Marea: An Efficient Application-Level Object Graph Swapper

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

JOURNAL OF OBJECT TECHNOLOGY
Published by AITO — Association Internationale pour les Technologies Objets, ©JOT 2011
Online at http://www.jot.fm.

Marea: An Efficient Application-Level Object Graph Swapper

Mariano Martinez Peck[ab] Noury Bouraqadi[b] Marcus Denker[a]
Stéphane Ducasse[a] Luc Fabresse[b]

a. RMoD, Inria Lille Nord Europe, France http://rmod.lille.inria.fr

b. Univ. Lille Nord de France, Mines de Douai, France http://car.mines-douai.fr/

Abstract During the execution of object-oriented applications, several millions of objects are created, used and then collected if they are not referenced. Problems appear when objects are unused but cannot be garbage-collected because they are still referenced from other objects. This is an issue because those objects waste primary memory and applications use more primary memory than they actually need. We claim that relying on the operating system's (OS) virtual memory is not always enough since it cannot take into account the domain and structure of applications. At the same time, applications have no easy way to parametrize nor cooperate with memory management. In this paper, we present Marea, an efficient application-level object graph swapper for object-oriented programming languages. Its main goal is to offer the programmer a novel solution to handle application-level memory. Developers can instruct our system to release primary memory by swapping out *unused yet referenced objects* to secondary memory. Our approach has been qualitatively and quantitatively validated. Our experiments and benchmarks on real-world applications show that Marea can reduce the memory footprint between 23% and 36%.

Keywords Object Swapping; Unused objects; Virtual Memory.

1 Introduction

In OO software, some objects are only used in certain situations or conditions and remain not used for a long period of time. We qualify such objects as *unused*. Such objects are reachable, and thus cannot be garbage collected. This is an issue because unused objects waste primary memory [Kae86]. This means less applications running on the same hardware or slowdowns because of operating system virtual memory swapping.

Typical unused objects are part of the applications' runtime. They are loaded on startup but most of them are useless regarding application functionalities. Consequently, applications usually occupy more memory than they actually need. Section 8.4 presents benchmarks of three real applications. These benchmarks report 80% of *unused* objects. The situation is even worse in small systems. For example, a hello world application in Java SE 1.6 occupies 25MB of RAM.

Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, Luc Fabresse. *Marea: An Efficient Application-LevelObject Graph Swapper* . In Journal of Object Technology, vol. 12, no. 1, 2013, pages 2:1–30.
doi:10.5381/10.5381/jot.2013.12.1.a2

Unused objects can sometimes be a symptom of an even more serious problem: memory leaks [BM08]. In presence of memory leaks, applications use much more resources than they actually need. They may eventually exhaust the available memory and lead to system crashes.

Operating systems have been supporting virtual memory since a long time [Den70, Sta82, CH81, WWH87, KLVA93]. Virtual memory is transparent in the sense that it automatically swaps out unused memory organized in pages governed by some strategies such as the least-recently-used (LRU) [CH81, CO72, Den80, LL82]. As virtual memory is transparent, it does not know the application's memory structure, nor does the application have any way to parametrize or cooperate with the virtual memory manager.

In this paper, we propose Marea, an Object Graph Swapper (OGS) whose main goal is to offer the programmer a solution to handle application-level memory. Developers can instruct our system to release primary memory by swapping out *unused objects* to secondary memory. Marea is designed to: 1) save as much memory as possible *i.e.,* the memory used by its infrastructure is minimal compared to the amount of memory released by swapping out unused objects, 2) minimize the runtime overhead *i.e.,* the swapping process is fast enough to avoid slowing down primary computations of applications, and 3) allow the programmer to influence the objects to swap.

The contribution of this paper is twofold. On the one hand, we introduce a precise description of problems, challenges, algorithms and design aspects while building an OGS for object-oriented systems. On the other hand, we present Marea, our efficient solution and its implementation in the Pharo programming language [BDN+09]. We show that Marea can reduce the primary memory occupied by applications up to 36%. Our implementation also demonstrates that we can build such a tool without modifying the virtual machine (VM) yet with a clean object-oriented design.

The remainder of the paper is structured as follows: Section 2 starts with a brief analysis of the existing solutions and their limitations. Section 3 lists the requirements to build an OGS. Section 4 presents Marea and gives an overview. Section 5 explains the need and the difficulty of correctly handling shared objects. The algorithms of Marea's object swapper are presented in Section 6 together with the solution to graph intersections. Section 7 explains how to use Marea from the developer point of view. Section 8 provides benchmarks for swapping code and data while including an analysis of unused objects. Marea's implementation and requirements are described in Section 9. We evaluate Marea over a list of requirements and desiderata and discuss infrastructure-specific issues and optimizations in Section 10. Finally, in Section 11, related work is presented before concluding in Section 12.

2 Limits of Existing Solutions

The memory that is occupied but unused can be split in two parts. One part is used by the code of the application and linked libraries. The other part stores data generated as a result of the execution. Reducing both parts is of interest.

Reduced and Specialized Runtime. Solutions belonging to this family decrease the memory usage by *removing code* and building small runtimes. For example, J2ME is a stripped-down version of Java for embedded devices that contains a strict subset of the Java-class libraries. However, decisions behind this reduction are taken by the developers of J2ME. From developers perspective, J2ME is a monolith that cannot be adapted.

J2ME degrades the Java environment and APIs right from the specification. Moreover, some J2ME APIs are not compatible with J2SE, breaking the rule "compile once, run everywhere." For instance, if an application needs to directly connect to a relational database, it

is not possible to do it in the same way it is done in standard Java because J2ME does not provide the JDBC API.

Custom and Specific Runtime. Contrary to the previous alternative where the runtime's developers decide what to include, another alternative is to let developers decide what each application needs and create a specific and customized runtime for it. JITS (Java In The Small) is a tool that allows the developer to customize the runtime to avoid the need of embedding unused packages or features [CGV10]. The idea of JITS is to develop using standard Java. Then, for deployment, JITS creates a tailored runtime according to the application's needs. JITS developers often develop applications in a subset of Java that eases the identification of unused classes (no inheritance or polymorphism and procedural programming). Similar solutions are implemented in other programming languages. For example, VisualWorks Smalltalk provides a runtime packager[1], which makes smaller runtimes by explicitly removing classes and packages.

Still, with this strategy, developers need to know *with absolute certainty* what is required by their applications. At development stage, it is difficult, time-consuming and sometimes impossible to figure out what an application needs for all possible execution paths even with static analyzers. Most static analyzers do not take into account reflective features that are often used to support application evolution and dynamic code loading [BSS+11].

Application Data. Certain types of applications, such as graphics editors, often have to manipulate images with a size that is bigger than primary memory. Developers address this issue by often building their own *ad hoc* internal memory management system [EGK95]. Nevertheless, the traditional way to handle data is to save it to files or databases and load them when necessary. While database connectors can be reused, developers still have to handle data storage and loading explicitly.

Operating System Virtual Memory. While operating systems (OS) provide virtual memory, this solution is not satisfactory to address the unused objects issue because of the following reasons :

- Garbage Collection: Memory not used can, in theory, be swapped by the operating system to disk. However, the objects on disk also need to be garbage collected. As GC's working set involves all objects (including those that are on disk), this can trigger memory thrashing (traffic between primary and secondary memory) that degrades performance by orders of magnitude [YBK+06, HFB05]. In Section 11 we discuss some solutions to this problem.
- Persistence: In image-based systems such as Smalltalk, we can persist the current object memory state in a disk snapshot. As the virtual machine is not aware of the memory management of the OS, it cannot save its memory in a state where some parts are swapped out while others are not. Thus, saving the system means swapping in all data and writing out the complete heap.
- The OS only knows about memory pages or similar structures used by virtual memory implementations. Therefore, the OS cannot guess which objects are the most appropriate ones to swap. The information about object usage is only available at the application-level [EKO95, EGK95].

[1] Explained in VisualWorks Application Developer's Guide.

3 Requirements for an Application-level Object Graph Swapper

We argue that a novel OGS for OO languages should comply with the following requirements:

- **Efficient Object-Based Swapping Unit.** Since we are targeting an application-level object-oriented system, the swapping unit has to be at the object level instead of pages of bytes. It can be one object or several objects together. However, it has to provide an *efficient granularity* to generate as little object faulting as possible and when swapping in to load the minimum of unnecessary objects.
- **Uniformity.** Some languages reify constructs and runtime parts. For example, in Smalltalk, method execution contexts, classes, methods, closures, processes, semaphores, among others are all first class objects. The OGS should be uniform and swap any kind of object, be it application data or a language reification.
- **Automatic swapping in.** If an object that was swapped out is needed, the system must *automatically* swap it in.
- **Automatic swapping out.** Although the programmer's involvement is a good complement for current memory management, providing also automated mechanism to swap out is appealing *e.g.,* for the case when nothing is specified by the developer.
- **Transparency.** From the point of view of an application, the system must be transparent: the application will get the same results whether it is swapping objects or not. From the application development point of view, the application code should not be polluted with code related to swapping.

Besides those requirements, there are also desiderata:

- **Cooperation with the application.** The solution should allow application programmers to parametrize what, when and how to swap. This opens up many new possibilities as we can take domain knowledge into account for the decision of what to swap. The idea is that, since we are at the object level, much more sophisticated strategies can be developed than the simple LRU replacement available with OS virtual memory [CH81, CO72].
- **Portability.** As much as possible, the solution should provide the OGS as a tool that is completely implemented on the language side without requirements on the virtual machine nor the OS.

4 Marea Overview

Marea temporarily moves object graphs to secondary memory releasing part of the primary memory. In particular, our solution addresses the problem of shared objects: objects that are swapped but also referenced from outside the swapped graph as discussed in Section 5.

