Mariano Martinez Peck - PhD

ImageSegments

marianopeck@gmail.com

1 Introduction

ImageSegment is something similar to serialize a graph of objects or to cut a piece of image. In this case you have an array of root objects. Those root objects contain anothers objects and those contain more and so on. So you have a graph of objects that you want to put in a segment. You can then write this segment to disk and remove the original objets from the Smalltalk image, as it is explained later.

But not all the objets from the graph are included in the segment. Objects that are **only** accessible from objects that are inside the segment are included. To resolve this problem — to know which objects to include — ImageSegment uses *Garbage Collector* techniques. In addition, the operations of storing and loading the graph of objects are implemented by primitives. This means that it is C code that it is executed and this is why ImageSegments is really fast and has good performance.

There are two main use cases of ImageSegment

- Swap objects of the same image
- Export a segment or piece of an image to another one

2 First details of ImageSegment implementation

There are two main classes which are ImageSegment and ImageSegmentRootStub. The first one represent an image segment and has some important instance variables like outPointers and the segment itself. The *segment* instance variable is a WordArrayForSegment that extends from WordArray and there is where the objects of the array of roots are stored in a binary way. The *outPointers* is also important and keeps the object pointers to all of the objects that are outside of the segment but they are also pointed from objects inside the segment.

The ImageSegment responsabilities and behavior include generating the segment, extracting it and replacing original objects by proxies — ImageSegmentRootStub — and vice-versa, writing and reading the segment from disk, among others.

Another important implementation detail is that the ImageSegment class has a kind of *State Machine*. It is not done by a *State Design Pattern* but stores symbols in an instance variable called *state*. Examples are *active*, *onFile*, *imported*, *inactive*, etc. These states are changed with the actions that are done to the image segment.

Finally, it is important to know that ImageSegment uses two primitives to store and load the graph of objects from the segment; *primitiveStoreImageSegment* — primitive 98 —

and *primitiveLoadImageSegment* — primitive 99 — . With these primitives ImageSegment achieves good performance.

3 Swap objects in the same image

This features is almost used in Squeak *Projects*. You can swap out a *Project* from the image to disk and then load it again when needed. Of course it can be used in other places than *Projects*.

What you must do is to create an ImageSegment instance for the array of root objects. Then you can extract the segment; the original root objects from the array will be replaced by objects that act like a proxy. The proxy extends from ProtoObject and redefines the *doesNotUnderstand:* so that to automatically load the segment again to the image when needed. In another word, it acts in a lazy way.

The original objects of the graph will be removed from the image and it will be written in the segment. However, this removal is not automatically done by ImageSegments. It only guarantees that there are no more references to those objects. You have to explicity evaluate a *Smalltalk garbageCollect* if you want that. At this moment you have a segment object with that array of bytes. After that you may want to write the segment to a file.

In summary, this use case is similar to what it is known as *Virtual Memory* in *Operating Systems*.

To install the segment again, you have two ways:

- Send any message to one of the root objects.
- Manually install the segment sending a particular message.

Swap in step by step

These are the steps to swap and then load again the graph of objects to a file. See figure 1 for a state diagram.

1. Create the segment.

Method 1: *This is an example of how to create the segment*

```
root := Array with: (TestCase new) with: (TestCase new).

segment := ImageSegment new
   copyFromRoots: root
   sizeHint: 5000
   areUnique: true.
```

