturnAddress – 1
```

```
    CogIA32Compiler methods for inline cacheing
    rewriteInlineCacheAt: callSiteReturnAddress tag: cacheTag
target: callTargetAddress
        "Rewrite an inline cache to call a different target for a new
tag. This variant is used
         to link unlinked sends in ceSend:to:numArgs: et al. Answer
the extent of the code
         change which is used to compute the range of the icache to
flush."
        | callDistance |
        callDistance := (callTargetAddress – callSiteReturnAddress)
signedIntToLong.
        objectMemory
            byteAt: callSiteReturnAddress – 1 put: (callDistance >>
24 bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 2 put: (callDistance >>
16 bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 3 put: (callDistance >>
8 bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 4 put: (callDistance
bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 6 put: (cacheTag >>
24 bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 7 put: (cacheTag >>
16 bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 8 put: (cacheTag >> 8
bitAnd: 16rFF);
            byteAt: callSiteReturnAddress – 9 put: (cacheTag
bitAnd: 16rFF).
        ^10
```

Recall that the send site looks like

```
        0000440f: movl %ss:(%esp), %edx : 8B 14 24
        00004412: movl $0x003f199c=#basicNew, %ecx : B9 9C 19
3F 00
        00004417: call .+0xffffc014 (0x00000430=ceSend0Args) : E8
14 C0 FF FF
    IsSendCall:
        0000441c: pushl %edx : 52
```

As with any numerous beast a send site isn't guaranteed to survive this far. In typical Smalltalk programs as much as 40% of generated sends may never be executed and hence never linked.

```
    | sends linkedSends |
    sends := linkedSends := 0.
    methodZone methodsDo:
        [:m|
        m cmType = CMMethod ifTrue:
            [self mapFor: m do:
                [:annotation :pc|
                annotation = IsSendCall ifTrue:
                    [| entryPoint |
                     sends := sends + 1.
                     entryPoint := backEnd
```

```
callTargetFromReturnAddress: pc.
                                        entryPoint > methodZoneBase ifTrue: "it's
a linked send"
                                               [linkedSends := linkedSends + 1]].
                                false "keep scanning"]]].
       { sends. linkedSends. linkedSends / sends roundTo: 0.01 } #(6352
3943 0.62)
```

(Being able to script the simulator makes gathering statistcs fun).

Continuing with the example, linkSendAt:… above rewrites the send to the following

```
               0000440f: movl %ss:(%esp), %edx : 8B 14 24
               00004412: movl $0x0013de38=Dictionary class, %ecx : B9
38 DE 13 00
               00004417: call .+0x0000155c (0x00005978=basicNew@20)
: E8 5C 15 00 00
       IsSendCall:
               0000441c: pushl %edx : 52
```

The load of the class register with the selector has been converted into a load of the class of the current receiver (in this case Dictionary class, since the first use is to instantiate a Dictionary). We call the "checked entry-point" of the target method. But for now we don't use the send site. Instead control continues with self executeNewMethod. But the *next* time the send is executed, possibly with a different receiver altogether, the checked entry-point will be called, and its job is to see if the current receiver is the same, in which case the target method is valid, and can be executed. If the receiver's class is different however, the execution of the method must be aborted and a message lookup must be done. So we've converted the costly hash-probe in the interpreter into a fetch of the receivers class and a compare, and we've replaced the indirect jumps involved in activating a method with a direct send of the method, across which the processor can prefetch instructions. We win big.

Inline Cache Tags; a Class by any Other Name is Still as Sweet

An aside; when I emmigrated from the UK and arrived at ParcPlace Systems in 1995 the first work I did on the VisualWorks VM, HPS, was to reimplement inline cacheing. The existing implementation always stored the class of receiver in the inline cache (the load of %ecx above). But that meant that checking sends to the tagged immediates, SmallInteger and Character, were a problem. In VisualWorks SmallIntegers are 30-bit 2's complement integers with least significant two bits equal to 3, and Characters are unsigned 30-bit integers with least significant two bits equal to 1. To get their class, those lsb tag bits have to be masked off, and used to index the sysOopRegistry, VisualWorks' array of objects known to the VM. This code sequence is loooong (tens of insructions) and putting it in the checked entry-point of every method causes bloat. So Peter Deutsch decided that the best approach was to abort if the receiver was an immediate, and have the single copy of the abort code extract the classes of immediates, and reenter if they matched. The checked-entry point looked like:

```
       Lmiss:
               call rtSendMiss
               …
```

```
    Lentry:
        mov %ebx, %eax              mask of the reciever's tags and
abort the send if tagged
        and #3, %eax
        jnz Lmiss
    Lnon-imm entry:                 if the compiler can prove the
receiver isn't tagged avoid the tag check via non-immediate send
bytecodes
        mov 4(%ebx), %eax       load the receiver's class from its
header
        cmp %eax, %ecx              if the receiver's class doesn't
match the inline cache, abort the send
        jnz Lmiss
    Lno-check entry:                no need to check super sends or
sends to constants (by super and no-check send bytecodes)
        method body                 we're good to go
```

But that meant an extra call-return (of the abort sequence) for all sends to immediates. Indeed, as Ian Piumarta reported, fibonacci written on SmallInteger went slower than fibonacci written with blocks.

So I hit on the idea that the inline cache didn't have to contain the class necessarily, simply something that could identify the receiver class. So I rewrote cacheing to store the tag pattern for immediates and otherwise the class. So now the check became

```
    Lmiss:
        call rtSendMiss
        …
    Lentry:
        mov %ebx, %eax              mask of the reciever's tags and
skip class load if tagged
        and #3, %eax
        jnz Lcompare
    Lnon-imm entry:                 if the compiler can prove the
receiver isn't tagged avoid the tag check via non-immediate send
bytecodes
        mov 4(%ebx), %eax       load the receiver's class from its
header
    Lcompare:
        cmp %eax, %ecx              if the receiver's class doesn't
match the inline cache, abort the send
        jnz Lmiss
    Lno-check entry:                no need to check super sends or
sends to constants (by super and no-check send bytecodes)
        method body                 we're good to go
```

Suddenly immediate sends were just as fast as normal sends and the triply-recursive Takeuchi benchmark went 10 times faster (!!).

```
    Integer methods for benchmarks
    takeuchiWith: y and: z
        ^y >= self
            ifTrue: [z]
            ifFalse:
                [(self – 1 takeuchiWith: y and: z)
                    takeuchiWith: (y – 1 takeuchiWith: z and:
self)
                    and: (z – 1 takeuchiWith: self and: y)]
```

I'd been with the company a month, understood inline cacheing, cleaned it up usefully, and got a useful across-the-board speedup in our macro benchmarks. I remember getting the thing working and playing with it until about 2am. The next day I went out and bought a car.

Squeak's object model is optimized for space, and so the above is even more important. Squeak provides "compact classes", an array of up to 32 classes whose instances contain the index into the compact class table, not the class itself. This means the most common objects have a one-word header but it also means that the fetch class sequence is tedious:

```
    ObjectMemory methods for interpreter access
    fetchClassOf: oop
        | ccIndex |
        <inline: true>
        (self isIntegerObject: oop) ifTrue:
            [^ self fetchPointer: ClassInteger ofObject:
specialObjectsOop].

