{ 2013 09 05 }

A Spur gear for Cog

I've got tentative permission at my day job to work on improving Cog's performance, specifically garbage collection. The existing Squeak GC isn't much to write home about; it has an interesting attempt at space optimization but is rather slow because it's based on a pointer-reversal scan-mark-compact algorithm which, while it avoids the space overhead of a mark stack (implemented in the pointer reversal), ends up writing every field scanned three times, twice during scan-mark, and once during compaction, which is expensive. Generation scavenging by comparison writes each field just once. We have recent benchmark results at Cadence that show the VM spending 40% of its time in garbage collection (!!). While this isn't a good comparison, in my experience with the VisualWorks VM, HPS, I saw overheads in the 2% to 5% range on memory-intensive loads. In any case a good GC should be a *lot* cheaper than 40% of entire execution time.

The Squeak object representation also isn't that spiffy. It has three header sizes, from a one to a two word header. one-word header objects must be instances of up to 31 classes (the "compact classes"); there is a bit field in the first header word and if non-zero the class of the object is in a 31-element Array off to the side. This is a good idea but as we'll see it doesn't go far enough. If the size of the object is less than 254 bytes (and not one of the compact classes), it has a two word header, the class reference taking up a word. Large objects have a three-word header with the full size taking a word. The parts of the VM that deal with object access (inst var access, object indexing etc) need to be insulated from this variety; it would be a really bad idea if one had to adjust an inst var offset by the header size whenever fetching the inst var of an object, so an object pointer always points at the first word of the header. But a memory manager must enumerate objects (e.g. privately for compaction, or as a service for allInstances) and hence needs to be able to look at the word following an object and tell what kind of header its got. So the Squeak format puts the header size in the least significant two bits of all header fields. Hence when fetching the class of a two- or three-word header the VM must mask off the least significant two bits.

All this complexity has two main impacts on the Cog JIT. First, the inline cache probe at the beginning of each method, the code that checks that the receiver's class is what's expected, is about twice as long as that in HPS because it must cope with compact classes and with stripping the least-significant two bits from a full class field. Second, implementing basicNew or at:put: in machine code is a too tedious to contemplate; every object instantiation (including block creation) and every at:put: involves calling out of machine code into the interpreter's run-time to use the existing Squeak VM code, which is doubly slow, slow because the switch from machine code to C involves switching stacks, and slow because the existing run-time code is already complex, given that the object representation is intrinsically complex.

So revamping the garbage collector and changing the object representation appear to offer about a factor of two in performance, in part from cutting into the 40% GC overhead, and in part from the object representation, a much more efficient implementation of at:put:, basicNew and block creation, and a shorter simpler method prologue. If you look at the computer language benchmark game you'll see VisualWorks is roughly twice as fast as Squeak on Cog.

OK then, so what does this mean concretely? I'm changing the garbage collector and the object representation, a project I've nicknamed Spur (geddit??). The GC is being replaced by a generation scavenger, which is well-known technology. The object representation is influenced by the scheme I came up with when implementing a 64-bit version of VisualWorks, and I'll spend some time describing it here. The main problem with a 64-bit Smalltalk object representation is that blowing a whole word on a class reference implies at least a 16-byte header, 8 bytes for the class and 8 bytes for the remaining information, and that's a lot of memory to spend on an object header, given that statically the average Smalltalk object has about 7 inst vars (there are a lot of methods in the image), and dynamically probably a lot less (things like Point and Rectangle, which have two inst vars each, are going to be instantiated much more frequently using the IDE than methods). And the way out is … the class index idea. Use a field in the header to contain a class index and hold classes off to the side. Given that Smalltalk VMs of all stripes, interpreted or compiled, contain extensive method cacheing, the slower class access isn't an issue. The VM uses the class index as the cache tag in the inline and global method caches, indirecting to fetch the actual class object only when method lookup in the cache fails and the class hierarchy must be searched (slow anyway, adding an extra indirection won't be noticed), or in the class primitive.

