{ 2013 09 13 }

Lazy Become and a Partial Read Barrier for Spur

Lazy Become and a Partial Read Barrier for Spur

One of the stated design goals for Spur is to support pinned objects via lazy become. This is one of the less well-thought-out parts of the design. If I can make lazy become work it'll be (as I hope to convince you in this article) a significant improvement over the current Squeak VM's become implementation. But it must work, and to that end I thought I'd write a post describing my current design of the scheme, and allow y'all to criticise it, either shooting it down in flames, saving me the effort of tilting at windmills, or helping fill in the blanks and offer improvements to help the scheme's success. So let's have a tilt at describing become:.

How becoming

Become is Smalltalk-speak for an operation that changes all references to one object into references to another. There are two becomes. two-way, where two objects exchange identities, and one-way where references to a first object are replaced by references to a second object, and the first object becomes inaccessible, to be swept away by the garbage collector. This operation has traditionally been used to grow collections in Smalltalk, and to update live instances when a class definition changes, adding or removing variables. It can also be used to replace a movable copy of an object with a non-movable copy in some more suitable space, for example effectively morphing a movable object in new space with an immobile one in old space. Being able to pin objects down is invaluable in a Foreign Function Interface (FFI) where one wants to pass storage to other components.

The traditional implementation of become in the earliest Smalltalk-80 implementations was based around a two-part object representation. The system had a table of object headers, all of the same size, and each header contained some information used by the garbage collector (contents contain pointers or not, etc) *and* a pointer to the body of the object, the body containing the object's class, its size, and its fields. The two-way become operation is simple with this representation; simply swap the pointers to the object bodies, plus whatever other header bits are relevant. Here there is effectively a read barrier on every class and instance variable access, through the pointer, so-called "object table indirection". The VisualWorks memory manager still uses this scheme, updated to work with a generational collector.

Subsequently, Digitalk's Smalltalk/V used a flat object representation without object table indirection, and hence provided a one-way become which scanned through memory, replacing any reference to the first object with a reference to the second.

The existing Squeak/Cog garbage collector has a flat object representation, and supports both two-way and one-way bulk becomes. The input to the become primitive is two arrays of objects, and the become is pair-wise, performing either a two-way or a one-way become on each pair of objects in the two arrays. This is implemented above the VM's compaction system which stops the world, allocates some forwarding blocks, replaces the headers of objects to move or become, with pointers to their forwarding blocks, which contain their final position after become or compaction, scans memory following object references through forwarding blocks updating pointers, and then, if compacting, moving objects, finally discarding the forwarding blocks.

All approaches support grow, and instance migration (although IIRC Smalltalk/V didn't migrate instances on class redefinition), but while the cost of the object indirection version is O(1), with hidden costs in object table indirection, the flat implementation is O(n) in the size of the heap, and in the age of 64-bit machines that's a problem.

An alternative implementation, oft-used in Lisp systems, is to add a read barrier to all object access, and mark objects as forwarders. This can be used to implement lazy copying garbage collection where objects are copied from one semi-space to another in parallel to the main program (the "mutator"). To become, or move an object one replaces the object's header or first field with a *forwarding* pointer to the desired target or copy in a new location, marking the "corpse" as forwarded. The program checks the forwarded flag on each access. If there is hardware support, as in a Lisp machine, this can work well. But without hardware support, and like the object table representation, it has costs, slowing down program execution due to the scattering of forwarding checks and forwarding pointer follows throughout program execution.

The Spur Conjecture

Smalltalk provides strict object encapsulation; the only way to access the state of an object is to send it a message. OK, bald-faced lies are very popular these days, but with Smalltalk there's more than a grain of truth here. The only violations of this principle are though certain primitives. The add primitives, for example, be it on SmallIntegers (fixnums) or Large integers (bignums) access the state of both the receiver and the argument in computing the result. One might think that instVarAt: and instVarAt:put:, which can access the instance variables of any object from the outside, violate encapsulation, but one still has to send the message to the object one is accessing. Mirror primitives, however, do violate encapsulation, allowing one to reach inside an object without sending a message to it, passing the object to be accessed as an argument to the mirror primitive. The up side is that mirrors provide security, they are capabilities that only certain parts of a system are trusted with; instVarAt: and instVarAt:put: declare open season on internals and are thankfully used with discretion in Smalltalk systems.