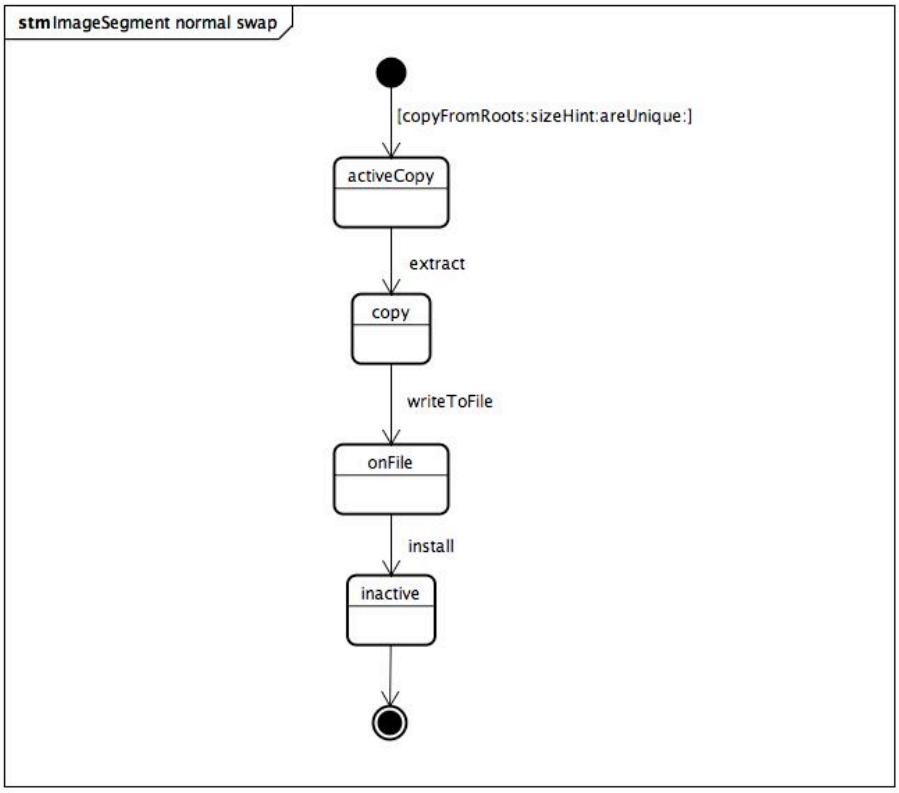

Figure 1: ImageSegment states during a normal swap operation.

As you can see, the most important message is *copyFromRoots: root sizeHint: 5000 areUnique:* that copies the graph of objects into a segment. The copied objects in the segment are not in the normal Squeak space.

After this message is sent, the segment status is *activeCopy* which means that the objects has been copied and the intention is to become *active*. The graph of objects has been encoded in the segment, but those objects are still present and referenced in the Squeak system.

2. Extract it.

Method 2: *To extract it you must do the following*

```
segment extract.
```

This operation replaces all the original roots of the segment with *ImageSegmentRootStubs*. Thus, the original objects will be reclaimed, and the root stubs will remain to bring the segment back in if it is needed. They are like proxies.

Once this operation is done, the state is *active*, which means that the segment is now the only holder of graph of objects. Each of the original roots has been replaced by

an ImageSegmentRootStub that refers back to this ImageSegment. Because of this, is very common to evaluate *Smalltalk garbageCollect* after the extract. I think that maybe it is a good idea to put it in at the end of the *extract* method.

3. Write it to disk.

Method 3: *This is the way to write the segment into a file*

```
segment writeToFile.
```

After this action, the sgment will have the state *onFile*.

4. Install it again.

Two to this, you have two options:

(a) Install it by sending a message to any of the root objects. We said root objects were replaced by ImageSegmentRootStub which are like a proxy. So when you send a message to any of those objects it will not be understood and then the method *doesNotUnderstand:* will load the segment and replace again the root objects with the original ones.

(b) Sending the message *install* to the segment.

4 Export a piece of the image to another one

This feature let you create a segment for an array of root objects and write it into a file so that it can be loaded in another image. In this case, the original root objects are not removed or even changed in the original image. They are just copied inside the segment and then written to a file. Another difference with the swap scenario is that in this case the file also includes all the *outPointers* and all what is needed to be able to load that segment in another image.

In the case you are swapping the graph of objects, you cannot get rid of the Image-Segmnt object. Even if you write the segment to disk, the object of the ImageSegment is still in memory. The only thing you do when you write the segment to disk, is to move the WordArray that represents the segment to a file. But again, the ImageSegment object is istill in memory (including the *outPointers*) and you need it to install the segment again. So, *outPointers* in this case, don't go to the file as they are kept in memory in the ImageSegment object. In addition, those objects pointed by *outPointers* will not be gabage collected as they are referenced by the *outPointers* itself.