        (ccIndex := (self compactClassIndexOf: oop)) = 0
            ifTrue: [^(self classHeader: oop) bitAnd:
AllButTypeMask]
            ifFalse: "look up compact class"
                [^ self fetchPointer: ccIndex – 1
                    ofObject: (self fetchPointer:
CompactClasses
                                                    ofObject:
specialObjectsOop)]
```

(Why not fix this straight away? Measurable improvements first). So in Cog I allow the inline cache to take either the SmallInteger tag (current Squeak has 31-bit immediate SmallIntegers and no immediate characters), the receiver's class, or the compact class index, suitably shifted to be unambiguous with the previous two forms.

```
    Cogit methods for compile abstract instructions
    compileEntry
        self AlignmentNops: (BytesPerWord max: 8).
        entry := self Label.
        objectRepresentation getInlineCacheClassTagFrom:
ReceiverResultReg into: TempReg.
        self CmpR: ClassReg R: TempReg.
        self JumpNonZero: sendMissCall.
        noCheckEntry := self Label.
        self recordSendTrace ifTrue:
            [self CallRT: ceTraceLinkedSendTrampoline]

    CogObjectRepresentationForSqueakV3 methods for compile
abstract instructions
    getInlineCacheClassTagFrom: sourceReg into: destReg
        "Extract the inline cache tag for the object in sourceReg into
destReg. The inline
         cache tag for a given object is the value loaded in inline
caches to distinguish objects
         of different classes. In SqueakV3 the tag is the integer tag
bit for SmallIntegers (1),
         the compact class index shifted by log: 2 word size for
objects with compact classes
```

```
          (1 * 4 to: 31 * 4 by: 4), or the class. These ranges cannot
overlap because the heap
          (and hence the lowest class object) is beyond the machine
code zone."
         | jumpIsInt jumpCompact |
         <var: #jumpIsInt type: #'AbstractInstruction *'>
         <var: #jumpCompact type: #'AbstractInstruction *'>
         cogit MoveR: sourceReg R: destReg.
         cogit AndCq: 1 R: destReg.
         jumpIsInt := cogit JumpNonZero: 0.
         "Get header word in destReg"
         cogit MoveMw: 0 r: sourceReg R: destReg.
         "Form the byte index of the compact class field"
         cogit LogicalShiftRightCq: (objectMemory
compactClassFieldLSB – ShiftForWord) R: destReg.
         cogit AndCq: ((1 << objectMemory compactClassFieldWidth)
– 1) << ShiftForWord R: destReg.
         jumpCompact := cogit JumpNonZero: 0.
         cogit MoveMw: objectMemory classFieldOffset r: sourceReg
R: destReg.
         "The use of signedIntFromLong is a hack to get round short
addressing mode computations.
          Much easier if offsets are signed and the arithmetic
machinery we have makes it difficult to
          mix signed and unsigned offsets."
         cogit AndCq: AllButTypeMask signedIntFromLong R:
destReg.
         jumpCompact jmpTarget: (jumpIsInt jmpTarget: cogit Label).
         ^0
```

So the prologue of the machine code Behavior>>basicNew method looks like:

```
    5958
         objhdr: 1003
         nArgs: 0       type: 2
         blksiz: 148
         method: 7C8DE8
         mthhdr: 1C8D
         selctr: 3F199C=#basicNew
         blkentry: 0
         stackCheckOffset: BB/5A13
              00005970: xorl %edx, %edx : 31 D2
              00005972: call .+0xffffac71
(0x000005e8=ceMethodAbort) : E8 71 AC FF FF
              00005977: nop : 90
         entry:
              00005978: movl %edx, %eax : 89 D0
              0000597a: andl $0x00000001, %eax : 83 E0 01
              0000597d: jnz .+0x00000010
(0x0000598f=basicNew@37) : 75 10
              0000597f: movl %ds:(%edx), %eax : 8B 02
              00005981: shrl $0x0a, %eax : C1 E8 0A
              00005984: andl $0x0000007c, %eax : 83 E0 7C
              00005987: jnz .+0x00000006
(0x0000598f=basicNew@37) : 75 06
              00005989: movl %ds:0xfffffffc(%edx), %eax : 8B 42 FC
              0000598c: andl $0xfffffffc, %eax : 83 E0 FC
```

```
                    0000598f: cmpl %ecx, %eax : 39 C8
                    00005991: jnz .+0xffffffdf (0x00005972=basicNew@1A)
: 75 DF
          noCheckEntry:
```

So while the inline cache check avoids indirecting to Squeaks specialObjectsArray to fetch the classes of immediates and compact instances, the sequence is still quite long and one of the reasons why I’m eager to implement a streamlined object representation when time and funding permit. By the way, jumping to 5972 invokes ceMethodAbort with %edx containing the receiver and vectors to handling a send miss. Jumping to 5970 invokes ceMethodAbort with %edx zero, is called by method frame build on stack overflow, and vectors to stack overflow and event-checking code, allowing the abort call to do double duty.

The Life-Cycle of the Lesser-Spotted Inline Cache; Emergence as a Polymorphic Inline Cache

So far so good. For those 90% of active send sites, sends are now much faster. The method code sequence is a simple register load of a constant, a direct call, an overly complex class tag extraction and a compare against the tag register. This can be faster than a C++ V-table dispatch because there’s no indirect call through the V-table. Modern processors provide support for indirect branch accelleration but these facilities can be overwhelmed by a large working set. Of course, the class fetch can cause read stalls with a large working set also. Your mileage will vary.

But what happens when a send does miss? A natural approach would be to retry the lookup with the new class and reloink the snd site to the new lookup. In HPS Peter Deutsch crafted inline machine code to probe the first-level method lookup cache and relink the send site if a match was found. In 1995 that code, handling 10% of the active send sites, took fully 15% of entire execution time in the macro benchmark suite, by far the highest single cost in the VM. The Self team had already shown in 1991 that polymorphic inline caches (PICs) were a more effective solution. Instead of rebinding the send site on every send, allocate a jump table, essentially a small machine code method that extracts the class and then does a sequence of compares and jumps to target methods, and rebind the send site to call it. The PIC will grow over time. When first created it will handle two cases, the send site’s original class and the class of the new receiver. As more receiver classes are encountered the PIC can grow to accomodate them. So one issue is how big to allow the PIC to grow. In my experience there is a near exponential decay in the number of cases. So large PICs are unlikely to be profitable, and as we’ll see, there’s a better way to handle megamorphic send sites. In HPS I used 8 entries and in Cog I use 6 entries.