The input for Marea is a set of *user-defined* graphs. Swapped out graphs are swapped in as soon as one of their elements is needed. Graphs to swap can have *any* shape and can contain *any* type of object. This includes classes, methods, closures, and even the execution stack, which are all first-class objects in Marea's implementation language [BDN+09]. This means that our solution works with both of the scenarios mentioned in Section 2 *i.e.,* code (runtime) and application-specific objects.

When Marea swaps a graph, it correctly handles all the references from outside and inside the graph. When one of the swapped objects is needed, its graph is *automatically* brought back into primary memory. To achieve this, Marea replaces original objects with proxies [GHVJ93]. Whenever a proxy intercepts a message, it loads back the swapped graph from secondary memory. Since we are changing the living object graph at runtime, this process is completely transparent for the developer and the application. Any interaction with a swapped graph has the same results as if it was never swapped.

Figure 1 shows that Marea is built on top of four main subsystems: (1) object graph swapper, (2) proxy toolbox, (3) serializer and (4) object storage. We describe them in the following.

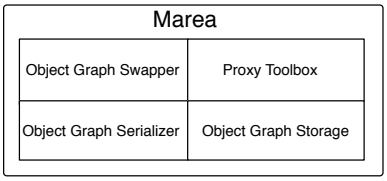


Figure 1 – Marea subsystems.

Object Graph Swapper. Its task is to efficiently swap graphs between primary and secondary memory. As we explain later in Section 5, detecting and correctly handling the *shared objects* of a graph is a challenging task that the OGS addresses. The OGS uses a serializer and a proxy toolbox to save, load and replace objects.

Proxy Toolbox. The OGS replaces some objects of the graph[2] being swapped with proxies. The references to proxified objects are replaced with references to proxies. We refer to this functionally as *object replacement*.

To deal with the uniformity requirement, Marea requires that proxy toolbox supports proxifying any kind of objects including language runtime objects (classes, methods, contexts) since in Smalltalk such elements are objects too.

Object Serializer. When an OGS needs to swap out a graph, the first step is to serialize it. An important requirement of Marea is to have access to a fast serializer. When a swapped out object is needed, it is essential to be able to load it back as fast as possible to avoid application slowdowns. Furthermore, the serializer must be able to correctly serialize and materialize any kind of objects such as classes, methods, contexts or closures in accordance with the implementation language.

Object Graph Storage. Its main responsibility is to store and load serialized graphs (each serialized graph is an array of bytes). The graph storage responsibilities are reified in a separate class following the strategy design pattern. This allows Marea to easily support different backends and the user can choose which one to use or even create its own. Current backends are the local file system, Riak[3] and MongoDB[4] NoSQL databases.

[2] Section 6 explains which objects are actually replaced by proxies.

[3] http://wiki.basho.com/Riak.html

[4] http://www.mongodb.org/

5 The Main Challenge: Dealing Efficiently with Shared Objects

This section defines the concepts and vocabulary used after. Then it presents the main challenge that an OGS should address: the case of "shared objects." In Section 6, we describe in detail how Marea deals with them.

5.1 Vocabulary

In an object-oriented system, related objects reference each other and form a graph. Objects are nodes of such a graph, while references are the edges of it. When dealing with an application-level OGS for object-oriented languages, we actually end up handling objects that are part of graphs. We need to deal with object graphs when analyzing the system to identify unused objects and also when swapping out and in. Therefore, Figure 2 shows an example of an object graph (surrounded by a rectangle). Through this example we define the terms used in this paper to avoid confusion:

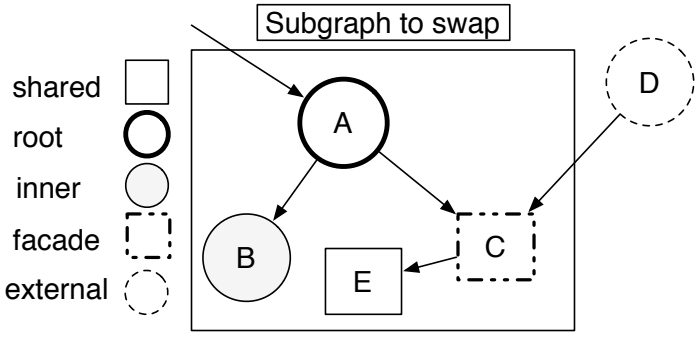


Figure 2 – Object classification based on graph structure.

A root object is a starting node that allows other nodes of the subgraph to be retrieved by following all the edges (references) *e.g.,*A.

Internal objects are all the objects belonging to the graph *i.e.,* A, B, C, and E.

External objects are outside the graph to swap such as D.

Shared objects are *internal objects* that are accessible not only through the roots of graph, but also from outside the graph *e.g.,* C and E.

Facade objects are *shared objects* that are directly referenced from *external objects e.g.,* C.

Inner objects are *internal objects* that are only accessible from the *roots e.g.,* B.

5.2 The Case of Shared Objects

Detecting and correctly handling shared objects of a graph (*e.g.,*C and E in Figure 2) is a challenging task that an OGS should address. Whether *shared objects* should be swapped or not, is an important decision. In any case, it is necessary to correctly deal with them because it is common to have shared objects inside graphs. In addition, since Marea allows the programmer to freely select any object graph to swap, the probability to get shared objects increases.

The following illustrates the problems introduced by shared objects. Figure 3 shows that if we simply replace all objects of the graph with proxies, we still need to know that object C is shared. Otherwise, after having reloaded the graph, D will continue to reference Pc. The correct behavior is to recreate the same graph as it was before the swapping. Therefore,

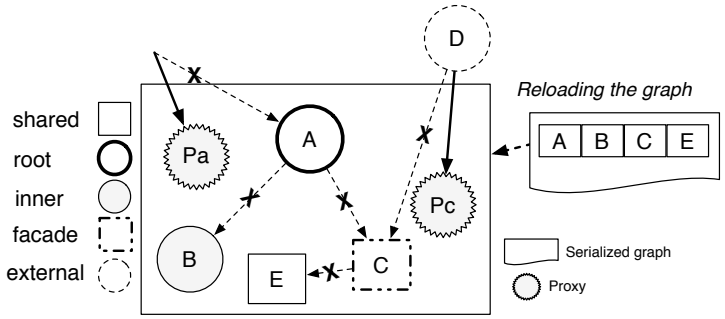


Figure 3 – The need to manage shared objects.

when we reload the graph, D should be updated to refer the materialized C, and C should refer to the materialized E. The problem is that programming languages do not provide an easy or incremental way of detecting *shared objects*. Objects do not have back-pointers to the objects that refer to them. Hence, a costly full memory traversal is often required as ImageSegment does [MPBD+10a]. By using weak collections that hold references to proxies and letting the garbage collector do its job, Marea avoids a full memory scan as we explain in the next section.

6 Marea's Object Swapper Algorithms

Marea provides an efficient approach to detect and correctly handle shared objects while avoiding a full memory scan. Marea creates a proxy for *every* object of the graph (whether it is a root, an inner or a shared object). Then each object is replaced by its associated proxy (previous object pointers now point to the associated proxy). As a result, proxies for inner objects are not referenced from any other object. Indeed, inner objects were *only* referenced from inside the graph and all objects were replaced by proxies. Hence, as soon as the GC runs, it will garbage collect all inner objects and all proxies leaving only those proxies for *facade objects* and the root of the graph. During the swap in, the graph is materialized and all proxies associated with the graph are replaced by the appropriate materialized objects. The subsequent sections explain in detail the swapping out and in operations.

6.1 Swapping Out

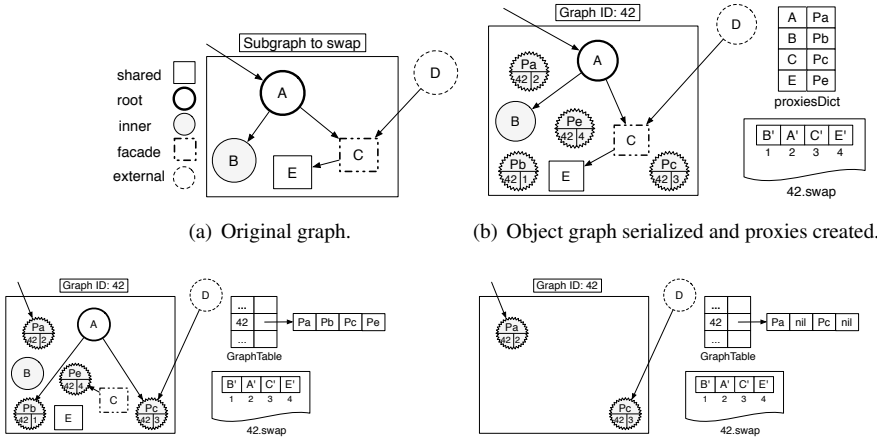
The swapping out phase is triggered explicitly, *i.e.,* the programmer has to instruct the OGS to swap out a graph. Marea's strategy to *swap out* object graphs decomposes into the following steps.

1. *Initialization:* Marea assigns an automatically generated and unique ID (a number) to each graph. In the example of Figure 4, Marea assigns the number 42 to the graph.

2. *Serialize the object graph:* with the default configuration, Marea serializes the graph into a single file located in a secondary memory (*e.g.,* on hard disk). The filename is the graph ID (42.swap in our example).

3. *Create proxies:* Marea creates a proxy for each object of the graph. We introduce proxiesDict, an identity dictionary that maps objects to proxies. In the example, it maps A, B, C, and E to their respective proxies, namely Pa, Pb, Pc, and Pe. The result is

depicted on Figure 4(b)[5]. We label the objects serialized into the serialization stream as "prime." For example, the object A is the original one and A' represents its serialized version. The stream with contents B' A' C' E' represents the serialized graph A B C E. Each proxy knows the position in the stream of the proxified object apart from its graph ID (*e.g.,*Pa knows the position of A'). This information stored in the proxy is then used during the swap in process which is explained in Section 6.2.



