

# Cog Blog

Speeding Up Terf, Squeak, Pharo and Croquet with a fast open-source Smalltalk VM

## HOME

{ 2011 03 04 }

## An Arranged Marriage

### 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

Can an arranged marriage be a perfect one, or can only a love-match result in the truly perfect union? Is perfection when two hearts beat as one, when Kate submits to Petruchio, or when Mr. accepts that Mrs. is wearing the trousers? Personally I don't think there's such a thing as a perfect marriage. All marriages, especially the long-lived ones take work. There are complex issues to be resolved, which requires good communication and division of responsibility. And of course a marriage won't last unless the partners are fundamentally compatible. And so it is with the two senior partners in the Cog JIT VM, the ColInterpreter, older spouse full of experience (the old interpreter, facelifted for the new context, the existing primitive set), and the Cogit, the younger partner, creative tear-away, needing lots of support from their other half, but bringing real energy to the marriage.

But is it prudent, the anxious parents wonder, to arrange a marriage between these two souls in the first place? After all many JITs don't have an interpreter. Well, doing without an interpreter means having to JIT all code at all times and this can be very tricky. For example, looking back at the [Inline Cacheing](#) post, think what has to happen when a send miss requires the allocation of a Closed PIC but the system runs out of code memory and needs to do a reclamation to free up code memory. It has to keep hold of the method containing the send site and the old and new target methods while these are moving beneath its feet as the reclamation compacts the code in the machine code zone. Or consider being faced with a huge autogenerated bytecode method whose jitted form may be ten times larger. The former problem is tricky; the latter is often terminal.

*Aside: You may ask why restrict the amount of memory used to generate code? One reason is to bound footprint, but a better one is to increase performance; a smaller working set can result in far better instruction cache performance and hence better overall performance even if code reclamations are common. Still better for a dynamic language is that a limited contiguous code space is extremely easy to manage, for example when scanning all send sites in machine code to unlink sends when methods are redefined. Hence the way I've always implemented JITs is to compile code in a fixed-size space (the size of which can be determined at startup). It works well.*

Further, specific to Cog, the precursor StackInterpreter already had lots of infrastructure I needed to either use directly or carry forward in some way, such as the context-to-stack mapping machinery and the existing primitive set. So a ColInterpreter solves a number of problems. It is already the home for the existing primitive set; it is a fall-back to interpreted execution at times when it would be tricky to continue in machine code (such as during machine code reclamations); it allows

avoiding jitting code in unproductive cases (seldom used or huge methods).

## Stacked to Attract

The most fundamental area of compatibility is the stack, and we choose two different formats to allow interpreter and JIT to share the Smalltalk stack, and carefully manage the C stack to allow the Cogit to call on services in the CoInterpreter. If you haven't already done so you might like to read [the post on context to stack mapping](#) that gives an in-depth account of Smalltalk stack management in Cog. In the StackInterpreter there was a single frame format:

```

                                receiver for method activations/closure for block
activations
                                arg0
                                ...
                                argN
                                caller's method ip/base frame's sender context
fp->    saved fp
                                method
                                frame flags (num args, is block flag, has context flag)
                                context (uninitialized)
                                receiver
                                first temp
                                ...
sp->    Nth temp
```

I was alarmed to read in [the paper on the HiPE \(High-Performance Erlang\) VM](#) that the implementors decided to keep the interpreter and machine code stacks separate. The paper mentions the apparently many bugs they had keeping the two different kinds of frames on the same stack, that the co-located scheme was abandoned and then goes into some detail explaining the difficulties keeping the two stacks separate imposes. If some thought is given to how the two kinds of frame can interface I contend there is manageable complexity and nothing particularly hairy about the approach and I offer Cog as an existence proof.

The two keys to the interpreter frame format are a) that it support the existing interpreter, hence having much the same layout as the above and b) that an interpreter caller frame can be returned to via a native return instruction executed by a machine code callee frame. Returning from machine code to a specific bytecode in an interpreted method can't be easily done with a single return pointer (it *can* be done if one is prepared to synthesize a thunk of machine code for each particular return, but this doesn't seem very sensible to me). Instead, the native return address can be that of a sequence of code that trampolines control back into the interpreter. We must be able to return to the interpreter from many points; for example in the middle of a process switch primitive, called from machine code that finds that the target process has an interpreted frame for its suspendedContext. Hence we establish a setjmp handler immediately prior to entry into the interpreter and return to the interpreter via longjmp. I'll come back to [cogit ceCaptureCStackPointers](#) later.

*CoInterpreter methods for initialization*

**enterSmalltalkExecutivImplementation**

"Main entry-point into the interpreter at each execution level,  
where an execution

level is either the start of execution or reentry for a callback.  
 Capture the C stack  
 pointers so that calls from machine-code into the C run-time  
 occur at this level.  
 This is the actual implementation, separated from  
 enterSmalltalkExecutive so the  
 simulator can wrap it in an exception handler and hence  
 simulate the setjmp/longjmp."  
 <inline: false>  
 cogit assertCStackWellAligned.  
 cogit ceCaptureCStackPointers.  
 "Setjmp for reentry into interpreter from elsewhere, e.g.  
 machine-code trampolines."  
 self sigset: reenterInterpreter jmp: 0.  
 (self isMachineCodeFrame: framePointer) ifTrue:  
 [self returnToExecutive: false postContextSwitch: true  
 "NOTREACHED"].  
 self setMethod: (self ifframeMethod: framePointer).  
 instructionPointer = cogit ceReturnToInterpreterPC ifTrue:  
 [instructionPointer := self ifframeSavedIP:  
 framePointer].  
 self assertValidExecutionPointe: instructionPointer r:  
 framePointer s: stackPointer imbar: true.  
 self interpret.  
 ^0

In Smalltalk the setjmp/longjmp pair is simulated via exception handling

```
Notification subclass: #ReenterInterpreter
  instanceVariableNames: 'returnValue'
  classVariableNames: "
  poolDictionaries: "
  category: 'VMMaker-JITSimulation'
```

*CoInterpreter methods for cog jit support*

```
sigset: aJumpBuf jmp: sigSaveMask
  "Hack simulation of sigsetjmp/siglongjmp.
  Assign to reenterInterpreter the exception that when
  raised simulates a longjmp back to the interpreter."
  <doNotGenerate>
  reenterInterpreter := ReenterInterpreter new returnValue: 0;
  yourself.
  ^0
```

```
siglong: aJumpBuf jmp: returnValue
  "Hack simulation of sigsetjmp/siglongjmp.
  Signal the exception that simulates a longjmp back to the
  interpreter."
  <doNotGenerate>
  aJumpBuf == reenterInterpreter ifTrue:
    [self assertValidExecutionPointe: instructionPointer r:
  framePointer s: stackPointer imbar: true].
  aJumpBuf returnValue: returnValue; signal
```