On the contrary, in the case of export, you can get rid of the ImageSegment object. You can set a nil to it and then garbage collect it. The *outPointers* in this case also go to the file. Thus, you can then read the file from disk and install it. In details, when you

export an image segment it just serializes the whole ImageSegment instance and all its referenced objects (that means also the *outPointers*) with a SmartRefStream.

In summary, this scenario is a kind of persistence, serialization or exportation mechanism.

Export step by step

See figure 2 for a state diagram.

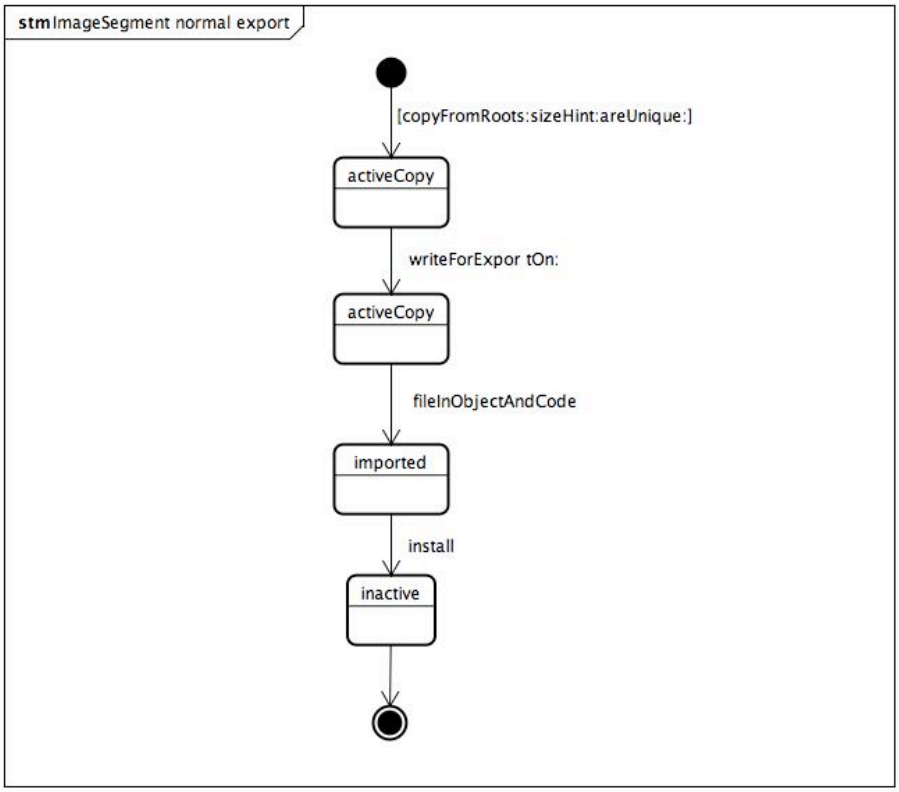

Figure 2: ImageSegment states during a normal export operation.

1. Create the segment.

   This step is exactly the same as swapping.

2. Export it to file.

Method 4: *This is the way to export the segment to a file*

```
| stream |
stream := FileStream forceNewFileNamed: 'fileName'.
[ segment writeForExportOn: stream ]
```

```
    ensure: [ stream close ].
```

With this code we write the segment on the disk with all the information needed to reconstruct it in a new image. The biggest difference with the swap approach is that in this case the *outPointers* are encoded as normal objects on the disk too. The other big difference is that the original root objects are not removed neither replaced by proxy objects. This is because the aim of this use case is to export a root of objects to a file so that it can be loaded in another image.

Of course, there are a lot of details you have to deal with. For example, suppose the class of one of the objects has changed between both images, or even more, it was deleted.

3. File in the segment into another image.

Method 5: *Load the segment file into the image*

```
| stream |
stream := FileStream oldFileNamed: self fileName.
↑ stream fileInObjectAndCode install arrayOfRoots first
```

With this little code we file in the image segment file into the image. After this, the state is *imported*. In addition, this message not only loads the ImageSegment instance to memory but also installs the image segment. More in details, the message *fileInObjectAndCode* calls to method *comeFullyUpOnReload:* which inside calls to the primitive to load the segment — *self loadSegmentFrom: segment outPointers: outPointers.* — , which is exactly what the *install* method do.

