Object Swapping Challenges: an Evaluation of ImageSegment

Mariano Martinez Peck[1,2,*], Noury Bouraqadi[2], Stéphane Ducasse[1], Luc Fabresse[2]

Abstract

In object-oriented systems, runtime memory is composed of an object graph in which objects refer to other objects. This graph of objects evolves while the system is running. Graph exporting and swapping are two important object graph operations. Exporting refers to copying the graph to some other memory so that it can be loaded by another system. Swapping refers to moving the graph to a secondary memory (*e.g.,* a hard disk) to temporary release part of the primary memory (*e.g.,* RAM).

Exporting and swapping are achieved in different ways and the *speed* in presence of large object graphs is critical. Nevertheless, most of the existing solutions do not address well this issue. Another challenge is to deal with common situations where objects outside the exported/swapped graph point to objects inside the graph. To correctly load back an exported subgraph, it is necessary to compute and export extra information that is not explicit in the object subgraph. This extra information is needed because certain objects may require to be reinitialized or recreated, to run specific code before or after the loading, to be updated to a new class definition, etc.

In this paper, we present all general problems to our knowledge about object exporting and swapping. As a case of study, we present an analysis of ImageSegment, a fast solution to export and swap object graphs, developed by Dan Ingalls. ImageSegment addresses the speed problems in an efficient way, as shown by the results of several benchmarks we have conducted using Pharo Smalltalk. However, ImageSegment is not a panacea since it still has other problems that hampers its general use.

Keywords: Object swapping, serialization, ImageSegment, Smalltalk, Object-oriented programming

1. Introduction

The object-oriented programming paradigm has been widely accepted in the last decades. Nowadays, it is the most common programming paradigm and is applied from very small systems to large ones as well as from small devices to huge servers. Since generally in this paradigm objects point to other objects, the runtime memory is represented by an object graph.

This graph of objects lives while the system is running and dies when the system is shutdown. However, sometimes it is necessary, for example, to backup a graph of objects into a non-volatile memory to load it back when necessary, or to export them so that they can be loaded in a different system. The same happens when doing migrations or when communicating with different systems.

Most applications need to persist graphs of objects so that they are not lost when shutting down the system. This is known as "persistency" and a general solution is to have a database that takes care of this problem.

In addition to this, large applications may occupy a lot of memory (hundreds of megabytes or even gigabytes). Therefore, application spatial scalability requires to temporarily swap out unused object graphs from primary memory (*e.g.,*RAM) to secondary memory (*e.g.,*hard disk) [Kae86]. The intention behind this is to save primary memory or, even more, to be able to run more applications in the same amount of memory. The same happens with systems that run in embedded devices or in any kind of hardware with a limited amount of memory like robots, cellphones, PDAs, etc. In these cases, swapping out unused objects saves memory, but it should not lead into thrashing as this will degrade the

[☆]This work was supported by Ministry of Higher Education and Research, Nord-Pas de Calais Regional Council and FEDER through the 'Contrat de Projets Etat Region (CPER) 2007-2013'.

[*]Corresponding author

Email addresses: marianopeck@gmail.com (Mariano Martinez Peck), noury.bouraqadi (Noury Bouraqadi), stephane.ducasse@inria.fr (Stéphane Ducasse), luc.fabresse@mines-douai.fr (Luc Fabresse)

[1]RMoD Project-Team, Inria Lille–Nord Europe / Université de Lille 1.

[2]Université Lille Nord de France, Ecole des Mines de Douai.

system's performance. By trashing we refer to the situation where data is rapidly written to and read from different types of memory, *i.e.,* constant data swapping.

Approaches and tools to export and swap object graphs are needed. One of the biggest problems (and difficult to solve) with export or swap solutions is their performance. The approach must scale to large object graphs. However, most of the existing solutions do not solve this issue properly. This is usually because there is a trade-off between speed and other quality attributes such as readability/independence from the encoding. For example, exporting to XML [SIX] or JSON [JSO] is more readable than exporting to a binary format, since the programmer can open it and edit it with any text editor. But a good binary format is faster than a text based serializer when reading and writing. Depending on the user usage of an object serializers, the performance can be a crucial aspect.

There are serializers like *pickle* [pic] in Python or Google Protocol Buffers [pro], that lets the programmer choose between text and binary representation. For debugging or while developing one can just use text based, which is easy to see, inspect and modify, and then, at production time, one can switch to a binary format.

An important question is, do we need an object serializer? What is wrong with just using a binary write stream? The problem is that binary write streams receive a binary array (for example, a ByteArray) as input. So we first need to serialize the object subgraph, and this is where we find problems like performance, cycles, etc. The object serializer takes an object graph as input and answers a binary array. Once we have such array, then we can do whatever we want, like write it on a binary write stream or on a socket.

It is common to have objects from outside the exported/swapped object graph pointing to objects inside the graph. This makes it a challenge to detect which objects should be swapped or exported and how they should be handled appropriately.

A usual problem is that the class has been changed or is different. For example, a graph of objects from an accounting system is exported and loaded in an enterprise resource planning system. In the graph, there are objects of the class *User*. But class *User* can be different in both systems (it might even not exist). It is also possible to swap out objects, change their classes (suppose an instance variable named *creditCard* was added, and *age* was removed) and then load back instances of the old class. At writing time, the tool should store all the necessary information (related to class shape) to deal with these changes, and at load time, objects must be updated in case they are required. There are object serializers that are quite limited in this aspect. For example, the Java Serializer [jav] supports adding or removing a method or a field, but does not support changing an object's hierarchy or removing the implementation of the Serializable interface.

Ungar [Ung95] claims that the most important and complicated problem was not to detect the subgraph to export, but to detect the implicit information of the subgraph that was necessary to correctly load back the exported subgraph in another image. Examples of this information can be whether to export an actual value or a counterfactual initial value or whether to create a new object in the new image or to refer to an existing one. In addition, it may be necessary that certain objects run some specific code once they are loaded in a new image.

Defining how to serialize, where, when and what to export, which file format, etc., are just a few more problems that have to be addressed too [MLW05].

The contribution of this paper is twofold. On the one hand, we introduce a precise description of objects swapping and exporting, and related challenges. On the other hand, we provide a detailed analysis of the ImageSegment solution and compare it with related work.

The remainder of the paper is structured as follows: Section 2 defines and unifies the concepts and names that are used throughout the paper. Section 3 presents the problem. We describe the major needed steps to export and swap a graph of objects, and also explain the most common problems and challenges. Section 4 presents a deep analysis of ImageSegment, a solution for both, objects export and swapping. Section 5 explains how ImageSegment uses Garbage Collector facilities in an interesting way to detect objects to be swapped out. Benchmarks and discussions are shown in Section 6. In Section 7, we describe the issues and the opportunities for improvement. Finally, in Section 8 related work is presented, before concluding in Section 9.

2. Glossary

To avoid confusion, we define a glossary of terms used in this paper. As example we use the object graph shown in Figure 1 to explain these concepts.