*CoInterpreter methods for initialization*

```
enterSmalltalkExecutive
  "Main entry-point into the interpreter at each execution level,
  where an
  execution level is either the start of execution or reentry for
```

a callback."

```
<macro: '() enterSmalltalkExecutiveImplementation()'>
"Simulation of the setjmp in
enterSmalltalkExecutiveImplementation for reentry into interpreter."
[[self enterSmalltalkExecutiveImplementation]
 on: ReenterInterpreter
 do: [:ex| ex return: ex returnValue]] =
ReturnToInterpreter] whileTrue
```

We then save the bytecode pc to at which to resume interpretation in an additional interpreter frame slot. Hence in Cog an interpreter frame has the following format:

```

receiver for method activations/closure for block
activations
  arg0
  ...
  argN
  caller's saved ip/this stackPage (for a base frame)
fp-> saved fp
      method
      context (initialized to nil)
      frame flags (num args, is block, has context)
      saved method bytecode pc
      receiver
      first temp
      ...
sp->  Nth temp
```

*CogInterpreter methods for trampolines*

**ceReturnToInterpreter:** anOop

```
<api>
self assert: ((objectMemory isIntegerObject: anOop) or:
[objectMemory addressCouldBeObj: anOop]).
self assert: (self isMachineCodeFrame: framePointer) not.
self setStackPageAndLimit: stackPage.
self setMethod: (self ifFrameMethod: framePointer).
self assertValidExecutionPointe: (self ifFrameSavedIP:
framePointer)
  r: framePointer
  s: stackPointer
  imbar: true.
instructionPointer := self ifFrameSavedIP: framePointer.
self push: anOop.
self siglong: reenterInterpreter jmp: ReturnToInterpreter.
"NOTREACHED"
^nil
```

Since with the JIT the VM spends the bulk of its time in machine code and since the JIT can use knowledge such as the method's argument count at compile-time we can make the machine code frame two slots smaller:

```

receiver for method activations/closure for block
activations
  arg0
  ...
  argN
  caller's saved ip/this stackPage (for a base frame)
```

```

fp->    saved fp
        machine-code method and lsb flags (has context, is
block)
        context (initialized to nil)
        receiver
        first temp
        ...
sp->    Nth temp

```

Since machine code methods are aligned on an 8-byte boundary the least significant three bits of the method address are zero and can be used for three flags, two of which are used (had context and is block); see **initializeFrameIndices** below. The smallest frame, a zero-argument method with no local temps hence looks like

```

        receiver
        caller's saved ip
fp->    saved fp
        machine-code method and lsb flags
        context
sp->    receiver

```

The minimum frame size is important; it defines the maximum number of contexts that may be allocated to flush a stack page to the heap, for which the garbage collector maintains reserve free space. Compared to a typical C frame layout method and context are extra. The former makes finding the metadata associated with a machine code frame (its argument count etc) trivial since all this is accessible via the machine code method. The context slot is required to allow the Smalltalk-specific mapping of stack frames to contexts. Of course the two formats mean that the ColInterpreter, which naturally inherits from the StackInterpreter must provide different offsets for the receiver, and in a number of cases must implement two variants of an accessor depending on the frame format:

ColInterpreter class methods for initialization

#### **initializeFrameIndices**

"Format of a stack frame. Word-sized indices relative to the frame pointer.

##### Terminology

Frames are either single (have no context) or married (have a context).

Contexts are either single (exist on the heap), married (have a context) or widowed (had a frame that has exited).

Stacks grow down:

```

        receiver for method activations/closure for block
activations
        arg0
        ...
        argN
        caller's saved ip/this stackPage (for a base
frame)
fp->    saved fp
        method
        context (initialized to nil)
        frame flags (interpreter only)
        saved method ip (initialized to 0; interpreter only)
        receiver

```

first temp  
...  
sp-> Nth temp

In an interpreter frame  
frame flags holds  
the number of arguments (since argument  
temporaries are above the frame)  
the flag for a block activation  
and the flag indicating if the context field is valid  
(whether the frame is married).  
saved method ip holds the saved method ip when the  
callee frame is a machine code frame.

This is because the saved method ip is actually the  
ceReturnToInterpreterTrampoline address.

In a machine code frame  
the flag indicating if the context is valid is the least  
significant bit of the method pointer  
the flag for a block activation is the next most  
significant bit of the method pointer

Interpreter frames are distinguished from method frames by  
the method field which will  
be a pointer into the heap for an interpreter frame and a  
pointer into the method zone for  
a machine code frame.

The first frame in a stack page is the baseFrame and is  
marked as such by a saved fp being its stackPage,  
in which case the first word on the stack is the caller context  
(possibly hybrid) beneath the base frame."

```
| fxCallerSavedIP fxSavedFP fxMethod fxIFrameFlags  
fxThisContext fxIFReceiver fxMFReceiver fxIFSavedIP |  
fxCallerSavedIP := 1.  
fxSavedFP := 0.  
fxMethod := -1.  
fxThisContext := -2.  
fxIFrameFlags := -3. "Can find numArgs, needed for  
fast temp access. args are above fxCallerSavedIP.  
Can find "is block" bit  
Can find "has context" bit"  
fxIFSavedIP := -4.  
fxIFReceiver := -5.  
fxMFReceiver := -3.
```

"For debugging nil out values that differ in the  
StackInterpreter."

```
FrameSlots := nil.  
IFrameSlots := fxCallerSavedIP - fxIFReceiver + 1.  
MFrameSlots := fxCallerSavedIP - fxMFReceiver + 1.
```

```
FoxCallerSavedIP := fxCallerSavedIP * BytesPerWord.  
"In Cog a base frame's caller context is stored on the first  
word of the stack page."
```

```
FoxCallerContext := nil.  
FoxSavedFP := fxSavedFP * BytesPerWord.  
FoxMethod := fxMethod * BytesPerWord.  
FoxThisContext := fxThisContext * BytesPerWord.  
FoxFrameFlags := nil.
```

```

FoxIframeFlags := fxIframeFlags * BytesPerWord.
FoxIFSavedIP := fxIFSavedIP * BytesPerWord.
FoxReceiver := #undeclared asSymbol.
FoxIReceiver := fxIReceiver * BytesPerWord.
FoxMReceiver := fxMReceiver * BytesPerWord.