```
| picCounts |
picCounts := Bag new.
methodZone methodsDo:
    [:m|
    m cmType = CMClosedPIC ifTrue:
        [picCounts add: m cPICNumCases]].
picCounts sortedElements {2->231 . 3->57 . 4->8 . 5->6 . 6->55}
```

Clearly there's an anomaly here with 6-entry PICs being relatively abundant, but things are reasonable at first 🙂 What happens if we double the maximum size to 12?

```
| picCounts |
picCounts := Bag new.
methodZone methodsDo:
	[:m|
	m cmType = CMClosedPIC ifTrue:
		[picCounts add: m cPICNumCases]].
picCounts sortedElements {2->236 . 3->58 . 4->8 . 5->6 . 6->5 . 7->5 . 8->2 . 9->9 . 10->5 . 11->2 . 12->34}
```

So the peak at the end is caused by those send sites on their way to becomming megamorphic, which suggests an obvious optimization I'll discuss later. But for now let's see what a closed PIC looks like. Since we need to introspect over a closed PIC (e.g. when the garbage collector updates class references in the code) it needs to be well laid-out. So early in startup I compile a prototype and derive various offsets and the size from it:

```
Cogit methods for inline cacheing
compileClosedPICPrototype
	"Compile the abstract instructions for a full closed PIC used to initialize closedPICSize"
	| numArgs jumpNext |
	<var: #jumpNext type: #'AbstractInstruction *'>
	numArgs := 0.
	self compilePICProlog: numArgs.
	jumpNext := self compileCPICEntry.
	self MoveCw: 16r5EAF00D R: ClassReg.
	self JumpLong: (self cCoerceSimple: 16r5EEDCA5E to: #'void *').
	jumpNext jmpTarget: (endCPICCase0 := self Label).
	1 to: numPICCases – 1 do:
		[:h|
		self CmpCw: 16rBABE1F15+h R: TempReg.
		self MoveCw: 16rBADA550 + h R: ClassReg.
		self JumpLongZero: (self cCoerceSimple: 16rBEDCA5E0 + h to: #'void *').
		h = 1 ifTrue:
			[endCPICCase1 := self Label]].
	self MoveCw: 16rAB5CE55 R: ClassReg.
	self JumpLong: (self cCoerceSimple: (self cPICMissTrampolineFor: numArgs) to: #'void *').
	^0
```

The first comparison in a closed PIC is very similar to the checked entry code in a method; it fetches the class of the receiver and then compares it against the class tag loaded at the send site (waste not, want not). Instead of falling through on a match it jumps to the unchecked entry-point of the send site's original target method (the jump to 16r5EEDCA5E in the prototype above). Each subsequent comparison compares the class tag in TempReg (%eax) against a constant, jumping if there's a match and finally calling a specialized miss routine if no cases match. Here I have an admission to make; the constant loads of ClassReg above as yet serve no purpose. They're there to speed-up doesNotUnderstand: but that's currently unimplemented; so for now please ignore those constant loads. Apologies.

When first created then, the closed PIC for basicNew above looks like

```
    5F98
        nArgs: 0        type: 4
        blksiz: A0
        selctr: 3F199C=#basicNew
        cPICNumCases: 2
        00005fb0: xorl %ecx, %ecx : 31 C9
        00005fb2: call .+0xffffa699 (0x00000650=cePICAbort) : E8
99 A6 FF FF
        00005fb7: nop : 90
    entry:
        00005fb8: movl %edx, %eax : 89 D0
        00005fba: andl $0x00000001, %eax : 83 E0 01
        00005fbd: jnz .+0x00000010 (0x00005fcf=basicNew@37) :
75 10
        00005fbf: movl %ds:(%edx), %eax : 8B 02
        00005fc1: shrl $0x0a, %eax : C1 E8 0A
        00005fc4: andl $0x0000007c, %eax : 83 E0 7C
        00005fc7: jnz .+0x00000006 (0x00005fcf=basicNew@37) :
75 06
        00005fc9: movl %ds:0xfffffffc(%edx), %eax : 8B 42 FC
        00005fcc: andl $0xfffffffc, %eax : 83 E0 FC
        00005fcf: cmpl %ecx, %eax : 39 C8
        00005fd1: jnz .+0x0000000a (0x00005fdd=basicNew@45) :
75 0A
        00005fd3: movl $0x00000000, %ebx : BB 00 00 00 00
        00005fd8: jmp .+0xfffff9b6 (0x00005993=basicNew@3B) :
E9 B6 F9 FF FF
    ClosedPICCase0:
        00005fdd: cmpl $0x00139164=SharedQueue class, %eax :
3D 64 91 13 00
        00005fe2: movl $0x00000000, %ebx : BB 00 00 00 00
        00005fe7: jz .+0xfffff9a6 (0x00005993=basicNew@3B) : 0F
84 A6 F9 FF FF
    ClosedPICCase1:
        00005fed: movl $0x00005f98=basicNew@0, %ecx : B9 98
5F 00 00
        00005ff2: jmp .+0xffffa6c1
(0x000006b8=ceCPICMissTrampoline) : E9 C1 A6 FF FF
```

The cePICAbort call is unused; part of the same unimplemented optimization for MNU. The abort when no matches are found is the final jump to ceCPICMissTrampoline. Both cases jump to 5993=basicNew@3B, which is the start of the basicNew method whose entry code is shown above. The ClosedPICCase0 and ClosedPICCase1 labels are offsets computed from the prototype and allow the VM to find class and target method references for garbage collection and send site unlinking. And the send site is patched to call the PIC, not the original target:

```
        0000440f: movl %ss:(%esp), %edx : 8B 14 24
        00004412: movl $0x0013de38=Dictionary class, %ecx : B9
38 DE 13 00
        00004417: call .+0x00001b9c (0x00005fb8=basicNew@20) :
```

```
E8 9C 1B 00 00
        IsSendCall:
```

If we let the system run until the site becomes megamorphic (in our case more than 6 cases) then the PIC grows to the following:

```
        5F98
                nArgs: 0        type: 4
                blksiz: A0
                selctr: 3F199C=#basicNew
                cPICNumCases: 6
                00005fb0: xorl %ecx, %ecx : 31 C9
                00005fb2: call .+0xffffa699 (0x00000650=cePICAbort) : E8
99 A6 FF FF
                00005fb7: nop : 90
        entry:
                00005fb8: movl %edx, %eax : 89 D0
                00005fba: andl $0x00000001, %eax : 83 E0 01
                00005fbd: jnz .+0x00000010 (0x00005fcf=basicNew@37) :
75 10
                00005fbf: movl %ds:(%edx), %eax : 8B 02
                00005fc1: shrl $0x0a, %eax : C1 E8 0A
                00005fc4: andl $0x0000007c, %eax : 83 E0 7C
                00005fc7: jnz .+0x00000006 (0x00005fcf=basicNew@37) :
75 06
                00005fc9: movl %ds:0xfffffffc(%edx), %eax : 8B 42 FC
                00005fcc: andl $0xfffffffc, %eax : 83 E0 FC
                00005fcf: cmpl %ecx, %eax : 39 C8
                00005fd1: jnz .+0x0000000a (0x00005fdd=basicNew@45) :
75 0A
                00005fd3: movl $0x00000000, %ebx : BB 00 00 00 00
                00005fd8: jmp .+0xfffff9b6 (0x00005993=basicNew@3B) :
E9 B6 F9 FF FF
        ClosedPICCase0:
                00005fdd: cmpl $0x00139164=SharedQueue class, %eax :
3D 64 91 13 00
                00005fe2: movl $0x00000000, %ebx : BB 00 00 00 00
                00005fe7: jz .+0xfffff9a6 (0x00005993=basicNew@3B) : 0F
84 A6 F9 FF FF
        ClosedPICCase1:
                00005fed: cmpl $0x0013f76c=Delay class, %eax : 3D 6C F7
13 00
                00005ff2: movl $0x00000000, %ebx : BB 00 00 00 00
                00005ff7: jz .+0xfffff996 (0x00005993=basicNew@3B) : 0F
84 96 F9 FF FF
        ClosedPICCase2:
                00005ffd: cmpl $0x0013da5c=OrderedCollection class,
%eax : 3D 5C DA 13 00
                00006002: movl $0x00000000, %ebx : BB 00 00 00 00
                00006007: jz .+0xfffff986 (0x00005993=basicNew@3B) : 0F
84 86 F9 FF FF
        ClosedPICCase3:
                0000600d: cmpl $0x0013dd94=Set class, %eax : 3D 94 DD
13 00
                00006012: movl $0x00000000, %ebx : BB 00 00 00 00
                00006017: jz .+0xfffff976 (0x00005993=basicNew@3B) : 0F
84 76 F9 FF FF
        ClosedPICCase4:
            0000601d: cmpl $0x001404b8=UnixFileDirectory class,
%eax : 3D B8 04 14 00
            00006022: movl $0x00000000, %ebx : BB 00 00 00 00
            00006027: jz .+0xfffff966 (0x00005993=basicNew@3B) : 0F
84 66 F9 FF FF
      ClosedPICCase5:
            0000602d: movl $0x00005f98=basicNew@0, %ecx : B9 98
5F 00 00
            00006032: jmp .+0xffffa681
(0x000006b8=ceCPICMissTrampoline) : E9 81 A6 FF FF
```

So as each new case is encountered the jump table is extended until the limit is reached:

```
      Cogit methods for inline cacheing
      cogExtendPIC: cPIC CaseNMethod: caseNMethodOrNil tag:
caseNTag
            <var: #cPIC type: #'CogMethod *'>
            | operand target address size end |
            coInterpreter
                  compilationBreak: cPIC selector
                  point: (objectMemory lengthOf: cPIC selector).
            self allocateOpcodes: 5 bytecodes: 0.
            self assert: (objectRepresentation inlineCacheTagIsYoung:
caseNTag) not.
            operand := 0.
            target := (coInterpreter cogMethodOf: caseNMethodOrNil)
asInteger + cmNoCheckEntryOffset.
            self CmpCw: caseNTag R: TempReg.
            self MoveCw: operand R: SendNumArgsReg.
            self JumpLongZero: (self cCoerceSimple: target to: #'void *').
            self MoveCw: cPIC asInteger R: ClassReg.
            self JumpLong: (self cCoerceSimple: (self
cPICMissTrampolineFor: cPIC cmNumArgs) to: #'void *').

            self computeMaximumSizes.
            address := self addressOfEndOfCase: cPIC cPICNumCases
– 1 inCPIC: cPIC.
            size := self generateInstructionsAt: address.
            end := self outputInstructionsAt: address.
            processor flushICacheFrom: address to: cPIC asInteger +
closedPICSize.
            cPIC cPICNumCases: cPIC cPICNumCases + 1.
            ^0
```

So what to do for megamorphic sites? We could simply handle the send in the miss routine, but that's going to be slow; apart from the unnecessary comparisons there's the cost of calling the miss routine and then repeating the interpreter's slothful lookup in the first-level method lookup cache. Much better is to create a first-level method lookup cache probe in machine code, but specialised for the send site's selector. If so, the selector and its hash become constants; that saves time finding the selector and reduces register pressure (which is an issue when we improve the code generator to pass arguments in registers). And directly calling the lookup code from the send site gets us to the lookup pretty damn quick.

```
      SimpleStackBasedCogit methods for inline cacheing
      compileOpenPIC: selector numArgs: numArgs
```

```
		"Compile the code for an open PIC. Perform a probe of the
first-level method
		 lookup cache followed by a call of ceSendFromOpenPIC: if
the probe fails."
		| jumpSelectorMiss jumpClassMiss itsAHit jumpBCMethod
routine |
		<var: #jumpSelectorMiss type: #'AbstractInstruction *'>
		<var: #jumpClassMiss type: #'AbstractInstruction *'>
		<var: #itsAHit type: #'AbstractInstruction *'>
		<var: #jumpBCMethod type: #'AbstractInstruction *'>
		self compilePICProlog: numArgs.
		self AlignmentNops: (BytesPerWord max: 8).
		entry := self Label.
		objectRepresentation genGetClassObjectOf:
ReceiverResultReg into: ClassReg scratchReg: TempReg.

		"Do first of three probes. See
CoInterpreter>>lookupInMethodCacheSel:class:"
		self flag: #lookupInMethodCacheSel:class:. "so this method
shows up as a sender of lookupInMethodCacheSel:class:"
		self MoveR: ClassReg R: SendNumArgsReg.
		self XorCw: selector R: ClassReg.
		self LogicalShiftLeftCq: ShiftForWord R: ClassReg.
		self AndCq: MethodCacheMask << ShiftForWord R:
ClassReg.
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheSelector << ShiftForWord)
			r: ClassReg
			R: TempReg.
		self annotate: (self CmpCw: selector R: TempReg) objRef:
selector.
		jumpSelectorMiss := self JumpNonZero: 0.
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheClass << ShiftForWord)
			r: ClassReg
			R: TempReg.
		self CmpR: SendNumArgsReg R: TempReg.
		jumpClassMiss := self JumpNonZero: 0.

		itsAHit := self Label.
		"Fetch the method. The interpret trampoline requires the
bytecoded method in SendNumArgsReg"
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheMethod << ShiftForWord)
			r: ClassReg
			R: SendNumArgsReg.
		"If the method is compiled jump to its unchecked entry-point,
otherwise interpret it."
		objectRepresentation
			genLoadSlot: HeaderIndex sourceReg:
SendNumArgsReg destReg: TempReg.
		self MoveR: TempReg R: ClassReg.
		jumpBCMethod := objectRepresentation
genJumpSmallIntegerInScratchReg: TempReg.
		jumpBCMethod jmpTarget: interpretCall.
		self AddCq: cmNoCheckEntryOffset R: ClassReg.
		self JumpR: ClassReg.
		"First probe missed. Do second of three probes. Shift hash
right one and retry."
		jumpSelectorMiss jmpTarget: (jumpClassMiss jmpTarget: self
Label).
		self MoveR: SendNumArgsReg R: ClassReg.
		self XorCw: selector R: ClassReg.
		self LogicalShiftLeftCq: ShiftForWord – 1 R: ClassReg.
		self AndCq: MethodCacheMask << ShiftForWord R:
ClassReg.
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheSelector << ShiftForWord)
			r: ClassReg
			R: TempReg.
		self annotate: (self CmpCw: selector R: TempReg) objRef:
selector.
		jumpSelectorMiss := self JumpNonZero: 0.
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheClass << ShiftForWord)
			r: ClassReg
			R: TempReg.
		self CmpR: SendNumArgsReg R: TempReg.
		self JumpZero: itsAHit.