Given strict encapsulation it seems to me that we can rely on message sending to follow forwarding pointers. (almost) Every message send in Smalltalk involves some form of check of the class of the receiver. In the abstract, a message is activated by fetching a receiver object's class, and searching the method dictionaries in the class's hierarchy until a matching method is found. In practice, the process is optimized using method lookup caches. Every send involves fetching either the object's class, or a unique class identifier (e.g. an index into a class table) from the object and comparing it against a possibly matching tag in a cache. If they match, the right method is in the cache. If they don't match a full lookup is needed. By arranging that forwarding objects have their class tags changed, the message send cache probe becomes a forwarding object trap. Whenever a message is sent to a forwarding object the cache probe will fail and the failure path can follow the forwarding pointer.

The scheme then implements pinning and become using forwarding pointers. The old space collector will not move pinned objects, and to avoid pinned objects filling up new space, an unpinned young object is pinned by allocating a pinned copy in old space and marking the young object as a forwarding object pointing to the copy. Two-way become is implemented by either swapping contents if the two objects have the same size, or allocating two copies, and forwarding the originals to the exchanged copies. One-way become is implemented by marking the first object as forwarded, pointing to the second object. This seems great in principle, but there are a number of pitfalls, not just the primitive argument one mentioned above.

My Spur conjecture is that using a message send trap is an effective approach because the pitfalls are limited and manageable. So what are the pitfalls?

Falling With Style

The first pitfall is that Smalltalk accesses two other objects far more than the receiver, which are the current method and the active context. Smalltalk-80's execution model executes bytecodes and references literals stored in the current CompiledMethod object. In most good Smalltalk implementations, methods are compiled on-the-fly to machine code and the machine code is executed. In fact, Cog is a hybrid interpreter+JIT, interpreting essentially only at start-up. So the becoming of methods is really only an issue in the interpreter. Smalltalk-80's execution model also provides access to the underlying activation record, a context, and in a fast Smalltalk implementation (and Cog's not too shabby) contexts are essentially virtual, created on-demand only when the program asks to see them; the VM's actual activation records are more-or-less conventional stack frames, and contexts are proxy objects for them. And while message sending is heavily optimized, when a method cache probe does fail the class hierarchy of the receiver is searched, and any object there-in could have been becomed.

The more-or-less conventional stack frames are housed not on the machine stack but in a *stack zone* of quite limited size, organized as a set of pages. The stack zone acts much like a young generation for contexts, storing the most recently active N stack frames. While the number of pages in the stack zone is a settable parameter, the zone's size is typically less than 100 kb. So it is practicable and quick to scan the stack zone and follow forwarding pointers after a become operation. We can know that the receiver in a stack frame is never a forwarding pointer and hence not need a read barrier when reading its instance variables. We can know that the method in an interpreted frame is never a forwarding pointer and hence not need a read barrier when interpreting bytecodes or accessing literals. However, a literal itself may be a forwarded object, and so, for example, we may have to follow the message selector on a message probe miss. We must scan the first-level method lookup cache and remove forwarded entries there-in; again the first-level method lookup cache is quite small, perhaps 4k entries.

To avoid checking class fetch (where the class table is indexed by an instance's class index to locate its class object for method lookup) we can scan the class table after a become operation. There can be up to 4M classes in Spur, but this is still a tiny fraction of a large heap – ignoring page density, faulting in 4M pages could be expensive ;-). But in the common case the class table contains a few thousand classes, most of them in the first few pages of the table, and that's a tiny amount of memory to scan. There is a wrinkle; the class table contains only classes that have been entered into the table, either by being instantiated, or by being entered into a hash table (this because a class's index into the table is its identityHash). But method lookup traverses a class's superclass chain, and e.g. abstract superclasses on the chain may not have been entered into the table. So the scan for forwarding pointers needs to follow the superclass chain of each class in the table until a class with an index is found; classes with indices are in the table so will be seen in scanning the table.