```

"N.B. There is room for one more flag given the current 8 byte alignment of methods (which is at least needed to distinguish the checked and unchecked entry points by their alignment."

```

MFMethodFlagHasContextFlag := 1.
MFMethodFlagsBlockFlag := 2.
MFMethodFlagsMask := MFMethodFlagHasContextFlag +
MFMethodFlagsBlockFlag.
MFMethodMask := (MFMethodFlagsMask + 1) negated

```

*ColInterpreter methods for frame access*

**frameMethodObject:** theFP

```

<inline: true>
<var: #theFP type: #'char *>
^(self isMachineCodeFrame: theFP)
  ifTrue: [(self mframeHomeMethod: theFP)
methodObject]
  ifFalse: [self iframeMethod: theFP]

```

**frameReceiver:** theFP

```

<inline: true>
<var: #theFP type: #'char *>
^(self isMachineCodeFrame: theFP)
  ifTrue: [self mframeReceiver: theFP]
  ifFalse: [self iframeReceiver: theFP]

```

**iframeReceiver:** theFP

```

<inline: true>
<var: #theFP type: #'char *>
^stackPages longAt: theFP + FoxIReceiver

```

**mframeReceiver:** theFP

```

<inline: true>
<var: #theFP type: #'char *>
^stackPages longAt: theFP + FoxMReceiver

```

**iframeNumArgs:** theFP

```

"See encodeFrameFieldHasContext:numArgs:"
<inline: true>
<var: #theFP type: #'char *>
^stackPages byteAt: theFP + FoxIframeFlags + 1

```

**mframeNumArgs:** theFP

```

^(self mframeCogMethod: theFP) cmNumArgs

```

etc.

Returning from the interpreter to machine-code is quite straight-forward; the interpreter can test the return pc and since all machine code pcs are lower than the object heap (another convenient consequence of having a fixed-size code zone) the interpreter simply compares against the heap base (localIP asUnsignedInteger < objectMemory startOfMemory):

*ColInterpreter methods for return bytecodes*

**commonCallerReturn**

```

"Return to the previous context/frame (sender for method
activations, caller for block activations)."
| callersFPOrNull |
<var: #callersFPOrNull type: #'char *'>
<sharedCodeNamed: 'commonCallerReturn' inCase: 125>
"returnTopFromBlock"

callersFPOrNull := self frameCallerFP: localFP.
callersFPOrNull == 0 "baseFrame" ifTrue:
    [self assert: localFP = stackPage baseFP.
     ^self baseFrameReturn].

localIP := self frameCallerSavedIP: localFP.
localSP := localFP + (self frameStackedReceiverOffset:
localFP).
localFP := callersFPOrNull.
localIP asUnsignedInteger < objectMemory startOfMemory
ifTrue:
    [localIP asUnsignedInteger ~= cogit
ceReturnToInterpreterPC ifTrue:
    ["localIP in the cog method zone indicates a
return to machine code."
     ^self returnToMachineCodeFrame].
    localIP := self pointerForOop: (self iframeSavedIP:
localFP)].
    self internalStackTopPut: localReturnValue.
    self setMethod: (self iframeMethod: localFP).
    ^self fetchNextBytecode

returnToMachineCodeFrame
"Return to the previous context/frame after assigning localIP,
localSP and localFP."
<inline: true>
cogit assertCStackWellAligned.
self assert: localIP asUnsignedInteger < objectMemory
startOfMemory.
self assert: (self isMachineCodeFrame: localFP).
self assertValidExecutionPointe: localIP asUnsignedInteger
r: localFP s: localSP imbar: false.
self internalStackTopPut: localIP.
self internalPush: localReturnValue.
self externalizeFPandSP.
cogit ceEnterCogCodePopReceiverReg
"NOTREACHED"

```

The `baseFrameReturn` phrase is necessary since the Smalltalk stack is actually organized as a set of small (4kb) stack pages, reasons for which are explained in the stack mapping post. Handling a base frame return in machine code is tricky since by the time the return instruction has executed the frame has already been torn down (the frame pointer set to the saved frame pointer, which in the case of a base frame is zero). Base frames are connected to other pages or the rest of the sender context chain via a reference to the sender context (which may itself be a context on the sender stack page). So we must be able to locate the spouse context and sender context of a base frame after the base frame has been torn down. The uppermost top two words on a stack page hold these two contexts (<blush>and yes I should define accessors for these</blush>). Again the use of a special return address



causes the VM to jump to the code for a machine code base frame return.

*CoInterpreter methods for trampolines*

**ceBaseFrameReturn:** `returnValue`

"Return across a stack page boundary. The context to return to (which may be married)

is stored in the first word of the stack. We get here when a return instruction jumps

to the `ceBaseFrameReturn:` address that is the return pc for base frames. A consequence

of this is that the current frame is no longer valid since an interrupt may have overwritten

its state as soon as the stack pointer has been cut-back beyond the return pc. So to have

a context to send the `cannotReturn:` message to we also store the base frame's context

in the second word of the stack page."

<api>

| contextToReturnTo contextToReturnFrom isAContext  
thePage newPage frameAbove |

<var: #thePage type: #'StackPage \*'>

<var: #newPage type: #'StackPage \*'>

<var: #frameAbove type: #'char \*'>

contextToReturnTo := stackPages longAt: stackPage  
baseAddress.

"The stack page is effectively free now, so free it. We must free it to be

correct in determining if contextToReturnTo is still married, and in case

makeBaseFrameFor: cogs a method, which may cause a code compaction,

in which case the frame must be free to avoid the relocation machinery

tracing the dead frame. Since freeing now temporarily violates the page-list

ordering invariant, use the assert-free version."

stackPages freeStackPageNoAssert: stackPage.

isAContext := self isContext: contextToReturnTo.

(isAContext

and: [self isStillMarriedContext: contextToReturnTo])

ifTrue:

[framePointer := self frameOfMarriedContext:

contextToReturnTo.

thePage := stackPages stackPageFor:

framePointer.

framePointer = thePage headFP

ifTrue:

[stackPointer := thePage headSP]

ifFalse:

["Returning to some interior frame, presumably because of a sender assignment.

Move the frames above to another page (they may be in use, e.g. via coroutining).

Make the interior frame the top frame."

frameAbove := self findFrameAbove:

```

framePointer inPage: thePage.
    "Since we've just deallocated a page
we know that newStackPage won't deallocate an existing one."
    newPage := self newStackPage.
    self assert: newPage = stackPage.
    self moveFramesIn: thePage through:
frameAbove toPage: newPage.
    stackPages
markStackPageMostRecentlyUsed: newPage.
    framePointer := thePage headFP.
    stackPointer := thePage headSP]]

ifFalse:
    [(isAContext
    and: [objectMemory isIntegerObject:
(objectMemory

    fetchPointer: InstructionPointerIndex

    ofObject: contextToReturnTo))] ifFalse:
        [contextToReturnFrom := stackPages
longAt: stackPage baseAddress - BytesPerWord.
    self
tearDownAndRebuildFrameForCannotReturnBaseFrameReturnFrom:
contextToReturnFrom
    to: contextToReturnTo
    returnValue: returnValue.
    ^self externalCannotReturn: returnValue
from: contextToReturnFrom].
    "void the instructionPointer to stop it being
incorrectly updated in a code
    compaction in makeBaseFrameFor:."
    instructionPointer := 0.
    thePage := self makeBaseFrameFor:
contextToReturnTo.
    framePointer := thePage headFP.
    stackPointer := thePage headSP].
    self setStackPageAndLimit: thePage.
    self assert: (stackPages stackPageFor: framePointer) =
stackPage.
    (self isMachineCodeFrame: framePointer) ifTrue:
        [self push: returnValue.
*        cogit ceEnterCogCodePopReceiverReg.
        "NOTREACHED"].
    instructionPointer := self stackTop.
    instructionPointer = cogit ceReturnToInterpreterPC ifTrue:
        [instructionPointer := self iframeSavedIP:
framePointer].
    self setMethod: (self iframeMethod: framePointer).
    self stackTopPut: returnValue. "a.k.a. pop saved ip then push
result"
    self assert: (self checkIsStillMarriedContext:
contextToReturnTo currentFP: framePointer).
*    self siglong: reenterInterpreter jmp: ReturnToInterpreter.
    "NOTREACHED"
    ^nil