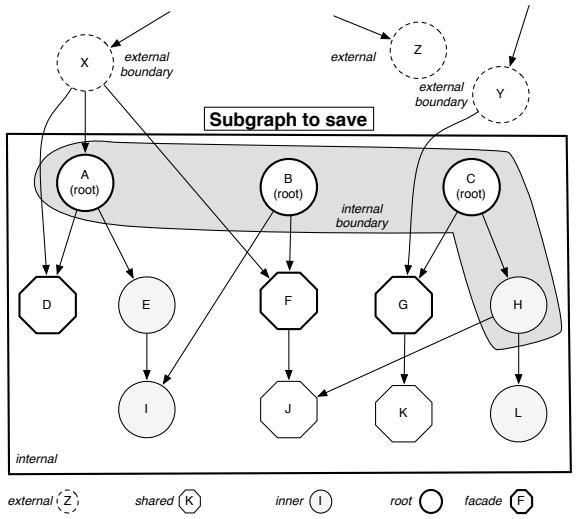

Figure 1: A graph to be saved.

External objects are objects outside the graph to process. Example: X, Y and Z.

Root objects are user-defined objects. They are the input provided by the user to the serializer. Example: A, B and C.

Internal objects are *root objects* and all the objects that are accessed through them. Example: A, B, C, D, E, F, G, H, I, J, K and L.

External boundary objects are *external objects* that point to *internal objects*. Example: X and Y.

Shared objects are *internal objects* that are accessed, not only through the roots of graph, but also from outside the graph. Example: D, F, G, J and K.

Internal boundary objects are *internal objects* which point to *shared objects*. Example: A, B, C and H.

Facade objects are *shared objects* which are pointed from *external objects*. Example D, F and G.

Inner objects are *internal objects* that are only accessible from the *root objects*. Example: E, H, I and L.

3. General Steps and Challenges

Before analyzing any particular solution to the problem of exporting and swapping graphs of objects, it is necessary to understand, not only the general steps that have to be followed, but also their challenges and problems. These steps, together with their problems and challenges, are completely general and they are independent of the technology.

We explain 'object export' before describing 'object swapping'. These two operations are similar: *swapping* can be considered as an *export* with an additional constraint on object identity. Nevertheless, this constraint may imply a completely different set of problems and solutions. This is the reason why in this paper we differentiate both operations.

3.1. Export

Export is usually needed when wanting to transfer an object graph from one system or application to another. For such purpose, the graph of objects should be written in a file or sent through a Socket. In this case, the objects from the original system are not modified neither removed. They are just serialized (converted into a sequence of bytes) and then written into a file, a Socket, etc.

The following is a possible list of general steps for a general solution to export an object graph including its most common problems and challenges:

1. *Identify first sets of objects:* starting from user defined *root objects*, the first step is to compute the *internal objects* set. This means that the graph has to be traversed and processed. It starts from the roots of the graph and iterates over their references. For each processed (depending on the implementation, processed can mean copying the object into an array, updating objects pointing to it, check flags to avoid cycles, etc) object, its referenced objects are then recursively processed too.

   While trying to put this step into practice, it is likely to face some problems like speed and cycles. Speed may not be very important for small graphs, but it definitively is in moderate and large graphs. When we refer to a graph size we mean the number of objects in the graph. The only way to scale and be able to compute large graphs is having a decent performance.

   Another problem that has to be addressed is the cycles inside the graph. In an object graph, objects may contain references to other objects and generate cycles or loops. The selected approach has to be able to deal with these cycles properly and without generating an infinite loop. Creating a collection with exported objects and checking for every object whether it is already in the collection, and

adding the object if it was not exported yet, is not a scalable solution. With large object graphs, this approach is slow and uses considerable amount of memory. This is why certain object serializers like Oracle Coherence[ora] does not support cycles inside the graph.

The result of this step may identify: *internal objects* set and, depending on the implementation, other lists such as the *shared objects set* and the *inner objects set*. In that case, the solution may require a complete memory traversal. (to detect shared and inner objects).

2. *Export a set of objects to a file*: once the graph is computed and the list of objects is ready, such list can be written into a file. Once again, performance is a big issue. In this case, two different facets should be considered: (a) the time to write (export) the graph to a file and (b) the time to read it and load it in memory.

   Sometimes, a good performance is really needed only while loading but not that much while writing. This scenario is the motivation behind Parcels [MLW05], a fast binary format developed in Visualworks Smalltalk. The assumption is that the user may not have a problem if saving code takes more time, as long as loading is really fast. In the context of software centralized repository, developers of a software project commit versions of their components and final users download them into their own environments. Load (read) speed is really important as final users should be able to download and install the software as fast as possible. This is at the expense of a slower writing time since commit may be done one or few times while, on the other hand, loading may be done hundreds of times.

   Another decision is which kind of format to use. The answer to this depends on the goals of the tool. For example, a binary format or a text format can be used. Binary formats are faster than text formats but the latter ones are more readable by humans and computers. Because XML is text, it is quite easy to understand by a human. From the computer point of view, it is easy also since the file can be opened and edited with any text editor program. In this situation there are trade-offs. Each approach has its own advantages and disadvantages.

   To conclude, it is really important to define the goals of the tool (exporter) to define proper solutions and achieve the needed performance.

3. *Load a set of objects from a file:* When a graph of objects is exported, it will be probably loaded back later on.

   The first problem faced is where that graph will be loaded. That can be either in the same system where it was originally exported or in another one. The main reason behind this question is that the answer may determine which objects to include in the export and, of course, enable to avoid duplicates or inconsistencies. For example, should all *internal objects* be included no matter if the graph will be loaded back in the same system or in a different one? Or should only *inner objects* be included?

   Once again, performance is an important issue.

3.2. Swapping Objects

Swapping is a combination of exporting and a constrain on object identity. The aim is to be able to load an object graph back later in the same system where it was originally exported. When having a graph of objects in primary memory which probably will be not used, that graph is swapped out to secondary memory and loaded back when needed to use less memory. This was the idea behind LOOM [Kae86] (Large Object-Oriented Memory), which implemented a swapping mechanism between primary and secondary memory for Smalltalk-80.

One of the key points of swapping is how and which objects have to be discarded or replaced to be able to automatically load back the swapped out graph when it is necessary. Generally, Proxy objects are used. These proxies can then load back the swapped objects when certain events happen, for example, when they receive a message.

The steps and challenges in this case are quite similar to the export case. The main difference is that *shared objects* may not be exported in the file as they are being referenced from outside the graph. The objects that are exported are the *inner objects*. Possibly, another implementation can also export the *shared objects* and then, at the moment of loading, prevent duplicates and instead use the objects that are present in the image where the import is happening.

One of the challenges of swapping is to carefully choose which objects have to be replaced by a proxy and which ones are directly swapped out.

Another problem is, not only how to select which objects to swap, but more importantly, how to fix the *internal boundary objects*. It is common to have objects from outside the graph pointing to objects inside the graph. In those cases, it is necessary to determine how to detect

and what to do with them. The solution has to assure consistence between *external and internal objects* and prevent duplicates.

