HAL
open science

Problems and Challenges when Building a Manager for Unused Objects

Mariano Martinez Peck, Marcus Denker, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse

To cite this version:

Mariano Martinez Peck, Marcus Denker, Stéphane Ducasse, Noury Bouraqadi, Luc Fabresse. Problems and Challenges when Building a Manager for Unused Objects. Smalltalks International Workshop, FAST (Federación Argentina de Smalltalks), Nov 2011, Bernal, Buenos Aires, Argentina. inria-00635793

HAL Id: inria-00635793

https://inria.hal.science/inria-00635793v1

Submitted on 25 Oct 2011

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Problems and Challenges when Building a Manager for Unused Objects

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

Abstract

Large object-oriented applications may occupy hundreds of megabytes or even gigabytes of memory. During program execution, a large graph of objects is created and constantly changed.

Most object runtimes support some kind of automatic memory management based on garbage collectors (GC) whose idea is the automatic destruction of unreferenced objects. However, there are referenced objects which are not used for a long period of time or that are used just once. These are not garbage-collected because they are still reachable and might be used in the future. Due to these unused objects, applications use much more resources than they actually need.

In this paper we present the challenges and possible approaches towards an unused object manager for Pharo. The goal is to use less memory by swapping out the unused objects to secondary memory and only leaving in primary memory only those objects which are needed and used. When one of the unused objects is needed, it is brought back into primary memory.

Keywords: Object Swapping, Serialization, Proxies, Smalltalk, Object-Oriented Programming

1. Introduction

In object-oriented programming languages like Smalltalk – and Java to a certain extent –, everything is an object. Objects are allocated and they occupy a certain amount of memory. The execution of an application forms a graph of interacting objects. During program execution, a large graph of objects is built, changed and reconfigured. Most object runtimes support some kind of automatic memory management based on garbage collectors (GC) [Jon96] whose idea is the automatic destruction of unreferenced objects. The GC collects objects that are not being referenced anymore, *i.e.,* it works by reachability.

During a typical run of an application, several millions of objects are created, used and then collected when not referenced. But a problem appears when there are objects which are not used but cannot be garbage-collected because they are still reachable (*i.e.,* are referenced by other objects). Some objects are used just once, are used only in certain situations or conditions or are not used for a long period of time but, in all cases, they are kept in memory. For example, memory leaks create these kinds of unused objects [BM08]. This is a problem because, in presence of memory leaks, applications use much more resources than what is actually needed and might even exhaust the available memory. Therefore unused objects lead to slower systems and can even be the cause of severe system crashes.

One solution to ensure the spatial scalability of applications is to temporarily move objects to secondary memory (*e.g.,*hard disk) to temporary release part of the primary memory (*e.g.,*RAM) [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 mentioned problem not only happens with large applications or servers running many programs, but also with systems that run in embedded devices or in any kind of hardware with a limited amount of mem-

☆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@mines-douai.fr (Noury Bouraqadi), marcus.denker@inria.fr (Marcus Denker), 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.

ory like robots, cellphones, etc.

In this paper we present the main challenges, problems and possible approaches towards building an *Unused Object Manager* (UOM) for Pharo. An UOM automatically manages the memory occupied by unused objects. The goal of this system is to use less memory by detecting and moving the unused objects to secondary storage and only leaving in primary memory those which are currently needed and used [MPBD+10]. When one of these swapped objects is then needed, it is brought back into primary memory. To achieve this, the system replaces the original (unused) object with a proxy [GHVJ93]. Whenever a proxy receives a message, it loads back the swapped object from secondary memory [Kae86, BM08].

An UOM must be carefully designed so that: 1) it saves as much memory as possible *i.e.,* it does not use more memory with proxies and other temporally required data, than the one that can be released by swapping unused objects; 2) it minimizes the overhead *i.e.,* should not slow down too much the computation when detecting unused objects or when swapping them between memories.

We did several experiments by implementing some parts of the whole UOM with Pharo [BDN+09]. We compared alternatives to implement different parts of an UOM.

The contributions of this paper are:

- A description of the main parts of an UOM for Pharo: the unused object detector, object proxies, the object serializer and the object swapper.
- An analysis of the main issues related to these different parts of an UOM.
- A catalog of inadequate solutions facing these issues in the context of Pharo. It is important to know that some solutions should not be used in an UOM.

The remainder of the paper is structured as follows: Section 2 defines and unifies the concepts and names used throughout the paper. Section 3 decomposes an UOM into smaller subsystems and gives a general overview of them. The most basic issues that appear when building an UOM are explained in Section 4. Section 5 shows all the related problems with proxies and its relation with memory addresses. In Section 6 we describe the problem of shared objects inside graphs and we list some possible alternatives to solve this issue. Section 7 shows that there are even more problems related to the whole swapping mechanism. The first steps of Marea project are presented in Section 8. Finally, in Section 9 related work is presented, before concluding in Section 10.

2. Glossary

Starting from a graph of interconnected objects, an UOM detects the unused objects and swaps to secondary memory a subgraph of these unused objects. Figure 1 shows an example of an object graph (surrounded by a rectangle) that we want to swap.

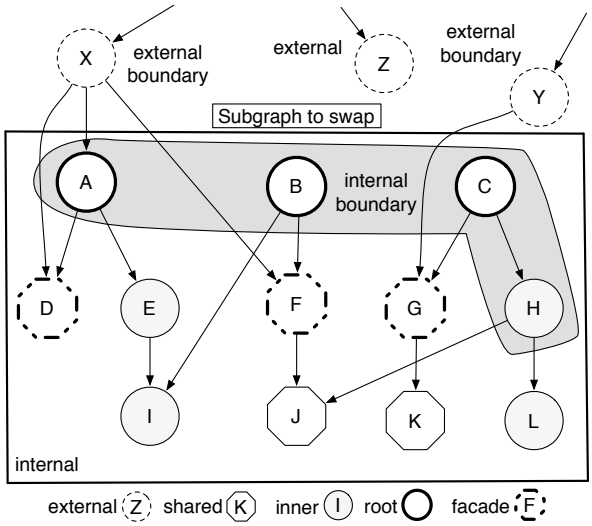

Figure 1: A graph to be swapped.