There are knock-on advantages. Class indexes are constant; the GC doesn't update them when class objects are moved during garbage collection, and so they don't need to be visited in in-line caches in machine code during garbage collection. Class indexes are constant; when instantiating a well-known class such as Array, Message or BlockClosure, the VM doesn't need to go look up the class object in a table, it simply uses the appropriate class index. But, you ask, isn't general instantiation really slow because with a class in hand, wanting to instantiate it, don't we have to search the class table looking for the class to determine its class index? No…

Another issue in the Squeak system, and many Smalltalk VMs that use a GC that moves objects, is the identityHash. The system must provide a per-object hash that doesn't change as an object moves. In the original 16-bit Smalltalk-80 implementation there was a fixed object table containing object headers, each with a pointer to their object body. The body could move but the header never did and so the object's address could serve as its hash. But in 32-bit VisualWorks there's a 14-bit hash field and in Squeak it's only 11 bits, and that means lots of collisions in IdentityDictionary et al. So in my class index design there's a much bigger identityHash field, and intentionally it is the same size (*) as the class index, and the system arranges that a class's identityHash is also its index in the class table. The first time a class gets instantiated or sent a message or put in an identity hash table the VM notices the class has no identityHash yet, finds an unused slot in the class table, and assigns this index as the class's identityHash field. So to instantiate a class the VM copies the class's identityHash to the new instance's classIndex field; simple. (*) *in fact it needs only to be as large as the class index. If one needs header bits or a better identityHash one could reduce the size of the classIndex field and increase the identityHash field. All that's needed is that a classIndex fit in the identityHash field.*

One really good idea in the Squeak object representation is the format field. This is a 4-bit per-object field that defines the object's basic type, is it pointers, bits, indexable, etc, and if bytes, how many unused bytes in the object, the "odd bytes", given that its header size is a word count. The existing Squeak format is

```
ObjectMemory methods for header access
formatOf: oop
" 0 no fields
1 fixed fields only (all containing pointers)
2 indexable fields only (all containing pointers)
3 both fixed and indexable fields (all containing pointers)
4 both fixed and indexable weak fields (all containing pointers).

5 unused
6 indexable word fields only (no pointers)
7 indexable long (64-bit) fields (only in 64-bit images)

8-11 indexable byte fields only (no pointers) (low 2 bits are low 2 bits of size)
12-15 compiled methods:
# of literal oops specified in method header,
followed by indexable bytes (same interpretation of low 2 bits as above)
"
	<inline: true>
	^((self baseHeader: oop) >> 8) bitAnd: 16rF
```

What's good here is that a lot of properties, usually implemented as single-bit flags, some of which are exclusive, are combined in a single field. These bits would be a pointers bit, unset in byte objects, an indexable bit, 0 in e.g. Point, (and often stored in the class, from where its slow to get, slowing down at: & at:put:), an isWeak bit, set in WeakArray et al, an Ephemeron bit, and two bits for the number of odd bytes. So instead of 5 or 6 bits we have 4, and these bits are often related during at: and at:put: (isPointers and odd bytes are both relevant).

So I'm keeping this scheme, but extending it to cope with 64-bit objects. Another design goal of Spur is to share as much of the object representation between a 32-bit and a 64-bit implementation as possible. In VisualWorks there's a lot of difference between the two, 32-bits having direct class pointers, 64-bits having class indexes, and that means a complex code base that's sometimes hard to read. Hence the Spur format field is

```
Spur32BitMemoryManager methods for header access
formatOf: objOop
    "0 = 0 sized objects (UndefinedObject True False et al)
     1 = non-indexable objects with inst vars (Point et al)
     2 = indexable objects with no inst vars (Array et al)
     3 = indexable objects with inst vars (MethodContext AdditionalMethodState et al)
     4 = weak indexable objects with inst vars (WeakArray et al)
     5 = weak non-indexable objects with inst vars (ephemerons) (Ephemeron)
     6,7,8 unused
     9 (?) 64-bit indexable
     10 - 11 32-bit indexable
     12 - 15 16-bit indexable
     16 - 23 byte indexable
     24 - 31 compiled method"
    self flag: #endianness. "longAt: objOop + self wordSize in a big-endian version"
    ^(self longAt: objOop) >> self formatShift bitAnd: self formatMask
```