The fact of being smart, for example being aware of the memory and CPU overhead resulting from the swapping process, is also a challenge. Indeed, the swapping tool will instantiate its own objects and, thus, use memory. At the same time, the swapping mechanism requires, not only memory, but CPU. It is useful to check before swapping if it is worth doing it or not.

An interesting question is why should an application program (from the viewpoint of the operating system, a Smalltalk virtual machine is nothing else) be blamed for moving parts of itself form primary to secondary memory? Why that task cannot be left to the operating system and its efficient management of virtual memory? There are some reasons:

The Garbage Collector. Objects on disk also need to be garbage collected. But if we want to swap something via the OS, we are not allowed to touch it at all. If not, it gets loaded in. It may be easier to do this when we control the swapping directly. Doing this with the OS where we cannot control anything, is not easy.

Persistency. Memory swapped by the OS is by definition not persistent. What happens if a Smalltalk image is bigger than main memory and we just want to quit the Virtual Machine? We can not swap everything in and then write it on disk and then back when starting again, as all the objects do not fit into memory.

Used and unused objects. There are objects being referenced by other objects but that are not used (accessed). The garbage collector works by reachability. As those objects are reachable, they are not garbage collected even if they were unused. A custom Smalltalk virtual memory implementation can take advantage of this and just swap out unused objects. The Operation System will mix them as it is not aware of objects nor whether they are used or not.

Granularity. Most of the Operating System virtual memory approaches use pages for grouping elements. A Smalltalk virtual memory implementation can provide fine-grained paging by using object instead of page granularity.

3.3. Still more problems

Now that we have already discussed about the general steps and challenges of exporting and swapping an object graph, it is time to analyze more problems that should be faced. For example, Ungar [Ung95] claims that a directly constructed concrete program is not complete. Although the objects comprising a program contain all the information needed to run it, they lack information needed to save and reload it into another world of objects.

Class changes. Consider a graph with instances of some class X that is exported from a system A and loaded into a system B. The problem arises if class X of system B defines a structure different from class X in system A. We face the same situation when swapping if a class of some swapped out objects is changed before they are loaded back.

There are different kinds of changes like adding, removing or renaming a method, class or instance variable, or changing the superclass, etc. Not all solutions solve all these change types. Indeed, most of solutions do not solve all of them, and they have a limited number of supported change types.

Objects duplication. Should all exported objects be created in the system where they are loaded? Or some should point to already existing objects? For example, there are certain objects in Smalltalk that have to be unique and should not have duplicates. For instance, *true*, *false*, and *nil* are the unique instances of the classes True, False and UndefinedObject respectively. Hence, if in the subgraph there are objects pointing to any of those objects, it is necessary to avoid the duplication when loading them in a new image and make them point to the already existing objects. For swapping, this is not a problem because those objects like *true*, *false* and *nil* are referenced by other objects in the system (in this case, these objects are pointed from the *specialObjectArray*) so they behave like any other *shared object*.

The biggest problem is how to detect which objects should be created and which ones should be point to existing ones. Ungar et al. [Ung95] decided to annotate objects with the needed information for dealing with these situations.

Recreate and reinitialize objects. When the swapped or exported subgraph is loaded back into memory, should those objects be recreated and reinitialized? One solution can just load the bytes of the object (header and instance variables) while another one can create a new instance of the same class using the normal message for instance creation and then

copy one by one each instance variable. Both solutions are very different. For example, in Smalltalk, the message to create an object is new (which sends the initialize message) or basicNew:. Hence, the object can be initialized again.

Suppose there is an object that has an instance variable with the Operating System name, and initialize method does the job of initializing such instance variable. In this case, we need to reinitialize this object in case it was loaded in a computer with different Operating System.

Should the solution do that or not? Does we need a unique global solution or a per object decision?

Code executed after loading. It may be necessary that the exported or swapped objects do specific tasks once they are loaded in the new system. For example, all Set instances can be rehashed (because the hash of the containing elements might have changed), a Process rescheduled, etc. Even the reinitialization of objects can be a particular case of this one.

In addition, there are also domain specific tasks. Objects of domain classes may need specific code to run once they are loaded in a new system. The tool should be flexible enough to support this.

4. ImageSegments

In this section, we describe ImageSegment, a software library implemented in Pharo [BDN+09] Smalltalk. ImageSegment was originally developed in Squeak by Dan Ingalls[IKM+97].

ImageSegment provides most of the features mentioned in the previous section and also addresses some of the issues already presented. In addition, it supports both: object export and object swapping. ImageSegments is therefore a really good candidate to understand the deep issues that are involved in building a fast serializer.

4.1. ImageSegment Object Swapping Principles

In the ImageSegment's object swapping implementation, there is a list of user defined root objects that are the base of the graph. The graph is then stored in an ImageSegment. Once this is done, the ImageSegment can be swapped to disk and the original objects are removed from the Smalltalk image.

In ImageSegment not all the objets from the graph are included in the swapped graph. Only the objects which are **only** accessible from objects inside the graph are included. These objects are what we have already defined as *inner objects* in section Section 2. To resolve the problem of identifying *inner and shared objects*, ImageSegment uses *Garbage Collector* facilities.

An ImageSegment is represented by an object that contains three sets of objects:

1. *root objects*: these objects are provided by the user and should be the starting point of object graph,
2. *inner objects*, and
3. *shared objects*.

Once the ImageSegment is created and the above sets are computed, it can be swapped out and the root objects are replaced by proxies. The *inner and root objects* of the graph are then written into a file.

Once the roots are replaced by proxies, there are no more references from outside the graph to the objects that were written into the file and, therefore, the garbage collector deletes them. As a consequence of this, an amount of memory is released.

To install back the ImageSegment from file, there are two different ways:

- Sending any message to one of the proxy objects: remember that roots were replaced by proxies. So, all the objects that were pointing to roots are now pointing to proxies. Whenever a proxy receives a message it will load back the object graph in memory.
- Sending a provided message to the ImageSegment instance.

4.2. Object Swapping Step by Step

Before talking about ImageSegment details, it is necessary to explain a few concepts behind its object swapping mechanism. It is also mandatory to define a new term in our glossary: *serialized objects*. These are the objects that are serialized and then swapped out. In ImageSegment, *serialized objects* is a WordArray that represents *inner objects* together with the *root objects* and the array that references them. This means that all those objects are serialized and stored as *words* (32-bit unsigned Integer values) into a WordArray.

When swapping, an ImageSegment instance is created for the array of root objects. An ImageSegment instance has three important instance variables: an array with references to the *root objects*, a WordArray representing the *serialized objects* and an array with references to the *shared objects*. In this paper, those

terms are used. However, in the current ImageSegment implementation, those instance variables have different names: *root objects array* is called *arrayOfRoots*, *serialized objects* array is just *segment* and *shared objects* array, *outPointers*.

To understand ImageSegment object swapping, it is important to analyze in details what is done step by step. We continue with the object graph of Figure 1. Here are all the necessary steps to swap and load back a graph of objects into a file.

1. Create and setup the ImageSegment object.

   The first task is to create an ImageSegment instance for an array of root objects that represents the graph. Once this is done, ImageSegment needs to identify the list of *shared objects* and the list of *inner objects*. To do this, it uses Garbage Collector facilities as it is explained later.