Through this exhaustive example we define a glossary of terms used in this paper to avoid confusion:

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

Root objects are those defined by the user or by the UOM. Commonly, operations begin with these objects. Example: A, B and C.

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

External boundary objects are *external objects* that refer 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 refer to *shared objects*. Example: A, B, C and H.

Facade objects are *shared objects* which are referenced 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. Unused Object Manager Prerequisites

Figure 2 shows the different subsystems needed by an UOM to perform its tasks. An UOM should involve at least four different tasks: mark objects when they are used and unmark them when they are not, select which objects to swap, replace them with proxies, serialize (write the object graph into a sequence of bytes) and materialize (recreate the object graph from a sequence of bytes) them.

The following is a general overview of such subsystems.

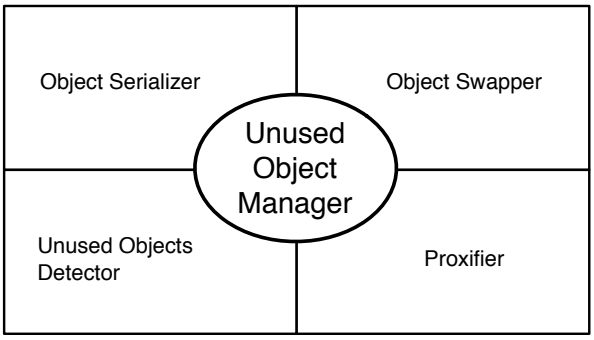

Figure 2: UOM subsystems.

3.1. Unused Objects Detector

While building an UOM, it is important to understand which parts of the system are used and which ones are not. We define a used object as an object that receives a message or that is directly used by the virtual machine during a specific period of time.

The main challenge here is how to store the usage status of each object. The most naive idea, adding an instance variable to Object and storing there a boolean leads to multiple problems: as it changes the memory layout of all objects (even those whose layout is known by the virtual machine (*i.e.,* classes), it is actually not possible in most languages without many VM level changes. In addition, the requirement of an extra reference for each object leads to a huge increase in memory.

The usage mark may be used later on in different ways and a boolean might not be enough. For example, if we want to compute the *lowest usage frequency object* or the *least recently used object* which are different policies for selecting which objects to swap, we need to store more information than a boolean, such as counters or timestamps.

Besides providing a place to store the usage status, it is necessary to modify the language or the VM so that the flag is turned on whenever an object is used or receives a message.

The information obtained from tracing objects usage is heuristic and it does not mean that a given component is not used at all. It just means it was not used during a period of time. A certain object that was considered unused may be needed later on.

All objects must be unmarked regularly, otherwise once one is marked as used, it is marked for ever. When to mark or unmark objects or when to swap them, is a decision of the UOM.

The UOM should have policies to mark and unmark unused objects as well as to start a swapping process.

3.2. Proxy Objects

The UOM replaces the roots of the graphs being swapped with proxies [GHVJ93]. As it is explained later, the UOM may not only replace roots by proxies but also other objects of the graph, *e.g., shared objects*. Two constraints are attached to the proxy framework: the first one is that, since the language can consider some special entities as first-class objects, the mechanism to track unused objects must be the same. For example, Smalltalk considers packages, classes, methods, processors, etc., as first-class objects. Therefore, we must swap a graph no matter what kind of objects are inside, *i.e.,* we need a reliable proxy implementation that can proxify any kind of object.

The second constraint is that, if the UOM replaces each object with a proxy, the amount of released memory varies significantly depending on the memory footprint of the proxies.

The UOM should use a proxy library able to handle any kind of object. In addition the memory footprint of proxies should be as low as possible.

3.3. Object Serializer

Serializer *speed* is an important aspect since it enables extreme scenarios such as the one of an UOM: saving objects to disk and loading them at the exact moment of their execution.

Two different functionalities of the serializer should be measured independently: the serialization speed and

the materialization speed. In our case, the materialization speed is much more important than the serialization speed. A graph is swapped out because it is not being used so it is not really important if the serialization takes more time. To the contrary, when a swapped out object is needed, it is very important to be able to load it back as fast as possible because it is being needed *at that exact moment*, *i.e.,* there is application code that is waiting.

The UOM should be fast at loading objects.

3.4. Object Swapper

The main responsibility of an object swapper is to efficiently swap graphs between primary and secondary memory. It takes care of replacing objects with proxies and serializing graphs to secondary memory. As we explain later in Section 6, detecting and correctly handling the shared objects of a graph is a challenging task that an object swapper should address.

The object swapper should be efficient when swapping graphs

4. Basic Swapping Issues

4.1. Swapping unit

The first question that appears while implementing an UOM is whether it should swap individual objects or groups. In Figure 1, object A is referencing E and E is *only* referenced from A. If A is unused and we want to swap it, we replace it with a proxy. But the proxy does not reference E. Hence, when the garbage collector runs, E is garbage collected which means that, if we only serialized object A and not E, then we lost the object E.

That could be solved by always keeping the outbound references of the graph in an array. However, such implementation occupies more memory and its memory footprint will be similar to the original object saving nothing or very little memory.

Conclusion: we do not want to just swap objects individually and create a proxy per object because, doing so, we release nothing or very little memory.

In a basic implementation, proxies are regular objects. The only difference regarding memory footprint between the original object and its proxy, is the amount of instance variables they have. For large objects (objects with several instance variables or *e.g.,* Collection instances), there can be a significant difference. But, for smaller objects, which have zero or a few instance variables, it would be almost the same (take into account that the proxy object may have instance variables *e.g.,* to know where the swapped object was stored).

Conclusion: *to be efficient, we need to group objects and be able to replace several objects with one or a few proxies.*

A naive thought is why not to group objects in pages, *i.e.,* groups of a fixed number of unrelated objects. Objects that are not being used can be grouped in pages and then swapped out all together. The problem with this approach is that we still need a proxy for each object of each page.

Because of all the mentioned reasons, in our experiments, we swap object *graphs*. Why considering graphs as the swapping unit is a good idea? First, because we avoid the problem of objects being lost by the GC. Second, because we can just replace the root (or roots) of the subgraph with proxies. This way we need only one or a few proxy instances. Depending on the solution, we may need more proxies than those for the roots. When we swap an object graph, we need to consider all the objects from outside the graph that are referencing objects inside. The most clear example are the roots of the graph. Nevertheless, it may be necessary to also create proxies for the *facade objects*.

