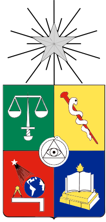


UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

DISCOVERING MEMORY OPTIMIZATION OPPORTUNITIES BY ANALYZING SHAREABLE OBJECTS

TESIS PARA OPTAR AL GRADO DE MAGÍSTER EN CIENCIAS, MENCIÓN COMPUTACIÓN

MEMORIA PARA OPTAR AL TÍTULO DE INGENIERO CIVIL EN COMPUTACIÓN

ALEJANDRO JOSÉ INFANTE RICA

PROFESOR GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
LUIS MATEU BRULÉ
ÉRIC TANTER
PAUL LEGER MORALES

Este trabajo ha sido parcialmente financiado por
CONICYT-PCHA/Magíster-Nacional/2015-22150809

SANTIAGO DE CHILE
2017

Resumen

Los lenguajes modernos de programación orientada a objetos han aliviado de manera importante a los programadores la tarea de administrar memoria. A pesar de la eficiencia de los recolectores de basura y herramientas de análisis de programas en tiempo real, aún existe una porción importante de memoria siendo desaprovechada.

El desaprovechamiento de memoria en software posee graves consecuencias, incluyendo frecuentes interrupciones en la ejecución debido a la presión ejercida sobre el recolector de basura y el uso ineficiente de dependencias entre objetos.

Hemos descubierto que supervisar los lugares de producción de objetos y la equivalencia de los objetos producidos es clave para identificar ineficiencias causadas por objetos redundantes. Hemos implementado optimizaciones para reducir el consumo de memoria de seis aplicaciones industriales, obteniendo una reducción superior al 40% en el uso de memoria en la mitad de las aplicaciones sin poseer conocimiento previo de las mismas.

Nuestros resultados replican parcialmente los resultados obtenidos por Marinov y O'Callahan y exploran nuevas formas de identificar objetos redundantes.

Abstract

Modern object-oriented programming languages have greatly alleviated the memory management for programmers. Despite the efficiency of garbage collection and Just-In-Time program analyzes, memory still remains prone to be wasted.

A bloated memory may have severe consequences, including frequent execution lags due to a high pressure on the garbage collector and suboptimal object dependencies.

We found that dynamically monitoring object production sites and the equivalence of the produced objects is key to identify wasted memory consumption caused by redundant objects. We implemented optimizations for reducing the memory consumption of six applications, achieving a reduction over 40% in half of the applications without having any prior knowledge of these applications.

Our results partially replicate the results obtained by Marinov and O'Callahan and explore new ways to identify redundant objects.

Acknowledgements

I would first like to thank my thesis advisor Prof. Alexandre Bergel of the DCC at Universidad de Chile. He has encouraged me to do research since 2013 and supported me all the way through. He consistently pushed me to reach new heights and helped me whenever I needed. I deeply appreciate all he has done, without the support of Prof. Bergel I could not have been successful on this.

I would also like to thank all members of SCG group for receiving me for 6 weeks in Bern in 2015. I appreciate all of you as college researchers, but also as friends, who warmly received me as one of yours. I would like to specially thank Prof. Oscar Nierstrasz for inviting me to Bern and including me in SCG group.

I would also like to thank the people of Lam Research for kindly receiving me as an intern in 2014 and supporting me through the last years. I would like to specially thank Chris Thorgrimsson and C.H. from Lam for giving me that amazing opportunity.

I would also like to thank all the people of Inria Chile for kindly receiving me at your office meanwhile I was working on my thesis. I deeply appreciate that you have included me as part of your group and given me your friendship.

I would also like to thank Renato Cerro from Universidad de Chile for helping me to improve the english writing of my thesis.

I would also like to thank CONICYT-PCHA/Magíster-Nacional/2015-22150809 program for supporting me during my masters program.

Finally, but not least, I would like to thank all members of my family, my friends and my girlfriend Tere for being with me during this stage of my life. Without all of you, I would have not been strong enough to accomplish this.

Contents

1 Introduction **1**

2 Object Shareability **6**
2.1 Overview . . . 6
2.2 Shareability relation . . . 7
2.2.1 Definition . . . 7
2.2.2 Properties . . . 8
2.3 Redundant objects and groups . . . 9
2.4 Shareability Oracle . . . 10

3 Object Equivalence **11**
3.1 Intuition . . . 11
3.2 Object Equivalence . . . 12
3.3 Equivalence example . . . 14
3.4 Properties . . . 15
3.5 Equivalent object group . . . 15

4 Experiment design **16**
4.1 Pharo object model . . . 16
4.2 Object characterization . . . 16
4.3 System specific objects . . . 17
4.4 Experiment . . . 18
4.5 Metrics . . . 19
4.6 Benchmarks . . . 20
4.7 Summary . . . 21

5 Results **22**
5.1 Metrics analysis . . . 22
5.2 Avoiding equivalent object creation . . . 24
5.3 Summary . . . 27

6 Solving object redundancy **29**
6.1 Tool report and use strategy . . . 29
6.1.1 Large equivalent object groups . . . 30
6.1.2 Shareable object groups with shared allocation context . . . 30
6.2 Classification of optimizations . . . 31
6.2.1 Single object cache (5 implementations) . . . 31

6.2.2 Map of objects cache (2 implementations) 33
6.2.3 Weak map of objects cache (5 implementations) 35
6.2.4 Others (2 implementations) 36
6.3 Optimizations performed 38