   Figure 2 shows the results in our example.

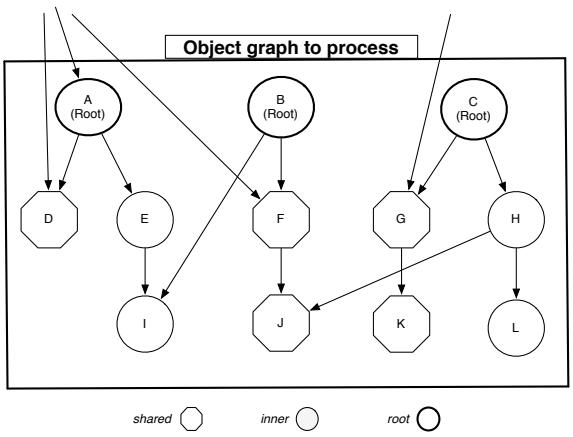

Figure 2: Identifying shared and inner objects

   The next operation copies the *words* representing the *serialized objects* into the WordArray. The copy mechanism used is not a standard copy done in Smalltalk, but a special copy done in the Virtual Machine that just copies the *words* representing objects. In the Squeak VM, an object is composed by an object header which is a sequence of bits for the GC, hash, pointer to its class, etc and a set of instance variables. An instance variable can store the address of another object (when an object points to another object) or directly store special objects, such as SmallInteger instances.

   So, the copy is like a chunk of memory copy which copies, for each object, the object header plus its instance variables. The *serialized objects* WordArray has the same binary format than the Pharo image file.

   This sounds easy but there are still some problems to solve. In our example, object *E* points to *I*. This means that *E* has an instance variable with *I* address (we assume that *I* is a fixed fields object, and not a SmallInteger, CompiledMehtod, BlockClosure, etc). When those objects are written into the file, the memory address does not make sense anymore. Furthermore, in Squeak VM, objects do not have a unique identifier. So, how can *E* still point to *I* when they are in file? What is more, how can the *serialized objects* WordArray be loaded back correctly?

   When ImageSegment writes an object to file, it checks whether it is pointing to an *inner object* or a *shared object*. If it is pointing to a *inner object*, then its instance variable containing the address is updated so that it points to the offset of the object in the WordArray. In the case of a *shared object* it is the same but it points to the index in the *shared objects array* of ImageSegment. Remember that this array is never swapped out and remains in primary memory. If any of the *shared objects* is moved by the Garbage Collector, then the *shared object array* is automatically updated. Hence, when loading back the subgraph, the pointers will be correct.

   The copied objects in the *serialized objects* WordArray are not in the normal Pharo memory space. This means that those copied objects are not really seen by the system as standard objects. They are just represented as *words* inside a WordArray of an ImageSegment object.

   In summary, the graph is computed and traversed while the *inner objects*, the *root objects* and the array that references them are encoded in the *serialized objects* WordArray instance variable of the ImageSegment object. In addition, ImageSegment has an instance variable with the *shared objects*. However, at this point, the original *inner objects* are still present and referenced in the runtime system (although, at the same time, they were copied as *words* in the WordArray)

2. Extraction.

   This operation replaces all the roots with proxies by using the method elementsForwardIdentityTo: otherArray implemented in Array, which delegates to a Virtual Machine primitive that does a bulk become with

both arrays. In our case, one array contains the root objects while the other one the proxies.

Once that step is done, the *serialized objects* WordArray is the only holder of those objects, not as normal objects but as a WordArray. The original objects are reclaimed by the garbage collector and the proxies remain to bring the *serialized objects* WordArray back if necessary. If one of the proxy receives a message, all the *serialized objects* are loaded back in memory.

On the other hand, *shared objects* are kept in the original image (they are not garbage collected) as they are referenced not only from the ImageSegment instance variable pointing to them, but also from *external boundary objects*.

Figure 3 shows the objects state of the example after the ImageSegment creation and extraction steps.

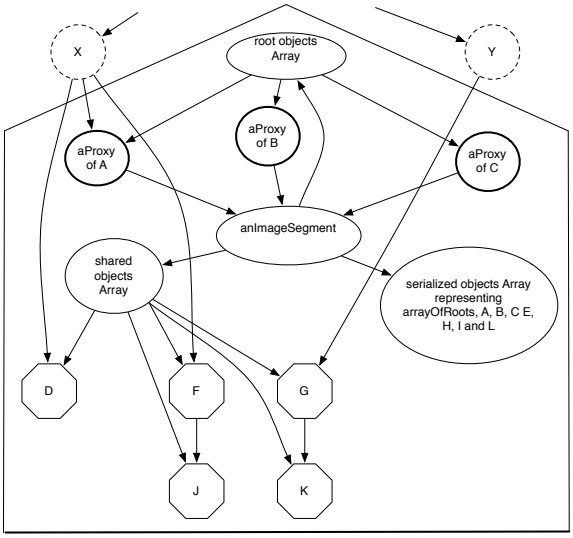

Figure 3: Objects state after the ImageSegment extraction

3. Write the *serialized objects* WordArray into a file using the method nextPutAll: of MultiByteFileStream. Afterwards, the instance variable that refers to the WordArray is put to nil so that the Garbage Collector can reclaim memory space.

4. Load the *serialized objects* WordArray from a file.

   This means reading from file and restoring the *serialized objects* WordArray in primary memory. ImageSegment object is still in memory and only the *serialized objects* were swapped out.

   At this point, it is necessary to perform the opposite of what has been done when creating the ImageSegment instance. Objects inside the graph have to be updated so that they point to the real addresses instead of the objects, and not to an offset or index in an array. With *inner objects* it is easier because objects were already loaded and they already have a memory address, so we only need to update its pointers. In the case of *shared objects* we have to first fetch the object pointer that is the *shared objects array* and then update the pointers.

   It is important to note that none of the *serialized objects* are recreated or reinitialized. ImageSegment does not use new nor basicNew to create them. The *serialized objects* WordArray is just loaded into primary memory as a chunk of memory and then the pointers are updated.

   To load the *serialized objects* back, there are two options:

   (a) Sending a message to any of the proxy objects. When sending a message to any of those proxies, it will load the *serialized objects* WordArray and replace the proxies with the original *root objects*. To achieve this, the proxy implements the doesNotUnderstand: message to load back *serialized objects*.

   (b) Sending the message install to the ImageSegment object.

ImageSegment supports class evolution. This means that when the *serialized objects* WordArray is being loaded back in primary memory, it has to check if the classes of those objects have changed since the time they were swapped out. If such is the case, those objects are fixed and updated to the new class definition.

4.3. Exporting an Object Graph

This feature allows one to create an ImageSegment for an array of root objects, to write it into a file, and to finally load it in another image. In this case, the objects of the graph are not removed or even changed in the original image.

The first step is exactly the same as the "Create the ImageSegment object" of the swapping scenario: the graph is computed and traversed, the *serialized objects* are encoded in the WordArray and the *shared objects* are also in an instance variable of ImageSegment.