In Squeak super sends are implemented using a method's *method class association*; this is its last literal and is an association from class name to class object of the class that owns the method, the method's method class (quite different from the method's class). A superclass send fetches the value of the method class association and then the superclass of the method class, and starts the lookup in this superclass. To avoid a read barrier on accessing the superclass, the stack zone scan needs to follow the method class association to the class.

Since the stack zone houses only some activation records, return from the base stack frame in a stack page may attempt to return out of the stack zone into a context. At this point the VM "faults in" the context into a new stack page (which I term marrying it with its stack frame). This process must now check for forwarding pointers throughout the context and the method's method class association. Since marrying a context is an expensive operation this adds hopefully negligible overhead.

Access to literals other than the method class association are of three kinds, message selectors, global variable associations, and ordinary literals. Let's consider these in turn.

- If a message selector is a forwarding pointer then, provided that we scan the method cache and follow there-in, the forwarded message selector will cause a cache miss and the message selector can be followed prior to the class-hierarchy lookup.
- Smalltalk implements global variable access by using as a literal the Association for the variable that is owned by some global dictionary (a key,value pair e.g. of class name to class object, e.g. in Smalltalk or a class's classPool). push/store/popLiteralVariable all fetch a literal, and either read or write the literal's value field. The fetch of the literal needs an explicit check (otherwise we would have to scan all literals in all methods in the stack zone, and the entire method on return, and

global variables are relatively rare; in my work image 8.7% of literals are globals).
- ordinary literals are simply objects pushed onto the stack. These, like the referents of the receiver's inst vars may be forwarded. No read barrier is needed here; access will be handled further down the line in method activation.

With the above approaches all bases appear to be covered except for primitives (have I forgotten anything else?). And the primitive issue, which we introduced at the beginning of this section, is access to sub-state of the receiver and/or access to arguments. These may be forwarding pointers.

First of all, the message send trap catches access to the receiver itself; this cannot be a forwarding pointer, and if the receiver is a flat non-pointer array such as a Float, a ByteArray or a String (all implemented as arrays of bits), the state cannot be forwarding pointers.

Secondly, not all state-accessing primitives are affected. e.g. in at:put: (almost *) no checks are required because the receiver will have been caught by the message send trap, the index is a SmallInteger (an immediate, and immediates can't be becomed), and the argument is either an immediate Character (in String>>at:put:) or a possibly forwarded object that will simply be stored into the array; i.e. the last argument's state is inspected only if it is an immediate. (*) the exception is 64-bit indexable and 32-bit indexable arrays, these are floats & bitmaps which could take LargeIntegers whose contents are copied into the relevant field). But e.g. in beCursor extensive checks are required because the primitive inspects the state of the receiver to some depth, accessing a couple of form instances, and a point that are sub-state of the Cursor object.

How should the design solve the issue of state access in primitives? One approach would be an explicit call in the primitive, to follow the pointers in the receiver, and in accessing the arguments from the stack. This can be made convenient via providing something like ensureNoForwardingPointersIn: obj toDepth: n, which in beCursor's case would look like interpreter ensureNoForwardingPointersIn: cursorObj toDepth: 3 (the offset point of the mask form). Another approach would be to put an explicit read barrier in store/fetchPointer:ofObject:[withValue:] et al, but provide an additional api (e.g. store/fetchPointer:ofNonForwardedObject:[withValue:] et al) and use it in the VM's internals. The former approach is error-prone, requiring extensive testing, but the latter approach is expensive, potentially ugly, touching nearly all of the core VM code. It would appear that one of these two approaches must be chosen. I favour the former, but I could be quite mad.

Finally, the Squeak VM cognoscenti amongst you will know that the Squeak VMs support primitive plugins that can be dynamically-loaded and interface to the VM via interpreterProxy. interpreterProxy's API is a simple place to introduce read barriers as required.

Pleas

OK, please respond with conversation, contradiction, criticism, positive or negative. OK, please respond with conversation, contradiction, criticism, positive or negative.

Talk talk talk, it's only *talk*. Comments, cliches, commentary, controversy. Chatter, *chit–chat, chit–chat, chit–chat, Conversation*, contradiction, *criticism.*

Send article as PDF Enter email address Send

Posted by admin on Friday, September 13th, 2013, at 11:51 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.