4.2. Not Everything Can Be Swapped

Instances of Array can be swapped without problems. However, one cannot swap the particular instance specialObjectsArray because it is used by the virtual machine. The same happens with objects such as nil, true, false, etc. This means that there are specific objects that cannot be swapped.

Another example are classes. In Smalltalk, a class is an instance of its metaclass. Classes are swappable. In fact, a good UOM should be able to swap out unused classes as well. However, at the same time, there are certain classes that cannot be swapped because they are needed by the minimal possible code execution. Examples are: ProtoObject, Object, Array, Symbol, BlockClosure, CompiledMethod, MethodDictionary, SmallInteger, etc.

We need a way to tag system classes and objects.

Finally, the UOM should not swap out objects that it needs to swap in. This seems logical but the solution still needs to take this into account. The idea is that the system automatically swaps out subgraphs of unused objects. It might happen that methods or classes used by the code to swap in, were swapped out. When such method or class is used by the swap in code, the system will detect that those objects are on disk and need to be swapped in. Therefore, there will be an endless loop which can end up in a system crash.

The answer sounds easy: do not swap what we need to swap in. Nonetheless, it is difficult to know which objects are going to be needed. For example, we can ensure that our own classes and methods (those which implement the swap logic) are not swapped. Even more, we can enforce not to swap the serializer's classes either. Still, the object serializer uses other objects, classes, and methods which may have been swapped out. What is important is that, for each possible runtime execution path, different objects can be used. In conclusion, it is a challenge to detect in advance all those objects that the swap in code use.

The idea is to detect as much needed code to swap in as possible, and never swap it. If the whole UOM is covered by an exhaustive Unit Tests suite, then when the system starts, it can run all the tests and detect all the used objects to do so. All those objects are then marked as "excluded" which means that they will not be swapped out.

Notice that the results of this approach are not always correct. For example, it may happen that we are not testing a particular scenario so some objets needed to swap in were not used and, consequently, swapped out. That leads to the mentioned problem of an endless loop which can end up in a system crash.

5. Proxies and Memory

5.1. Common Problems with Proxies

The following is a list of problems that we often find while using proxies, and an UOM is not an exception.

Methods Not Intercepted. Proxies can be used in different scenarios. In an UOM, proxies must intercept any message send and then load the swapped out graph back in primary memory.

Another common problem when dealing with proxies is the existing optimizations for certain methods. In Pharo, as well as in other languages, there are two kinds of optimizations that affect proxies. The first ones are those optimizations done by the compiler. For example, messages like ifTrue:, ifNil:, and:, to:do:, etc. are detected by the compiler and are not compiled as a regular message send. Instead, those methods are directly replaced by jump bytecodes (this is known as "inlining") which means that those methods are never executed. If they are not executed, they cannot be intercepted by proxies.

We would like to handle those messages the same way than regular ones. The easiest yet naive way of dealing with it is to modify the compiler so that it does not inline those methods. However, disabling all optimizations brings two important problems. The first one is that the system gets significantly slower. The second one is that if those optimizations are disabled, those methods are executed and there can be unexpected and random problems which are extremely difficult to find. For instance, in Smalltalk, everything related to managing processes, threads, semaphore, etc., is implemented in Smalltalk itself. The processes' scheduler can only switch processes between message sends. This means that there are some parts in the classes like Process, ProcessorScheduler, Semaphore, etc., that have to be atomic, *i.e.,* they cannot be interrupted and switched to another process. In Pharo, there is no way to easily and explicitly define that. As a consequence, sometimes sending *e.g.,* messages like whileTrue:, ifFalse:, etc, is used as a way to avoid generating a suspension point for the scheduler. If we disable the optimizations, such code is not atomic anymore.

The second type of optimization is between the compiler and the virtual machine. There is a special list of selectors that the compiler does not compile like a regular message send. Instead, each of those selectors is associated with a special bytecode that the VM can then directly interpret. Again, it means that those methods are not executed. From an UOM point of view, the two most important optimizations are the method == which answers whether two variables refer to the same object and the message class which answers the class of an object.

Not being able to intercept messages is a problem because those messages will be directly executed by the proxy instead of being intercepted. This leads to different execution paths in the code. For example, given the following code:

```
(anObject class = User)
    ifTrue: [ self doSomething]
    ifFalse: [self doSomethingDifferent]
```

If anObject is an instance of User and it is swapped out, the reference anObject points to a proxy. That means that if class is not intercepted, it will execute self doSomethingDiffrent instead of loading back the original graph and executing self doSomething.

With == it is a different scenario. Given the following code:

```
(anObject  == anotherObject)
    ifTrue: [ self doSomething]
    ifFalse: [self doSomethingDifferent]
```

If anObject is swapped out, the reference anObject points to a proxy. Even if the proxy cannot intercept such message, this is not a problem in an UOM. The

UOM replaces a target object with a proxy, *i.e.,* all objects in the system which were referring the target, will refer to the proxy. Since all references has been updated, == continues to answer correctly. For instance, if anotherObject was the same object as anObject, == answers true since both are referencing the proxy now. If they were not the same object, == answers false. Hence, identity is not a problem for an UOM.

We did a benchmark to estimate the impact of removing the special bytecode for class. We run all the tests (8003 unit tests) present in a PharoCore 1.3 - 13204 image, twice: first with the class optimization and then without it. The overhead of removing that optimization was only about 4%, which means that it is only slightly perceptible in general system interactions.

5.2. *Mapping objects from primary memory to secondary memory*

This problem is also known as *pointer swizzling*, and it is the conversion of references based on name or position (indexes) to direct pointer references. It is typically performed during the deserialization (loading) of a relocatable object from disk. The reverse operation, replacing pointers with position-independent symbols or positions, is sometimes referred to as unswizzling and is performed during serialization (saving). This technique is frequently used in object-oriented databases.

In primary memory, there are objects (proxies) that refer to other objects in secondary memory and vice-versa. In primary memory, the memory address is used but, in secondary memory (*e.g.,* hard disk), it is different. Often, for secondary memory, relative offsets or IDs can be used as addresses.

The UOM can store objects using different backends. For example, it can use the local filesystem or a database.