```

So between an interpreter caller and a machine code callee the return address must be ceReturnToInterpreterTrampoline and the caller's

ifframeSavedIP: must be valid, whereas between a machine-code caller and an interpreted callee the machine code return address is fine. Similarly, the convention for the top frame on other than the active stack page is that it always contain a machine-code pc; ceReturnToInterpreter for an interpreted frame and the machine code return address for a machine code frame. The return pc for a base frame is ceBaseFrameReturnTrampoline.

## Managing the C Stack

So far we've seen how trampolines are used to jump from machine-code to C, and how setjmp/longjmp can be used to get back into the interpreter from any point. But how do we manage the C stack so that we can call into the run-time and know that the C frame containing the setjmp is valid? Since the C stack is contiguous and grows in one direction (typically down) – at least on the OS's Cog runs on today (Windows, Mac OS X, linux) – all that's needed is to record the C stack in **enterSmalltalkExecutivImplementation** via cogit [ceCaptureCStackPointers](#). Any jump into machine code will occur from a point further down the C stack. Any jump back to the C stack discards those intervening frames, cutting the C stack back to the activation of **enterSmalltalkExecutivImplementation**. What we *mustn't* do is capture the C stack pointers every time we jump into machine-code. That could cause uncontrolled stack growth.

*Cogit methods for initialization*

### **generateStackPointerCapture**

"Generate a routine [ceCaptureCStackPointers](#) that will capture the C stack pointer, and, if it is in use, the C frame pointer. These are used in trampolines to call run-time routines in the interpreter from machine-code."

```
| oldMethodZoneBase oldTrampolineTableIndex |
self assertCStackWellAligned.
oldMethodZoneBase := methodZoneBase.
oldTrampolineTableIndex := trampolineTableIndex.
self generateCaptureCStackPointers: true.
self perform: #ceCaptureCStackPointers asSymbol.
(cFramePointerInUse := self isCFramePointerInUse) ifFalse:
    [methodZoneBase := oldMethodZoneBase.
     trampolineTableIndex := oldTrampolineTableIndex.
     self generateCaptureCStackPointers: false]
```

### **generateCaptureCStackPointers: captureFramePointer**

"Generate the routine that writes the current values of the C frame and stack pointers into variables. These are used to establish the C stack in trampolines back into the C run-time."

This is a presumptuous quick hack for x86. It is presumptuous for two reasons. Firstly the system's frame and stack pointers may differ from those we use in generated code, e.g. on register-rich RISCs. Secondly the ABI may not support a simple frameless call as written here (for example 128-bit stack alignment on Mac OS X)."

```
| startAddress |
```

```

<inline: false>
self allocateOpcodes: 32 bytecodes: 0.
initialPC := 0.
endPC := numAbstractOpcodes - 1.
startAddress := methodZoneBase.
captureFramePointer ifTrue:
    [self MoveR: FPReg Aw: self cFramePointerAddress].
"Capture the stack pointer prior to the call."
backEnd leafCallStackPointerDelta = 0
    ifTrue: [self MoveR: SPReg Aw: self
cStackPointerAddress]
    ifFalse: [self MoveR: SPReg R: TempReg.
self AddCq: backEnd
leafCallStackPointerDelta R: TempReg.
self MoveR: TempReg Aw: self
cStackPointerAddress].
self RetN: 0.
self outputInstructionsForGeneratedRuntimeAt:
startAddress.
self recordGeneratedRunTime: 'ceCaptureCStackPointers'
address: startAddress.
ceCaptureCStackPointers := self cCoerceSimple:
startAddress to: #'void (*)(void)'

isCFramePointerInUse
<doNotGenerate>
"This should be implemented externally, e.g. in
sqPlatMain.c."
^true

```

and from e.g. platforms/Mac OS/vm/sqMacMain.c

```

#if COGVM
/*
 * Support code for Cog.
 * a) Answer whether the C frame pointer is in use, for capture of the C stack
 * pointers.
 */
# if defined(i386) || defined(__i386) || defined(__i386__)
/*
 * Cog has already captured CStackPointer before calling this routine. Record
 * the original value, capture the pointers again and determine if CFramePointer
 * lies between the two stack pointers and hence is likely in use. This is
 * necessary since optimizing C compilers for x86 may use %ebp as a general-
 * purpose register, in which case it must not be captured.
 */
int
isCFramePointerInUse()
{
    extern unsigned long CStackPointer, CFramePointer;
    extern void (*ceCaptureCStackPointers)(void);
    unsigned long currentCSP = CStackPointer;

    currentCSP = CStackPointer;
    ceCaptureCStackPointers();
    assert(CStackPointer < currentCSP);
    return CFramePointer >= CStackPointer && CFramePointer <= currentCSP;
}

```

```

}
# endif /* defined(i386) || defined(__i386) || defined(__i386__) */
#endif /* COGVM */

```

So two variables define the C stack, CStackPointer and CFramePointer. On x86 the machine-code for `ceCaptureCStackPointers` looks like the following; the `addl $4, %eax` skips the return pc.

```

ceCaptureCStackPointers:
    00000418: movl %ebp, %ds:0x10d530 : 89 2D 30 D5 10 00
0x10d530 = &CFramePointer
    0000041e: movl %esp, %eax : 89 E0
    00000420: addl $0x00000004, %eax : 83 C0 04
    00000423: movl %eax, %ds:0x10d538 : A3 38 D5 10 00
0x10d538 = &CStackPointer
    00000428: ret : C3
    00000429: nop : 90
    0000042a: nop : 90
    0000042b: nop : 90
    0000042c: nop : 90
    0000042d: nop : 90
    0000042e: nop : 90
    0000042f: nop : 90

```

The C stack needs to be captured and restored on callbacks. Lets follow this through assuming a call to the FFI primitive from machine-code. First of all the machine-code method calls the interpreter's FFI primitive, using the current values of CFramePointer and CStackPointer as established in a previous **enterSmalltalkExecutivImplementation** invocation. The primitive then marshalls arguments to the C stack and calls out to external code. If that code calls-back into the VM the C stack between the FFI call-out and the call-back must be preserved for the VM to return from the call-back. When the callback enters the VM it must do so via **enterSmalltalkExecutivImplementation**, and set-up a new `reenterInterpreter jmpbuf` for jumping into the interpreter at this level. So `callbackEnter` must both save and restore the C stack pointers *and* the `reenterInterpreter jmpbuf`.