{ 17 }
Comments

1. **Reinout Heeck** | 15-Sep-13 at 1:46 am | Permalink

   Allow me to shoot a bit from the hip (if only to get some conversation started here).

   1) I miss an expose on GC interaction. For instance: does the GC use forwarding pointers internally and if so will they be the same thing? Will GC eliminate forwaring pointers or will it treat them as normal objects? If the latter: where will various color/mark bits live?

   2) How will assignment work? You say that #at:put: will store forwarding ponters unaltered, but will that be sufficient in the light of repeated (chained) becomes? Can one create chains of forwarders in your proposed system? Perhaps samething along the lines of:

   ```
   |holder|
   holder := Object new asValue.
   [ forwarder:=Object new.
   forwarder oneWayBecome: holder value.
   holder value: forwarder
   ] repeat.
   ```

   An what about identity tests on such chained forwarders (if they exist), is the extra run time considered a bug or a feature?

2. **Carl Watts** | 15-Sep-13 at 7:31 pm | Permalink

   Eliot, you are undoubtedly one of the finest minds in Computer Science EVER.
   And undoubtedly this is another brilliant idea in VM technology.
   But I wonder if:
   a) you should get out a bit more (maybe a picnic on the grass in the park with friends) (like me) 🙂
   b) smalltalk VM's are getting way too complicated for most mortals to comprehend; trying to optimize the efficiency of operations that, while efficient to implement in the days of object tables, are not any longer and should perhaps be eliminated from the standard system?

I realize I may be close to the LEAST qualified person, amongst those you expected to read your essay, to comment on VM architecture, but I've often wondered if we might be better off simplifying the whole problem by:
1) Getting rid of become. Resizable Collections can be implemented simply by holding a private Array and allocating a larger one when necessary (copying elements and throwing away the old Array when done). Proxies wouldn't become the real object, they just hold onto the real object and forward messages to it, etc. Objects would morph when instance variables are added/removed from their class using a world-stopping full-memory-scan become (the one case where become still is allowed because it only happens during development time not (typically) at run time).
2) Enforcing a maximum object size of something reasonable (1-64MB would be the range in which my vote for the max would lay). LargeArrays (of sizes greater than the max for a single Array) would simply be implemented as Arrays of arrays.
3) Not moving any object in RAM after it is allocated. EVER. That way the reference is always just the pointer to the object. And the memory fragmentation problem is solved NOT by compaction garbage-collection but by:
a) attempting to allocate new objects to fill the free gaps between live objects in an existing object-memory page.
b) always coalescing any two adjacent free object slots (within a single object-memory page) into a single larger free object slot, this can be done efficiently just by ensuring that the last byte in any slot indicates whether the slot is empty and, if so, that byte and the 2-3 bytes before it indicate how big the slot is (in machine words).
c) keeping free lists of most sizes of slots (from 3-64 machine words in size in 1 machine word increments) in each memory-page (say 64MB) and always attempting to allocate new objects in the correct-sized slot. Larger slots are just kept in a "large slot" list. The lists are just singlely-linked linked lists of slots (but each object memory page has it's own set of free lists).

I now rest my case and prepare to be edumacated as to why my suggestion is ridiculously naive (preferably while on a picnic blanket, having a picnic, in Precita Park). 🙂

3. **Eliot Miranda** | 16-Sep-13 at 3:15 pm | Permalink

Hi Reinout!

yes, forwarding objects are internal to the VM. The image should never see them. But they are not internal to the GC. The VM needs to do most of the work in following forwarding pointers, catching them during message send. A message send to a forwarded object should always fail, so on that rarely-taken failure path the VM will check for, and follow forwarding pointers. This is key to the design.

As far as the GC eliminating them or treating them as normal objects, it will do the best it can. For example, a forwarding object in oldSpace will live for quite a long time, until the old space incremental collector (or a global scan-mark) eliminates it. So forwarding objects have colour/mark bits like any other. Its just that only their first field is live (rather like CompiledMethod having only literals live).

Assignment works like it does now. Storing a forwarding object into a field just stores a reference to the forwarding pointer. The store operation doesn't check for storing a forwarding pointer, so objects can acquire references to forwarding pointers naturally. But it is, of course, illegal to store into a forwarding object. The VM must make sure that the receiver is never forwarded.