To conclude, there is no need to explicitly install it.

4. Install it.

This step is optional, as this is also done in step before. It is quite similar to the swap case. The only difference is that in this case you have to explicitly install it — there are no proxies to do it for you — .

If you send this message after sending the message *fileInObjectAndCode*, the primitive to load the segment is invoked twice. However, it seems the second call takes zero time. If you look at the code of the primitive, I think this is due to the fact that *segOop* is not <= then *endSeg* so, it does not loop anything and just returns.

5 ImageSegment and Garbage Collection techniques

It was already said that ImageSegment uses '*Garbage Collection* techniques to detect which objets of the graph should be copied in the WordArray. Let see in details how this is achieved.

In order to understand each step we will use an example. See figure 3 and you will notice the *arrayOfRoots* that we will use.

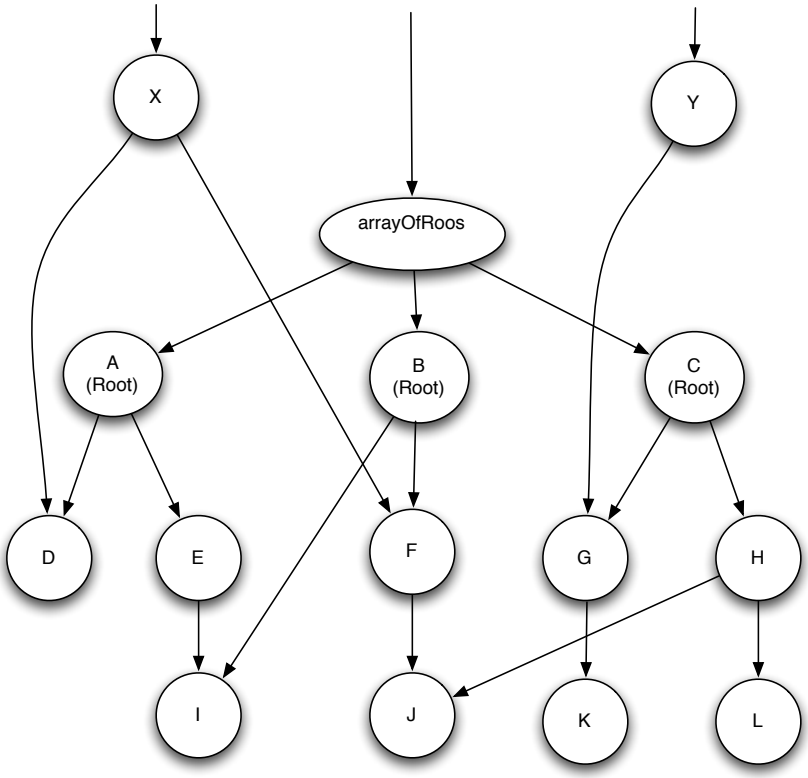

Figure 3: Example we will use to understand the GC techniques.

So, let's see these steps:

1. First, it manually marks the rootArray and all root objects. See figure 4.
2. Then do a mark pass over all objects. This will stop at our marked roots, thus leaving our segment unmarked in their shadow. See figure 5.
3. Finally unmark the rootArray and all root objects. See figure 6.
4. All external objects, and only they, are now marked.
5. The objects that are unmarked in the graph of objects of the arrays are those who will be put in wordarray and those who are marked, to the *outPointers*. In our example A, B, C E, H, I and L go to the *WordArray* and D, F, G, J and K go to *outPointers*.