After that, the ImageSegment is exported into a file using a SmartRefStream. A difference with the swapping scenario is that in such case the ImageSegment instance is kept (included the *shared objects* Array) in the

image to load back the serialized objects in memory. On the contrary, in the exporting scenario the file also includes the *shared objects* Array and the ImageSegment instance is garbage collected after writing the file.

In details, when exporting an ImageSegment, it just serializes the whole ImageSegment instance and all its referenced objects (this means also the *shared objects*) with a SmartRefStream.

As said, when exporting, *shared objects* are also written into the file. There are some objects in Smalltalk that have to be unique like *nil*, *true*, *false*, etc. How does ImageSegment avoid duplicating these objects when loading the ImageSegment in another image that already has these objects? The solution is provided by the class SmartRefStream as it has specific ways of reading certain objects. This is explained more in detail in Section 4.4.

This capacity of exporting an object graph is also a kind of persistency mechanism. For example, it can be used to persist a web application.

4.4. How does ImageSegment solve the problems?

ImageSegment solves most of the mentioned problems in Section 3.3. As ImageSegment's export functionality uses SmartRefStream, the former solves some problems and the latter solves others.

Class changes. This problem depends whether we are swapping or exporting. When swapping, if we modify a class, it will trigger the update of its instances. Hence, accessing those instances that were swapped out to a file, requires loading back the ImageSegment into memory. This is exactly the normal case where an ImageSegment is loaded in primary memory because one of the proxies received a message. In this case, the received message is something related to the class reshape or instance update.

In the export case, it is more complicated since it is necessary to write enough information in the file about the names of the instance variables of all outgoing classes. Note that not only an instance variable can be renamed, but also a class. All these tasks related to class reshape are performed by SmartRefStream. Indeed, this is the extra functionality that SmartRefStream provides over ReferenceStream.

When an object is written into the file, no one knows how the classes will change in the future. Therefore, all conversion must be done when the file is read. SmartRefStream stores enough information in the file about the names of the instance variables of all outgoing classes. The conversion of old objects is done by a method in each class called convertToCurrentVersion: varDict refStream: smartRefStrm. At writing time, a prototype of this method is created. The programmer must edit this method to (1) test if the incoming object needs conversion, (2) put non-nil values into any new instance variables that need them, and (3) save the data of any instance variable that are being deleted. For more details, read the class comments of SmartRefStream.

Object duplication. This distinction between creating new objects or pointing to existing ones happens only when exporting since, in swapping, the *shared objects* remain in memory. SmartRefStream solves this problem using a Dictionary where *keys* are class names and *values* selectors. For example, there is an element in the dictionary which has *True* as key and the selector *readTrue* as value. This means that when an instance of True is read, it is done by sending the message *readTrue*. This method simply returns the *true* object. This way we ensure that there is always a single *true* object in the image.

In addition to this, the class SmartRefStream allows the programmer not only to specify the way instances of certain classes are read, but also to decide how they are written. Examples are CompiledMethod, Symbol, Class, among others.

Code executed after loading. One can implement the method startUpFrom: anImageSegment in any class. That method has to answer the selector to be run in its instances. When the *serialized objects* WordArray is loaded in another image using SmartRefStream, it checks for each object if its class implements such message. If true, it sends startUpFrom: anImageSegment message and gets the selector as result. Finally, sends that message to the object.

The message is free to do anything. It can be used, for example, to rehash Set instances, to reinitialize objects (remember ImageSegment does not use new nor basicNew when loading objects) or simply to do specific tasks.

Notice that this feature is implemented between SmartRefStream and ImageSegment. When an object subgraph is exported using SmartRefStream, at loading time, it sends the mes-

sage comeFullyUpOnReload: aSmartRefStream to each object of the subgraph (there is a default implementation in Object that does nothing). When exporting an ImageSegment with SmartRefStream, the ImageSegment instance is just one of those objects, and it will receive the message comeFullyUpOnReload: aSmartRefStream. The trick is that ImageSegment implements the message comeFullyUpOnReload: with the responsibility of sending the message *startUpFrom: anImageSegment* to the classes of its objects.

The previous paragraph means that ImageSegment supports code execution and object reinitialization only when exporting. This makes sense, since objects usually need those features only when loading them in another image.

A problem of this solution is that it is at class level. This means that one cannot work with a particular object but only with a particular class that will apply such behavior to all its instances.

Recreate and reinitialize objects. With ImageSegment this problem is a subset of the previous one. The code executed after loading can take care of the objects recreation or reinitialization.

5. ImageSegment and a Smart Use of Garbage Collection Facilities

ImageSegment uses *Garbage Collection* facilities to identify which objets of the graph are *inner objects* and which ones are *shared objects*. Once these objects are discovered, defining the list of *serialized objects* is easy.

To explain each step of this solution, we use the same example used so far (see Figure 1). The steps are:

1. First, all *root objects* and the array referencing them are marked by setting the Garbage Collector bit in their object headers. This will prevent marking objects reachable from them in the next step (see Figure 4).

2. Afterwards, a mark pass is done over all the objects in the image by recursively marking all objects reachable from the roots of the system. This process will stop at our marked roots leaving *inner objects* unmarked (see Figure 5).

3. *Root objects* and the array referencing to them are unmarked, leaving unmarked the transitive closure of objects accessible from the roots and no where else (see Figure 6).

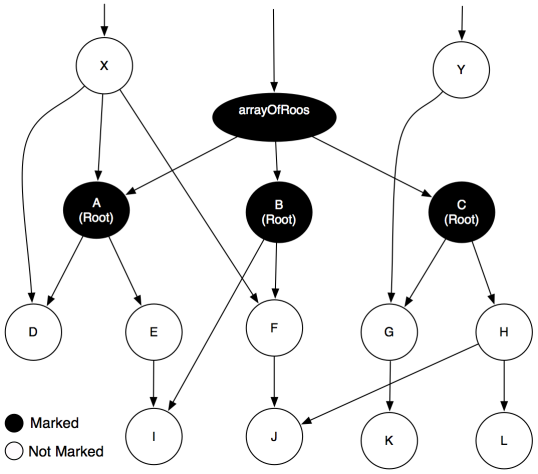

Figure 4: First step: marks *root objects* and the array referencing to them.

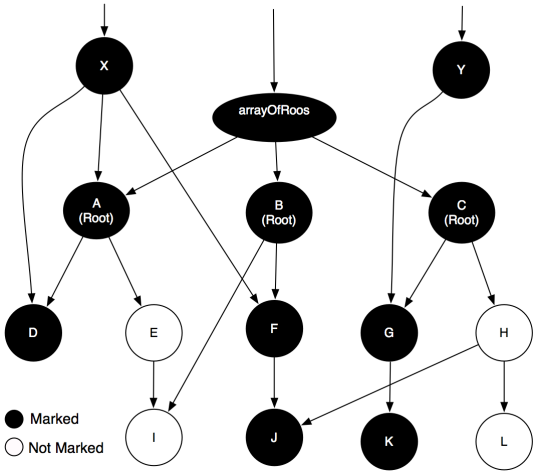

Figure 5: Second step: do a mark pass over all image.