*CoInterpreter methods for callback support*

**callbackEnter:** `callbackID`

```

    "Re-enter the interpreter for executing a callback"
    | currentCStackPointer currentCFramePointer
savedReenterInterpreter
    wasInMachineCode calledFromMachineCode |
    <volatile>
    <export: true>
    <var: #currentCStackPointer type: #'void *'>
    <var: #currentCFramePointer type: #'void *'>
    <var: #callbackID type: #'sqlInt *'>
    <var: #savedReenterInterpreter type: #'jmp_buf'>

    "Check if we've exceeded the callback depth"
    (self assert: jmpDepth < MaxJumpBuf) iffFalse:
        [^false].
    jmpDepth := jmpDepth + 1.

    wasInMachineCode := self isMachineCodeFrame:
framePointer.

```

```

calledFromMachineCode := instructionPointer <=
objectMemory startOfMemory.

    "Suspend the currently active process"
    suspendedCallbacks at: jmpDepth put: self activeProcess.
    "We need to preserve newMethod explicitly since it is not
activated yet
    and therefore no context has been created for it. If the caller
primitive
    for any reason decides to fail we need to make sure we
execute the correct
    method and not the one 'last used' in the call back"
    suspendedMethods at: jmpDepth put: newMethod.
    self transferTo: self wakeHighestPriority from:
CSCallbackLeave.

    "Typically, invoking the callback means that some
semaphore has been
    signaled to indicate the callback. Force an interrupt check as
soon as possible."
    self forceInterruptCheck.

i.e. from here:
    "Save the previous CStackPointers and interpreter entry
jmp_buf."
    currentCStackPointer := cogit getCStackPointer.
    currentCFramePointer := cogit getCFramePointer.
    self mem: (self cCoerceSimple: savedReenterInterpreter to:
#'void *')
        cp: reenterInterpreter
        y: (self sizeof: #'jmp_buf' asSymbol).
    cogit assertCStackWellAligned.
    (self setjmp: (jmpBuf at: jmpDepth)) == 0 ifTrue: "Fill in
callbackID"
        [callbackID at: 0 put: jmpDepth.
        *
        self enterSmalltalkExecutive.
        self assert: false "NOTREACHED"].

    "Restore the previous CStackPointers and interpreter entry
jmp_buf."
    cogit setCStackPointer: currentCStackPointer.
    cogit setCFramePointer: currentCFramePointer.
    self mem: reenterInterpreter
        cp: (self cCoerceSimple: savedReenterInterpreter to:
#'void *')
        y: (self sizeof: #'jmp_buf' asSymbol).
:upto here
    "Transfer back to the previous process so that caller can
push result"
    self putToSleep: self activeProcess yieldingIf:
preemptionYields.
    self transferTo: (suspendedCallbacks at: jmpDepth) from:
CSCallbackLeave.
    newMethod := suspendedMethods at: jmpDepth.    "see
comment above"
    argumentCount := self argumentCountOf: newMethod.
    self assert: wasInMachineCode = (self
isMachineCodeFrame: framePointer).
    calledFromMachineCode

```

```

        ifTrue:
            [instructionPointer >= objectMemory
startOfMemory ifTrue:
                [self ifframeSavedIP: framePointer put:
instructionPointer.
                    instructionPointer := cogit
ceReturnToInterpreterPC]]
        ifFalse:
            ["Even if the context was flushed to the heap and
rebuilt in transferTo:from:
                above it will remain an interpreted frame
because the context's pc would
                remain a bytecode pc. So the instructionPointer
must also be a bytecode pc."
            self assert: (self isMachineCodeFrame:
framePointer) not.
            self assert: instructionPointer > objectMemory
startOfMemory].
        self assert: primFailCode = 0.
        jmpDepth := jmpDepth-1.
        ^true

```

when the return from callback is done longjmp-ing back to the setjmp the C stack and reenterInterpreter set-up in **enterSmalltalkExecutiveImplementation** is cut-back and restored to their previous values.

## Keeping Up Appearances

Of course while the relationship between ColInterpreter and Cogit has to work in private (i.e. when compiled to C and executed in use) it also has to put on a good show in social settings. When being developed, simulated machine code must also be able to gain intimate access the ColInterpreter's internals. This is on the one hand about faithful simulation, and on the other hand about the difference between, in the production vm, machine code calling into C code, and, in the simulator, turning a machine code call into a Smalltalk message send to a given receiver. This needs to be done when simulating run-time calls, and so we may as well use the same mechanism for accessing ColInterpreter variables as well. This of course takes us into the heart of the machine-code simulator, and I think this bit is particularly interesting.

A simple way of breaking-out of machine code is simply to dole out illegal addresses for variables and run-time routines:

```

    Cogit methods for initialization
    simulatedAddressFor: anObject
        "Answer a simulated address for a block or a symbol. This is
an address that
            can be called, read or written by generated machine code,
and will be mapped
            into a Smalltalk message send or block evaluation."
        <doNotGenerate>
        ^simulatedAddresses
            at: anObject
            ifAbsentPut: [(simulatedAddresses size + 101 *
BytesPerWord) negated bitAnd: self addressSpaceMask]

    simulatedVariableAddress: getter in: receiver
        "Answer a simulated variable. This is a variable whose value

```

can be read

by generated machine code."

<doNotGenerate>

| address |

address := self simulatedAddressFor: getter.

simulatedVariableGetters

at: address

ifAbsentPut: [MessageSend receiver: receiver selector:

getter].

^address

**simulatedReadWriteVariableAddress:** getter in: receiver

"Answer a simulated variable. This is a variable whose value

can be read

and written by generated machine code."

<doNotGenerate>

| address |

address := self simulatedVariableAddress: getter in: receiver.

simulatedVariableSetters

at: address

ifAbsentPut:

[| setter |

setter := (getter, ':') asSymbol.

[:value| receiver perform: setter with: value]].

^address

*CoInterpreter methods for trampoline support*

**framePointerAddress**

<api>

<returnTypeC: #usqInt>

^self cCode: [(self addressOf: framePointer)

asUnsignedInteger]

inSmalltalk: [cogit

simulatedReadWriteVariableAddress: #framePointer in: self]

*Cogit methods for trampoline support*

**genLoadStackPointers**

"Switch back to the Smalltalk stack. Assign SPReg first  
because typically it is used immediately afterwards."

self MoveAw: coInterpreter stackPointerAddress R: SPReg.

self MoveAw: coInterpreter framePointerAddress R: FPRReg.