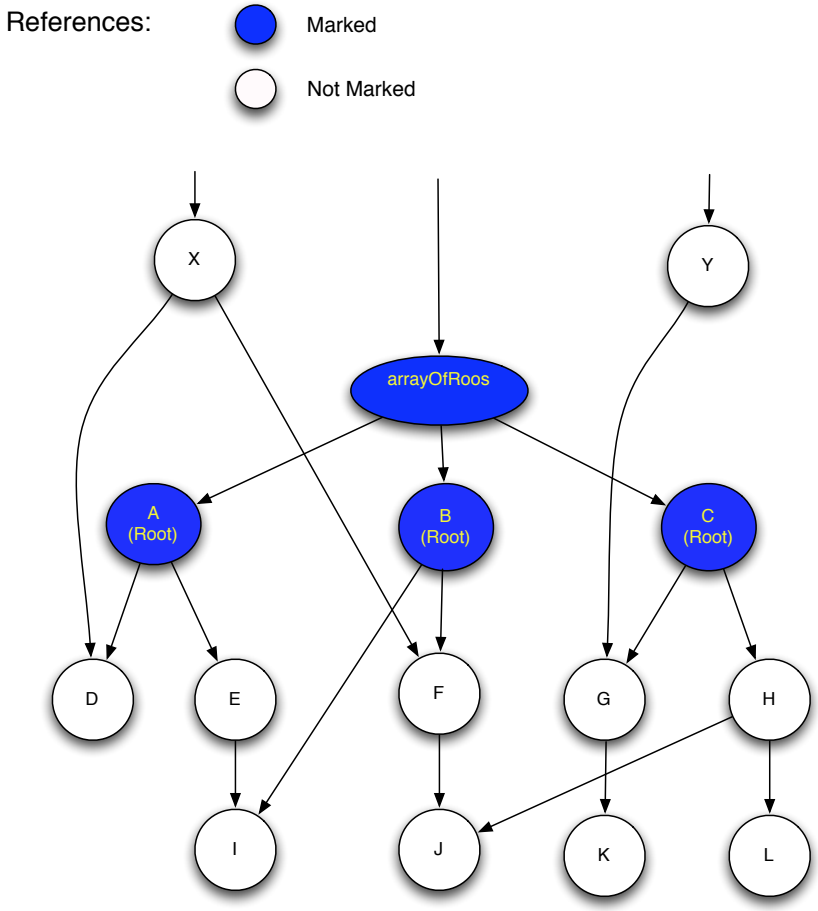

Figure 4: First step: it manually marks the rootArray object and the roots.

6 Creating and storing a segment in depth

The creating of the segment and the storage on it is done by *primitiveStoreImageSegment*. Some primitives are directly written in C. Others are written first in Smalltalk, actually in a limited Smalltalk called Slang, and then they are automatically translated to C. In this case, both primitives are written with Slang. So if you want you can browse them and see the code. In order to this you must first install the VMMaker package in your image. Once that is done, you will be able to see the Interpreter methods *primitiveStoreImageSegment* and *primitiveLoadImageSegment*.

Regarding ImageSegment we will now see the steps that are performed during the creation and storage of the segment.

1. It automatically decides the WordAray size of *segment* and the size of the *outPointers* array using the *hintSize* passed as parameter as a base.

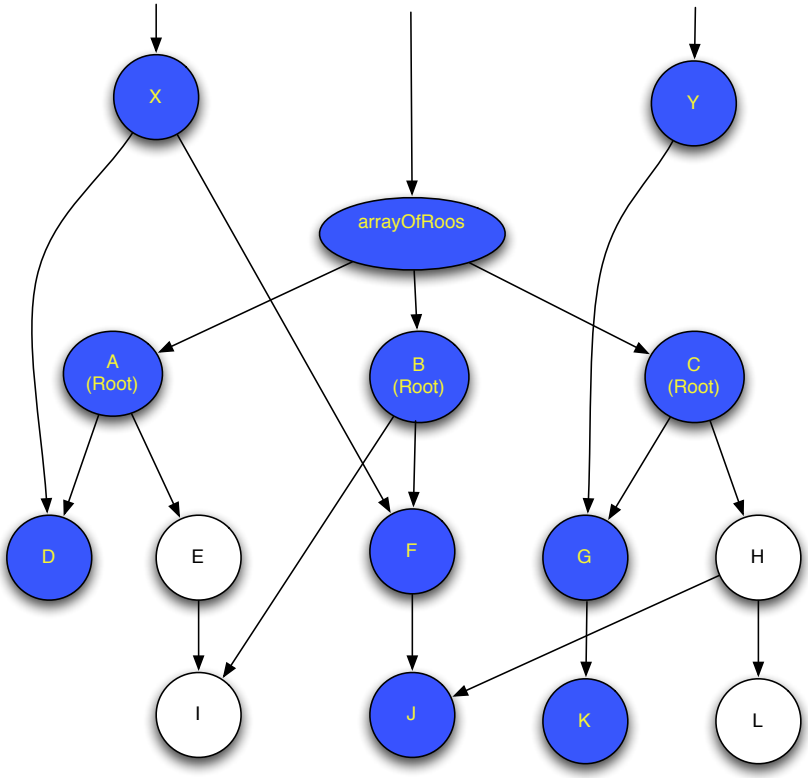

Figure 5: Second step: Do a mark pass over all objets.