4. Finally, the graph has to be traversed (starting by the roots of the graph) to detect the *inner objects* and serialize them into the WordArray. All the unmarked and reachable objects from the roots of the graph are the *inner objects*. On the other hand, all the marked and reachable objects from the roots of the graph are the *shared objects*.

   In our example, E, H, I and L are identified as *inner objects* and D, F, G, J and K as *shared objects*.

An important last remark is that the step of marking all objects in the image and the step of traversing the object graph are both implemented in the Virtual Machine side. They are both implemented as primitives and the main problem with this is that one does not have control

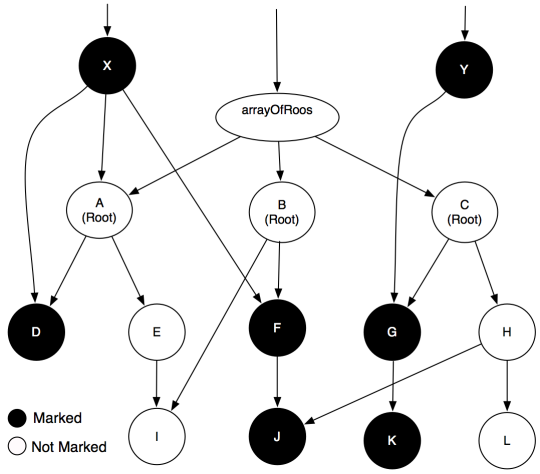

Figure 6: Third step: unmarks *root objects* and the array referencing to them.

over it.

6. Benchmarks and Discussions

We have done some benchmarks to compare ImageSegment and SmartRefStream. These benchmarks were run in the virtual machine Squeak 4.2.5.beta1U on Mac OS X, with a Pharo-1.1-11360-alpha1dev10.05 image. To measure the time, we use the method MessageTally time: and the benchmarks were developed as unit tests. The operations were run five times and the final result was obtained from the average.

SmartRefStream is an object serializer in Squeak and Pharo. To do these experiments we use object graphs of different sizes. For such graphs, we used different models extracted using Moose, an open-source reengineering platform [NDG05]. Such graphs represent source code entities at various levels of details.

Since SmartRefStream does not support object swapping, we compare export durations. We then present a separate performance analysis of ImageSegment object swapping.

6.1. Benchmark and Analysis of Objects Export

Unsurprisingly, our experiments show that ImageSegment is much faster then SmartRefStream. In Figure 7 we present a benchmark done with the Moose default model object which size is small/medium. The amount of *internal objects* of such graph is 741 037. This benchmark shows that ImageSegment is ten times faster to export and thirty times faster to import.

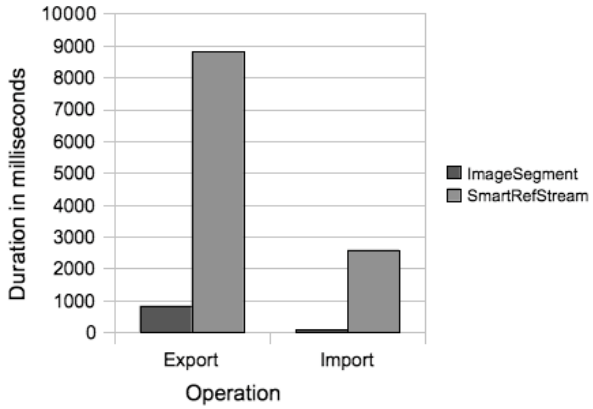

Figure 7: Moose default model chart.

In Figure 8 we used a bigger graph which is the Moose network model. This graph has 2.701.763 *internal objects*. In this case (the graph is bigger) the difference between ImageSegment and SmartRefStream is much larger too: for both operations it is approximately eighty times faster.

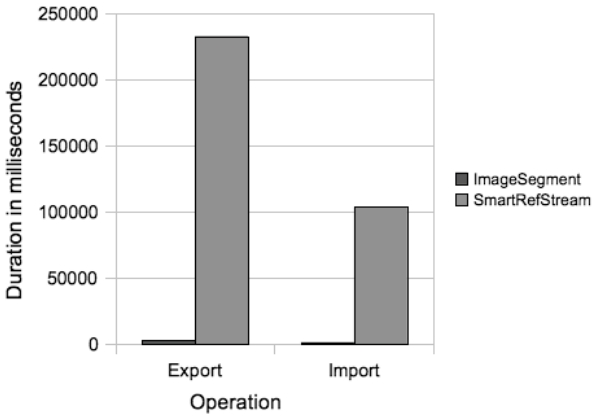

Figure 8: Moose network model chart.

Still, ImageSegment export implementation uses SmartRefStream to serialize the ImageSegment object. This leads us to an interesting question: Why serializing an ImageSegment (that contains an object graph) with SmartRefStream is much faster than just serializing the same object graph directly with SmartRefStream (without using ImageSegment)?

In SmartRefStream, the complete object graph has to be traversed in the image side (Smalltalk). But when an ImageSegment is created, the traversal of the object graph is done at the Virtual Machine level. As a result of such procedure, the ImageSegment has an

encoded WordArray representing the processed object graph. This array is ready to be exported, there is no need to traverse it. Furthermore, ImageSegment uses the GarbageCollector marking algorithm which is optimized and also performed at the Virtual Machine side.

Taking into account the previous explanation, it is necessary to point out that the percentage of *shared objects* is a key aspect. The larger this percentage is, the slower ImageSegment will be, and the smaller the difference with SmartRefStream will be. In both, ImageSegment and SmartRefStream, the *shared objects* array is an array with pointers to the real objects, and thus, it has to be traversed in image side. This means that for an object graph with a big amount of *shared objects*, the performance of ImageSegment and SmartRefStream is much closer to each other.

As it was already explained, the ImageSegment export implementation creates an ImageSegment in the same way it is done for object swapping. Once that such instance is created, it is then exported using a SmartRefStream. When the ImageSegment is created, several steps that are only related to swapping are done. These steps require time, memory and CPU usage. The problem is that for object exporting, those steps are not needed at all: such as the identification of *shared objects* and *inner objects* – this involves computing the whole graph, and what is more, a full Garbage Collector marking phase in the whole image. In addition, ImageSegment has to keep an array with references to the *shared objects*. This array occupies memory.

To conclude, for the exporting point of view, our results show that ImageSegment is faster that SmartRefStream but only because it is done in the Virtual Machine. We believe that if SmartRefStream was implemented in the Virtual Machine it will be even faster than ImageSegment, since ImageSegment export performs all the extra work done for supporting object swapping, work that is not used for export.

6.2. *Performance Analysis for ImageSegment Object Swapping*

For object swapping, we distinguished three different operations that are interesting to analyze: (a) ImageSegment and object sets (*roots, shared and inner*) creation, (b) ImageSegment export (swap out) to a file, and (c) its import (swap in).