^0

**genSaveStackPointers**

self MoveR: FPRReg Aw: coInterpreter framePointerAddress.

self MoveR: SPReg Aw: coInterpreter stackPointerAddress.

^0

The Bochs simulator will raise an exception whenever machine code tries to access an out-of-bounds address, which the internals of the Bochs plugin catch and turn into a primitive failure:

platforms/Cross/plugins/BochsIA32Plugin/sqBochsIA32Plugin.cpp:

int

runCPUInSizeMinAddressReadWrite(void \*cpu, void \*memory, ulong byteSize,

ulong minAddr, ulong

minWriteMaxExecAddr)



```

{
    BX_CPU_C *anx86 = (BX_CPU_C *)cpu;

    if (anx86 != &bx_cpu)
        return BadCPUInstance;
    theMemory = (unsigned char *)memory;
    theMemorySize = byteSize;
    minReadAddress = minAddr;
    minWriteAddress = minWriteMaxExecAddr;
*   if ((theErrorAcorn = setjmp(anx86->jmp_buf_env)) != 0) {
*       anx86->gen_reg[BX_32BIT_REG_EIP].dword.ery = anx86->prev_rip;
*       return theErrorAcorn;
    }

    blidx = 0;
    bx_cpu.sregs[BX_SEG_REG_CS].cache.u.segment.limit_scaled
        = minWriteMaxExecAddr > 0 ? minWriteMaxExecAddr - 1 : 0;
    bx_cpu.sregs[BX_SEG_REG_DS].cache.u.segment.limit_scaled =
    bx_cpu.sregs[BX_SEG_REG_SS].cache.u.segment.limit_scaled = byteSize;
    bx_cpu.sregs[BX_SEG_REG_CS].cache.u.segment.limit = minWriteMaxExecAddr
>> 16;

    bx_cpu.sregs[BX_SEG_REG_DS].cache.u.segment.limit =
    bx_cpu.sregs[BX_SEG_REG_SS].cache.u.segment.limit = byteSize >> 16;
    anx86->eipFetchPtr = theMemory;
    anx86->eipPageWindowSize = minWriteMaxExecAddr;
    bx_pc_system.kill_bochs_request = 0;
    anx86->cpu_loop(0 /* = "run forever" until exception or interrupt */);
*   if (anx86->stop_reason != STOP_NO_REASON) {
*       anx86->gen_reg[BX_32BIT_REG_EIP].dword.ery = anx86->prev_rip;
*       if (theErrorAcorn == NoError)
*           theErrorAcorn = ExecutionError;
*       return theErrorAcorn;
    }
    return blidx == 0 ? 0 : SomethingLoggedError;
}

...
void BX_CPU_C::access_read_linear(bx_address laddr, unsigned len, unsigned curr_pl,
unsigned xlate_rw, void *data)
{
    if (laddr < minReadAddress
        || laddr + len > theMemorySize)
        longjmp(bx_cpu.jmp_buf_env, MemoryBoundsError);
    last_read_address = laddr; /* for RMW write cycles below */
    memcpy(data, theMemory + laddr, len);
}

```

Up in Smalltalk the primitive failure gets turned into a suitable exception:

*BochsIA32Alien methods for primitives*

**primitiveRunInMemory:** **memoryArray** "<Bitmap|ByteArray>"

**minimumAddress:** **minimumAddress** "<Integer>"

**readOnlyBelow:** **minimumWritableAddress** "<Integer>"

"Run the receiver using the argument as the store. Origin the argument at 0. i.e. the first byte of the

**memoryArray** is address 0. Make addresses below **minimumAddress** illegal. Convert out-of-range

call, jump and memory read/writes into register instructions into **ProcessorSimulationTrap** signals."

```

<primitive:
'primitiveRunInMemoryMinimumAddressReadWrite' module:
'BochsIA32Plugin' error: ec>
    ^ec == #'inappropriate operation'
    ifTrue: [self handleExecutionPrimitiveFailureIn:
memoryArray
                                minimumAddress: minimumAddress
                                readOnlyBelow:
minimumWritableAddress]
    ifFalse: [self reportPrimitiveFailure]

BochsIA32Alien methods for error handling
handleExecutionPrimitiveFailureIn: memoryArray "
<Bitmap|ByteArray>"
    minimumAddress: minimumAddress "<Integer>"
    readOnlyBelow: minimumWritableAddress "<Integer>"
    "Handle an execution primitive failure. Convert out-of-range
call and absolute
    memory read into register instructions into
ProcessorSimulationTrap signals."
    "self printIntegerRegistersOn: Transcript"
    "self printRegistersOn: Transcript"
    | pc opcode |
    ((pc := self eip) between: minimumAddress and:
memoryArray byteSize - 1) ifTrue:
        [opcode := memoryArray byteAt: pc + 1.
        ^self
            perform: (OpcodeExceptionMap at: opcode + 1)
            with: pc
            with: memoryArray
            with: minimumWritableAddress].
    ^self reportPrimitiveFailure

BochsIA32Alien class methods for class initialization
initialize
    "BochsIA32Alien initialize"
    PostBuildStackDelta := 0.
    OpcodeExceptionMap := Array new: 256 withAll:
#handleExecutionPrimitiveFailureAt:in:readOnlyBelow:.
    OpcodeExceptionMap
        at: 1 + self basicNew callOpcode put:
#handleCallFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew jmpOpcode put:
#handleJmpFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew retOpcode put:
#handleRetFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew movALObOpcode put:
#handleMovALObFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew movObALOpcode put:
#handleMovObALFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew movGvEvOpcode put:
#handleMovGvEvFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew movEvGvOpcode put:
#handleMovEvGvFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew movGbEbOpcode put:
#handleMovGbEbFailureAt:in:readOnlyBelow;;
        at: 1 + self basicNew movEbGbOpcode put:
#handleMovEbGbFailureAt:in:readOnlyBelow:

```

and e.g.

*BochsIA32Alien methods for error handling*

```
handleCallFailureAt: pc "<Integer>"
  in: memoryArray "<Bitmap|ByteArray>"
  readOnlyBelow: minimumWritableAddress "<Integer>"
  "Convert an execution primitive failure for a call into a
  ProcessorSimulationTrap signal."
  | relativeJump |
  relativeJump := memoryArray longAt: pc + 2 bigEndian:
false.
  ^((ProcessorSimulationTrap
    pc: pc
    nextpc: pc + 5
    address: (pc + 5 + relativeJump) signedIntToLong
    type: #call)
    signal

handleMovGvEvFailureAt: pc "<Integer>"
  in: memoryArray "<Bitmap|ByteArray>"
  readOnlyBelow: minimumWritableAddress "<Integer>"
  "Convert an execution primitive failure for a register load into
  a ProcessorSimulationTrap signal."
  | modrmByte |
  ^(((modrmByte := memoryArray byteAt: pc + 2) bitAnd:
16rC7) = 16r5) "ModRegInd & disp32"
    ifTrue:
      [(ProcessorSimulationTrap
        pc: pc
        nextpc: pc + 6
        address: (memoryArray
unsignedLongAt: pc + 3 bigEndian: false)
        type: #read
        accessor: (#(eax: ecx: edx: ebx: esp:
ebp: esi: edi:) at: ((modrmByte >> 3 bitAnd: 7) + 1)))
        signal]
    ifFalse:
      [self reportPrimitiveFailure]
```

The exception handler for ProcessorSimulationTraps is in the Cogit's core machine-code simulation entry-point. I'm going to include it warts and all, since I think there's lots of interest here. The exception handler almost right at the end.