Now we can define the complete Spur header. There are two formats; an 8-byte header, common between the 32-bit and 64-bit implementations, and a 16-byte header which has the overflow size in an 8-byte field prepended to the standard 8-byte header:

```
SpurMemoryManager methods for header format
headerForSlots: numSlots format: formatField classIndex: classIndex
    "The header format in LSB is
     MSB:   | 8: numSlots          | (on a byte boundary)
            | 2 bits               |
            | 22: identityHash     | (on a word boundary)
            | 3 bits               |
            | 5: format            | (on a byte boundary)
            | 2 bits               |
            | 22: classIndex       | (on a word boundary) : LSB
     The remaining bits (7) need to be used for
            isGrey
            isMarked
            isRemembered
            isPinned
            isImmutable
     leaving 2 unused bits."
    <returnTypeC: #usqLong>
    ^ (numSlots << self numSlotsFullShift)
    + (formatField << self formatShift)
    + classIndex
```

That’s room for 4 million classes, 4 million identityHashes, and objects with up to 1020 bytes before they need an overflow size field (2040 in the 64-bit system). While the 4 million class limit will be breached some day, it’ll stand for a few years yet, so it shouldn’t be as alarming as Bill’s famous 64k byte remark. For object parsing the overflow size word also contains a numSlots field. If the numSlots field is maxed out at 255 slots, then there’s an overflow size word and its numSlots is also 255. If the word following an object has 255 in the most significant byte then that following object has a 16-byte header, 8 bytes otherwise.
This is a work in progress so I’ve yet to write the instantiation routine, but hopefully you can see its a lot simpler than this:

```
ObjectMemory methods for allocation
instantiateClass: classPointer indexableSize: size
    "NOTE: This method supports the backward-compatible split instSize field of the
    class format word. The sizeHiBits will go away and other shifts change by 2
    when the split fields get merged in an (incompatible) image change."
    <api>
    | hash header1 header2 cClass byteSize format binc header3 hdrSize sizeHiBits bm1 classFormat |
    <inline: false>
    self assert: size >= 0. "'cannot have a negative indexable field count"
    hash := self newObjectHash.
    classFormat := self formatOfClass: classPointer.
    "Low 2 bits are 0"
    header1 := (classFormat bitAnd: 16r1FF00) bitOr: (hash bitAnd: HashMaskUnshifted) << HashBitsOffset.
    header2 := classPointer.
    sizeHiBits := (classFormat bitAnd: 16r60000) >> 9.
    cClass := header1 bitAnd: CompactClassMask. "compact class field from format word"
    byteSize := (classFormat bitAnd: SizeMask + Size4Bit) + sizeHiBits.
            "size in bytes -- low 2 bits are 0"
    "Note this byteSize comes from the format word of the class which is pre-shifted
            to 4 bytes per field. Need another shift for 8 bytes per word..."
    byteSize := byteSize << (ShiftForWord-2).
    format := self formatOfHeader: classFormat.
    format < 8
        ifTrue:
            [format = 6
                ifTrue: ["long32 bitmaps"
                    bm1 := BytesPerWord-1.
                    byteSize := byteSize + (size * 4) + bm1 bitAnd: LongSizeMask. "round up"
                    binc := bm1 - ((size * 4) + bm1 bitAnd: bm1). "odd bytes"
                    "extra low bit (4) for 64-bit VM goes in 4-bit (betw hdr bits and sizeBits)"
                    header1 := header1 bitOr: (binc bitAnd: 4)]
                ifFalse: [byteSize := byteSize + (size * BytesPerWord) "Arrays and 64-bit bitmaps"]]
        ifFalse:
            ["Strings and Methods"
            bm1 := BytesPerWord-1.
            byteSize := byteSize + size + bm1 bitAnd: LongSizeMask. "round up"
            binc := bm1 - (size + bm1 bitAnd: bm1). "odd bytes"
            "low bits of byte size go in format field"
            header1 := header1 bitOr: (binc bitAnd: 3) << 8.
            "extra low bit (4) for 64-bit VM goes in 4-bit (betw hdr bits and sizeBits)"
            header1 := header1 bitOr: (binc bitAnd: 4)].
    byteSize > 255 "requires size header word/full header"
        ifTrue: [header3 := byteSize. hdrSize := 3]
        ifFalse: [header1 := header1 bitOr: byteSize. hdrSize := cClass = 0 ifTrue: [2] ifFalse: [1]].
    ^self allocate: byteSize headerSize: hdrSize h1: header1 h2: header2 h3: header3 doFill: true format: format

allocate: byteSize headerSize: hdrSize h1: baseHeader h2: classOop h3: extendedSize doFill: doFill format: format
    "Allocate a new object of the given size and number of header words. (Note: byteSize already
     includes space for the base header word.) Initialize the header fields of the new object and
     fill the remainder of the object with a value appropriate for the format. May cause a GC"

    | newObj remappedClassOop |
    <inline: true>
    <var: #i type: 'usqInt'>
```

```
<var: #end type: 'usqInt'>
"remap classOop in case GC happens during allocation"
hdrSize > 1 ifTrue: [self pushRemappableOop: classOop].
newObj := self allocateChunk: byteSize + (hdrSize - 1 * BytesPerWord).
hdrSize > 1 ifTrue: [remappedClassOop := self popRemappableOop].

hdrSize = 3
    ifTrue: [self longAt: newObj put: (extendedSize bitOr: HeaderTypeSizeAndClass).
        self longAt: newObj + BytesPerWord put: (remappedClassOop bitOr: HeaderTypeSizeAndClass).
        self longAt: newObj + (BytesPerWord*2) put: (baseHeader bitOr: HeaderTypeSizeAndClass).
        newObj := newObj + (BytesPerWord*2)].

hdrSize = 2
    ifTrue: [self longAt: newObj put: (remappedClassOop bitOr: HeaderTypeClass).
        self longAt: newObj + BytesPerWord put: (baseHeader bitOr: HeaderTypeClass).
        newObj := newObj + BytesPerWord].

hdrSize = 1
    ifTrue: [self longAt: newObj put: (baseHeader bitOr: HeaderTypeShort)].
"clear new object"
doFill ifTrue:
    [| fillWord end i |
     fillWord := format <= 4
                    ifTrue: [nilObj] "if pointers, fill with nil oop"
                    ifFalse: [0].
     end := newObj + byteSize.
     i := newObj + BytesPerWord.
     [i < end] whileTrue:
        [self longAt: i put: fillWord.
         i := i + BytesPerWord]].
^newObj
```