(a) Original graph.

(b) Object graph serialized and proxies created.

(c) Objects replaced by proxies and GraphTable update.

(d) Final result after a GC.

Figure 4 – Steps to swap out: Serialization, proxy creation, objects replacement and GC execution.

4. *Replace original objects with proxies:* all references to an original object are updated and changed to point to the appropriate proxy as shown in Figure 4(c). We replace each key (original object) of proxiesDict with its associated value (its proxy)[6]. In the example, A now points to Pb and Pc, D points to Pc, and C points to Pe.

5. *Update GraphTable*: the OGS maintains a global table called GraphTable. This is a dictionary in which a key is a graph ID and a value is a collection holding *weak* references to the proxies associated with the graph. We need the GraphTable because during the swap in of a graph we need to retrieve all its proxies and replace them with the appropriate materialized objects. In our example, this step consists of adding graph 42 (*i.e.,* the graph with ID 42) into GraphTable as shown in Figure 4(c).

6. *Cleaning:* we can now discard the temporary proxiesDict. Once this step is done, none of the internal objects are strongly referenced anymore, *i.e.,* there are no strong references to objects A, B, C, or E. Consequently, when the next GC runs, all these objects are removed. In addition, all weak references to those proxies for inner objects are replaced by *nil* (cleared) in the GraphTable. Figure 4(d) shows the final result after a GC execution. It only remains in memory the proxy for the *root* (Pa in this example) and the proxies for the *facade objects* (only Pc in this example).

[5]For sake of clarity, we do not show the object references from structures like proxiesDict that are external to the graph.

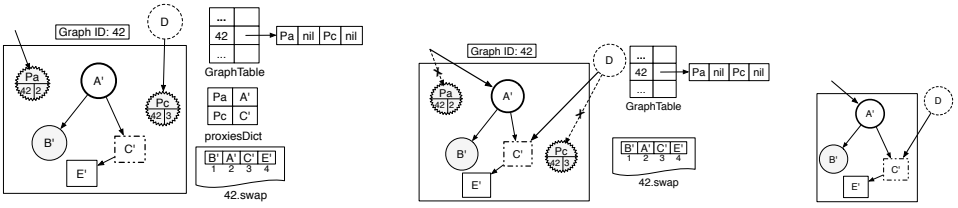
[6]In Marea's implementation this is solved by using a reflective Smalltalk operation called "become" that swaps atomically all the references to the receiver to the argument and the inverse. This is explained in more detail in Section 9.2 and Section 10.2.

6.2 Swapping In

The swap in of a graph is triggered when one of its proxies is accessed, for example, via a message send or an instance variable access. This situation, *i.e.,* when the system needs an object that was swapped out, causes what is known as "object faulting" [HMB90].

When a proxy intercepts an action it has certain information about it. For example, if the action is a message send, the proxy knows which message was sent and its arguments. The proxy passes all the interception's information to a handler. The handler first makes the OGS to swap in the graph associated with the proxy. Then, the handler forwards the original message to the object that was initially replaced by the proxy. Section 9.1 gives more details about Marea's classes and their responsibilities.

We continue our example from Figure 4(d). We assume that the proxy Pa receives a message. The resulting *swap in* decomposes into the following steps:



(a) Object graph is materialized and proxies are associated.

(b) Proxies are replaced by materialized objects.

(c) Final result.

Figure 5 – Steps to swap in: Materialization, association of proxies with materialized objects, proxy replacement and cleaning.

1. *Materialize the object graph:* this is done by first getting the file named after the proxy's graph ID. Once we have the stream, we materialize the object graph (with all the objects references) into primary memory.

2. *Associate proxies with materialized objects:* The graph ID is also used to retrieve the list of proxies from the GraphTable. This list includes only *alive* proxies *i.e.,* proxies that have survived GC because of being referenced. Each proxy stores the position in the stream of the corresponding proxified object. With this information, we can identify the materialized object associated with a given proxy. In Figure 5(a), Pa corresponds to the object at the position 2 in the serialization stream of graph 42. Thus, Pa is associated to A'. The result of this process is a temporary proxiesDict in which the keys are proxies and the values are the associated materialized objects.

3. *Replace proxies with original objects:* all references to each proxy in the proxiesDict are updated so that they now point to the corresponding materialized object. The proxies we need to replace are those that are stored as keys in the proxiesDict and their associated materialized objects are the values of the dictionary. Figure 5(b) illustrates the result of this step.

4. *Cleaning:* we discard proxiesDict, the current graph is removed from the GraphTable and the file for the serialized graph is deleted[7]. Figure 5(c) presents the final state after a GC run. The result of the swapping in is a graph equal to the one that was swapped out, as Figure 2 shows. Notice that even if the materialized objects are called A', B', C', and E', they are equal to A, B, C, and E.

[7] Marea's current implementation offers an API to trigger an automatic compaction of the GraphTable. Such compaction can also be done automatically after each GC.

Pre- and post-actions. Some objects may have to execute specific actions before being swapped out or after being swapped in. Another problem while swapping object graphs is to detect implicit information that is necessary to correctly load it back later [Ung95]. For example, when swapping out Smalltalk objects like true, false, nil, Transcript or Processor, we do not want to recreate them when swapping in. Instead, we have to refer to the already existing ones. This is why Marea provides:

- Pre- and post-actions for serialization and materialization[8]. Marea requires that the serializer provides a hook to execute user-defined actions once an object has been serialized or materialized. This is critical to ensure a correct serialization or materialization for some core objects and for classes. For example, hashed collections need to be rehashed after being materialized because they may refer to objects that have changed their hash. This hook may also be used by developers to define custom actions.
- Marea provides similar hooks so that programmers can define arbitrary actions that are executed before or after application-specific objects and classes are swapped out or in.

6.3 Handling Graph Swapping Intersection

Object graphs are complex and it is inevitable that a programmer may end up trying to swap out a graph that contains proxies introduced by the swapping of another graph. We call this situation *graph swapping intersection* and it is an issue that should be addressed. Otherwise, *swapping out* a graph may lead to *swapping in* another unnecessary graph. Even worse, graphs can be swapped in with an inconsistent shape.

Problem of Shared Proxies. Following with the example of Figure 2, imagine that we swap out graph 42. We end up in the situation of Figure 4(d). Suppose that now the user wants to swap out a graph that includes the object D *e.g.,* graph 43. This raises the question of swapping proxies since the graph of D shown in Figure 4(d) includes a proxy resulting from the previous swapping out of graph 42. Indeed Pc is a proxy of a facade object. We call such a proxy a *shared proxy*.

Swapping graphs with proxies is an issue since it may lead to the loss of shared proxies, which in turn results into corrupted graphs. In our example, imagine that we apply the regular swapping out mechanism for the graph of root D and, therefore, we swap out the proxy Pc as if it were a regular object. This will produce two proxies: Pd which replaces D and Ppc which replaces Pc. Once D is garbage collected, the only remaining references to Ppc are weak since they are from the GraphTable. This leads to Ppc being collected too. If now graph 42 is swapped back in, the result will be the correct materialization of A', B', C' and E'. The problem appears when swapping in graph 43. We will get D' referencing a proxy Pc' that has no relation at all with the already materialized object C' from graph 42. And if now Pc' receives a message, there will be an error because its graph (ID 42) has already been swapped in.

Solution part 1: Swapping Out with Shared Proxies. During the swapping out phase, Marea only creates proxies to replace plain objects (*i.e.,* objects that are not proxies). Proxies found in a graph are kept unchanged, and they are inserted into the SharedProxiesTable. This table is a dictionary where a key is a proxy ID and a value is a *strong* reference to the proxy. A proxy ID is unique in the system and it is easily computed from the graph ID and the position in the stream.

[8] We use the concept of "materialization" to refer to the "deserialization."

Following our example, Figure 6 shows the result of swapping out first the graph 42 and then the graph 43 with the handling of shared proxies. When swapping out graph 43, Pc is added to both, GraphTable and SharedProxiesTable.

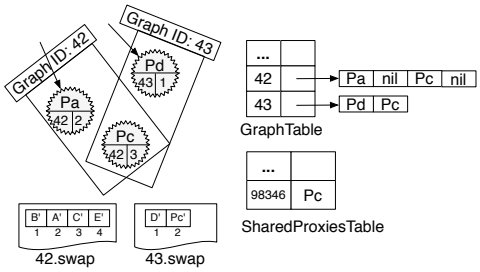


Figure 6 – Swapping out two graphs with shared proxies. First, graph 42 and then graph 43.

Solution part 2: Swapping In with Shared Proxies. To handle swapping in graphs with shared proxies, the swap in algorithm performs as described in Section 6.2 except for proxies found in the materialized graph. Those proxies are shared ones. Each materialized shared proxy is replaced by the appropriate object from the SharedProxiesTable which is looked up based on the proxy's ID.

To illustrate the algorithm, consider again our example with two graphs of Figure 2. After swapping graph 42 (starting with object A) and graph 43 (starting with object D) in this order, we obtain a structure depicted by Figure 6. Now, two swapping in scenarios may occur: graph 42 is swapped before 43 or vice-versa. We will consider those scenarios and show that both graphs are correctly rebuilt.

In the first scenario, graph 42 is swapped in first. None of the materialized objects (A', B', C', and E') is a proxy. We simply replace proxies Pa and Pc found at entry 42 of the GraphTable with the right objects, namely: A' and C'. The replacement, which is system-wide, affects the SharedProxiesTable since C' replaces Pc. When graph 43 is then swapped in, Marea first replaces the proxy Pd found in entry 43 of the GraphTable by the materialized object D'. Entry 43 of the GraphTable also includes C' which is ignored because it is not a proxy. Then, Marea detects a shared proxy issue since there is a proxy (Pc') in the materialized graph. Next, we use the ID of Pc' to get C' from the SharedProxiesTable. Finally, Pc' is replaced by C'. Thus, both graphs are reconstructed correctly and sharing is preserved.