There's no way to keep become lazy and avoid the possibility of forwarders to forwarders, so whenever forwarders are followed there is a loop to reach the final target.

As far as identity tests, since #== is a primitive that takes an argument the VM will have to check the argument for being a forwarding pointer. Given that #== is actually inlined without sends it will have to check the receiver as well.

4. **Eliot Miranda** | 16-Sep-13 at 3:49 pm | Permalink

Hi Carl :-),

yes I should come hang out. Name a day. It'd be lovely to come over to Precita.
Now addressing the post, yes getting rid of become is an excellent idea unless it's difficult to live without. One of the things I want the Spur memory manager to support well is pinning objects, i.e. modifying an object so that it won't move, and hence allowing it to be passed through the FFI with less ceremony.

One could allow new space to fill up with pinned objects and eventually allocate a new new space. But that has downsides in managing the scavenger, a performance-critical component, and has no solution for eventual fragmentation.

Other approaches require, in effect, a become:. The object is moved to some location where its immobility poses less of a problem and all references to it are updated. This is cheap to do for a newSpace object; references to it are either in other new space objects or in the remembered set (or on the stack). So moving a newSpace object to oldSpace is reasonably cheap.

But that still leaves the existing Squeak system, and it uses become a lot (not least for instance migration) and I don't want to block Spur's adoption by forcing a significant image change, in the attempt to do without become:. So instead I chose this more ambitious path, to get the two birds cheap become: and cheap pinning with the one lazy become stone.

On your point 1. Squeak already does this.
2.'s a good idea.
3 is nearly what Spur will do with old space. It'll compact though, using best fit and forwarding to avoid sliding things around.
I implemented 3c for VisualWorks a while back. IMO it's a must have.

cheers!

5. **tim Rowledge** | 16-Sep-13 at 4:05 pm | Permalink

   Obvious, trivial and probably worthwhile optimisation for two-way become – copy the contents of the smaller object into the space of the larger one and stick in an empty slot header for the remains, then you only have to allocate one new space.

   Oh, and you (and indeed Carl) should come and hangout at my place sometime. VI is a nice break from the bustle of silicon valley.

6. **Igor Stasenko** | 18-Sep-13 at 1:43 am | Permalink

   Hi, Eliot

   i am still don't feel comfortable about hard limit of number of behaviors in system imposed by object header format.

   I know, that surpassing the limit is something which is highly improbable in smalltalk-based systems, well, unless we want to experiment and implement a prototype-based system to run on VM:

   (1 to: (big number)) collect: [:i | Behavior new new yourself]

   i just thinking , if there's a cheap way to avoid hard limit, like reserve the highest possible
   value of class index for 'extenternal' class ref, and instead of placing class reference into additional slot of object, use an identity-dictionary lookup, maintained by VM itself.

   Like that, in traditional smalltalk systems, where number of behaviors fit into class table,
   everything is ok, and upon surpassing the limit, fetching a class ref for some objects will be much slower.

   Concerning forwarding pointers, 2 things:
   – i think most problematic is primitives (need to review all of them and put checks where they accessing object's state). For most of them it is

mainly ensuring that primitive inputs contain no forwarded references.
But some primitives is following pointers from primary inputs, which can also contain forwarded references.

– indeed, it would be cool to not have a forward reference chains.. but how you going to ensure that?

consider following simple case:

we have 3 objects, a, b ,c allocated on heap.

array := { a. b . c }.
a becomeForward: b
b becomeForward: c.

after first become, a's header is replaced with forwarding ref to b, like that
(array at: 1) has to follow reference.

after second become, b's header is replaced with forwarding ref to c, like that
(array at: 2) has to follow reference, but the problem is that at this point,
your input is reference to b, and you cannot know if there's anyone who having forwarding reference to it or not
and unless you fix a's header, it will still point to location of b (which we just forwarded),
forming a forwarding chain.

To avoid chaining, the only thing which you can do is to again, scan whole heap
to fix all forwarding pointers to b, to forward them to c… which leads to getting where
we started from 🙁

7. **Eliot Miranda** | 18-Sep-13 at 10:20 am | Permalink

   Hi Tim, good idea, I'll add that RSN!