*Cogit methods for simulation only*

```
simulateCogCodeAt: address "<Integer>"
  <doNotGenerate>
  | stackZoneBase |
  stackZoneBase := coInterpreter stackZoneBase.
  processor eip: address.
  [[singleStep ifTrue:
    [[processor sp < stackZoneBase ifTrue: [self halt].
    self recordRegisters.
    printRegisters ifTrue:
      [processor printRegistersOn: coInterpreter
transcript]].
    self recordLastInstruction.
    printInstructions ifTrue:
      [Transcript nextPutAll: lastNInstructions last; cr;
```

```

flush].

(breakPC isInteger
  ifTrue:
    [processor pc = breakPC
      and: [breakBlock value: self]]
  ifFalse:
    [breakBlock value: self] ifTrue:
      ["printRegisters := printInstructions := true"
       "self reportLastNInstructions"
       "coInterpreter printExternalHeadFrame"
       "coInterpreter printFrameAndCallers:
coInterpreter framePointer SP: coInterpreter stackPointer"
       "coInterpreter shortPrintFrameAndCallers:
coInterpreter framePointer"
       "coInterpreter printFrame: processor fp WithSP:
processor sp"
       "coInterpreter printFrameAndCallers: processor
fp SP: processor sp"
       "coInterpreter shortPrintFrameAndCallers:
processor fp"
       "self disassembleMethodFor: processor pc"
       coInterpreter changed: #byteCountText.
       self halt: 'machine code breakpoint at ',
                    (breakPC isInteger
                      ifTrue: [breakPC hex]
                      ifFalse: [String
streamContents: [:s| breakBlock decompile printOn: s indent: 0]])]
value]. "So that the Debugger's Over steps over all this"
singleStep
  ifTrue: [processor
            singleStepIn: coInterpreter memory
            minimumAddress: guardPageSize
            readOnlyBelow: coInterpreter
cogCodeSize]
  ifFalse: [processor
            runInMemory: coInterpreter memory
            minimumAddress: guardPageSize
            readOnlyBelow: coInterpreter
cogCodeSize].

((printRegisters or: [printInstructions]) and: [clickConfirm])
ifTrue:
  [(self confirm: 'continue?') ifFalse:
    [self halt]].
true] whileTrue]
  on: ProcessorSimulationTrap
  do: [:ex| self handleSimulationTrap: ex].
true] whileTrue

```

One of the first things to notice is that it's peppered with useful expressions I can evaluate when in the debugger. The second thing to notice is that when single-stepping this is like the In-Circuit-Emulator from the 22nd century. You can define just about any break-point function you can imagine and assign it to breakBlock, and it'll be evaluated every instruction. The single-stepper can capture the last N instructions and register states so that one can look back at the instructions executed immediately before an error. It will print individual instructions and the registers to the transcript. It makes machine-code

debugging verge on the enjoyable, and certainly leaves things like gdb in the dust.

OK, so here's the core exception handler

*Cogit methods for simulation only*

```
handleSimulationTrap: aProcessorSimulationTrap
    <doNotGenerate>
    aProcessorSimulationTrap type caseOf:
        { [#read] -> [self handleReadSimulationTrap:
aProcessorSimulationTrap].
        [#write] -> [self handleWriteSimulationTrap:
aProcessorSimulationTrap].
        [#call] -> [self handleCallOrJumpSimulationTrap:
aProcessorSimulationTrap].
        [#jump] -> [self handleCallOrJumpSimulationTrap:
aProcessorSimulationTrap] }
```

and e.g.

```
handleReadSimulationTrap: aProcessorSimulationTrap
    <doNotGenerate>
    | variableValue |
    variableValue := (simulatedVariableGetters at:
aProcessorSimulationTrap address) value asInteger.
    processor
        perform: aProcessorSimulationTrap registerAccessor
        with: variableValue signedIntToLong.
    processor pc: aProcessorSimulationTrap nextpc
```

and (gulp)

```
handleCallOrJumpSimulationTrap: aProcessorSimulationTrap
    <doNotGenerate>
    | evaluable function result savedFramePointer
savedStackPointer savedArgumentCount rpc |
    evaluable := simulatedTrampolines at:
aProcessorSimulationTrap address.
    function := evaluable
        isBlock ifTrue: ['aBlock; probably
some plugin primitive']
        ifFalse: [evaluable selector].
    function ~~ #ceBaseFrameReturn: ifTrue:
        [coInterpreter assertValidExternalStackPointers].
    (function beginsWith: 'ceShort') ifTrue:
        [^self perform: function with:
aProcessorSimulationTrap].
    aProcessorSimulationTrap type = #call
        ifTrue:
            [processor
                simulateCallOf: aProcessorSimulationTrap
address
                    nextpc: aProcessorSimulationTrap nextpc
                    memory: coInterpreter memory.
                self recordInstruction: {'(simulated call of '.
aProcessorSimulationTrap address. '/'. function. ')'}]
        ifFalse:
            [processor
                simulateJumpCallOf:
```

```

aProcessorSimulationTrap address
    memory: coInterpreter memory.
    self recordInstruction: {'(simulated jump to '.
aProcessorSimulationTrap address. '/. function. ')}].
    savedFramePointer := coInterpreter framePointer.
    savedStackPointer := coInterpreter stackPointer.
    savedArgumentCount := coInterpreter argumentCount.
    result := ["self halt: evaluable selector."
        evaluable valueWithArguments: (processor

postCallArgumentsNumArgs: evaluable numArgs

in: coInterpreter memory)]
    on: ReenterMachineCode
    do: [:ex| ex return: ex returnValue].

coInterpreter assertValidExternalStackPointers.
    "Verify the stack layout assumption
compileInterpreterPrimitive: makes, provided we've
    not called something that has built a frame, such as closure
value or evaluate method, or
    switched frames, such as primitiveSignal, primitiveWait,
primitiveResume, primitiveSuspend et al."
    (function beginsWith: 'primitive') ifTrue:
        [coInterpreter primFailCode = 0
            ifTrue: [(#(
                primitiveClosureValue
primitiveClosureValueWithArgs primitiveClosureValueNoContextSwitch
                primitiveSignal primitiveWait
primitiveResume primitiveSuspend primitiveYield

                primitiveExecuteMethodArgsArray primitiveExecuteMethod
                    primitivePerform
primitivePerformWithArgs primitivePerformInSuperclass
                    primitiveTerminateTo
primitiveStoreStackp primitiveDoPrimitiveWithArgs)
                    includes: function) ifFalse:
                        [self assert: savedFramePointer
= coInterpreter framePointer.
                        self assert: savedStackPointer
+ (savedArgumentCount * BytesPerWord)
= coInterpreter
stackPointer]]

                    ifFalse:
                        [self assert: savedFramePointer =
coInterpreter framePointer.
                        self assert: savedStackPointer =
coInterpreter stackPointer]].
        result ~~ #continueNoReturn ifTrue:
            [self recordInstruction: {'(simulated return to '.
processor retpcIn: coInterpreter memory. ')}].
        rpc := processor retpcIn: coInterpreter memory.
        self assert: (rpc >= codeBase and: [rpc < methodZone
zoneLimit]).
        processor
            smashCallerSavedRegistersWithValuesFrom:
16r80000000 by: BytesPerWord;
            simulateReturnIn: coInterpreter memory].
        self assert: (result isInteger "an oop result"