In the second scenario, graph 43 is swapped in first. Marea first replaces the proxy Pd by the materialized object D'. Entry 43 of the GraphTable also includes Pc which is ignored because its graph ID is 42 and not 43. Marea detects a shared proxy issue since there is Pc' in the materialized graph. We use the ID of Pc' to get Pc from the SharedProxiesTable. Next, we replace Pc' by Pc. When graph 42 is then swapped in, Marea replaces proxies Pa and Pc found at entry 42 of the GraphTable with the right objects, namely: A' and C'. The replacement makes D' reference C'. Again, both graphs are reconstructed correctly and sharing is preserved.

Cleaning SharedProxiesTable. Since GraphTable contains weak references, Marea can listen when the GC clears those weak references and then do an automatic cleanup and compaction of the table. However, SharedProxiesTable holds strong references to its values (they can be proxies or normal objects). To clear this table, Marea needs to analyze GraphTable. If a proxy of SharedProxiesTable is not referenced from any entry in GraphTable, it means that all its related graphs were swapped in so it can be removed.

We could make Marea trigger this cleaning when swapping in a graph, or hook into the GC and trigger it automatically after each GC. However, this adds an unnecessary overhead

for a small gain in memory. Because of this, the SharedProxiesTable is cleaned by Marea sporadically, that is, after having swapped in a customizable number of graphs. Nevertheless, the API allows application programmers to trigger such cleaning when desired.

7 How to Use Marea

The API of Marea is very straightforward. Swapping in is automatic. Thus, developers only have to trigger explicitly swap outs by providing a root object as following:

```
ObjectGraphSwapper new  swapOutGraph: rootObject
```

Marea detects the boundaries of a graph by computing the transitive closure starting from the root instance variables. Any object can be considered as root including classes. Since object graphs related to classes are complex, Marea's API provides some helper methods such as swapOutClass:, swapOutClassAndSubclasses:, swapOutClassWithInstances: and swapOutClassWithSubclassesWithInstances:. The following is an example to swap out the maximum number of classes possible.

```
MareaExamples>>swapOutClassesForMaxPossiblePackages
1  | classesToSwap |
2  classesToSwap := self classesOfAllNoneKernelPackages.
3  classesToSwap do: [:each |
4         ObjectGraphExporter new swapOutClassWithSubclassesWithInstances: each].
```

In this example, each class is considered as a root of an object graph that is swapped out. The graph includes the method dictionary, compiled methods, the class organizer, the instances, the subclasses as well as the sub-instances. In line 2, we obtain all the classes in the system except those of the kernel that should not be swapped out. Kernel classes are those central to the execution of the Pharo system and Marea and this is why we do not to swap them out. ObjectGraphSwapper maintains a list of packages and classes that cannot be swapped out. In line 3-4 we swap out the classes.

8 Benchmarks and Case Studies

In this section, we benchmark Marea with real applications and show the results in terms of speed and memory consumption. We performed several experiments to measure how much memory could be gained using Marea. Our experiments on real-world applications show that Marea reduced the memory footprint between 23% and 36%.

Experiment Setup. The following is the setup of our experiments.

1. We took the PharoCore 1.3 runtime as available from the Pharo project site[9]. PharoCore is a small environment with the minimal toolsets and libraries (no external packages or developer tools like refactoring engine or code assist). In addition, we used a PharoCore configuration dedicated for production usage that aggressively cleans caches, fonts, help information and other meta-data and also removes code *e.g.,* all unit tests resulting in a 6.9 MB runtime (excluding the VM size).

2. We loaded some real applications.

[9] http://www.pharo-project.org

3. We ran some scripts (similar to the one of Section 7) to swap out as many objects as possible. In some experiments the roots of object graphs were classes, in others they were packages.

4. We ran the application over several real-case scenarios *i.e.,* we used it and we ran actions on it. While doing so, needed graphs were swapped in.

5. We measured the memory used and the swapping speed.

Then, we performed the same experiments but on a PharoCore *without* object swapping and analyzed the gained memory.

Environment Configuration. Our experiments were run with PharoCore 1.3 build number 13327 and Cog Virtual Machine version 'CoInterpreter VMMaker-oscog-EstebanLorenzano.139'. The operating system was Mac OS 10.6.7 running in a 2.4 GHz Intel Core i5 processor with 4 GB of primary memory DDR3 1067 MHz. The hard disk was SATA, 500 GB and running at 7200 RPM. The swapping was done with the default object graph storage, *i.e.,* files in the local filesystem creating one file per swapped graph.

8.1 Applications

DBXTalk CMS Website. It is the web application of the DBXTalk project[10]. It is developed with Pier[11], a CMS based on the Seaside Web framework [DRS+10].

Moose Suite (version 4.7). Moose[12] is a platform for software and data analysis [NDG05]. It allows users to handle different large models that represent the packages and data that are being analyzed.

Dr. Geo (version 11-12g). This is an interactive geometry software that allows one to create geometric sketches. The Pharo runtime used by Dr. Geo is already reduced. It is based on a PharoCore using the production configuration. Besides, developers compacted it even further by removing part of the code included in PharoCore. Since the runtime is only 11.2 MB, it is an interesting challenge.

Pharo Infrastructure (version 1.3). Apart from benchmarking the memory consumption of different applications built on top of Pharo, we measure the Pharo infrastructure (the IDE and the runtime) itself without anything extra loaded on it. The idea is to compare the infrastructure versus applications and analyze if we can compact Pharo's runtime even more.

All these selected real-world applications have different characteristics with the purpose of being representative. DBXtalk CMS is a web application, Moose is a large standalone software (around 174 packages 1364 classes and 121010 lines of code), DrGeo is a mobile app and Pharo is the infrastructure itself.

8.2 Swapping Out Code

Marea can swap different user-provided object graphs. However, a common scenario is when the user wants to swap out "unused code" to make its application's runtime smaller and less memory consuming.

[10] The website is currently located in http://dbxtalk.smallworks.com.ar/
[11] www.piercms.com
[12] http://www.moosetechnology.org/

One use case in which Marea is very useful is when the application developer performs a pre-deployment step. In this stage, Marea can swap out all possible classes and methods. Then the user runs the tests of the application causing the swap in of the needed graphs. Finally, the user saves a snapshot of the runtime and he deploys it together with the swapped graphs. With this scenario Marea releases memory and there is low overhead at runtime because most of the needed graphs were already swapped in during the pre-deploy step.

Application (swap unit)	**Size (MB) before swapping**	**Objects before swapping**	**Size (MB) after swapping**	**Objects after swapping**	**Size (MB) after experiments**	**Objects after experiments**	**Average objects per graph**	**Average % of shared objects per graph**	**Gain**
DBXTalk (class)	21.7	460801	10.6	254696	13.8	317107	170	17%	36.4%
DBXTalk (package)	21.7	460801	16.8	327594	19.7	386907	1340	24%	9.2%
Moose (class)	82.9	2899393	13.2	408549	63.7	2349701	670	15%	23.2%
DrGeo2 (class)	10.2	307131	6.9	174221	7.7	193086	232	17%	24.5%
Pharo (class)	10.7	233730	6.9	171460	7.2	153835	205	16%	32.7%

Table 1 – Primary memory footprints showing the benefit of using Marea.

Results. The resulting runtimes after the swapping were working correctly and all examples behaved normally. Table 1 shows that, after having navigated and used the applications, the amount of released memory was between 23% and 36% of the original memory footprint.

To explain the columns of Table 1, we consider the first line. "DBXTalk (class)" means that we are benchmarking the application DBXTalk and that we consider classes as roots. The table then presents the amount of memory used (in MB) and the total amount of objects before and after performing the swapping. The memory occupied by the code of Marea itself is not included in the measurement of "before swapping."

After, it shows the same information but after having run the experiment (using the application). In this case, the values are higher than "after swapping" because, during the experiment, there were graphs swapped in. Moreover, there is memory occupied by internal data structures and proxies. At the same time, these values are smaller than "before swapping" meaning that we are actually releasing primary memory. The last columns give information about the size of the graph and the amount of shared objects. Finally, the table presents the percentage of memory gain between the original scenario (before swapping) and the last scenario (after swapping and experiments).

- With DBXTalk, when considering classes as graphs, the runtime was 36.4% smaller than the original one which was already compacted and cleaned for production.
- In the Moose case, the memory was reduced up to 23.2%. The average number of objects per graph (670) is bigger than the previous example. This is because Moose handles models that are rather large. The graph of a model can include from few hundred thousands of objects to a couple of millions of objects. In this example, we have run *all* visualizations and *all* tools provided by Moose, which is not always needed by all users. Using less of them would cause less swapping in and result into more memory released. Therefore, 23.2% is the minimum gain to expect in the worst case scenario.

- In the case of Dr. Geo we reduced 24.5% of the runtime, which is significant knowing that the developers already compacted Dr. Geo as much as they could.
- Regarding Pharo's infrastructure we were able to gain 32.7% of memory. The actions performed on Pharo were to start it, browse some classes, create a new class and add some methods to it. This analysis shows that Pharo's environment can still be reduced.

Measuring Internal Data. Besides measuring the amount of memory released as part of the swapping, it is also interesting to know how much of the resulted memory is occupied by proxies and other internal data structures of Marea. To answer that question, we evaluated some of the applications.

Application (swap unit)	**Memory released (MB) after experiments**	**Proxies**	**Proxies memory (bytes)**	**Internal data structures (bytes)**	**Total internal (bytes)**	**Total internal / memory released**	**Total internal / size after experiments**
DBXTalk (class)	7.89	7619	135020	377392	512412	6.08%	3.55%
DrGeo2 (class)	7.7	4599	64608	126852	191460	7.23%	2.34%

Table 2 – Measuring Internal Data.