		"Second probe missed. Do last probe. Shift hash right two
and retry."
		jumpSelectorMiss jmpTarget: self Label.
		self MoveR: SendNumArgsReg R: ClassReg.
		self XorCw: selector R: ClassReg.
		ShiftForWord > 2 ifTrue:
			[self LogicalShiftLeftCq: ShiftForWord – 1 R:
ClassReg].
		self AndCq: MethodCacheMask << ShiftForWord R:
ClassReg.
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheSelector << ShiftForWord)
			r: ClassReg
			R: TempReg.
		self annotate: (self CmpCw: selector R: TempReg) objRef:
selector.
		jumpSelectorMiss := self JumpNonZero: 0.
		self MoveMw: coInterpreter methodCacheAddress
asUnsignedInteger + (MethodCacheClass << ShiftForWord)
			r: ClassReg
			R: TempReg.
		self CmpR: SendNumArgsReg R: TempReg.
		self JumpZero: itsAHit.

		"Last probe missed. Call ceSendFromOpenPIC: to do the full
lookup."
		jumpSelectorMiss jmpTarget: self Label.
		self genSaveStackPointers.
		self genLoadCStackPointers.
		methodLabel addDependent: (self annotateAbsolutePCRef:
(self MoveCw: methodLabel asInteger R: SendNumArgsReg)).
		cStackAlignment > BytesPerWord ifTrue:
			[backEnd
				genAlignCStackSavingRegisters: false
				numArgs: 1
```

```
                    wordAlignment: cStackAlignment /
BytesPerWord].
        backEnd genPassReg: SendNumArgsReg asArgument: 0.
        routine := self cCode: '(sqInt)ceSendFromInLineCacheMiss'
                        inSmalltalk: [self
simulatedAddressFor: #ceSendFromInLineCacheMiss:].
        self annotateCall: (self Call: routine)
        "Note that this call does not return."
```

Again allocating a prototype at startup allows us to know how much space to allocate. The open PIC has the following machine code. Notice that since the first-level method lookup cache includes classes, not class tags (something that when time allows could be changed), the entry fetches the receiver's class, not merely its class tag (hence the load of SmallInteger at 00017279, and the load of the compactClassTable at 00017280).

```
    17240
        nArgs: 0        type: 5
        blksiz: 108
        selctr: 3F199C=#basicNew
        00017258: xorl %ecx, %ecx : 31 C9
        0001725a: call .+0xfffe93f1 (0x00000650=cePICAbort) : E8
F1 93 FE FF
        0001725f: nop : 90
    entry:
        00017260: movl %edx, %eax : 89 D0
        00017262: andl $0x00000001, %eax : 83 E0 01
        00017265: jnz .+0x00000012 (0x00017279=basicNew@39) :
75 12
        00017267: movl %ds:(%edx), %eax : 8B 02
        00017269: shrl $0x0a, %eax : C1 E8 0A
        0001726c: andl $0x0000007c, %eax : 83 E0 7C
        0001726f: jnz .+0x0000000f (0x00017280=basicNew@40) :
75 0F
        00017271: movl %ds:0xfffffffc(%edx), %ecx : 8B 4A FC
        00017274: andl $0xfffffffc, %ecx : 83 E1 FC
        00017277: jmp .+0x0000000d (0x00017286=basicNew@46)
: EB 0D
        00017279: movl $0x0013aa60=SmallInteger, %ecx : B9 60
AA 13 00
    IsObjectReference:
        0001727e: jmp .+0x00000006 (0x00017286=basicNew@46)
: EB 06
        00017280: movl %ds:0x10da80=a(n) Array(%eax), %ecx :
8B 88 80 DA 10 00
    IsObjectReference:
        00017286: movl %ecx, %ebx : 89 CB
        00017288: xorl $0x003f199c=#basicNew, %ecx : 81 F1 9C
19 3F 00
    IsObjectReference:
        0001728e: shll $0x02, %ecx : C1 E1 02
        00017291: andl $0x00003ff0=critical:@268, %ecx : 81 E1 F0
3F 00 00
        00017297: movl %ds:0x108148(%ecx), %eax : 8B 81 48 81
10 00
        0001729d: cmpl $0x003f199c=#basicNew, %eax : 3D 9C 19
3F 00
```

```
        IsObjectReference:
                000172a2: jnz .+0x0000001f (0x000172c3=basicNew@83) :
75 1F
                000172a4: movl %ds:0x10814c(%ecx), %eax : 8B 81 4C 81
10 00
                000172aa: cmpl %ebx, %eax : 39 D8
                000172ac: jnz .+0x00000015 (0x000172c3=basicNew@83) :
75 15
                000172ae: movl %ds:0x108150(%ecx), %ebx : 8B 99 50 81
10 00
                000172b4: movl %ds:0x4(%ebx), %eax : 8B 43 04
                000172b7: movl %eax, %ecx : 89 C1
                000172b9: andl $0x00000001, %eax : 83 E0 01
                000172bc: jnz .+0xffffff9c (0x0001725a=basicNew@1A) : 75
9C
                000172be: addl $0x0000003b, %ecx : 83 C1 3B
                000172c1: jmp %ecx : FF E1
                000172c3: movl %ebx, %ecx : 89 D9
                000172c5: xorl $0x003f199c=#basicNew, %ecx : 81 F1 9C
19 3F 00
        IsObjectReference:
                000172cb: shll $1, %ecx : D1 E1
                000172cd: andl $0x00003ff0=critical:@268, %ecx : 81 E1 F0
3F 00 00
                000172d3: movl %ds:0x108148(%ecx), %eax : 8B 81 48 81
10 00
                000172d9: cmpl $0x003f199c=#basicNew, %eax : 3D 9C 19
3F 00
        IsObjectReference:
                000172de: jnz .+0x0000000a (0x000172ea=basicNew@AA)
: 75 0A
                000172e0: movl %ds:0x10814c(%ecx), %eax : 8B 81 4C 81
10 00
                000172e6: cmpl %ebx, %eax : 39 D8
                000172e8: jz .+0xffffffc4 (0x000172ae=basicNew@6E) : 74
C4
                000172ea: movl %ebx, %ecx : 89 D9
                000172ec: xorl $0x003f199c=#basicNew, %ecx : 81 F1 9C
19 3F 00
        IsObjectReference:
                000172f2: andl $0x00003ff0=critical:@268, %ecx : 81 E1 F0
3F 00 00
                000172f8: movl %ds:0x108148(%ecx), %eax : 8B 81 48 81
10 00
                000172fe: cmpl $0x003f199c=#basicNew, %eax : 3D 9C 19
3F 00
        IsObjectReference:
                00017303: jnz .+0x0000000a (0x0001730f=basicNew@CF) :
75 0A
                00017305: movl %ds:0x10814c(%ecx), %eax : 8B 81 4C 81
10 00
                0001730b: cmpl %ebx, %eax : 39 D8
                0001730d: jz .+0xffffff9f (0x000172ae=basicNew@6E) : 74
9F
                0001730f: movl %ebp, %ds:0xfffffe6c=&framePointer : 89 2D
6C FE FF FF
                00017315: movl %esp, %ds:0xfffffe68=&stackPointer : 89 25
68 FE FF FF
```