```

```

or: [result == coInterpreter
or: [result == objectMemory
or: [#(nil continue continueNoReturn) includes:
result]]]).
processor cResultRegister: (result
    ifNil: [0]
    ifNotNil: [result isInteger
        ifTrue:
            [result]
        ifFalse:
            [16rF00BA222]])

```

Now **handleCallOrJumpSimulationTrap**: is anything but simple, but it does a lot. Its main complication comes from assert-checking. Most of the time when the simulator sends a message to the CoInterpreter (e.g. that of a run-time routine) we expect that the stack frame has not changed when we return. But for a particular set of primitives and run-time routines that isn't so and hence the stack assertions must be avoided in their case. On return to machine code any caller-saved registers could contain arbitrary values, and on reentry to machine code any and all registers could contain arbitrary values so we smash them to avoid values persisting in the simulated processor from the time of a simulated call. This is also one place where the simulation of enilopmarts, jumps back into machine-code, has to be handled specially to avoid uncontrolled Smalltalk stack growth, hence the signal of ReenterMachineCode.

*Cogit methods for simulation only*

```

simulateEnilopmart: enilopmartAddress numArgs: n
    <doNotGenerate>
    "Enter Cog code, popping the class reg and receiver from
the stack
    and then returning to the address beneath them.
    In the actual VM the enilopmart is a function pointer and so
sends
    of this method end up calling the enilopmart to enter
machine code.
    In simulation we either need to start simulating execution (if
we're in
the interpreter) or return to the simulation (if we're in the run-
time
called from machine code. We should also smash the
register state
since, being an abnormal entry, no saved registers will be
restored."
    self assert: (coInterpreter isOnRumpCStack: processor sp).
    self assert: ((coInterpreter stackValue: n) between:
guardPageSize and: methodZone zoneLimit - 1).
    (printInstructions or: [printRegisters]) ifTrue:
        [coInterpreter printExternalHeadFrame].
    processor
        smashRegistersWithValuesFrom: 16r80000000 by:
BytesPerWord;
        simulateLeafCallOf: enilopmartAddress
        nextpc: 16rBADF00D
        memory: coInterpreter memory.
    "If we're already simulating in the context of machine code
then

```

```

        this will take us back to handleCallSimulationTrap:.
Otherwise
    start executing machine code in the simulator."
    (ReenterMachineCode new returnValue:
#continueNoReturn) signal.
    self simulateCogCodeAt: enilopmartAddress.
    "We should either longjmp back to the interpreter or
    stay in machine code so control should not reach here."
    self assert: false

```

With the above machinery machine code can freely access variables and methods of the written-in -Smalltalk simulated VM. So for example, here's the trampoline that calls ceReturnToInterpreter:


```

ceReturnToInterpreterTrampoline:
    00000958: movl %ebp, %ds:0xfffffe6c=&framePointer : 89
2D 6C FE FF FF
    0000095e: movl %esp, %ds:0xfffffe68=&stackPointer : 89 25
68 FE FF FF
    00000964: movl %ds:0x10d538, %esp : 8B 25 38 D5 10 00
    0000096a: movl %ds:0x10d530, %ebp : 8B 2D 30 D5 10 00
    00000970: subl $0x0000000c, %esp : 83 EC 0C
    00000973: pushl %edx : 52
    00000974: call .+0xffff49f
(0xfffffe18=&ceReturnToInterpreter:) : E8 9F F4 FF FF
    00000979: movl %ds:0xfffffe68=&stackPointer, %esp : 8B 25
68 FE FF FF
    0000097f: movl %ds:0xfffffe6c=&framePointer, %ebp : 8B 2D
6C FE FF FF
    00000985: ret : C3
    00000986: nop : 90
    00000987: nop : 90

```

To save time when simulating I hold CStackPointer and CFramePointer in the byte array that holds the entire heap. But framePointer & stackPointer, instance variables of the ColInterpreter, and ceReturnToInterpreter:, a method in the ColInterpreter all have illegal addresses and get accessed via the ProcessorSimulationTrap exception machinery above. The end result is that in the simulator I generate exactly equivalent machine code to that executed by the production generated-to-C VM and I'm pretty confident that this approach has resulted in far fewer code generation bugs in unsimulated code than otherwise. Almost all machine code generation bugs show up in the simulator, and as you can imagine it's quite a powerful development tool.

So an arranged marriage, certainly, with some complex communications issues, but a successful one I think.

	Send article as PDF	<input type="text" value="Enter email address"/>	<input type="button" value="Send"/>
---	---------------------	--	-------------------------------------

Posted by admin on Friday, March 4th, 2011, at 8:32 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.



## Comments

1. **Ben Coman** | 05-Sep-13 at 5:57 pm | [Permalink](#)

> The single-stepper can capture the last N instructions and register states so that one can look back at the instructions executed immediately before an error. It will print individual instructions and the registers to the transcript. It makes machine-code debugging verge on the enjoyable, and certainly leaves things like gdb in the dust.

Not that I've done any machine-code debugging in the last 20 years, but that is particularly interesting. Perhaps a video of that in operation would be attractive to those outside the Smalltalk community – perhaps in conjunction with efforts to port Cog to ARM & embedded platforms where perhaps people deal with machine code more often.

2. **Eliot Miranda** | 06-Sep-13 at 10:32 am | [Permalink](#)

Ben, I must do this. It's a great suggestion. Thanks.

### Post a Comment

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

Name  \*

Email  \*

Website

Message

« [BUILD ME A JIT AS FAST AS YOU CAN...](#) [A SPUR GEAR FOR COG](#) »