Filesystem and granularity problem. When an object is written to disk, there can be several options regarding where and how to write such object. For example, we can use the same file for all graphs, a file per graph or a file per group of graphs, etc. If we use the same file for all objects, then proxies should use an offset inside such file and we have to manage free spaces, compaction, etc. This means we need to implement something similar to a filesystem.

A simpler possibility is to write each graph in a separate file and store such filename as an instance variable in the proxy.

Databases as backends. Another approach is to use a relational or object database as a backend for storing graphs. The problem with this approach is that the database adds functionalities that we do not particularly need and that have performance impact, for example, transactions support, security and validations, etc. In addition, we need to maintain in primary memory all the objects related to the database driver.

Some recent NoSQL databases may *not* provide those functionalities avoiding that extra overhead. Databases can be used in two scenarios: when there is more than one client accessing the database and when there is only one.

When there is more than one client, there are some of these NoSQL database *e.g.,* CouchDB or Riak that expose their interface (API) through the network. That means that we only need a HTTP client library. This approach is useful when we desire a distributed UOM where graphs are not swapped to the same host where the system is running but instead to a particular server which stores graphs from different systems. Notice that sending the graph by HTTP may not be as fast as directly storing it in a local file.

If there is only one client, we can use NoSQL databases which are more accurate for single machine, *e.g.,* Tokyo Tyrant. In this case, the database provides a client library which can be written in C and easily called from any other language.

The way to store data in these databases is usually following the convention of key/value, *i.e.,* at a certain key we put certain value. A possible solution would be to create IDs for each graph (key) and serialize the graph into an array of bytes (value). This way it is easy to use the client library to store a BLOB [3] representing our object graph. The proxy can then store such ID to search and load back the graph.

One drawback when using a database as backend is that the solution depends on an external technology. Therefore, the database driver is also part of what should not be swapped out.

5.3. *Making Proxies Use As Little Memory As Possible*

In Smalltalk, everything is an object. An object in the virtual machine is represented by an object header plus slots that can contain pointers to other objects or directly store bytes. Since proxy objects also require memory, they must be as small as possible.

[3] BLOB stands for "binary large object", a database type for storing a collection of binary data

To make proxies have the minimum memory footprint possible without modifying the virtual machine, in Pharo it is possible to do the following:

- Proxy class can be a "Compact Class". This means that, in a 32 bits system, their instances' object headers are only 4 bytes long instead of 8 bytes for instances of regular classes. For instances whose "body" part is more than 255 bytes and whose class is compact, their header will be 8 bytes instead of 12. The first word in the header of regular objects contains flags for the garbage collector, the header type, format, hash, etc. The second word is used to store a reference to the class. In compact classes, the reference to the class is encoded in 5 bits in the first word of the header. These 5 bits represent the index of a class in a compact classes array. This array is set from the image[4] and it is accessed by VM. With these 5 bits, there are 32 possible compact classes. Thus, declaring the proxy classes as compact, enables proxies to have a smaller header and a smaller memory footprint.

- Proxies should only keep the minimal state they need. In the case of an UOM, each proxy has to store the necessary information to load back the swapped out subgraph. One alternative is to simply store a string with a file name. But when using a filename, we have to pay the cost of the string and its object header. If we use a number, it can be an instance of SmallInteger which is an immediate object in Smalltalk. Immediate objects are those that are *directly* encoded in the memory address and do not require an object header nor slots so they consume less memory.

  The mentioned approach is with the assumption that we are able to get the exact place on disk from the number value. A simple option is to serialize each graph into a new file. Such file has a specific number (ID) as file name which is directly stored in the proxy.

Even if the previous ideas help from the memory footprint point of view, there is still room for improvement and we have the possibility of using *immediate objects* for proxies. For example, GemStone [BOS91] database uses this strategy. In a 32 bits VM, it is complicated because it needs bits to tag the immediate objects but, at the same time, it needs a large range for addresses. GemStone, which is a 64 bits VM, uses 61 bits for addresses and 3 for immediate objects where it can encode true, false, nil, characters, small floats, small ints, etc.

One problem that appears when defining proxies as immediate objects is *where* to store the reference to the target object, the address on disk, the filename or whatever is needed. The ideal case is to be able to store such information in the bits that are designated for the object pointer. Otherwise, we end up needing again another object for such state. Normally, as in the case of an UOM, the memory address of a target object, an identifier or an offset in a file fits in the object pointer space.

The advantage with this approach is that we do not need extra memory (for object headers) for the proxy instances since we can directly tag the references. For example, in 61 bits, we have enough space to encode the address of the original object in disk. In addition, we do not have extra cost on having to fetch the proxy because we have everything we need in the reference, which means better performance. Still, notice that tagging memory addresses instead of creating proxies and replacing objects, can significantly change the mechanism of a particular solution.

5.4. Special Proxies

Certain classes have instances that cannot be easily replaced by a regular proxy. For example, in Pharo all immediate objects (those who are directly encoded in the memory address) like SmallInteger cannot be replaced by a proxy using the primitive become:. This is because with the method become: all references from the system to a particular instance of SmallInteger must be updated to refer to a proxy instance. Since SmallInteger are directly encoded in the memory address, this task is more complicated. Each memory address must be accessed, check that its contents is a SmallInteger instead of a regular reference, and finally it needs to check if the content is the same or not to the value we are searching for.

That being said, notice that it does not make sense to create proxies for SmallInteger instances because they occupy less memory than what proxies do.

Some object-oriented programming languages like Smalltalk, represent classes and methods as first-class objects, *i.e.,* they are not more than just instances from other classes known as the Metaclass and CompiledMethod respectively.

Imagine that one of the roots is a class or a method, it will then be replaced by a proxy. If we then send a message to an instance of such class (which is now the

[4] See methods SmalltalkImage»compactClassesArray and SmalltalkImage»recreateSpecialObjectsArray

proxy instance), the virtual machine crashes while trying to perform the method lookup. This is because the VM imposes specific constraints on the memory layout of objects representing classes and methods. Hence, if we replace them with objects that do not respect that shape, the VM crashes or throws an error.

A good proxy toolbox must solve this problem, for example, by creating special proxies for classes and methods that respect the shape needed by the VM.