To explain the columns of Table 2, we consider the first line. In this case, "Memory released (MB) after experiments" is 21.7 (size before swapping) - 13.8 (size after experiments) = 7.89. The next columns are self-explanatory. The table shows that on average the internal data structures of Marea and the proxies themselves, represent only between 6.08% and 7.23% of the released memory and between of the 2.34% and 3.55% of the resulting memory.

Classes vs. Packages as Roots. Part of our experiments was to compare the impact of considering packages as the roots of the graph rather than classes. We report here only the experiments with DBXTalk since the results with others applications were similar. As we can see in the first two rows of Table 1, the gain when using classes as swap unit is much bigger than with packages. The main reason is that considering a package as root involves larger object graphs. The average number of objects per graph is 1340 with packages while it is 170 with classes. Then as soon as one of these objects is needed, the whole graph is swapped in. One conclusion from this experiment is that deciding which graphs are chosen to swap is important and directly impacts on the results. Another conclusion is that classes are a good default candidate.

Analyzing Shared Objects. If we only consider experiments with classes as roots, shared objects represent between 15% and 17% of each graph. This means that the compaction of GraphTable is worthy since 83% to 85% is full of nils. In the example of DBXTalk, the compaction results in a reduction of memory footprint by 1MB which is already subtracted from the 13.8MB mentioned in Table 1.

Understanding Memory Savings. Table 3 describes with more details the actual objects swapped out. We distinguished between classes, methods and plain instances. The analysis shows that the results on average are approximately the same.

Scenario	Graphs	Total objects	Plain objects	Classes	Methods
Swap out	3565	596339	543650	3627	49062
Swap in	402	109330	100317	464	8549
% saved	88.7%	81.6%	81.5%	87.2%	82.5%

Table 3 – Understanding memory savings of the DBXTalk example.

The "% saved" item shows the difference between the original runtime of DBXTalk and the final one (after swapping out and having used the application). The explanation lies in the fact that a lot of classes (and their instances) are not used in this web application. This table also shows that the percentage saved is quite similar (between 80% and 90%) for all the measured items, *i.e.,* the number of graphs, the total objects, plain object, classes and methods. It makes sense since the number of methods per class is linear and the average smoothes the outliers. What this analysis shows is that the execution of this web application uses only a limited set of the available classes. The rest of the applications showed similar results.

Conclusions. One conclusion we got from these benchmarks is that, even if the runtimes were small, none of the applications uses 100% of them. Furthermore, different applications need different parts (classes and libraries) of the environment.

Our experiments prove that Marea significantly decreases primary memory consumption. To gain the described 23% to 36%, all we needed was a few lines of code that simply swapped out as much as possible. No analysis was required. When swapped out code was actually needed by applications, it was just swapped in allowing applications to run smoothly. Nevertheless, as we saw with the experiment of considering packages as roots, the graphs chosen to swap directly impacts on the results. Hence, with certain knowledge in the domain, the developer may be able to choose graphs that may lead to better results.

8.3 Swapping Out Data

Marea can swap any type of object graph, not necessary those related to code. Measuring the efficiency of data swapping is difficult to assess without building applications like GIMP[13] that require a lot of data. We performed an experiment showing that Marea supports such a scenario by using it to swap graphs of plain objects.

Moose Example. Depending on what is being analyzed, Moose models can be quite large *e.g.,* 2 million objects. Because of this, the final user is limited regarding the number of models loaded at the same time. If several models are opened, Moose uses a very large amount of memory. However, at a given point in time, one analyzes only a subset of models.

Our solution is to use Marea to automatically swap out unused models and then automatically swap them in if needed by the user. In our experiments, we created three different models for different projects: Networking, Morphic and Marea itself. Each model (instance of MooseModel) was considered as root when swapping. Table 4 presents the results.

Results. By using Marea, we were able to automatically manage different models while only leaving in primary memory those models we wanted to analyze at a particular moment in time. An average Moose model between 300.000 and 600.000 objects took between 12 and

[13]Gimp is an image retouching and editing tool that implements its own memory management.

Model	Objects	Swapping out time (ms)	Swapping in time (ms)
Morphic	2102672	170202	5360
Network	539127	22751	1476
Marea	323138	12127	835

Table 4 – Swapping different large Moose models.

22 seconds to swap out and approximately 1 second to swap in. Considering the size of the experimented graphs we conclude that Marea can be used by applications to automatically swap data. The swapping out of large graphs is expensive regarding execution time. On the Moose example, those 12 to 22 seconds to swap out a large model have to be compared with alternatives such as reconstructing the whole model each time or just writing it on disk and re-reading it. Our conclusion is that Marea's overhead is not significant compared to alternatives while it brings a lot of benefits to the application developers.

8.4 Measuring Unused Objects

We gathered statistics about the memory consumption and objects usage [MPBD+10b] based on a Pharo VM that we modified to support the identification of unused objects. For each experiment, we start the analysis, we use the system for a while, we stop the analysis and finally collect the results. For each application we got the following information: the percentage of used and unused objects and the percentage of memory that used and unused objects represent. The results are presented in Table 5.

App.	% Used objects	% Unused objects	% Used objects memory	% Unused objects memory
DBXTalk	19%	81%	29%	71%
Moose	18%	82%	27%	73%
DrGeo	23%	77%	46%	54%
Pharo	10%	90%	37%	63%

Table 5 – Measuring used and unused objects.

Table 5 shows that, after having navigated all the pages of the DBXTalk website and doing our best to cover all its functionalities, only 19% of the objects were used representing 29% of the runtime memory size.

For Moose, we got that 18% of the objects were used by our experiment. This makes sense because Moose is a really large suite of tools that provides many visualizations and integration with multiple programming languages. In our case, we just imported Smalltalk projects and we run all the visualizations and tools.

Dr. Geo example demonstrates that its runtime is already reduced and that is the reason why its percentage of used objects is bigger than the other examples.

In the Pharo infrastructure example, only 10% of the objects were used representing 37% of the runtime memory size. Why do 10% of the objects represent 37% of the memory? This is because the runtime includes all the bitmaps to render the environment. There are a few bitmaps (312 in our example) but each of them is large in memory consumption. Those 312 bitmaps occupy 4.8 MB which represent almost half of the consumed memory.

Conclusion. Using Marea we were able to decrease the amount of memory used. However, the gained memory does not match the expected percentage of unused object previously analyzed. For example, in DBXTalk, we could save 36.4% of the memory even though we previously measured in our experiments that 71% was unused. One reason is the granularity

of the swapping unit. Depending on the chosen graphs, there may be several unused objects that are swapped in. Another reason is the fact that we did a "blind swapping", *i.e.,* we swapped as much as we could taking classes as roots.

The conclusion from this experiment is that there is still room for improvement and that with certain knowledge in the domain the memory released can be even bigger.

8.5 Speed Analysis

So far, we have only benchmarked memory consumption. In this section, we also measure the speed to swap out and swap in objects.

Objects per graph	**Swapping out time (ms)**	**Swapping out time (ms) / object**	**Swapping in time (ms)**	**Swapping in time (ms) / object**
51	40	0.7	37	0.7
236	47	0.2	44	0.2
777	39	0.05	41	0.05
5758	130	0.02	50	0.008
21753	256	0.01	122	0.005

Table 6 – Measuring swapping time.

Absolute Runtime Overhead. In Section 8.2, we saw that when considering classes as roots, the average number of objects per graph was between 170 and 670. Based on measurements provided by Table 6, we can conclude that swapping out an average class takes approximately 40 milliseconds, which is negligible most of the cases. Another characteristic is that the graph size does not significantly impact the swapping time. Section 10.2 explains that the object replacement used to replace objects with proxies is slow in Pharo. It takes, on average, 60% of the swapping time and it does not change much with the size of the graph. We can observe this with the columns that show the swapping time per object.

This benchmark also demonstrates that the swapping in is faster than the swapping out. This is mostly because of the serializer's performance whose materialization (deserialization) is faster than the serialization.

From the user of the application the swapping in is unnoticeable with the graphs we used. Therefore, the experiences show that we can swap out advantageously classes and reload them on use without significantly performance penalties for the user.

Relative Runtime Overhead. Previously, we measured the absolute time to swap out and in object graphs of different sizes. How much relative overhead Marea adds to a running application is a difficult question because the overhead is not constant. The overhead depends on how many objects graphs are being swapped. That being said, we took the example of the DBXTalk website and we analyzed the relative overhead.

Table 7 presents the results of the experiment. Each action is a row and they were executed in order from top to bottom. For each action we measured the original time (without swapping) to perform it and then we measured the time to perform the action considering that we have already swapped out. Finally, we measured the percentage of overhead.

Section 10.2 explains that the object replacement in Pharo is slow and it takes between 30% and 70% of the swapping time. Other Smalltalk dialects have already solved this and provide an object replacement with negligible overhead. Therefore, we also measure the swapping time without considering the time for the object replacement.

Action	Original time (ms)	Time after swapping (ms)	% time for object replacement	Swap ins	% of overhead	% of overhead without obj. repl.
Start app.	2	202	72.3%	13	101x	28x
/home	50	4298	59.5%	201	85.96x	34.78x
/tools	21	60	40%	2	2.85x	1.71x
/doc	27	100	54%	4	3.7x	1.7x
/download	27	41	31.7%	1	1.51x	1.03x
/support	18	16	0%	0	0x	0x
/news	37	95	40%	3	2.56x	1.54x

Table 7 – Relative overhead to the DBXTalk website.