```
                0001731b: movl %ds:0x10d538, %esp : 8B 25 38 D5 10 00
                00017321: movl %ds:0x10d530, %ebp : 8B 2D 30 D5 10 00
                00017327: movl $0x00017240=basicNew@0, %ebx : BB 40
72 01 00
        IsAbsPCReference:
                0001732c: subl $0x0000000c, %esp : 83 EC 0C
                0001732f: pushl %ebx : 53
                00017330: call .+0xfffe8b17
(0xfffffe4c=&ceSendFromInLineCacheMiss:) : E8 17 8B FE FF
        IsRelativeCall:
                00017335: addb %al, %ds:(%eax) : 00 00
                00017337: addb %al, %ds:(%eax) : 00 00
                00017339: addb %al, %ds:(%eax) : 00 00
                 16r1727E IsObjectReference (16r17347)
                 16r17286 IsObjectReference (16r17346)
                 16r1728E IsObjectReference (16r17345)
                 16r172A2 IsObjectReference (16r17344)
                 16r172CB IsObjectReference (16r17342)
                 16r172DE IsObjectReference (16r17341)
                 16r172F2 IsObjectReference (16r17340)
                 16r17303 IsObjectReference (16r1733F)
                 16r1732C IsAbsPCReference (16r1733D)
                 16r17335 IsRelativeCall (16r1733C)
```

At 000172b4 through 000172c1 is the test that the method is cogged and the indirect jump to the unchecked entry-point (16r3b/0x3b) if so. 22 instructions ot dispatch in the shortest case, but more importantly at least 4 memory references and an indirect jump. At the end of the open PIC is metadata that identifies where all the object references are, and (on x86) the relative call of the run-time miss routine ceSendFromInLineCacheMiss:. The plumbing of the call into C is from 0001730f through 0001732c where we write-back stack and frame pointers to the interpreter variables and at 0001731b & 00017321 switch to the C stack, preserving platform stack alignment while passing the open PIC as the argument at 0001732c & 0001732f.

So that's almost the complete life-cycle of a send site, born unlinked, hatching as a monomorphic in-line cache, experiencing a growth spurt as a closed PIC and then maturing as an open PIC. The final steps in the life-cycle are either rebirth or death, to be unlinked, for example when a method with the send site's selector is redefined, or to disappear completely when code reclamation destroys the machine code method containing the send site. These final two steps, plus some chicanery we can play with PICs and MNUs, will wait for another post on method management and method metadata.

An Obvious Optimization

Writing a VM in a high-level language with an IDE makes exploring the VM very easy. Once I'd implemented open PICs in Cog I wondered whether it was worth skipping creating a Closed PIC when a site becomes polymorphic if an Open PIC with the site's selector already exists. The benefit is avoiding lots of code space modifications and an allocation. The downside is replacing faster Closed PIC dispatch with slower Open PIC dispatch. The question is how many send sites would be prematurely promoted to megamorphic, or how many Closed PICs have selectors for which there are Open PICs. Asking the question is easy:

```
    | cpics opics openSelectors |
    cpics := methodZone cogMethodsSelect: [:cm| cm cmType =
CMClosedPIC].
    opics := methodZone cogMethodsSelect: [:cm| cm cmType =
CMOpenPIC].
    openSelectors := (opics collect: [:ea| ea selector]) asSet.
    { cpics size. (cpics select: [:cpic| openSelectors includes: cpic
selector]) size }
     #(303 52)
    52 / 303.0 0.1716171617161716
```

Only 17% of polymorphic send sites would get prematurely promoted. So I've implemented a simple sharing scheme. The Cogit maintains a linked list of Open PICs, and before it creates a Closed PIC for a send site will patch it to an Open PIC if the list contains one for the send's selector. So here's the binder for monomorphic misses, called from ceMethodAbort if and when the class tag check fails. About half way down you'll see "pic := methodZone openPICWithSelector: targetMethod selector.".

```
    Cogit methods for inline cacheing
    ceSICMiss: receiver
        "An in-line cache check in a method has failed. The failing
entry check has jumped
         to the ceSendMiss abort call at the start of the method
which has called this routine.
         If possible allocate a closed PIC for the current and existing
classes.
         The stack looks like:
                 receiver
                 args
                 sender return address
         sp=>        ceSendMiss call return address
         So we can find the method that did the failing entry check at
            ceSendMiss call return address – sicMissReturnOffset
         and we can find the send site from the outer return
address."
        <api>
        | pic innerReturn outerReturn entryPoint targetMethod
newTargetMethodOrNil cacheTag extent result |
        <var: #pic type: #'CogMethod *'>
        <var: #targetMethod type: #'CogMethod *'>
        "Whether we can relink to a PIC or not we need to pop off
the inner return and identify the target method."
        innerReturn := coInterpreter popStack.
        targetMethod := self cCoerceSimple: innerReturn –
missOffset to: #'CogMethod *'.
        outerReturn := coInterpreter stackTop.
        entryPoint := backEnd callTargetFromReturnAddress:
outerReturn.

        newTargetMethodOrNil := self lookupAndCog: targetMethod
selector for: receiver.
        "We assume lookupAndCog:for: will *not* reclaim the
method zone"
        cacheTag := objectRepresentation
inlineCacheTagForInstance: receiver.
```