But will it blend? I've bootstrapped from the old format to the new format so I can show you what happens to the size of the heap. Take a guess. There is additional overhead when compared to the existing Squeak object representation. A further departure is that objects are always a multiple of 8 bytes, and always have at least one field past the header for forwarding, specifically for implementing lazy become:. So in the 32-bit system a zero-sized object occupies 16 bytes, 8 bytes for the header, and 4 bytes for the forwarding pointer, rounded up to 8 bytes to preserve 64-bit alignment. In the 64-bit system it's of course also 16 bytes, one extra 8 byte field for the forwarding pointer. And in Spur, Characters are 30-bit immediate values, with tag pattern 2, whereas SmallIntegers are unchanged, 31-bit immediate values with tag pattern 1. Let's compare the sizes of the heaps; for this I'm using a Squeak4.3 image containing 342533 objects occupying 15585400 bytes, 45.5 bytes, or 11.37 words per object.
The new image contains 348104 objects, occupying 17844440 bytes. There are fewer Characters but some extra space for the root and two pages of the sparse class table. So there's a significant increase in heap size of 14.5%, but not a huge increase given Spurs other advantages (*). Let's take a look at the overheads

```
| l s z o d r |
l := s := z := o := d := 0.
self allObjectsDo:
    [:j| | n |
    n := self numSlotsOf: j.
    n odd ifTrue: [d := d + 1].                    "d is number of odd-word sized objects, 138,919"
    n >= self numSlotsMask
        ifTrue: [l := l + 1]                   "l is number of large objects with an overflow size field, 963"
        ifFalse:
            [n = 0
                ifTrue: [z := z + 1]             "z = zero-sized objects, 10,642"
                ifFalse:
                    [n = 1
                        ifTrue: [o := o + 1]  "o = one-slot sized objects, including 1,2,3 & 4 byte strings, 54,502"
                        ifFalse:
                            [s := s + 1]]]]. "s = small objects with no overflow size field, 281,997"
r := { l. s. o. z }.                            "sum of r is total number of objects"
r, {d}, (r, {d} collect: [:v| v * 100.0 / r sum roundTo: 0.01]), { r sum }

=> #(963 281997 54502 10642 138919 0.28 81.01 15.66 3.06 39.91 348104)
```