In Figure 9 shows the duration of each of those operations for two different graphs: the Moose default model and a Moose model of network package. This chart points out some interesting results:

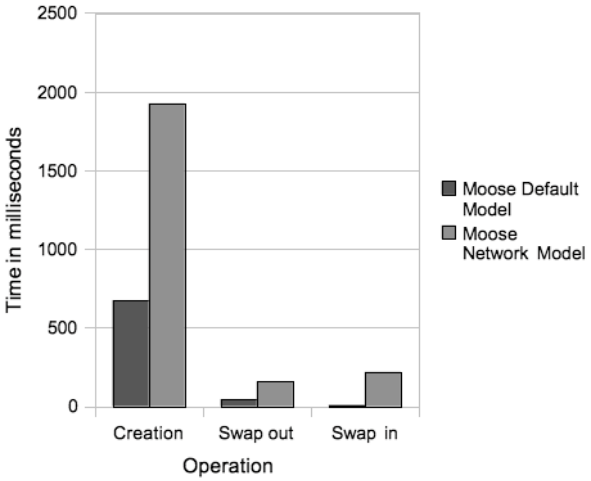

Figure 9: Swapping analysis Moose models.

1. The operation that requires much more time than the rest is the creation of the ImageSegment. The reason is that in this step is where all the object sets are computed and the object graph has to be traversed.

2. The duration of swapping out is similar to the duration of swapping in.

7. Issues and Opportunities for Improvement

ImageSegment is fast and seems simple to use but it also has its own problems and aspects that can be improved.

It can be slow. The biggest problem faced is that it is necessary to make sure that there are no or few *external boundary objects*. Otherwise, all the referenced objects will end up in the *shared objects* array, making ImageSegment very slow.

Difficult to manage in presence of shared objects. The fact that shared objects are not exported can be a serious usability concerns. This is a key observation relevant for any serious use of ImageSegments. As an example, the company netstyle.ch developed a Seaside web application and used ImageSegment as the persistency (export) scheme. The problem is that when the ImageSegment is created, external objects pointing into the graph are not wanted. In the mentioned scenario, many objects are put in *shared objects* array because there are many *external boundary objects*. This means that, for example, all Seaside sessions and

their state have to be deleted; all caches, cleared; background processes, terminated; etc. These kinds of objects hold references to objects inside the graph and are difficult to identify and control.

The extreme but useful solution they found was to:

1. Duplicate the current Smalltalk image to avoid locking-up the image. By duplicating a Smalltalk image, we refer to the creation of a new image based on an existing one and also the creation of an Operating System process for it. This can be achieved in Pharo by using for example, the OSProcess package.
2. Do all the needed cleaning: disable debugging, stop all network services like VNC (RFBServer) or Web Servers (WAKom for example), clean Magritte, terminate non-important processes, clean TestRunner and its results, run garbage collector, forget DoIts, etc.
3. Create the ImageSegment and export it to disk.
4. Kill the duplicated image.

Such solution clearly illustrates the problem ImageSegment faces in presence of shared objects in a complex system.

Another approach may be not to compute the *shared objects* and just include all the *internal objects* in the *serialized objects* array. The problem is that this solution is naive since at loading time, it has to be guaranteed that there are no duplicates of the *shared objects*.

This solution may save time and CPU usage at the cost of having bigger files for the swapped graphs.

Modularity: one for all. Another problem with ImageSegment is the granularity of the import. When swapping, the objects that are swapped are the *serialized objects*. Roots of the graph get replaced by proxies which will read the *serialized objects* array from the file and load it back in memory replacing the proxies with the real objects. The problem is that all the *serialized objects* are brought into memory even when only a single one is needed.

Something similar happens when sending messages like allClassesDo: or allInstancesDo: where the consequences are that all ImageSegments are brought back into memory. Originally metalevel iteration methods were defined and checked if the classes were in memory to avoid exactly this problem.

Memory usage. For anything done in an object-oriented system, objects are created, and, of course, they occupy space in memory. The swapping mechanism is not an exception. Therefore, when using ImageSegment, new objects are being instantiated.

In addition, in the current implementation, ImageSegment uses much more memory than it really needs. It automatically decides the sizes of the *inner objects* array and the *serialized objects* array. To do this, it repeatedly doubles the previous size and checks if it is enough. If it is enough, it finishes. Otherwise, it continues doubling the size. So, it effectively doubles the array size (or more, because it over-allocates) since the primitive that builds an ImageSegment requires pre-allocated arrays which are even longer than needed. The problem with this approach is that it is using much more primary memory than it is really needed and it is easy to run into low memory conditions. This is a big issue for large object graphs and this is exactly the problem that the online database DabbleDB had when using ImageSegment for saving the databases.

Swapping classes and methods. Since in Smalltalk classes and methods are normal objects, it is interesting to be able to swap them out. This is not completely solved with ImageSegment. One problem is that there are several objects outside the graph that may be pointing to a class. For example, depending on the Smalltalk implementation, a class can be referenced from its metaclass, from *SmalltalkDictionary*, from its subclasses or superclasses, from its instances (this is an implicit reference because objects do not have an instance variable with its class but a pointer to it in the object header), etc.

This makes swapping out classes very difficult to achieve and the same problem happens when trying to swap out methods. A possible workaround (not a solution) is to consider all classes (and methods) as *root objects*. In this case, classes are replaced by proxies and swapped out.

Extra information is at class level. We have already said that a subgraph needs extra information to load correctly in another system. The ImageSegment solution to most of those problems is implementing methods in classes. However, sometimes

this extra information is needed per object which means at object level.

8. Related Work

Most of the existing related work is about exporting without addressing swapping aspects.

Vegdah [Veg86] started to face the first problems trying to move objects between Smalltalk images. He found the following problems: cycles inside subgraphs, unique objects like *true*, *false*, *nil* or Symbol instances, Set instances rehash, class reshape and some particular issues with BlockContext, CompiledMethod and MethodContext.

Ungar explained that, when exporting an object subgraph, the subgraph in itself is not complete as it needs extra information (not explicit in the graph) to load correctly in another Self image [Ung95]. His solution was to annotate objects in order to provide the necessary information. He defined generally needed information and Self specific ones. The generally needed information is:

- Which module does a slot belong to?
- Use the slot actual contents vs. a fixed initial value?
- Should slot just reference a preexisting (global) object?
- Should identity of an object be respected?
- Is it possible to create an object with an abstract expression and if so what?

Most of them are discussed in Section 3.3.

The most common export example is a XML serializer like SIXX [SIX] or JSON [JSO]. In this case the object graph is exported into a portable text file. The main problem with test-based serialization is encountered with big graphs as it does not have a good performance and it generates very large files. Other alternatives are ReferenceStream or SmartReferenceStream. ReferenceStream is a way of serializing a tree of objects into a binary file. A ReferenceStream can store one or more objects in a persistent form including sharing and cycles. The main problem of ReferenceStream is that it is slow for large graphs.

A much more elaborated approach is Parcel [MLW05] developed in VisualWorks Smalltalk. Parcel is an atomic deployment mechanism for objects and source code that supports shape changing of classes, method addition, method replacement and partial loading. The key to making this deployment mechanism feasible and fast is a pickling algorithm. Although Parcel supports code and objects, it is more intended to source code than normal objects. It defines a custom format and generates binary files. Parcel has very good performance and the assumption is that the user may not have a problem if saving code takes more time, as long as loading is really fast.