```
        ((objectRepresentation inlineCacheTagIsYoung: cacheTag)
         or: [newTargetMethodOrNil notNil
                and: [objectMemory isYoung: newTargetMethodOrNil]])
ifTrue:
            [result := self patchToOpenPICFor: targetMethod
selector
                        numArgs: targetMethod cmNumArgs
                        receiver: receiver.
            self assert: result not. "If patchToOpenPICFor:..
returns we’re out of code memory"
            ^coInterpreter ceSendFromInLineCacheMiss:
targetMethod].
        "See if an Open PIC is already available."
        pic := methodZone openPICWithSelector: targetMethod
selector.
        pic isNil ifTrue:
            ["otherwise attempt to create a closed PIC for the two
cases."
            pic := self cogPICSelector: targetMethod selector
                        numArgs: targetMethod cmNumArgs
                        Case0Method: targetMethod
                        Case1Method: newTargetMethodOrNil
                        tag: cacheTag.
            (pic asInteger between: MaxNegativeErrorCode and: –
1) ifTrue:
                ["For some reason the PIC couldn’t be generated,
most likely a lack of code memory.
                 Continue as if this is an unlinked send."
                 pic asInteger = InsufficientCodeSpace ifTrue:
                     [coInterpreter
callForCogCompiledCodeCompaction].
                ^coInterpreter ceSendFromInLineCacheMiss:
targetMethod]].
        extent := backEnd
                    rewriteCallAt: outerReturn
                    target: pic asInteger + cmEntryOffset.
        processor
            flushICacheFrom: outerReturn – 1 – extent to:
outerReturn – 1;
            flushICacheFrom: pic asInteger to: pic asInteger +
closedPICSize.
        "Jump back into the pic at its entry in case this is an MNU
(newTargetMethodOrNil is nil)"
        coInterpreter
            executeCogMethodFromLinkedSend: pic
            withReceiver: receiver
            andCacheTag: (backEnd inlineCacheTagAt:
outerReturn).
        "NOTREACHED"
        ^nil
```

Let Me Tell You All About It, Let Me Quantify

Let’s finish up by looking at the numbers. First, how many send sites of each kind are there? I’ve taken the numbers in this post from running an image throguh start-up to a point where it prompts for user input. That’s not necessarily representative or unrepresentative of Smalltalk programs in general, simply a convenient source of numbers. Actual numbers will vary according to work-load but these are not hugely at variance with my experience with the VisualWorks VM. Take them with a pinch of salt.

```
    | nmethods nsends nunlinked nlinked nmono npoly nmega |
    nmethods := nsends := nunlinked := nmono := npoly := nmega :=
0.
    methodZone methodsDo:
        [:m|
        nmethods := nmethods + 1.
        m cmType = CMMethod ifTrue:
            [self mapFor: m do:
                [:annotation :pc|
                annotation = IsSendCall ifTrue:
                    [| entryPoint target offset |
                    nsends := nsends + 1.
                    entryPoint := backEnd
callTargetFromReturnAddress: pc.
                    entryPoint > methodZoneBase
                        ifTrue: "it's a linked send"
                            [offset := (entryPoint bitAnd:
entryPointMask) = checkedEntryAlignment
                                                ifTrue:
[cmEntryOffset]
                                                ifFalse:
[cmNoCheckEntryOffset].
                            target := self cCoerceSimple:
entryPoint – offset to: #'CogMethod *'.
                            target cmType = CMMethod
ifTrue:
                                [nmono := nmono + 1].
                            target cmType = CMClosedPIC
ifTrue:
                                [npoly := npoly + 1].
                            target cmType = CMOpenPIC
ifTrue:
                                [nmega := nmega + 1]]
                        ifFalse:
                            [nunlinked := nunlinked + 1]].
                false "keep scanning"]]].
    nlinked := nsends – nunlinked.
    { nmethods. nsends. nunlinked. nmono. npoly. nmega. nunlinked /
nsends roundTo: 0.001. nmono / nlinked roundTo: 0.001. npoly / nlinked
roundTo: 0.001. nmega / nlinked roundTo: 0.001 } #(1752 6352 2409
3566 307 70 0.379 0.904 0.078 0.018)
```

nmethods	1752		
nsends	6352	(nsends / nmethods 3.63)	
nunlinked	2409	nunlinked / nsends	0.379
nmono	3566	nmono / nlinked	0.904
npoly	307	npoly / nlinked	0.078
nmega	70	nmega / nlinked	0.018

And how fast do the sends perform? Let's send the message yourself, which answers the receiver, and so compiles to the checked entry-point followed by a single return instruction, to some homogenous and inhomogenous collections:

```
    | void homogenousImmediate homogenousCompact
homogenousNormal polymorphic megamorphic |
    void := Array new: 1024 withAll: nil.
    homogenousImmediate := Array new: 1024 withAll: 1.
    homogenousCompact := Array new: 1024 withAll: 1.0.
    homogenousNormal := Array new: 1024 withAll: 1 / 2.
    polymorphic := Array new: 1024.
    1 to: polymorphic size by: 4 do:
        [:i|
        polymorphic
            at: i put: i;
            at: i + 1 put: i asFloat;
            at: i + 2 put: i / 1025;
            at: i + 3 put: i * 1.0s1 "scaled decimal"].
    megamorphic := Array new: 1024.
    1 to: megamorphic size by: 8 do:
        [:i|
        megamorphic
            at: i put: i;
            at: i + 1 put: i asFloat;
            at: i + 2 put: i / 1025;
            at: i + 3 put: i * 1.0s1; "scaled decimal"
            at: i + 4 put: i class;
            at: i + 5 put: i asFloat class;
            at: i + 6 put: (i / 1025) class;
            at: i + 7 put: (i * 1.0s1) class].
    { void. homogenousImmediate. homogenousCompact.
homogenousNormal. polymorphic. megamorphic } collect:
        [:a|
        [1 to: 100000 do: [:j| a do: [:e| e ~~ nil ifTrue: [e yourself]]]]
timeToRun]
```

The sum of three different runs on my 2.66 GHz Intel Core i7 MacBook Pro is

```
    #(8260 9121 10895 10869 11691 13389)
```

Subtracting the void case gives the following times, in nanoseconds per send (timeToRun answers milliseconds).