6. Shared Objects Inside Graphs

Detecting and correctly handling shared objects of a graph is a challenging task that an object swapper should address. Take the graph example of Figure 3. If we swap such graph, object Y needs to be considered because G will be swapped. Whether *shared objects* should be swapped or not, depends on the implementation. In any case, it is necessary to handle that situation and to know which objects inside the subgraph to swap are shared. This is important because it is really common to have one or more shared objects inside subgraphs.

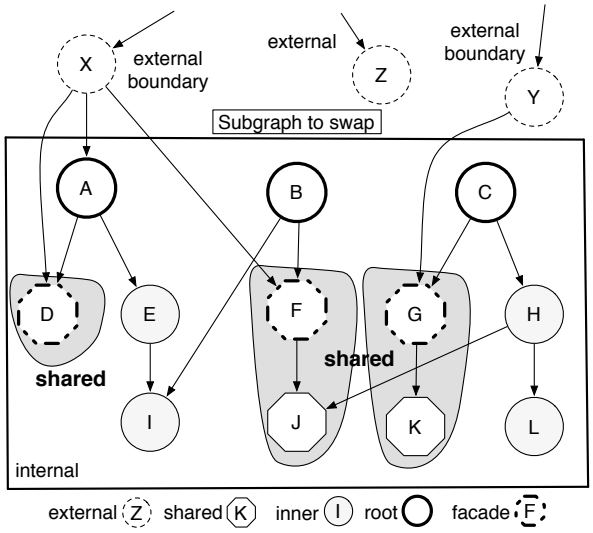

Figure 3: A graph to be swapped with shared objects.

6.1. Detecting Shared Objects

As recently explained, for every object in a graph, we need to know if it is shared or not. The problem is that there is no easy or incremental way of detecting *shared objects* because objects do not have back-pointers to the objects that refer to them.

We have analyzed several ways to make shared objects' detection fast. One of the key goals is to avoid a whole memory traversal. We think it is worth to list all the different possible alternatives because one of them may be more adequate to implement in a specific language.

Full Memory Scan. One way of solving the problem of shared objects is to traverse (scan) the full object memory, as ImageSegment does [MPBD+10]. By using garbage collection infrastructure, it identifies which objets of the graph are *inner objects* and which ones are *shared objects*. The steps followed by ImageSegment are:

1. All *root objects* are marked.
2. A mark pass is done over all the objects in the primary memory by recursively marking those reachable from the roots of the system. This process will stop at our marked roots leaving *inner objects* unmarked.
3. *Root objects* are unmarked while *inner objects* are left without being marked.

The problem is the overhead and time spent to do the full traversal of the whole memory. This is what we want to avoid.

Using a Reference Counting Garbage Collector. The Pharo VM is a generation scavenging mixed with a mark and sweep algorithm. The idea is to modify the current GC and merge it with a reference counting GC. In a typical reference counting GC, there is a counter stored in the object header which represents the amount of incoming pointers to each object. The GC takes cares of updating this counter to reflect new objects that have been created, removed, assigned, etc.

Taking this into account, we can traverse *only* the object subgraph and count the references to each object during that traversal. Another counter is used to store that information. Once we finish, we can determinate how many incoming references each object has from inside the subgraph.

Finally, we can compare each result with the reference counter of the GC. If it is the same, then the object is *inner object* because all its incoming references are from inside the subgraph. If it is less, then it is a shared object.

The advantage of this mechanism is that we only need to traverse the object subgraph. The drawbacks are that we need to modify the GC (which means significantly

changing the virtual machine), pay the overhead of updating the reference counters and pay the cost of the memory in the object header to store such counters.

Adding Back-Pointers to the Virtual Machine. Modify the virtual machine and add "back pointers". Each object contains not only the references to other objects, but also references to the objects that point to it, *i.e.,* incoming references. If there are back pointers, it is easy to know whether an object is shared or not.

The problems are the amount of memory needed to allocate those pointers for every object and the overhead in the VM to create, update and release such references.

First Class References. Most virtual machines have an important part whose responsibility is managing the memory, allocating objects, releasing, etc. In Pharo VM, such part is called Object Memory. In addition, the Object Memory defines the internal representation of objects, its references, its location, its object header, etc.

Regarding the references implementation, there are two possibilities which are the most common: object tables and direct pointers. With the first, there is a large table with two entries. When an object A points to B, it means that A points to an index in the table where the memory address of B is located. With direct pointers, when A points to B, it means that A has directly the memory address of B.

There are pros and cons for each strategy but such discussion is out of range for this paper. We will just mention some of them which are important for our domain. With an object table representation, moving an object or swapping it by another one is really fast since it is just updating one reference. With direct references, swapping an object is slow because it needs to scan all the memory do detect all the objects that are pointing to a particular one. On the other hand, with Object Tables, we have to pay the cost of accessing an extra indirection and this impacts on the overall performance of the system. With direct pointers, we do not have that problem. Finally, Object Table uses more memory since the table itself needs memory.

The idea is to implement *first class references* [ADD+10, LGN08]. This is something similar to Object Table but it is spread all over the memory and uses objects instead of just addresses. The idea is that an object does not refer directly to another object, but to an intermediate one that points to the target one. If there are different objects pointing to the same object, they will be all pointing to the same intermediate object which points to the target object.

Having first class references enables us to implement the following: traverse only the subgraph marking with a special flag the intermediate objects. We mark all intermediate objects of the subgraph without checking whether they are shared or inner. Once we write the subgraph into a file and we replace the roots with proxies, all the intermediate objects that were pointing to an inner object are not referenced any more and, consequently, they are removed by the garbage collector. Only those intermediate objects that were referencing to shared objects remain alive. If the tag used was "the object is on disk", we can use those intermediate objects as proxies. Ideally, we could store the filename or address in secondary memory in those intermediate objects.

The advantage of this approach is that we do not need to scan the whole memory. The disadvantages are the significant changes it requires in the virtual machine and the amount of memory used as there will be one more object (the intermediate one) for every single object. In addition, there could be a significant overhead in the garbage collector.

Serialized Objects in a Weak Collection. While swapping a graph, we write *all* objects (shared and inner) into the file. We do not spend time checking whether each object is shared or not. Then, as always, we replace roots with proxies. When a GC runs, all *inner objects* are removed and *shared objects* remain there because they are still being referenced by *external boundary objects*. At loading time, *i.e.,* when we want to swap in the graph, if we directly load the whole file, there will be duplicates for shared objects. This a problem because certain objects cannot have duplicates (imagine objects like "true", "false", "nil", etc) and, furthermore, because the references to those objects need to be updated to reference the original ones. Otherwise, the materialized graph will not be the same as the original one.