8. **Eliot Miranda** | 18-Sep-13 at 10:39 am | Permalink

   Hi Igor,

   good points.

   For instance-specific behaviours, one solution is allowing an object to be an instance of itself. This is an idea of Claus Gittinger's. e.g.
   | proto |
   proto := { Object. MethodDictionary new. Object format + 5. 'first inst var'. 'second inst var' }.
   proto changeClassTo: proto

   Now proto is an instance of itself, has room for 5 inst vars (superclass, methodDict, format and the two inst vars), and inherits from Object. Spur supports this by reserving 1 class index that means "the object is its own class". So now there is no limit other than address space on the number of instance-specific behaviours. Is that acceptable?

If not, one can always imagine adding an extension header for a full class pointer. But I would only implement that once the limit on the number of classes became a problem. I doubt that it will be for a while yet.

Spur avoids forwarding chains by following forwarders during become. The inputs to become are two arrays of the objects to be becommed. As long as Spur follows forwarders in these arrays it won't create forwarders to forwarders. There are asserts to ensure forwarders to forwarders are not created.

The primitive issue is sticky. I hope that I can catch many forwarding issues in primitiveFail[For:], checking for forwarding and retrying, but that approach probably won't work for primitives that deeply traverse their arguments' state. These primitives will need to be rewritten. But thankfully they're very rare.

9. **Igor Stasenko** | 18-Sep-13 at 3:00 pm | Permalink

a) instance-specific behaviours:
– yes, i think direct support for 'instance of itself' should relax most of problems related to class table limit.

b) "Spur avoids forwarding chains by following forwarders during become. "

Yes, that trivial optimization, but it won't guarantee you from not forming a forwarding chains!

Look:

array := { a. b . c }.

"array at: 1 points to &a
array at: 2 points to &b
array at: 3 points to &c "

a becomeForward: b

"array at: 1 still points to &a
array at: 2 still points to &b
array at: 3 still points to &c "

b becomeForward: c.

"array at: 1 still points to &a
array at: 2 still points to &b
array at: 3 still points to &c "

now look, when you accessing

(array at:1) you have to follow a forwarding chain:
&a -> &b -> c

but not:

&b -> c

because there's nothing which overrides reference to &a in given array during become operation.
Taking into account that 'array' can be any other object in system, not something which directly observable during become operation, the cost of

finding and replacing reference(s) in all such objects will be same as current become implementation: scanning whole heap.

Which kills the whole idea of having fast become, isn't?

So that means, either we will have fast become, but at cost of having forwarding chains, or we have slow become, but no forwarders chaining.

10. **Eliot Miranda** | 18-Sep-13 at 3:16 pm | Permalink

    Hi Igor,

    yes you're right. So when following a forwarder one must follow a potential chain. No problem because this is localised. Thanks!

11. **Klaus D. Witzel** | 24-Oct-13 at 12:48 pm | Permalink

    Hi Eliot,

    Igor's concerns are one thing, the other is your still open "have I forgotten anything else?"

    Perhaps, after implementation and some debugging, you'll have collected a good deal of helpful integrity checks. But these can be put into code right from the beginning and be invoked from the Smalltalk level. They might accept a selector symbol which they send to self with problematic objects as arguments (if there are any) or else return nil (for nothing found).

    Just a thought.

12. **Ben Coman** | 17-Dec-13 at 5:35 am | Permalink

    Hi Eliot,

    Just had a random neuron fire so wondering… Would your lazy #become: benefit a copy-on-write system, where one VM could run multiple images sharing objects. Use case might be a cloud provider where many clients might base off the same Squeak 4.5 or Pharo 3.0 image.

    I am thinking that when writing to an object, the original could be copied twice and replaced with a conditional-forwarder such that which copy is forwarded to depends on the image accessing the conditional-forwarder.

    cheers, Ben

13. **Eliot Miranda** | 17-Dec-13 at 11:55 am | Permalink

    Hi Klaus,

    that tends to be the way the VM is written. It is peppered with assert expressions, much like tests. These are always run when the simulator is in use. The asserts are also included when the VM is translated to C, but the VM is compiled in three different ways:

1. a production VM compiles out the assets by defining the assert macros to be void

2. a assert VM is compiled with moderate optimization; it can be used to check the asserts even though it is often an order of magnitude slower than the production VM

3. a debug VM compiled with no optimization for full debugging

So what's really useful is to augment the VM with Smalltalk code or images that stress it. These stress tests are then run on any or all of the four VMs, simulator, debug, assert or production.