Object serializers are needed and used not only by final users, but also for specific type of applications or tools. What it is interesting is that they can be used outside the scope of their project. Some examples are the object serializers of Monticello2 (a source code version system), Magma object database, Hessian binary web service protocol [has] or Oracle Coherence*Web HTTP session management [ora].

The main problem is that none of the mentioned solutions support object swapping. There are few experiments regarding object swapping and even fewer implemented and working solutions.

In the eighties, LOOM [Kae86] (Large Object-Oriented Memory) implemented a kind of virtual memory for Smalltalk-80. It defined a swapping mechanism between primary and secondary memory. The solution was good but too complex due to the existing restrictions (mostly hardware) at the time. Most of the problems faced do not exist anymore with today's technologies — mainly because of newer and better garbage collector techniques —. For example, LOOM had to do complex management for special objects that were created too frequently like MethodContext but, with a generation scavenging [Ung84], this problem is solved by the Garbage Collector. Another example is that LOOM was implemented in a context where the secondary memory was much slower than primary memory. This made the overall implementation much more complex. Nowadays, secondary memory is getting faster and faster, with random access showing more and more the same properties as RAM memory[3]. Finally, LOOM implies big changes in the Virtual Machine.

It is possible that a program will leak memory if it maintains references to objects that will never be used again. Leaked objects decrease program locality and increase garbage collection frequency and workload. A growing leak will eventually exhaust memory and crash the program. Melt [BM08] implements a tolerance approach that safely eliminates performance degradations

[3] "Solid-state drives" (SDD) or flash disks have no mechanical delays, no seeking and they have low access time and latency.

and crashes due to leaks of dead but reachable objects, giving sufficient disk space to hold leaking objects. Melt identifies "stale objects" that the program is not using and swaps them out to disk. If they are then needed, they are brought into primary memory. Its approach is quite similar to LOOM.

GemStone [gem] is a Smalltalk object server and database which manages primary and secondary memory as well. To provide its features, it has to implement object graph exporting, swapping, serializing and most of the concepts discussed in this paper. In addition, it has an excellent performance and is highly scalable. The main difference between GemStone and what has been previously discussed is that GemStone is not a tool for exporting or swapping an object graph, but a complete Smalltalk dialect that supports transactions, persistency and that also acts as an object server. It is more suitable for middle or big systems. ImageSegment or ReferenceStream, for example, are just small tools that only allow performing specific tasks like exporting or swapping a graph of objects. Another important difference between GemStone and solutions like ImageSegment is that they use the opposite approach. In GemStone, objects live permanently in secondary memory and are temporally loaded into primary memory and kept there while needed and then swapped out when not needed anymore. With ImageSegment, objects live in primary memory and they are just swapped out when not needed and loaded back when needed.

9. Conclusion and Future Work

In this paper, we have looked into the problem of exporting and swapping object graphs in object-oriented systems. We have analyzed not only most of the general major steps that are needed to export and swap object graphs, but also their problems and challenges. What is important is the fact that these steps, together with their problems and challenges, are completely general and they are independent of the technology.

These object graphs operations are very important to support virtual memory, backups, migrations, exportations, etc. In addition, object swapping may be an interesting approach for saving primary memory in embedded devices (robots, handheld, etc).

The biggest constraint in these kind of graph operations is speed. Any possible solution has to be fast enough to be actually useful. In addition, this problem of performance is the most common problem among the different solutions. Most of them do not deal properly with it.

We have deeply analyzed ImageSegment solution and we compared it with other alternatives. ImageSegment supports both: object swapping and export. It has good performance but as long as there are few shared objects between the swapped subgraph and the remaining objects. ImageSegment is fast mainly because the object graph is traversed in the Virtual Machine.

There are also some severe negative points with ImageSegment. First, it has been already explained that it can get slow when there are several *shared objects*. Second, the modularity of the solution is a problem too: when an object graph is swapped out and then only one single object from the graph is needed, the whole graph is loaded back in memory. There is no way to manage subgraphs. Finally, the export implementation of ImageSegment does extra work that is only swapping related and not needed at all when exporting. This extra work consumes time, CPU and memory. Finally ImageSegment is not simply portable because it is implemented by extending the Virtual Machine.

As said above, in the current ImageSegment implementation, all the *serialized objects* are brought into memory even when only a single one is needed. A first idea to solve this problem and, thus, be able to swap and load back subgraphs is to have more proxies. Instead of replacing only the roots of the graph by proxies, also the *facade objects* should be replaced. When one of the proxies of the *facade objects* receives a message, only its subgraph is loaded back and not the whole graph. We are not completely sure if this will work or if other changes are needed but we believe that swapping out and loading back subgraphs instead of the whole graph is a really necessary feature.

ImageSegment swaps even when it is not worth it. We plan to make ImageSegment smart so that it can automatically decide if swapping an object graph is worth it or not. If the memory occupied by the ImageSegment objects is the same or more than what it will be released because of swapping out the graph, then it is not worth it. A smarter strategy may also take into account the CPU costs to swap out and in.

Finally, we would like to address the problem of the memory usage by using a file to allocate the arrays of ImageSegment, instead of primary memory. There is lot of random access and thus, a file-based solution will be much slower than a primary memory based. Nevertheless, sometimes it is worth paying that cost to avoid growing the image while swapping. In addition, for large graphs, the current implementation does not work at all as it leads into an out of memory error.

References

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, 2009.

[BM08] Michael D. Bond and Kathryn S. McKinley. Tolerating memory leaks. In Gail E. Harris, editor, *OOPSLA: Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008.

[gem] Gemstone object server. http://gemstone.com/products/gemstone.

[has] Hessian. http://hessian.caucho.com.

[IKM+97] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'97)*, pages 318–326. ACM Press, November 1997.

[jav] Java serializer api. http://java.sun.com/developer/technicalArticles/Programming/serialization/.

[JSO] Json (javascript object notation). http://www.json.org.

[Kae86] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. *Proceedings OOPSLA '86, ACM SIGPLAN Notices*, 21(11):87–106, November 1986.

[MLW05] Eliot Miranda, David Leibs, and Roel Wuyts. Parcels: a fast and feature-rich binary deployment technology. *Journal of Computer Languages, Systems and Structures*, 31(3-4):165–182, May 2005.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In *Proceedings of the European Software Engineering Conference (ESEC/FSE'05)*, pages 1–10, New York NY, 2005. ACM Press. Invited paper.

[ora] Oracle coherence. http://coherence.oracle.com.

[pic] Pickle. http://docs.python.org/library/pickle.html.

[pro] Google protocol buffers. http://code.google.com/apis/protocolbuffers/docs/overview.html.

[SIX] Sixx (smalltalk instance exchange in xml). http://www.mars.dti.ne.jp/~umejava/smalltalk/sixx/index.html.

[Ung84] Dave Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984.

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings OOPSLA '95*, pages 73–87, 1995.

[Veg86] Steven R. Vegdahl. Moving structures between smalltalk images. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 466–471, New York, NY, USA, 1986. ACM.