For example, suppose that the object G of our example is the true object, unique instance of True class. We choose the true object because we are sure it is a *shared object* but the same happens with any *shared object*. The whole subgraph is written into a file but, at loading time, the object Y and C cannot refer to a duplicate instance of True. Instead, they must refer to the unique instance, true.

To solve this problem, at serialization time, we automatically create a WeakOrderedCollection that contains references to each object of the serialized graph. Weak collections are those which only hold weakly to its elements. This means that, whenever an object is only referenced by instances of a weak class, it will be garbage collected.

This WeakOrderedCollection is directly stored in the proxy object. After replacing the roots by proxies, nobody else refers to the *inner objects* of the graph meaning they are garbage collected. When inner objects are garbage collected, there will be a nil in the WeakOrderedCollection for each of them. The references to *shared objects* remain in the WeakOrderedCollection since they were not garbage collected. Actually, in the future, external boundary objects can be removed or can stop referencing the objects inside the graph and, hence, shared objects can be garbage collected as well and a nil will be placed in the WeakOrderedCollection.

Finally, at loading time, for each object to materialize, we check the WeakOrderedCollection to see if it is a nil or not. If it is not a nil, it means it was a shared object which was not garbage collected so we take that object instead of the one that we have materialized from file. If it was not nil, it means it was an inner object or a shared object that was garbage collected. In this case, there is nothing special to do and we just take the object we have materialized from file.

There is still a problem with this alternative and it is related to graph intersection. Imagine we first swap the graph of the example. There will be an WeakOrderedCollection referencing the shared object G. If we then serialize the graph considering object Y as the root, there will be another WeakOrderedCollection referencing the shared object G. Since now there are only weak references to G, it is garbage collected. Since during materialization, each serialized graph checks its own WeakOrderedCollection, both graphs will materialize the object G generating two copies. A possible solution to this problem can be a kind of shared array between all graphs.

The advantages of this option are that we do not need to traverse the whole memory and that we can spend the extra time at loading time instead of at writing. This is fruitful because several swapped out graphs will never be swapped in. The disadvantage is that we need to store a collection with the same size of the object graph.

6.2. Handling shared objects

There are two approaches to deal with shared objects: 1) swap them and also create proxies for them; 2) detect them but do not swap them.

Detecting and swapping shared objects. The first approach is the one that makes more sense from an unused object manager. If there is a graph of unused objects, we want to swap it out no matter whether its internal objects are inner or shared. With this approach, *i.e.,* to detect and swap them, we need to create a proxy per shared object since the object could be accessed from the outside of the swapped out graph. This makes the algorithm more complex since the original graph is split and, during serialization, we need to serialize each subgraph separated. Another possibility is to serialize the whole graph into the same file and then all proxies of the graph, whether they are proxies for roots or for shared objects, has a reference to such file.

Following the example of the objects Y and G, with this approach, we create a proxy for G and we replace it. Hence, object Y will reference the proxy. After the graph is swapped out, if object Y sends a message to the proxy, the proxy must search the file on disc and materialize the graph. Once materialized, it is necessary to replace the proxies back with the just materialized objects. Replacing the root is easy because, once we materialize the graph, we know which is the root. The problem appears with shared objects. For example, how does the proxy of G knows that he must be replaced by object G? This means that proxies for *facade objects* must also store an offset in the stream or something that allows them to identify the object they need to replace when they are materialized.

Detecting but not swapping shared objects. This strategy is to detect which are the shared objects and do not swap them. It looks easier but there are problems as well. Continuing with our example, suppose that object G is not swapped. In an object graph, all the references between objects inside the graph are based on memory addresses. When we serialize an object graph into a stream, those references are transformed to indexes inside the stream. During materialization, once the objects have been recreated, the references are updated to get the memory address based references back.

A problem arises when there are objects inside the graph which refer to objects outside the graph which are *not* serialized. In our example, how can we serialize object C if G is not serialized? The first attempt is to use the real memory address instead of using internal indexes. Unfortunately, this does not work because the garbage collector moves objects around. If it moves G, at the loading time, C we will be pointing to an incorrect place.

One alternative is having an unique ID for every object which is what most object databases do. The problem is where to put such amount of bytes. It might be too large to fit inside the object header. In addition, fetching another object for the ID, may be very expensive.

Another option is to have an internal table or array for

shared objects: when we swap out an object graph, we create an array that will remain in primary memory (it is not swapped). That array has references to the shared objects. If shared objects are moved by the GC, then the GC automatically updates such references in the array. When the inner objects are being written into the file, the references to shared objects are replaced by an offset in the array (which is never changed). In our example, suppose the object G was written in the array at position 4. Then, when object C is serialized in the instance variable that refers to G, we store the position number 4.

This way, the GC can freely move objects but, when the graph is loaded, the references from objects to shared objects are updated and fixed. This means, it takes its address (offset in the array) and retrieves the current address of the object.

This solution is the one implemented by ImageSegment.

7. Even More Problems

7.1. Swapping out unused objects or swapping in used objects

Even though both phrases sound similar, both are two completely different approaches. The first idea, is to have the whole Smalltalk image in memory and swap out to disk the unused objects. Objects live in primary memory and they are just temporally swapped out while they are not being used. In the second idea, objects live permanently in secondary memory and are temporally loaded into primary memory, kept there while needed and then deleted. The RAM is treated as a cache. This second approach is the one behind most object database *e.g.,* Gemstone.

7.2. Selecting graphs to swap out

There are two possibilities to select which graphs to swap out:

User-defined subgraphs. The user, as client of the system, knows and defines which object subgraphs he wants to swap out. The system receives a particular user-defined subgraph and swaps it out to disk. It is not up to the system to decide which subgraphs are needed to swap out. This approach is the one behind Squeak Etoys Projects and ImageSegment.

Automatic system-defined subgraphs. The system, without a user decision, *automatically* detects subgraphs which are good candidates to be swapped out. A good candidate is one that has not been used for a while and which is composed by as much unused objects as possible.