So the overhead for the forwarding pointer is z * 8 = (10642 * 8), a percentage overhead of 54376 * 100.0 / 17844440 or 0.48%. The overhead for rounding-up to 64-bits is d * 4 = 138919 * 4, a percentage overhead of 3.1%. The number of zero-sized objects rounded up to make room for their forwarding pointer is 10642, an overhead of 10642 * 8 * 100 / 17844440, or 0.48%. Interestingly the number of odd slot objects is 15.7%, significantly less than the expected 50%. Do we software engineers like powers of two or does binary computing encourage them? The saving on Characters is almost negligible; there are only 256 characters in the image I started with (no static occurrences of wide characters), that's 768 bytes + 1032 bytes for the Character table. So in summary, the forwarding pointer and 64-bit alignment impose a 3.6% overhead. Spur loses space because it uses 8 bytes of header for almost all objects whereas the old format's 1-word header for small instances of the compact classes manages to squeeze fully 62.5% of objects into the small header (the Squeak 4.3 image I used has 213947 1-word header out of 342533 objects). So given how effective the old scheme is, 14.5% heap growth isn't so bad.

(*) *an earlier version of this post contained a dreadful clerical error. I mistakenly subtracted the size of an empty space from the total Spur heap size and hence claimed* *a -2.4% shrinkage in heap size. Apologies; forgive me. One of the hazards of web publishing is the lack of review.*

I'd like to dedicate Spur to Andreas Raab, my dear friend and challenging and supportive colleague, who gave me the chance to implement Cog in the first place. Andreas died so young of a stroke earlier this year, but wonderfully he has a son. Kathleen may your life with Theodor be filled with joy! Andreas I miss you.

Posted by admin on Thursday, September 5th, 2013, at 10:31 am, and filed under Cog, Spur.
Follow any responses to this entry with the RSS 2.0 feed.
You can post a comment, or trackback from your site.

{ 15 } Comments

1. **Ryan Macnak** | 05-Sep-13 at 8:23 pm | Permalink

   Good to see work started on the new object representation!

   Does HPS have mixed-format objects like Squeak? E.g., named and indexable pointers (MethodContext) or indexable pointers and bytes (CompiledMethod). Do you think the savings in space is worth the complexity in at:[put:] or GC decoding the format?

2. **Philippe Marschall** | 06-Sep-13 at 12:41 am | Permalink

   We have anecdotal evidence that Seaside request handling is GC limited as well. Removing one or two #streamContents: from the request handling loop gives us about 100 req/s more.

3. **John Maloney** | 06-Sep-13 at 2:53 am | Permalink

   Nice design! Thanks for writing up your ideas and includes all those stats. I'm starting on a new project for which we're building an object memory and I'd come to some of the same conclusions that you have, such having only one header size and referring the classes by index rather than address. A radical idea I got from Dan (long ago) is to eliminate identity hash and replace hash tables with sorted arrays (sorting on the address of the key objects) as the way to implement identity sets and dictionaries. That can work with a Squeak-style compactor since the order of objects in memory never changes. It might not work with your generational GC. What do you think of that idea? If you're willing, I'd love to get your opinions on some of the other VM implementation choices I'm considering.

4. **Klaus D. Witzel** | 06-Sep-13 at 3:51 am | Permalink

   Brilliant plan to rethink GC for the Cog VM, very much appreciated.

   Do you also think about an accessible (and decidable) root of the world? In Smalltalk of the old days the receiver of #snapshot:andQuit: was the Smalltalk dictionary (represents the current image and runtime environment, etc) and therefore this (as receiver) was the root node of the object tree at the time of the ultimate GC.

   Not so in Squeak, IIRC their ultimate GC began at the specialObjectsArray (which includes the Smalltalk object and the rather "hard" pointers to the zero-size objects as they are known to the running VM).

   Can I have the original back please 🙂 This would bring back the possibility to make an arbitrary clone of the image, in-image, dramatically different just for its own #snapshot:andQuit: and without affecting any other part of the currently running image (until, of course, it stopped by the intended andQuit: true).