2. It tries to create the segment with that size. If there is an error, it assumes that it was due to the fact that the size of the arrays where too small. So, it duplicates the size and try again. And loops until the creation of the segment does not return an error — actually, it returns *nil* if there is an error — . In my opinion, this can lead into an infinitive loop in the case that an error is produced inside the creation of the segment due to another problem than the size of the segments.

3. Creates the segment using the primitive *primitiveStoreImageSegment* that receives the *arrayOfRoots*, the *aWordArray* and the *outPointersArray*.

   (a) Type validations. For example, it checks if the objects are indexable and the header is 3-word — headerType = HeaderTypeSizeAndClass — .

   (b) Split the *outPointers* array in two. It uses the top half of outPointers for saved headers.

   (c) Localize the end and the last segment.

   (d) Write a version number for byte order and version check in the first word. This is useful as the segment can be loaded in another image on another machine and maybe these configurations can change.

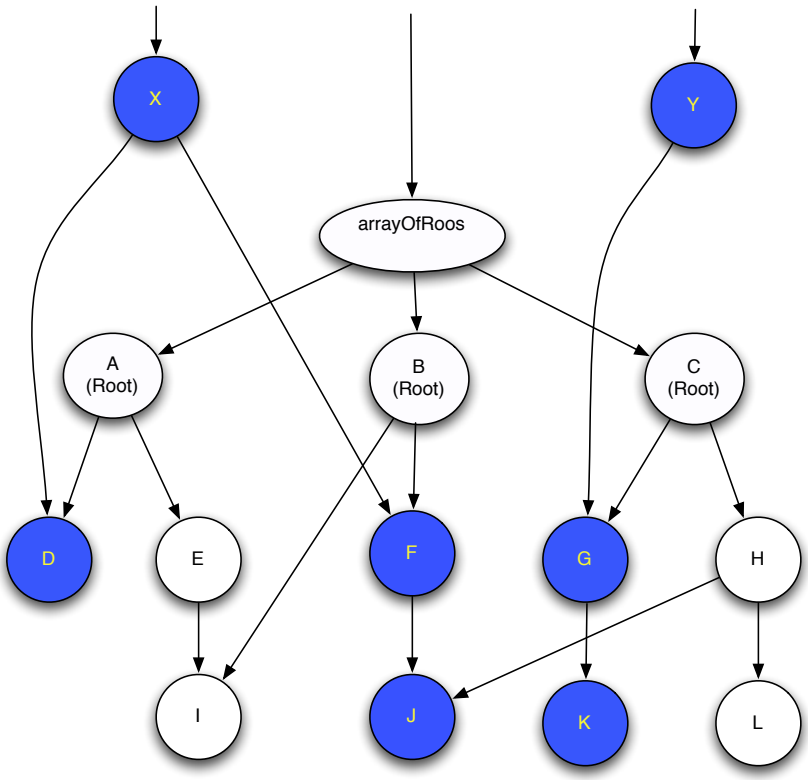

Figure 6: Third step: unmark the rootArray objet and all root objects.

(e) Allocate top 1/8 of segment for table of internal oops and saved headers.

(f) Now it has to search for those objects that have to be included in the segment. To do this, is uses the GC techniques as it has been already explained.

(g) Copy the object that represents the array of roots (and only the array, not also its contents) into the segment, and forward its oop. It stores the oop and the header of the *arrayOfRoots* in the segment and then forwards the old oop to the new one and marks it as forbidden (HeaderTypeFree).

(h) Clear root and mark bits of all headers copied into the segment.