In our case, we focus in automatic system-defined subgraphs of unused objects. But how we detect one graph in particular if *everything* can be a subgraph in the object memory? For each object, we can find a subgraph considering it as the root of the subgraph. This means that any object can be a subgraph. Therefore, the question is which graphs are worth to swap out. We need to limit and define constraints to be able to select special subgraphs.

One approach would be to have a process that at certain period randomly chooses objects that are candidates for roots. If a candidate is an unused object, it iterates over all its references to other objects checking whether such objects are unused too. The idea is to define a subgraph of unused objects taking the randomly selected object as root.

Once we have selected possible object graphs, we need to analyze them and check whether they are worth swapping or not. There must be policies that define that. The following is a list of a possible criteria:

Percentage of unused objects. Probably this is the easiest way to implement the algorithm. Yet, it may not be the optimal. For example, if there is an object graph of 1000 objects and 95% of the objects are unused, it is worth swapping it. In the contrary, if only 20% are unused, it does not make sense.

Percentage of shared objects. Similar to the case of unused objects, determining the percentage of shared objects is important as well. If the solution does replace shared objects with proxies, it is necessary to know the amount of shared objects because proxies also occupy memory space. If the solution does not swap shared objects, we need to know how many there are because, the more there are, the less memory that can be released.

Graph size. If the size of the graph is too small, not only we will release very little memory, but we will also add an unnecessary overhead in the system. If the graph is too big, the chances that an object inside it will be needed are bigger and we need to remember that, when loading back the graph, we load it back completely even if one single object was needed. If there is a solution that provides partial loading then this may not be a problem.

7.3. Graphs Intersections

One very common situation is graphs' interceptions. Imagine there is an object graph which was selected to be swapped out. If proxies are objects too, then there can be proxies inside the graph and such objects can be marked as unused. Thus, those proxy objects are swapped out together with the rest.

One problem in this scenario is that, while serializing an object, the serializer sends messages to the object to serialize *e.g.,*basicAt:. If that object is a proxy, such messages are intercepted and, consequently, the original graph is loaded back. This is not the expected behavior. One option is to adapt the serializer so that, in the case of proxies, it sends special messages that can be specially treated by the handler of the proxy. This way, the proxy can be serialized and materialized like any other object.

Notice that a proxy inside a graph can even be a facade object, in which case, the object swapper replaces it with a proxy. That means that we are creating a proxy for a proxy. Nevertheless, this is correct since both proxies handle different graphs.

7.4. Partial Loading

There are two scenarios where partial loading can worth it. The first one is when dealing with shared objects. Following our example of the objects Y and G, when swapping the graph of such example, object G is replaced by a proxy. If then object Y sends any message to the proxy, the whole graph is loaded back into memory. A smarter mechanism would be to only load back the graph considering G as the root. This is difficult to achieve because, during serialization, the whole graph was written. Therefore, we need support from the serializer to only materialize part of the graph.

The second scenario is in presence of large object graphs. Imagine a serialized graph that has 1000 objets. Even when one single object is needed, the *whole* graph is loaded back. For small and medium graphs it can be good enough but not for large graphs. Ideally, the serializer can serialize a graph but structure the data in packages or pages. Each page has a number of objects. Then, at materialization time, we could ask the serializer to materialize the graph but only up to certain number of pages.

7.5. Replacing Objects By Proxies Without a Full Memory Scan

So far, we have discussed different approaches to avoid a full memory scan while detecting shared objects. We never mentioned how we can replace objects with proxies. The common way to do this in Smalltalk is by using the become: message. The problem is that this is slow because it does a full memory scan to update all references from the system. Hence, even when avoiding a full memory scan for detecting shared objects, we will pay that cost for replacing objects by proxies.

There are other possibilities:

Object table. Use an Object Table memory representation in the VM. The become: method can be extremely fast since it just needs to swap two references.

First class references. If we implement first class references instead of direct pointers in the VM, then with both the previous scheme and this one, the become is fast since it requires just updating two references. These alternatives have the drawback that we need a lot of extra memory to keep the indirection (the object table or the intermediate objects of the firs class references). In addition, the overall performance will decrease since for each object access there is one indirection more to fetch.

Use the same traversal for several objects. The overhead of the method become: is the full memory scan. Nonetheless, the same method can do a bulk become, *i.e.,* we can become a list of objects to a list of proxies. The overhead of the bulk become is almost the same as the one of just becoming one single object.

In the previously mentioned random algorithm, we randomly choose "N" amount of objects that will be treated as roots. Each subgraph is analyzed to determine if it is worth swapping or not. The idea is that with the same memory traversal we can become by proxies all those roots that we want to swap. This way, we pay the cost of the memory traversal but, at least, we replace several objects.

8. Laying the First Stones of Marea: an Unused Object Manager for Pharo

We have already started to experiment building Marea, an unused object manager for Pharo. Marea already provides the basic functionality of an UOM but there are still several problems that need to be solved. Marea's swapping units are object graphs. This allows us to swap out several objects using only one or a few proxies.

8.1. Marea Subsystems

We have developed our custom *Unused Objects Detector* by modifying the Pharo VM so that we can use an empty bit of the object header to mark objects as used. By doing this, we do not use extra memory and it works efficiently. We have also modified the code of the VM that implements the message send so that it now turns on the bit when an object receives a message or when it is directly used by the VM. In addition, we have implemented all the necessary primitives from the language side to get the value of the bit, to mark and unmark all objects, etc.

An object is used when it receives a message or when it is directly used by the VM. To intercept these actions and mark the objects, we had to modify the VM. Several parts have been modified:

- The place where the normal method send is done, *i.e.,* the method lookup code.
- All the bytecoded primitives that do not go through the normal method send.
- All the places where the VM directly access to certain objects. For example, for a method lookup, the VM uses the receiver and, for each class in the hierarchy chain, it uses the class and the method dictionary until it finds a corresponding method or not.

Marea needs a reliable proxy implementation and this is the reason why we have developed Ghost [PBD+11], a uniform, light-weight and stratified proxy implementation. Ghost solves the two most important constraints of Marea: 1) it provides low memory footprint proxies; 2) it is able to proxify almost all kind of objects without problems.