5. **Eliot Miranda** | 06-Sep-13 at 8:12 am | Permalink

   Hi Ryan,

   yes, its great to get started. this has been potential work for a long time. Yes, hps has indexable objects with named fields. Ironically they're *not* used for MethodContext. There is a separate stack array in an inst var for that (IIRC). But OrderedCollection has the canonical representation, two named inst vars, firstIndex and lastIndex followed by some number of indexable inst vars. Set, Dictionary et al all have the same flat representation with tally as a named inst var and some number of following indexable inst vars. This works because become: is cheap, by virtue of the fact that in HPS all objects have an indirection pointer. As you know, a cheap become: is needed with the flat representation because become: is the only way to grow the number of indexable fields; when a collection encapsulates a separate array holding its indexable fields it can simply allocate a new array. So in HPS oldSpace is a set of segments, and each segment has an object table of object headers growing from one end towards object bodies growing from the other end. In newSpace object headers are contiguous with their bodies until a become: alters things.

   But CompiledMethod is not hybrid. The second inst var is bytecodes, and notionally a ByteArray holds the byte codes. However, to save space bytecodes and up to one literal can be SmallIntegers so that methods with 6 bytecodes or less don't use a separate ByteArray, saving much space. This reasonably works because HPS is a pure JIT. Interpreting such a format is likely a bitch, and hence I didn't consider it for Cog; the Cog project started with the StackInterpreter to deliver results as early as possible ad so efficient interpretation was a given, even if the JIT now spends almost no time in the interpreter. A further disadvantage of the bytecodes-in-SmalIntegers idea, apart form its intrinsic complexity, is that its wasteful in 64-bits. Yes one can have code that packs 7 bytecodes into a 64-bit SmallInteger, but then things like object serialization have to be clever enough to convert e.g. 64-bit 14-bytecode methods into 32-bit methods with an explicit bytecodes array, and that kind of complexity was too much for us to consider with HPS.

   As far as the complexity in at:put: for accessing e.g. OrderedCollection, my thought is that yes its worth it, because a) the cost of the indirection to fetch the Array object has to be considered when doing without the mixed named/indexable format, and b) the object header can be designed to fit. For example, in the 64-bit HPS I did just this, allocating an 8-bit field to hold the number of fixed fields (this is probably overkill; a much smaller field would probably serve just as well). So that at:put: can find the number of fixed fields from the header and not visit the class at all, keeping the complexity down.

   In fact that's a very good point. e.g. a 3-bit field would allow objects with up to 6 named inst vars to have indexable inst vars without at:[put:] having to visit the class to determine the number of fixed fields. Given that Spur provides a forwarding pointer become: is again cheap. That would be an interesting experiment. Perhaps its feasible to do in the Newspeak implementation?

6. **Eliot Miranda** | 06-Sep-13 at 8:14 am | Permalink

   Hi Philippe,

   perhaps you could try profiling with the VMProfiler (I'm available to hold hands) and really see where the VM spends its time in Seaside? I expect that Seaside would be a great real-world benchmark that I could use to stress-test and tune Spur. Perhaps you could hold my hand setting up a simple configuration for benchmarking and profiling?

7. **Eliot Miranda** | 06-Sep-13 at 8:23 am | Permalink

   Hi John,

   re Dan's idea, that looks sensible. In fact in HPS/VisualWorks we ditched the split Smalltalk-80 IdentityDictionary-style MethodDictionary format for a flat object organized as selector,method pairs with no free space, ordered by selector identityHash. a) linear search is faster for small MethodDictionaries than large, and b) binary search for large dictionaries is acceptably fast. It is reminiscent of a scheme used in some lisp implementations. The idea is to still use the address of an object as its hash, even though the GC moves objects, but to rehash identity collections whenever any of their contents is moved. I expect this implies the disadvantage that the VM knows what the hashing algorithm is (or at least that there is a broader interface between GC and the rest of the system to support the rehash). Perhaps a simpler approach is to have a needs-rehash flag in an inst var that the GC sets and have the accessing logic check it, but there are threading issues here if GC kicks in mid-way between an access.

   As regards discussing VM design, are you kidding me?!?!? Do bears sh^Hit in the woods? I'd love to talk, especially over food and/or drink.

8. **Eliot Miranda** | 06-Sep-13 at 8:39 am | Permalink

   Hi Klaus,

   one issue is that the VM also needs to be able to get at nil, true, false, #doesNotUnderstand: etc, and these aren't in Smalltalk (given that all classes are reachable from the class table root). Perhaps Smalltalk could have flat inst vars that include nil, true, false and the selectors? That would work. Interesting. If time allows we could build a prototype?