(i) Now run through the segment fixing up and remapping all the pointers. For it pointer it does a couple of things:

   i. It checks if the pointer is to a class or to a normal object.

   ii. It checks if it has already been forwarded (it has the hader type *Header-TypeFree*).

   iii. If it points to an unmarked object (an internal pointer) it copies the object into the segment, and forward its oop. The same as it was done for the roots.

iv. If it points to a marked object (an external pointer) it maps it as a tagged index in outPointers, and forward its oop.
v. Replace the oop by its mapped value.

7 Loading a segment in depth

The loading of a segment is done by the *primitiveLoadImageSegment*. It receives the segment WordArray and the outPointers array as parameters. This primitive returns the array of roots. Internally, those are the steps that are carried out:

1. Localize the end and the last segment.
2. Validations of both arrays, for example that they are indexable in the correct way.
3. Checks both the word reversal and the endianness of the machine it came from. Remember this was done by *primitiveStoreImageSegment* when creating the WordArray. If they are different — the opposite — it reverses the byte type objects.
4. Proceed through the segment remapping pointers. For each object of the array, it checks every object pointer of that object. If the pointer is to an external object (not from the same object) it looks in the *outPointers* array and remaps the pointer.
5. Again, proceed through the segment checking consistency.
6. Truncate the segment word array to size = BytesPerWord — version stamp only — .
7. Finally return the array of roots.

8 Tips, ideas, and others

Here is a list of future ideas, tips and things to be careful.

- Use SmartRefStream instead of ImageSegment. SmartRefStream is used by ImageSegment.
- It would be useful, maybe, if the VM had a primitive that simply let you invoke the content of any ByteArray as native code. This would also have the interesting side effect that you could eventually rewrite the entire low level VM entirely in live objects inside Squeak using Squeak, dumping Slang.

- Currently, the primitive that builds an image segment requires a pre-allocated WordArray. If you are creating an image segment of a significant portion of the data in the image, this effectively doubles your image size (or more, because it over-allocates). It would be great to have a version of this primitive that used a file instead. It seems to be that there is a lot of random access of the buffer that goes on, which means that a file-based primitive is going to be a **lot** slower than the memory based one, but in some cases it is better to pay that price to keep the image size low.
- ImageSegment is used mostly in Projects. Actually, several times ImageSegment code is hardcoded with Project stuff.

9 Warnings and known problems

- Image segments are much faster than reference streams, but you need to take care that there are no references pointing from the outside into the graph of objects you are saving (except for globally used objects like symbols). Else, these and all other indirectly referenced objects end up in a reference stream that in the segment.

  This is a key observation that is relevant for any serious use of image segments. At cmsbox.com they had quite some trouble because of objects being referenced from their model, that should not have been referenced (e.g., referencing a block, e.g., for sorting, does not work well at all). Another difficulty is that when you create the segment, you don't want to have any external objects pointing into your graph. This means that you have to delete, say, all Seaside sessions and their state, clear all caches, terminate background processes etc. They do this by forking off an image, then doing all the cleaning, then taking the segment and storing it to disk, and finally killing the forked image.

- John McIntosh talked about their experience with segments in Sophie was not helpful.

  They attempted to segment off their font menu since it was expensive to build, and really only needed to be changed if the fonts on the machine changed. Since the image was read only we would startup, build the font menu then image segment that off.

  On a restart they just read the segment in, confirmed the machine didn't have font files changed. This worked well in the lab.

  But when they push it to the public we started getting email about machines crashing. Tim and John were just unable to determine why... But it was always related to the point where it read the image segment, sometimes it would crash (rarely) mostly not.

  They backed that out, and settled with a image segment that really just stored forms of each font face for the menu. That seemed to work ok

- Problem with Symbols.

  It is probable that you get multiple same symbols (duplicates) after loading a segment. This seems to be because when creating the segment you do not hold onto these symbols. Like this they do not get into the outPointers ref stream but in the bytearray. This is due to the fact that the only object who is pointing to that symbol is inside the segment. To be precise, the symbols are also pointed to by the symbol table, but only by weak references. Since image segments use the GC mark logic, these pointers are not considered. When installing the segment again, with same symbols existing in the image already, then you get duplicates.