The results show that when the web application starts, some graphs are swapped in, causing overhead. Second action, *i.e.,* accessing the /home page, has a significant overhead because several graphs are swapped in. That means that for processing a request in the web application, several classes are needed. The next pages start decreasing the numbers of swap ins and hence of overhead. This is because each of these pages already has almost everything it needs in memory. It continues this way, until the /support page does has no swap in. Finally, /news has more swap ins, and this is because this page provides a blog and therefore it needs some special classes regarding blogs that were not needed so far.

Notice that the overhead of Marea is constant and independent of the time to execute the original task. For example, a request to /home takes 50 ms without Marea and 4289 with it. Now, consider another action that would originally take 2000 ms but results in the same amount of swap ins. In this case, the overhead of Marea is the same (4298-50) so the total time will be 2000 + 4298 - 50, which is 3.12x slower.

The last experiment we did to measure overhead is to run all the tests of the application with and without swapping. Without swapping, it took 2333 ms to run 726 tests. After having swapped out, it took 16286 ms, and 597 graphs were swapped in. That means an overhead of 6.98x. If we do not count object replacement, it is 2.85x.

It is also important to note that in all these experiments we are always measuring the worst case scenario, *i.e.,* when all needed graphs were already swapped out. In normal use, it is likely that some of the graphs are already in memory, and hence the overhead is smaller.

9 Implementation

Marea is fully implemented[14] in Pharo, an open-source Smalltalk-inspired programming language [BDN+09]. Marea is open-source and developed under the MIT license[15]. It relies on Ghost (an advanced proxy toolbox [MPBD+11, MPBD+12]) and Fuel (a fast serializer [DMPDA11]). In this section, we explain Marea's design and its requirements from the programming language.

9.1 Marea Design

Marea is built with a clean object-oriented design and *without any change to the virtual machine* making it easy to understand, maintain and extend. Marea consists of approximately

[14]The website of the project with its documentation is at: http://rmod.lille.inria.fr/web/pier/software/Marea. The source code is available in the SqueakSource3 server: http://ss3.gemstone.com/ss/Marea.html

[15]http://www.opensource.org/licenses/mit-license.php

2000 lines of code across 35 classes. The average number of methods per class is 9 while the average lines of code per method is 6 conforming to Smalltalk standards [KST96]. The 80 unit tests that cover all Marea's use cases are implemented in approximately 1900 lines of code, which is almost as long as Marea's implementation. Figure 7 shows a simplified UML class diagram of Marea's design.

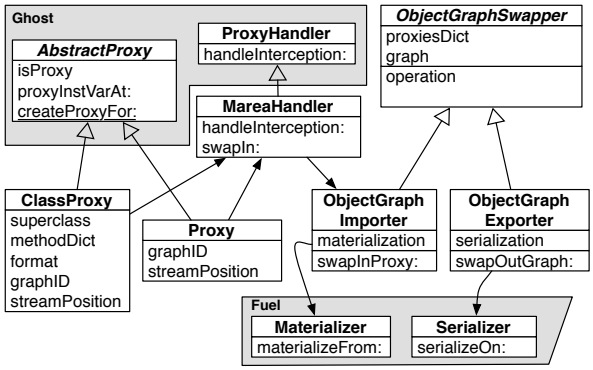


Figure 7 – Simplified UML class diagram of Marea.

Its key classes are:

- ObjectGraphExporter is the entry point for the final user. It provides methods to swap out graph of objects and relies on Fuel for the serialization. The algorithm is the one explained in Section 6. Similarly, ObjectGraphImporter implements the algorithm to swap in and its input is a proxy. It relies on Fuel for the materialization.

- Proxy and ClassProxy are concrete subclasses of Ghost's AbstractProxy. Their only purpose is to intercept messages to trigger the swapping in. ClassProxy is needed because we want to proxy classes themselves and the VM expects certain shape for classes. Once a message is intercepted, it is forwarded to its associated handler which is MareaHandler in our case. There are also certain special messages that the proxies implement and answer themselves rather than intercepting *e.g.,* isProxy or proxyInstVarAt:.

- MareaHandler is a concrete subclass of Ghost's ProxyHandler whose main goal is to manage interceptions. An interception is a reification of a message that was sent to a proxy and intercepted. Each interception contains everything needed to swap in the receiver of the intercepted invocation *i.e.,* the proxy that forwarded the interception, the receiver and the arguments. MareaHandler triggers the swap in of the graph associated with the proxy (by delegating to the ObjectGraphImporter and passing the proxy as parameter) that forwarded the interception and then it forwards to the just swapped-in object the message intercepted by the proxy.

9.2 Required Reflective Entry Points

Although implementing a whole OGS usually requires development on the virtual machine side or even at the OS level, our implementation relies on the default Pharo VM. Nevertheless, Marea takes advantage of the following reflective features and hooks provided by the VM and the language.

- *Object replacement:* the primitive Object»become: anotherObject atomically swaps the references of the receiver and the argument. All references in the entire system that

used to point to the receiver now point to the argument and vice versa. There is also becomeForward: anotherObject which is only one way. This feature enables us to replace target objects with proxies and vice versa.

- *Objects as methods:* Pharo lets us replace a method in a method dictionary with an object that is not an instance of CompiledMethod. While performing the method lookup, the VM detects that the object in the method dictionary is not a method so it sends the message run: aSelector with: arguments in: aReceiver to that object. Therefore, by handling run:with:in:, Ghost can even intercept method execution.

- *Class with no method dictionary:* the method dictionary is stored as an instance variable of a class, hence it can be changed. When an object receives a message and the VM does the method lookup, if the method dictionary of the receiver class (or of any other class in the hierarchy chain) is nil, then the VM directly sends the message cannotInterpret: aMessage to the receiver. However, the lookup for method cannotInterpret: starts in the *superclass* of the class whose method dictionary was nil. This hook allows Ghost to create proxies that intercept almost all possible messages.

"Objects as methods" is optional and it is just for intercepting method execution. "Class with no method dictionary" is used for implementing proxies that intercept all messages but other languages may have other ways of implementing proxies. What Marea needs is a proxy toolbox that is able to intercept all messages. Besides intercepting messages, the proxy toolbox should be able to intercept instance variable accessing. In the case of Pharo this is not a problem because instance variables are private to the instance holding them and the only way of accessing them from outside the instance is by sending a message to it.

The implementation details of a proxy library depend on the underlying language. For example, contrary to dynamic languages where a proxy needs to intercept all messages, in static languages it only needs to intercept those messages defined by the type of the object it is proxifying.

For instance, Java supports Java Dynamic Proxies but they need that at creation time the user provides a list of *Java interfaces* for capturing the appropriate messages. Later, more powerful proxies were created that are able to proxify classes and not only interfaces [Eug06]. Java Dynamic Proxies basically work by dynamically creating classes that implement the provided interfaces. In addition, it dynamically generates all the methods defined by the interfaces in a way that they invoke the proxy handler. Finally, intercepting instance variables access can be done with bytecode instrumentation, *e.g.,* replacing field accesses to invocations of getter/setter methods that are automatically generated for classes [Eug06].

Besides the proxy toolbox, to implement Marea in another language the other needed feature is "Object replacement." To our knowledge, Smalltalk is the only programming language that provides such a feature at the language side. Nevertheless, we believe that it is not that complex to implement it in different object-oriented languages. The reason is that several of them include a moving garbage collector, and if the GC moves objects around, then it needs to update pointers. Such task is the most significant and important part of the become: primitive. Moreover, if the VM is based on an object table instead of on direct pointers, the implementation of become: is even easier because it just means swapping two pointers.

Since Smalltalk provides become: at the language level and does not keep it hidden in the VM, we were able to implement Marea without any VM changes. In a language with a moving GC (such as the default Java collector) or based on object tables, implementing Marea would require to modify the VM either to provide a become primitive at the language level or to implement the whole Marea in the VM.

9.3 Using Low Memory Footprint Proxies

Marea's goal is to reduce the memory footprint of applications. However, Marea itself uses proxies, which also occupy memory. Ghost already reduces the memory footprint of proxies but we reduced it even more.

Compact Classes. We take advantage of the Pharo internal object representation: we declare our proxy classes as *compact*. In Pharo, up to 32 classes can be declared as compact classes. In a 32 bits system, compact class instances' object headers are only 4 bytes long instead of the 8 bytes that apply to instances of regular classes.

Reducing Instance Variables in Proxies. Secondly, we encode the proxy instance variables position and graphID in one unique proxyID. The proxyID is a SmallInteger which uses 15 bits for the graphID and 16 bits for the position. Since SmallInteger are immediate objects[16], we do not need an object header for the proxyID. With these optimizations, an instance of Proxy is only 8 bytes (4 for the header and 4 for the proxyID).

By using small integers for proxyIDs, we have a limit of 32767 for the graphID and 65535 for the position. Still, Marea may exceed those limits. Pharo can represent integers with an arbitrary large number of digits. However, instead of using SmallIntegers which are immediate objects, it uses instances of the class LargePositiveInteger which are plain objects and hence occupy more memory. Nevertheless, LargePositiveInteger is a compact class so its instances have a small header.

10 Discussion

10.1 Marea Evaluation over Requirements and Desiderata

The following is the evaluation of Marea over the listed requirements:

- **Efficient Object-Based Swapping Unit.** First, Marea's swapping unit consist of objects. Second, it does not swap objects individually but graphs of objects. This generates few object faults and when swapping in, it loads few unnecessary objects.
- **Uniformity.** Marea can handle all kinds of objects whether they represent code, runtime entities, application data, etc.
- **Automatic Swapping In.** As soon as a swapped object happens to be needed, it is automatically swapped in.
- **Automatic swapping out.** Currently, Marea only provides a small and simple prototype. Section 12 explains that this is the natural future work of this dissertation.
- **Transparency.** From the point of view of an application, Marea is transparent in the sense that the application will get the same results whether it is using Marea or not. From the application's development point of view, the application code is not polluted with code related to swapping. Instead, Marea's code is totally decoupled from the application. However, there is the triggering of the swapping out if the user wants to do it when an application-level event occurs.