   Once I have the bootstrap complete (today's work is to install a few changed methods into the bootstrapped image so that immediate characters work, classes have a new identityHash primitive, and classes can manipulate a revised format word) we'd have the framework to install such a SystemDictionary and replace the specialObjectsArray. Such a prototype might be a week's work or so.

9. **Klaus D. Witzel** | 06-Sep-13 at 12:32 pm | Permalink

   Hi Eliot,

   flat inst vars sounds very good, also brings clear structure instead of today's array. And yes, I'll take time for work on a prototype. But don't hurry, in a week from now I'm going to enjoy these beaches ./search?q=tenerife+beach for the rest of September.

   Some time ago Igor and I attempted to clone a baby image from inside by using his Hydra VM (Hydra was sponsored by Andreas) which made it into a new heap and process. Perhaps Igor's clone building code can be shared, I'll ask him. It worked already for launching a baby image which did just the idle loop with its own objects.

10. **Eliot Miranda** | 06-Sep-13 at 1:18 pm | Permalink

    Hi Klaus,

    great to hear and enjoy the Tenerife beaches !! But don't worry about the bootstrap. I have that and it is progressing well. I have bootstrapped the new image format, and now I'm working on modifying the few methods in the system that need to change for Spur. These methods concern Behavior's format inst var, Character methods that access the value inst var – with immediate characters that variable disappears and must be replaced by e.g. a send of value, and basicNew et al, which need to allow the generation scavenger to run on failure. I should have the bootstrap finished quite soon.

11. **tim Rowledge** | 16-Sep-13 at 5:01 pm | Permalink

I'd really love to see CompiledMethods be all grown up and tidy, neat, just-plain-objects, with a separate literals array and byte codes array and so on. Just because, well, tidy and shiny. I understand the concern about space usage though – but are we confident that *deployed* images would noticeably grow as a result? *Development* images seem obviously to be largely compiled methods but how about those actual user applications? And given that a few modest graphic images can swamp the memory size of even Alan Kay's squeak images, is it an issue? My current working image is 27Mb – but my camera makes bigger files than that sometimes. People send me bigger emails…

Assuming the decision is to go with a mixed-up crazy compiled method like the ones we all know and love, can we at least try to tidy them up a bit?
a) Having the source pointer be the last X bytes, mangled in some fashion, merged with the number you last thought of is a bit … ugly. At least let's move it to be up with the big kids as a literal's sibling.
b) the method header is a mess; we don't need 10 bits for prim numbers anymore. I'm reasonably sure we only actually need 7 bits max for numbered prims these days and add 1 to allow for the extended quick-return pseudoprims (yes, we'd lose some range, big deal, probably – I see 5 cm's with a return field likely to be impacted). Do we need a LargeFrame flag? Do we need the flag flag – I see a single usage in IslandVMTweaksTest.
c) with a tidied method header we could steal some of the 4 freed-up bits to specify whether the literals actually include a methodClass entry or the selector/properties, rather than the somewhat ham-fisted assumption of last literal and penultimate literal and a wasted slot if one but not the other unless it rained last Thursday…
d) there's surely more; we also have (ab)uses of prim number to decide if there is an external call spec or a plugin/named prim call and oh my, it looks like Traits is in there somehow too.

Whee! Fun!

12. **vigrx efek samping** | 20-Mar-14 at 9:18 am | Permalink

With havin so much written content do you ever run into any problems of plagorism or copyright
violation? My blog has a lot of unique content I've either created myself or outsourced but it appears a lot of it
is popping it up all over the web without my permission. Do you know any ways to help protect against content
from being stolen? I'd really appreciate it.

13. **admin** | 20-Mar-14 at 9:51 am | Permalink

Hi samping,

I don't know and I suppose I don't care enough. I'm more interested in getting the ideas out with the hoped for (and to some extent realised) effect of attracting collaborators. I'm also not sure how generally applicable all this content is, so I wonder whether there's much incentive to plagiarise. So for me I'm more interested in having people read what I write than preventing them steal it :p

14. **Lawson English** | 18-Oct-14 at 2:34 pm | Permalink

When I was doing my exploratory videos on OpenGL and Squeak, I ran into a GC problem that pretty much guarantees that a realtime combat game can't be written in Squeak (or Pharo also I assume):

the GC would kick in and eat up to .25 seconds of VM time so that rendering the game would have a noticeable pause on a regular basis. The ships and bullets would just… stop… and then the game would resume a quarter second later. This means that entire categories of games written in Squeak are simply not playable.