Even if Ghost provides special proxies for classes and methods that respect the shape needed by the VM, there are some special classes which Ghost cannot swap right now because it has not yet developed special proxies for them. One example are the instances of Process that is another class to which the VM imposes certain shape. This means that instances of Process cannot be replaced by proxies and, consequently, we cannot swap graphs whose root are instances of Process.

That being said, notice that swapping out classes and methods is something desired and likely to happen. In the contrary, swapping out Process instances is not that common.

Ghost makes a clear difference between interceptors and handlers. Proxies only play the role of interceptors and all they do is to forward intercepted messages to handlers. Each proxy must have an associated handler. Different proxies can use different handlers and vice versa. Handlers' responsibility is to deal with the method interceptions that the proxies trap.

That being said, it looks like we need two objects: one for the proxy and one for the handler. Normally, a proxy instance has a reference to a handler instance. Nonetheless, this is only necessary when the user needs one handler instance per target object which is not often the case. In Marea, the handler is stateless and can be shared among the different proxy instances. It can be referenced through a class variable, a global variable, a Singleton, etc. Therefore, apart from the low memory footprint provided out of the box from Ghost, we can even avoid the memory cost of a handler instance and a reference per proxy.

For serialization, Marea uses Fuel [DPDA11], a general purpose framework to serialize and deserialize object graphs using a pickle format which clusters similar objects. Fuel is highly customizable to cope with different objects, it does not need specific VM support, it has a clean object-oriented design and provides most of the required features for a serializer.

8.2. Marea in a Nutshell

Right now, the input is the desired graph to swap out. Marea performs the following steps:

1. Serialize the object graph.
2. Create a proxy instance and set the filename in its state.
3. Replace the root of the graph with the created proxy. Once this is done, there are no other references to the original root object. For that reason, the next time the Garbage Collector runs, all inner objects of the graph are removed saving memory.
4. Now, whenever the proxy receives a message, the file is searched on disc and the graph is materialized in memory.
5. Finally, the proxy is replaced by the materialized root.

This procedure is the simplest possible, *i.e.,* without taking into account *e.g.,* the problem with shared objects as discussed in Section 6.

9. Related Work And Future Work

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[5]. 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 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 back into primary memory. Its approach is quite similar to LOOM.

ImageSegment [MPBD+10] is an object swapping and serializer for Squeak Smalltalk. ImageSegment seems to be fast in certain scenarios. However, it is necessary to explain how ImageSegment works. Basically, ImageSegment receives a user defined graph and it needs to distinguish between *shared objects* and *inner objects*. To do that, it has to do a full memory traversal using the garbage collector infrastructure.

All *inner objects* are put into a byte array which is finally written into the stream using a primitive implemented in the virtual machine. *Shared objects* are not swapped. Moreover, there is an array which remains in primary memory that refers to them. ImageSegment is fast mostly because it is implemented in the virtual machine. The real problem is that it is difficult to control which objects in the system are referencing to objects inside the subgraph. For that reason, most of the times there are several *shared objects* in the graph. The result is that the more *shared objects* there are, the less memory that can be released.

Finally, notice that ImageSegment does not select which graphs to swap neither manages unused objects. In the contrary, ImageSegment's input is directly a user-defined object graph.

GemStone [BOS91] is a Smalltalk object server and database which manages primary and secondary memory as well. To provide its features, it implements 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 the other solutions 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 the others, objects live in primary memory and they are just swapped out when not needed and loaded back when needed.

10. Conclusion

In this paper, we looked into the problem of swapping unused objects between primary and secondary memory in object-oriented systems. We have analyzed not only most of the problems and challenges for building an unused object manager, but also we have presented our first steps in Marea project. What is important is the fact that most of the problems and challenges are completely general and independent of the technology.

We have demonstrated that to build a real unused object manager, it is necessary to provide a proxy implementation that can proxy almost all kind of objects, a fast object serializer and a way to efficiently identify graphs of unused objects. We presented Marea, our first experiment towards an unused object manager for dynamic languages. Marea already provides the basic

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

functionality of an object swapper but there are still several problems that need to be solved.

Once we are capable of detecting unused objects, replacing them with proxies and swapping them to secondary memory, we need to face more advanced problems. We gave a list of those problems which include how to deal correctly with objects inside the graph that are referenced also from objects outside, how to map addresses between primary and secondary memory, how to avoid using more memory with the solution than the one that can be released, how to make the possible solution efficient, how to select which graphs to swap out, etc.

For most of the problems, we proposed different alternatives but none of them was good enough regarding our two constraints: 1) saving as much memory as possible *i.e.,* do not use more memory with proxies and other temporally required data, than the one that can be released by swapping unused objects; 2) minimizing the overhead *i.e.,* the system should not slow down too much the computation when detecting unused objects or when swapping them between memories.

As future work, we plan to solve the mentioned problems in Marea and attempt to provide an unused object manager for Pharo – generalizable to dynamic languages – that fulfills the requirements we presented.

References

[ADD+10] Jean-Baptiste Arnaud, Marcus Denker, Stéphane Ducasse, Damien Pollet, Alexandre Bergel, and Mathieu Suen. Read-only execution for dynamic languages. In *Proceedings of the 48th International Conference Objects, Models, Components, Patterns (TOOLS-Europe'10)*, Malaga, Spain, June 2010.

[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.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991.

[DPDA11] Martin Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Clustered serialization with fuel. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2011)*, Edinburgh, Scotland, 2011.

[GHVJ93] Erich Gamma, Richard Helm, John Vlissides, and Ralph E. Johnson. Design patterns: Abstraction and reuse of object-oriented design. In Oscar Nierstrasz, editor, *Proceedings ECOOP '93*, volume 707 of *LNCS*, pages 406–431, Kaiserslautern, Germany, July 1993. Springer-Verlag.

[Jon96] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[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.

[LGN08] Adrian Lienhard, Tudor Gîrba, and Oscar Nierstrasz. Practical object-oriented back-in-time debugging. In *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *LNCS*, pages 592–615. Springer, 2008. ECOOP distinguished paper award.

[MPBD+10] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Experiments with a fast object swapper. In *Smalltalks 2010*, 2010.

[PBD+11] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Efficient proxies in smalltalk. In *Proceedings of ESUG International Workshop on Smalltalk Technologies (IWST 2011)*, Edinburgh, Scotland, 2011.

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