And here over the desiderata:

[16] In Smalltalk, immediate objects are objects that are encoded in a memory address and do not need an object header. In Pharo, integers are immediate objects.

- **Cooperation with the application.** It allows application programmers to parametrize or decide what, when and how to swap. This opens up many new possibilities as we can take domain knowledge into account for the decision of what to swap.
- **Portability.** Our solution is decoupled from both, the OS and the virtual machine. However, Marea requires some entry points from the virtual machine as explained in Section 9.2.

10.2 Infrastructure-Specific Issues

While implementing Marea in Pharo, we encountered some issues that are specific to Pharo but represent recurrent problems that most of the implementations will face.

Special Objects Are Never Swapped. The swap out algorithm replaces each object of the graph with a proxy. Since the graph is specified by the end programmer, it may contain any type of object *e.g.,* system objects, which cannot be replaced without breaking the system. In Pharo, there are three kinds of objects that cannot be replaced: (1) nil, true, false, Smalltalk, Processor, etc[17]. (2) Immediate objects *e.g.,* instances of SmallInteger. (3) Instances of Symbol and Character are also special in Smalltalk because they are created uniquely and shared across the system. In addition, symbols and characters are also used in critical parts of the machinery of method execution. Marea does not support instances of Process either since they require special management.

Marea handles this problem by systematically checking the objects to proxify. Special objects are not proxified and they are handled specifically during materialization. When a graph is rebuilt, Marea inserts references to the special objects that already exist in the system instead of creating duplicates.

Proxies and Primitives. Most programming languages have methods called primitives implemented in the virtual machine. For example, in Pharo, arithmetic operations, checking whether two references represent the same object or not, file and sockets management, graphics processing, etc., all end up using primitives at some point.

Primitives usually impose a shape in the receiver object or arguments. For example, they expect some objects to be an instance of a certain class or to have a specific format. The problem appears when we replace an object with a proxy that is then passed as an argument to a primitive. In Pharo, most primitives do not crash the VM in that situation but instead they notify their failure which can be managed at language level. By default, a primitive failure is handled by raising a PrimitiveFailed exception.

To be transparent for the user and avoid raising such exceptions, Marea captures the PrimitiveFailed exception. It gets the original receiver and arguments, swaps in proxies and then re-execute again the same method. Thanks to Pharo facilities, such solution is approx. 10 lines of code but it relies on the fact that primitives raise exceptions and that we can dynamically access the stack.

Another problem is if an application does object replacement. In this case, it could replace a Marea proxy, which is a problem with the current implementation. A workaround is to just throw an error in this scenario and do not perform the replacement.

Object identity could also be a problem when dealing with proxies and object replacement. However, from the execution point of view, it is not a problem. For example, given the following code:

[17] These objects are present in the specialObjectsArray and are known and directly used by the VM

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

Imagine that anObject is replaced by a proxy, *i.e.,* all objects in the system which were referring to the target (anObject), will now refer to the proxy. Since all references have been updated, ==[18] 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.

Special Proxies. Some objects can be replaced but only by proxies that respect certain shape. For example, during method lookup, the Pharo VM directly accesses some instance variables of objects representing classes. Ghost proxies solve this problem by creating special proxies for classes and methods that respect the shape needed by the VM.

Object Replacement in Pharo. The object replacement in Pharo is done by using the primitive become: which scans the whole memory to swap all references to the receiver and the argument. Additionally, Pharo provides a "bulk become" that replaces multiple objects at the same time. Marea uses a bulk become to convert all the proxies of a graph. That way, we pay the memory traversal only once.

While performing the benchmarks described in Section 8, we measured the time of the bulk become for each graph and we calculated which percentage of the total swapping time it represents. We find out that, on average, this primitive takes about 60% of the total swapping time.

This full memory traversal is not needed by Marea itself and it is actually a current limitation of Pharo. In fact, other Smalltalk dialects provide a fast become *e.g.,* VisualWorks and GemStone. This means that if the language provides a fast become, all the overhead measured in Section 8 will decrease %60, *i.e.,* Marea will be %60 faster.

10.3 Messages That Swap In Lots of Objects

Having classes and methods as first-class objects offers solid reflective capabilities. However, some system queries access *all* classes or methods in the system, that may cause the swap in of many of the swapped out graphs. A typical example is when a programmer asks for all the senders of a certain message. The system sends messages to all objects that are in the method dictionaries of classes causing the swap in if they happen to be proxies. This scenario and most of the similar ones happen during application development. As Marea is intended to reduce memory for deployed applications, this is not usually a problem. Still, we identified the following possible solutions.

Explicit Verification. This solution was introduced by ImageSegment (an object swapper developed by D. Ingalls for Squeak [IKM+97]). It modifies a large amount of queries of the base system to explicitly check which objects are in memory and only perform actions on them. The implementation is simple: one method isInMemory defined in Proxy returns false and one method in Object returns true. In other cases, ImageSegment swaps in the graph, sends the message to the objects that were just swapped in and then it swaps them out again. An example of this is when a class changes its shape and its swapped instances need to be updated. This type of solution raises the question of the transparency of an OGS.

[18] In Pharo the message == answers whether the receiver and the argument are the same.

Proxy Adaption. This solution requires to have certain messages handled by the proxy itself instead of forwarding it to the handler (which will swap in the graph). The proxy plays the role of a cache by keeping certain information of the proxified object. This is the solution chosen by Marea. For example, we use it for the case of class proxies. In Pharo, it is common that the system simply selects classes by sending messages such as isBehavior, isClassSide, isInstanceSide, instanceSide or isMeta. Therefore, we defined such methods in ClassProxy to answer appropriate results, *i.e.,*isBehavior and isInstanceSide answers true, isClassSide and isMeta answers false and instanceSide answers self. This way, we avoid swapping in classes.

We also applied this solution to metaclasses and traits. But, it generalizes to any kind of object. Developers can create specific proxy classes for the desired type of object. Nevertheless, such an approach is limited to return simple values and it is complemented by the following one.

Using Proxies as Caches. Another possible solution, which is an improvement of the previous one, is to make proxies cache some values from the proxified objects. For instance, a class proxy can cache the class name, a method proxy can cache the literals of the proxified method. There is a trade-off here between sizes of proxies and proxified objects. It is often worthwhile to have larger proxies if they avoid swapping in large objects or objects that are roots of relatively large object subgraph.

11 Related Work

11.1 Virtual Memory and Garbage Collected Languages

While offering numerous advantages regarding software engineering, garbage collectors do not interact well with the OS's virtual memory. The reason is that a full GC traverses all the objects and that can lead to page reloads causing so-called thrashing [HFB05]. Marea is not intended to solve this problem. However, since it reduces the number of objects loaded in primary memory, it also may reduce the number of pages and thus, the OS's thrashing.

LOOM. In the eighties, LOOM [Kae86] (Large Object-Oriented Memory) implemented a virtual memory for Smalltalk-80. It defined a swapping mechanism between primary and secondary memory. The main downfall of LOOM was that its swapping unit was too small as it was only one object. 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[19]. Finally, LOOM entails big changes to the virtual machine.

CRAMM. CRAMM (Cooperative Robust Automatic Memory Management) [YBK+06] consists of two parts: the virtual memory system and the heap sizing model. The former gathers information about the process being executed and the latter dynamically chooses the

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

optimal heap size that allows the system to maintain high performance and avoid thrashing [HFB05]. However, this approach as well as LOOM are independent from application domains. Developers have no way to give the system hints about application specifics.

11.2 Orthogonal Persistence and Object Databases

Even if orthogonally persistent systems [MBMZ00, HC99, AM95, HMB90] and object databases [BOS91] may look similar to Marea, they have several differences. The most important one is that their goal is to automatically *persist* a graph of objects into a non-volatile memory. They do not *swap* graphs and, therefore, they do not take into account shared objects. Besides, with object databases, objects live permanently in secondary memory and are temporally loaded into primary memory when needed. In Marea, objects live in primary memory and they are swapped out when they are not needed or when the users decide.

11.3 Memory Leaks

Melt [BM08] implements a tolerance approach that eliminates performance degradations and crashes due to leaks of dead but reachable objects ("stale objects"). The strategy is to have sufficient disk space to hold leaking objects. Melt identifies "stale objects" and swaps them out to disk. If they are then needed, they are brought back into primary memory.

One difference with Marea is that the granularity used by Melt is that of objects rather than graphs. In addition, Melt adds a few words for (1) each pointer from in-use to stale and (2) each in-use object pointed to by the stale space. Melt hopes that the number of pointers from in-use to stale should be relatively small (not nearly as big as the number of stale objects). In other words, Melt assumes that most (or even the vast majority of) stale objects will not be referenced directly by an in-use pointer. That is, presumably many stale objects are only pointed to by other stale objects. If the reality is different than the assumption, then Melt is not performant.

Memory leaks are different from general graphs, hence In Marea we cannot do such assumptions and we want to support the case where there are several in-use objects referencing stale objects. In Melt, the idea is that a graph that was swapped out will not be probably needed again since it is likely to be a memory leak. It Marea it is more frequent that a swapped out graph is later swapped in. Finally, Melt supports automatic swap out.

11.4 General-Purpose Object Graph Swappers

ImageSegment is an object swapper and serializer for Squeak Smalltalk developed by D. Ingalls [IKM+97]. To detect *shared objects* it does a full memory traversal using the garbage collector infrastructure [MPBD+10a]. *Shared objects* are not swapped out. Since users have no mean to estimate the amount of shared object given a graph, ImageSegment ends up releasing less memory than Marea.

12 Conclusion and Future Work

In this paper, we studied the problems that appear when building an efficient application-level object graph swapper for object-oriented programming languages. We introduced Marea, a solution to swap object graphs between primary and secondary memory. We explained the challenges and algorithms to provide fast swapping and the necessary subsystems.