Would Spur get around this issue?

15. **Eliot Miranda** | 19-Oct-14 at 5:40 pm | Permalink

Hi Lawson,

yes, that's very much the plan. But as of this writing that part of the GC has yet to be written. Let me outline the issues.

The Spur system has a scavenging garbage collector that is significantly more efficient than the existing Squeak V3 one, by about 2x. But it can only collect short-lived objects, the objects living in new space. As new space is garbage collected, whenever new space fills up with a substantial fraction of reachable objects, some of those objects get promoted to old space. Any objects that get promoted and later become unreachable are not collectable by the scavenger. They require collection by an old space collector.

Like the V3 system, Spur has a stop-the-world scan-mark-compact GC and it will cause noticeable pauses. This GC has exactly the same issues as the V3 one, and so is intended to be used only for GC on snapshot, or when the developer demands a GC.

The plan has always been to include an incremental scan-mark-compact collector that collects old space but does not cause noticeable pauses because it breaks up the steps in the stop-the-world scan-mark-compact GC into small increments. The VisualWorks VM, with which I'm very familiar, has a similar GC (because of internal details it does not compact). I have a good understanding of the VW incremental GC (IGC). When I started working at ParcPlace the IGC was essentially inoperative because its increments were set based on small heaps and slow machines and hence it typically never managed to complete an entire cycle and hence its work would be discarded and a stop-the-world GC would occur. I did some work to tune the IGC, determining the size of the increment based on the apparent speed of the system, and was able to resuscitate the IGC.

The plan for Spur is to drive the IGC from the scavenger, doing an increment of the complete cycle after each scavenge, based on how fast old space is growing. The system needs to be tuned so that the IGC completes a collection cycle of old space before old space grows by some delta, e.g. by 25%, which would be a settable parameter of the GC.

Let me quote some meaningless statistics to try and illustrate the scheme. These are meaningless because they're taken not from measurements of some real application, but straight from my working system, which hasn't been busy while I've been writing this and so doesn't really reflect a game. But it's illustrative. Right now, on my 2.2GHz Core i7 MBP Spur scavenges are taking about 0.4ms to complete with a 4Mb new space, and in a system with a ~ 140Mb heap (my working image) a stop-the-world GC takes 160ms. Let's say that an incrementalization of the stop-the-world algorithm is twice as slow, that's a 320 / 0.4, or an 800-to-1 ratio of runtime. Since startup my system has done one full GC and 975 scavenges. So in this case, if the IGC runs for about the same time as the scavenger it will complete a cycle once every 80 scavenges, and the total pause time will still be sub-millisecond (0.8ms).

Now take this with a grain of salt. That's the desire, but there are parts of the IGC algorithm that must be completed atomically. One is nilling out references in weak arrays, and the end of the mark phase. This doesn't require a scan of the entire system, which would cause a noticeable pause in a very large heap, because the IGC can maintain a set of weak arrays reached in the mark phase and enumerate just this set quickly. But without a read barrier on weak array, something Spur choses to do with out, all the weak arrays in old space need to be scanned in one go. Again appealing to my meaningless statistics, a full stop-the-world GC takes 160 ms and of which scanning weak arrays is a small part. Weak arrays are only a small fraction of the total heap, and so you can see that such a step will not cause long pauses. The other atomic step is similar, that for processing activated ephemerons (for pre-mortem finalization). This is also unlikely to cause noticeable pauses since the number of ephemerons is likely to be small, and the ammount of space they guard likely to be small too.

But of course the real issue is getting round to implement it. Realistically I won't get to starting the IGC until next year. I need to make significant progress on 64-bit Spur before year's end. But I'm confident enough that it'll get written and deployed in 2015. Wish me luck.

Post a Comment

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

Name	*
Email	*
Website	
Message	

Post

« AN ARRANGED MARRIAGE   LAZY BECOME AND A PARTIAL READ BARRIER FOR SPUR »