  The *right way* to do this is to strongly hold onto all symbols when creating a segment. For example, you can hold it keeping this into an instance variable of an object: *symbolHolder := Symbol allSymbols*. When ImageSegment uses the GC techniques to detect which objects are *only* pointed from inside of the segment, the symbols is not found (because it is accessible trough *systemHolder*) and thus, it goes to outPointers instead of ByteArray.

  And of course, if it is in outPointers instead of ByteArray when the segment is loaded again, yo don't create a symbol again but use the same object (the one of the oop).

- With lazy loading the probability is high that all your objects or classes are accidentally loaded. Adrian recently experimented with using image segments to stub out classes, or rather, group of classes. Somebody doing allObjectsDo: or allClasses etc. brings in all your segments. Probably a more sophisticated strategy is needed than just lazily load these back in.

10 Discovering used and unused classes

There is some work that it is slightly related to ImageSegment and it is used to discover used and unused classes during a period of time.

The Virtual Machine pays attention not to crash totally when you put a nil in a Method Category. In addition, there is a possibility to force a MethodDictionary fault with the message *ClassDescription»induceMDFault*. This method copies the *methodDict* in the *organization* slot. Actually, it creates an Array with the *methodDict* and the *organization* and set it in the *organization* slot. Finally, it sets a *nil* to the *methodDict* instance variable. After this is done, when that class is used and the *methodDict* is asked — it is very important to acces the *methodDict* using the accessor — it checks whether it is nil or not. If it is nil, it will search for that *methodDict* in the Array of the *organizaction* it will then copy it to the *methodDict* instance variable and will log in *Smalltalk at: #MDFaultDict*. Sometimes you will see that you have classes that were used but you do not know why or who used it. To solve this question, we can have that log. But in order to have that

trace, you have to first evaluate something like *Smalltalk at: #MDFaultDict put: Dictionary new.*. The log will be then put in that dictionary.

Taking that explanation in mind, you can follow these steps:

1. Send the message *ImageSegment»discoverActiveClasses* which forces a MethodDictionary fault in all classes. Yes! It sends the message *induceMDFault* to all classes except classes like Array, Object, Class, Message and MethodDictionary. After this, all classes but those ones, will have the *methodDict* with *nil*.

2. Do a few things, use the system for a while. Maybe even save and resume the image.

3. Now, it is easy. All classes that still have a nil in the *mehodDict* is because they were not used, and those who have something different from *nil* is because they were. With the message *ImageSegment»activeClasses* or *ImageSegment»unactiveClasses* you can restore all remaining MethodDictionary faults and return the active or unactive classes respectively. There is also the message *ImageSegment»swapOutInactiveClasses* which makes up segments by grouping unused classes by system category and write them to disk. There are some categories that are excluded because they have problems, like System and Kernel.

After doing all those steps you can then inspect *Smalltalk at: #MDFaultDict* and you will see that in the dictionary you will have an element per class. That element is a ByteString that contains all the *stack trace*. This shows you all the messages and why that class was used.

A very important thing is that if you send the message *discoverActiveClasses* you must then have to send the message *activeClasses* or *unactiveClasses* or *swapOutInactiveClasses* because otherwise you will have a lot of classes with the *methodDict* in *nil*.

I do not know other situation where the MD can be in *nil* rather than using the message *induceMDFault*. You can put it in nil using the message *instVarAt:put:* but even with this the recover will not work because the MD will not be found in the *organization* slot.

This approach of swapping out inactive classes have two big problems:

1. The granularity is too high: the class. As soon as only one method is invoked, the whole class is recovered.

2. It is very common to recover **all** classes doing a simple action. For example, searching for implementors or senders of a certain method. Because of this, when you are developing in your Smalltalk image is very likely that all classes are recovered in less than 5 minutes of work. It is not useful. Browsing or editing code seems not to be a problem, thus. Maybe this approach has more sense in production images rather than development.

As you may noticed ImageSegment uses this feature of Squeak/Pharo that let you force a MethodDictionary to discover used and unused classes. However, this should not be part of the ImageSegment class. It is just one possible uses of ImageSegments, but not part of it. This is why these methods should be moved to another classes or packages.