```
    #(8260 9121 10895 10869 11691 13389) collect: [:ea| (ea – 8260)
* 1e6 / (3 * 1024 * 100000) roundTo: 0.1]
    #(0.0 2.8 8.6 8.5 11.2 16.7)
```

homogenous immediate	2.8 nsecs
homogenous compact	8.6 nsecs
homogenous normal	8.5 nsecs
polymorphic	11.2 nsecs
megamorphic	16.7 nsecs

Closed PICs are faster than Open PICs, but the cost of class access is a significant percentage of overall send costs: 8.6 – 2.8 = 5.8, or 35% of the Open PIC lookup cost. Interesting. One thing masking the difference between the other sends and the indirect branch in Open PIC dispatch here is that in the benchmark it only has the one target Object>#yourself, and we can be confident the processor is correctly predicting the branch. The indirect jump could be much more costly in a larger working set.

Let me explain…no, there is too much. Let me sum up.

One thing I've found in moving from the VisualWorks VM, written in C, to Cog, written in Smalltalk, is that using a high-level language results in a significantly improved ability to push the design further, in no small part because programming in an IDE, with no compile-link-run cycle, yields much swifter gratification, so there's much more encouragement to ask questions of the system; they're easier to pose and the answers easier to act upon. Anything as complex as a virtual machine really *needs* to be written in a high-level language. The only question is *how*? Slang has issues, but Gerardo Richarte and Javier Burroni have found a better way. And that's a subject for a subsequent post.

Posted by admin on Tuesday, March 1st, 2011, at 7:53 pm, and filed under Cog.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment, or trackback from your site.

{ 11 }

Comments

1. **StCredZero** | 03-Mar-11 at 5:01 am | Permalink

   How about, "Just-in-time, just-in-time, VM man, code me a JIT as fast as you can?" (It scans almost as well. The n and m consonants at the ends of syllables are a bit awkward.)

   After Cog, how would you feel about about an ECMAScript server VM that can do continuations? (Or maybe help someone else do one?)

2. **cremes** | 07-Mar-11 at 11:22 am | Permalink

   I contribute a little to the Rubinius project (Ruby in Ruby with a C++ interpreter). I was chatting with the lead developer today after a bug was uncovered and fixed related to JITting code.

   I asked the question if it was possible to spec/test JIT code for the purpose of guarding against regressions. He indicated that he hasn't figured out how to do so yet, so I volunteered to ask you if you had any insight into this issue.

   Is it feasible or advisable to write specs/test code for the purposes of fleshing out a runtime's JIT capability? If so, how does one go about doing it? What are the main challenges or pitfalls and how does one overcome them?

   If this is too much for an answer in a comment, feel free to expand on the idea in a blog post. 🙂

3. **Eliot Miranda** | 13-Mar-11 at 5:50 pm | Permalink

   Hi Cremes,

It is possible to do testing, both by running unit tests above a VM and by running tests within a simulation environment such as I've built for Cog. When running unit tests above the VM the procedure is simple; simply select a test suite composed of tests designed to stress the JIT. When running tests in the context of a simulator one can both run tests on generated code without running it (that is for example how I exhaustively test bytecode to machine code pc mapping) and, similarly to running unit tests above a real VM, run (in Smalltalk's case) an image above the simulated VM. When running unit tests above a simulated VM there is scope to run sophisticated assert-checking that one might not be able to run in the real VM. This is for example how I tests stack depth at run-time call points such as sends. This is interesting and I'll expand on it here if I may.

Using Smalltalk's InstructionStream tools I can scan a bytecoded method and compute the correct stack depth at each bytecode. I can then run the simulated JIT and cause it to check that the stack depth in JITted code agrees with the stack depth in the corresponding bytecoded method. This is possible in the simulator because
a) the simulator can call out to the Smalltalk system to compute the stack depth, and
because the method to be scanned is in a Smalltalk image being run by the simulator, not a method in the Smalltalk image holding the simulator,
b) Smalltalk allows me to construct a proxy object that allows me to scan bytecoded methods in the simulated image using objects in the image holding the simulator

In an analogous but reversed way I can compile any method in the Smalltalk system holding the simulator by using a proxy object that makes a bytecoded method in the image appear to be a bytecoded method in an image loaded by the simulator.

Let me know if this is interesting enough for a blog post and I'll try and rustle one up on the topic of testing and using proxies to interface level and meta-level.

4. **cremes** | 20-Apr-11 at 10:50 am | Permalink

   Thanks for the response on how to test the JIT. I *do* think it is interesting enough for a standalone blog post, so if you have the time then I know I would be interested to read it. I also know a few language runtime developers who would love to read it too.

5. **Craig Latta** | 03-Jul-11 at 12:17 pm | Permalink

   Nice Annie Lennox reference! 🙂

6. **Mark Willis** | 17-Sep-12 at 9:39 am | Permalink

   I noticed that you were a mentor in the Smalltalk Google Summer of code for implementing an ARM jitter for Squeak VM. How far did this get?

7. **Eliot Miranda** | 17-Sep-12 at 3:23 pm | Permalink

   Hi Mark,

   Lars Wassermann is the GSOC student, and he's really strong. He implemented a plugin to a working ARM simulator, ported tests that verify the plugin runs ARM machine code, disassembles, etc. He then wrote the ARM-specific compiler class that maps the abstract machine to the ARM machine that is at the core of the JIT's code generation model. He also implemented the C calling glue that allows generated code to call the C run-time that forms the rest of the VM.

   The system gets as far as the first attempt to relink a send. This is where the issues of ARM having a RISC calling convention with a link register vs the x86 having a CISC calling convention that pushes the return pc on the stack. So there's some work to do evolving the code that manages send relinking and inline caches so that it can work either with a link register or a stacked return pc. You can see above, e.g. in ceSICMiss: that return pcs are used to locate send sites (they're calls). So some redesign of method prologue plus the relinking machinery is required to make it deal conveniently with ether form of call. This can be as simple as pushing the link register at suitable points, but since it affects performance we may want to put in some effort to make the code quick.

   Another area requiring work is in constant generation. Constants in machine code can either be true constants (masks for accessing an object header for example) or object pointers (e.g. message selectors and classes in inline caches). The ARM has no support for single-instruction generation of an arbitrary 32-bit literal, instead requiring use of shift and add or shift and or to combine 12-bit literals into full literals. This means that generating an arbitrary literal requires 4 32-bit instructions. The code generator both needs to identify short sequences for true constants, and to use out-of-line access for object pointers. The ARM supports pc-relative addressing (as does x86_64) so the code generator needs to be extended to allow object pointer constants to be generated out-of-line and accessed via pc-relative addressing.

   In short, Lars got the system to a state where it actually runs Smalltalk in the simulator, but the system is still some months effort away from running a full image and generating compact efficient code. I have to say I'm delighted by how far Lars got in such a short amount of time. I reiterate, he's really strong. You can read Lars' progress reports on the vm-dev mailing list in the archive at http://marc.info/?l=squeak-vm-dev.

Thanks for your interest!

8. **Mark Willis** | 18-Sep-12 at 11:17 am | Permalink

   Thank you for the info and the mailing list link. That sounds very encouraging. I won't pretend to understand it all though! You may have guessed that I'm approaching this from a Scratch running on a Raspberry Pi perspective. What performance improvement would you expect an ARM JIT to make to this scenario?

9. **Eliot Miranda** | 18-Sep-12 at 2:54 pm | Permalink

   Hi Mark,

   well, when we applied the Cog JIT VM at Qwaq/Teleplace we saw a three-fold improvement in frame-rate, i.e. an application improvement of a factor of three. Certain benchmarks improve by a factor of five, but an interactive application has performance-iontensive activities (graphics) that the JIT has no effect on.

10. **Mark Willis** | 19-Sep-12 at 12:20 am | Permalink

    Thanks again. Is there any scope to help this project along? I don't know Smalltalk or ARM assembler so face a huge learning curve in that respect but do know C,C++,C# and Java and x86 assembler many moons ago!

11. **The Cog VM lookup | Clément Béra** | 12-Aug-13 at 1:16 am | Permalink

    […] like to explain my understanding. Eliot Miranda has already explained some of it in his post Build me a JIT as fast as you can. His post is really good but when I first looked at it I didn't understand everything because […]

Post a Comment

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

Name		*
Email		*
Website		
Message		

Post

« SOMETIMES SMALLTALK IS TRULY AWESOME #1

AN ARRANGED MARRIAGE »