Contrary to existing alternatives, Marea enables application developers to decide which object graphs to swap. Marea stores them in a secondary memory and replaces them with

a few proxies that act as triggers to automatically swap in needed objects. Our proposal discusses potential issues and how to handle them. This includes situations where different graphs sharing some objects are swapped in and out separately in an arbitrary order.

Our implementation of Marea demonstrates that we can build a fast application-level OGS. Thanks to some functionalities of the Pharo virtual machine like "object replacement" we were able to implement Marea without modifying the VM yet with a clean object-oriented design. It also allowed us to validate empirically our solution by experimenting Marea with different real-world applications. We have focused on measuring the efficiency and usefulness of it. Our benchmarks demonstrate that the memory footprint of different applications can be reduced from 23% to 36%.

A natural follow up to this work is to automate the process of swapping out graphs and to build an automatic system for freeing main memory on top of Marea. Right now we have a simple prototype that every certain amount of time randomly selects objects and checks if those objects are unused. If an object is unused, then Marea continues analyzing its transitive closure (subgraph). If all the objects of the subgraph are unused, Marea swaps out the graph. In this prototype each object has a flag to mark it as unused and with a customized virtual machine, we turn on the flag when the object is used. Every certain period of time, the flags of all objects are reset.

The challenge to implement a better automatic swapping out strategy is to detect graphs of *unused yet referenced objects* which are good candidates to be swapped out. First and foremost, we need to answer the question: how to identify unused objects? After using an object (*e.g.,* message reception), how long should the system wait before considering it as unused? Besides, different criteria should be considered to select graphs to swap out such as: the graph's size, the percentage of unused objects or the percentage of objects shared with other graphs.

Another interesting direction to explore is the relationship between object graph swapping and garbage collection. Could they take advantage one of the other? Can they reuse the same memory traversal? Can they share the information or flags stored in object headers?

One of the current limitations of Marea is when a graph changes while being swapped out. All our algorithms to swap and the serialization do not work properly when the graph changes in the middle of the operation. This is because Marea is implemented at the language side and therefore it is not atomic. We plan to study this problem and analyze possible solutions.

Acknowledgments We thank Igor Stasenko for his help with the discussions and development of Marea, Martin Dias for his work on Fuel serializer and Esteban Lorenzano for his help with the experiments and benchmarks with Seaside and Pier.

References

[AM95] Malcolm Atkinson and Ronald Morrison. Orthogonally persistent object systems. *VLDB JOURNAL*, 4:319–401, 1995.

[BDN+09] Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz, Damien Pollet, Damien Cassou, and Marcus Denker. *Pharo by Example*. Square Bracket Associates, Kehrsatz, Switzerland, 2009. URL: http://pharobyexample.org/.

[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, October 19-23, 2008, Nashville, TN, USA*, pages 109–126. ACM, 2008. URL: http://doi.acm.org/10.1145/1449764.1449774.

[BOS91] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Commun. ACM*, 34(10):64–77, 1991. doi:10.1145/125223.125254.

[BSS+11] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM. doi:10.1145/1985793.1985827.

[CGV10] Alexandre Courbot, Gilles Grimaud, and Jean-Jacques Vandewalle. Efficient off-board deployment and customization of virtual machine-based embedded systems. *ACM Transaction on Embedded Computer Systems*, 9:21:1–21:53, mar 2010. doi:10.1145/1698772.1698779.

[CH81] Richard W. Carr and John L. Hennessy. WSCLOCK a simple and effective algorithm for virtual memory management. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 87–95, New York, NY, USA, 1981. ACM. doi:10.1145/800216.806596.

[CO72] Wesley W. Chu and Holger Opderbeck. The page fault frequency replacement algorithm. In *Proceedings of the December 5-7, 1972, fall joint computer conference, part I*, AFIPS '72 (Fall, part I), pages 597–609, New York, NY, USA, 1972. ACM. doi:10.1145/1479992.1480077.

[Den70] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3):153–189, September 1970. doi:10.1145/356571.356573.

[Den80] Peter Denning. Working sets past and present. *IEEE Transactions on Software Engineering*, SE-6(1):64–84, January 1980. doi:10.1109/TSE.1980.230464.

[DMPDA11] 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. URL: http://rmod.lille.inria.fr/archives/workshops/Dia11a-IWST11-Fuel.pdf, doi:10.1145/2166929.2166930.

[DRS+10] Stéphane Ducasse, Lukas Renggli, C. David Shaffer, Rick Zaccone, and Michael Davies. *Dynamic Web Development with Seaside*. Square Bracket Associates, 2010. URL: http://book.seaside.st/book.

[EGK95] D.R. Engler, S.K. Gupta, and M.F. Kaashoek. AVM: Application-level virtual memory. In *Hot Topics in Operating Systems, 1995. (HotOS-V), Proceedings., Fifth Workshop on*, pages 72 –77, may 1995. doi:10.1109/HOTOS.1995.513458.

[EKO95] Dawson R. Engler, M. Frans Kaashoek, and James O'Tool. Exokernel: An operating system architecture for application-level resource management. In *SOSP*, pages 251–266, 1995. doi:10.1145/224056.224076.

[Eug06] Patrick Eugster. Uniform proxies for Java. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 139–152, New York, NY, USA, 2006. ACM. URL: http://doi.acm.org/10.1145/1167473.1167485, doi:10.1145/1167473.1167485.

[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. URL: ftp://st.cs.uiuc.edu/pub/papers/patterns/ecoop93-patterns.ps.

[HC99] Antony L. Hosking and Jiawan Chen. PM3: An orthogonal persistent systems programming language - design, implementation, performance. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 587–598, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. URL: http://dl.acm.org/citation.cfm?id=645925.671503.

[HFB05] Matthew Hertz, Yi Feng, and Emery D. Berger. Garbage collection without paging. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 143–153, New York, NY, USA, 2005. ACM. doi:10.1145/1065010.1065028.

[HMB90] Antony L. Hosking, J. E Moss, and Cynthia Bliss. Design of an object faulting persistent Smalltalk. Technical report, University of Massachusetts, Amherst, MA, USA, 1990.

[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. URL: http://www.cosc.canterbury.ac.nz/~wolfgang/cosc205/squeak.html, doi:10.1145/263700.263754.

[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. doi:10.1145/28697.28707.

[KLVA93] Keith Krueger, David Loftesness, Amin Vahdat, and Thomas Anderson. Tools for the development of application-specific virtual memory management. In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, volume 28, pages 48–64, October 1993. doi:10.1145/165854.165867.

[KST96] Edward J. Klimas, Suzanne Skublics, and David A. Thomas. *Smalltalk with Style*. Prentice-Hall, 1996.

[LL82] H. Levy and P. H. Lipman. Virtual memory management in the VAX/VMS operating system. *IEEE Computer*, 16(3):35, March 1982. doi:10.1109/MC.1982.1653971.

[MBMZ00] Alonso Marquez, Stephen M Blackburn, Gavin Mercer, and John Zigman. Implementing orthogonally persistent Java. In *In Proceedings of the Workshop on Persistent Object Systems (POS*, pages 218–232, 2000.

[MPBD+10a] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Experiments with a fast object swapper. In *Smalltalks 2010*, Concepción del Uruguay, Argentina, 2010. URL: http://rmod.lille.inria.fr/archives/workshops/Mart10b-Smalltalks2010-Swapper-ImageSegments.pdf.

[MPBD+10b] Mariano Martinez Peck, Noury Bouraqadi, Marcus Denker, Stéphane Ducasse, and Luc Fabresse. Visualizing objects and memory usage. In *Smalltalks 2010*, Concepción del Uruguay, Argentina, 2010. URL: http://rmod.lille.inria.fr/archives/workshops/Mart10a-Smalltalks2010-VisualizingUnusedObjects.pdf.

[MPBD+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'11)*, Edinburgh, Scotland, 2011. URL: http://rmod.lille.inria.fr/archives/workshops/Mart11a-IWST11-Ghost.pdf, doi:10.1145/2166929.2166937.

[MPBD+12] Mariano Martinez Peck, Noury Bouraqadi, Stéphane Ducasse, Luc Fabresse, and Marcus Denker. Ghost: A uniform and general-purpose proxy implementation (submitted + passed first review round). *Journal of Science of Computer Programming (SCP)*, October 2012.

[NDG05] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. The story of Moose: an agile reengineering environment. In Michel Wermelinger and Harald Gall, editors, *Proceedings of the European Software Engineering Conference*, ESEC/FSE'05, pages 1–10, New York NY, 2005. ACM Press. Invited paper. URL: http://scg.unibe.ch/archive/papers/Nier05cStoryOfMoose.pdf, doi:10.1145/1095430.1081707.

[Sta82] James William Stamos. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Technical Report SCG-82-2, Xerox Palo Alto Research Center, Palo Alto, California, may 1982.

[Ung84] Dave Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19(5):157–167, 1984. doi:10.1145/390011.808261.

[Ung95] David Ungar. Annotating objects for transport to other worlds. In *Proceedings of the tenth annual conference on Object-oriented programming systems, languages, and applications*, OOPSLA '95, pages 73–87, New York, NY, USA, 1995. ACM. doi:10.1145/217838.217845.

[WWH87] Ifor Williams, Mario Wolczko, and Trevor Hopkins. Dynamic grouping in an object-oriented virtual memory hierarchy. In J. Bézivin, J-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings ECOOP '87*, volume 276 of *LNCS*, pages 79–88, Paris, France, June 1987. Springer-Verlag.

[YBK+06] Ting Yang, Emery D. Berger, Scott F. Kaplan, J. Eliot, and B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *In USENIX Symposium on Operating Systems Design and Implementation*, pages 103–116, 2006.