But my question "have I forgotten anything else" was really an architectural one. Are there ways the VM processes objects not covered by message sending, primitive invocation, inst var access, and literal access? I think that's it :-).

14. **Eliot Miranda** | 17-Dec-13 at 11:56 am | Permalink

Hi Ben,

sounds really cool. care to expound a bit more?

15. **Klaus D. Witzel** | 04-Feb-14 at 8:43 am | Permalink

Hi Eliot,
while at FOSDEM I attended the presentation about Shenandoah – utilizing parallel garbage collection threads (Java) by Red Hat for OpenJDK. The aim is to evacuate one of many regions (the one which has produced most garbage), and only for determination of this region (and house keeping) to stop the world, while the relocation is then done on occasion (like lazy forwarding, etc). It's bleeding edge at the moment, needs 4 extra words per object in the prototype, but IMO interesting enough to keep an eye on.

16. **Mateusz Grotek** | 26-Mar-14 at 11:08 am | Permalink

How does this implementation prevent forming long chains of redirections?

Let's say I have a variable V1 pointing to an object O1. (V1 -> O1).
I have another variable V2 pointing to an object O2. (V2 -> O2).
After that I do: V1 becomeForward: V2.
Now O1 points to O2, so I have V1 -> O1 -> O2
Now I assign to V2 an object O3: (V2 -> O3)
And again do:
V1 becomeForward: V2.

So I have: V1 -> O1 -> O2 -> O3

I do it a couple of times. Now I have a long chain of redirections:
V1 -> O1 -> O2 -> O3 -> O4 -> O5 -> …. -> ON

If I save my image it gets saved inside it. So the longer I use my image
the longer the chain is. In the end it's longer than my memory and the
image crashes.

What am I missing?

17. **Eliot Miranda** | 26-Mar-14 at 11:22 am | Permalink

Hi Mateusz,

That's a great question! The answer is that there are several points at which forwarders get followed such that the references to the forwarders are updated to refer to the target of the chain of forwarders.

Forwarders get followed during garbage collection. That means after an object has been scanned by either the scavenger or the mark-sweep garbage collector, if it referred to forwarders before the scan it no longer does after the scan. It so happens that when the VM does a snapshot it also does a full scan-mark GC, so there are no forwarders in a snapshot.

Forwarders get followed on message send and primitive evaluation. If the receiver of a send is a forwarder then the VM follows forwarders in the arguments and temporaries of the current activation. If a primitive fails and the primitive has a non-negative "primitive accessor depth" then the VM traverses the transitive closure of the objects comprising the primitive's receiver and arguments to that depth, following and eliminating forwarders. If it finds a forwarder it retries the primitive. The accessor depth is the depth to which the primitive traverses the object graph of its arguments, and is computed by analysing the parse tree of the primitive. See Primitives and the Partial Read Barrier.

Since in Smalltalk instance variable access is direct, not by message send, the VM also scans the receivers of the frames in the stack zone after any become that forwards a pointer object, since that object could undergo an instance variable access. This avoids a read barrier on instance variable access. Alas there /is/ a read barrier on global variable access, since a global variable (an Association, ClassBinding, etc) could be becommed, and eliminating the read barrier would involve scanning all methods in the stack zone, which would be expensive. But the read barrier is cheap because the contents of the global variable are being accessed anyway (either reading or writing the value of the association). So the additional check on the class index of the global is cheap; it won't create any additional cache misses, and on current machines the test itself is very fast; it's the reads that cost.

Another thing limiting a potential explosion of forwarders is that forwarders only get created during become, or pinning, and become and pinning are relatively rare operations.

So in practice deep chains of forwarders don't get created, and forwarders get eliminated during normal processing.

Post a Comment

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

Name	*
Email	*
Website	
Message	

Post

« A SPUR GEAR FOR COG PRIMITIVES AND THE PARTIAL READ BARRIER »