7 Feedback from developers **40**
7.1 Roassal Case 40
7.2 NeoJSON and NeoCSV Case 42
7.3 Summary 44

8 Implementation **45**
8.1 Profiling Technique 45
8.1.1 Events and strategies for data collection 46
8.1.2 Architecture of the profiler 47
8.1.3 Unique id specification 47
8.2 Object graph replication 49
8.3 Graph analysis and object equivalence evaluation 49
8.4 Implementation difficulties 51

9 Discussion **53**
9.1 Memory impact on the system 53
9.2 Partial replication of Marinov *et al.* experiment 54
9.2.1 Similarities 54
9.2.2 Differences 54
9.2.3 Results and Conclusions comparison 55
9.2.4 Summary 57
9.3 Threat to validity 57
9.4 Limitations of our approach 58
9.4.1 Limitations of Object Equivalence definition 58
9.4.2 Limitations of System specific objects 58
9.4.3 Tool design 59
9.5 Extensibility to other languages 59

10 Related work **61**

11 Conclusion **63**

Bibliography **65**

A Specification of profiler events **67**

List of Tables

4.1 Metric definitions . . . 20
4.2 Applications metrics . . . 20

5.1 Results in number of objects . . . 22
5.2 Number of equivalent objects . . . 23
5.3 Results in memory usage (KB) . . . 23
5.4 Reduction of memory footprint due to our optimizations . . . 24

6.1 Classification of implemented optimizations for the applications in benchmark suite. . . . 39
6.2 Time spent for producing optimizations for the applications in benchmark suite. 39

9.1 OEP Marinov's mergeability results . . . 56

List of Figures

2.1 Object Shareability Relation 8

3.1 Object graph of example which shows that $a1 \approx a2$, $b1 \approx b2$, and $c1 \approx c2$. . . . 14

5.1 Distribution of objects and equivalent objects in benchmarks. 24
5.2 Comparison of initial memory consumption, optimal consumption and after optimizations consumption. 25
5.3 Comparison between initial number of domain objects, optimal number of domain objects and after optimizations number of domain objects. 26
5.4 Comparison between initial number of system specific objects, optimal number of system specific objects and after optimization number of system specific objects. 26
5.5 Comparison between initial number of external objects and after optimizations number of external objects. 27

8.1 Illustration for simplified Subgraph isomorphism problem for Object Equivalence. 50

A.1 SOEvent hierarchy class diagram. 72

Chapter 1

Introduction

Memory consumption is a major concern for most non trivial software [16]. Memory is used to store the resources used in software executions, which are constantly allocated and released.

Management of memory resources is a fundamental task in software development. In object-oriented languages the memory is usually consumed by objects, which are created and destroyed continuously during the execution of a program. Manually handling the memory is known to be a complex and prone-to-error activity. An example of this are the memory-leak problems caused by not freeing the resources when they are not used anymore.

The garbage collector is a sophisticated and optimized component for removing objects from memory that are no longer necessary. The use of an automatic memory management greatly alleviates the programming activity. On the other hand, the ease of creating objects without concern for their destruction contributes to conveying the feeling that creating objects leaves a small memory footprint. Studies show that it is not the case and objects are frequently and unnecessarily created [2, 6].

To illustrate the objective of the thesis, we use a contrived but representative example. The `Address` class is part of a model that holds (i) postal code, (ii) street, (iii) number and (iv) city. It also provides two factory methods for creating Address objects, one for addresses with known postal code and one for addresses with unknown postal codes.

```
class Address {
   PostalCode postalCode;
   Street street;
   int number;
   City city;

   // Addresses with unknown postal code
   static Address createAddress(Street street, int number, City city) {
      return new Address(new PostalCode(−1), street, number, city);
   }

   // Addresses with known postal code
   static Address createAddress(PostalCode postalCode, Street street, int number, City city) {
      return new Address(postalCode, street, number, city);
```

```
  }

  private Address(PostalCode postalCode, Street street, int number, City city) {
    this.postalCode = postalCode; this.street = street; this.number = number; this.city = city; }
}
```

This implementation uses `new PostalCode(-1)` to represent addresses with unknown postal codes. With this approach, a new PostalCode object is created every time a new address with an unknown postal code is created. Unfortunately, a representative dataset revealed that most of the addresses had unknown postal codes, creating many duplicated instances of postal codes *-1*.

In this case, `PostalCode` is a *value object*, *i.e.,* a simple immutable data entity for which equality does not depend on identity, but on its value. Then, the special postal code *-1* instance can be shared to avoid `PostalCode` duplication.

In 2003, Marinov and O'Callahan [9] produced a profiling technique called *Object Equality Profiling*(OEP). This technique identifies redundant objects by inferring groups of objects qualified as *mergeable at time t*.

To solve the object duplication problem we introduce a cache for postal code *-1*:

```
class Address {
  ...
  static PostalCode unknownPostalCode = new PostalCode(−1);

  // Addresses with unknown postal code
  static Address createAddress(Street street, int number, City city) {
    return new Address(unknownPostalCode,street, number, city);
  }
  ...
}
```

Mergeability is a relation defined by Marinov and O'Callahan that relates two objects at time t if:

- the objects are instances of the same class,
- each pair of corresponding fields of the objects reference the same object or reference a pair of mergeable objects,
- both objects are not mutated from time t,
- no identity operation is applied on the objects from time t.

If two objects are then mergeable at time t, one of them becomes redundant from time t up to the end of the execution. Each pair of mergeable objects have their own time t for the moment that they become mergeable, which corresponds to the time of the last mutation or identity operation applied to one of the objects.

In the Address example, `PostalCode(-1)` instances become mergeable after being initialized, *i.e.,* time t is just after initialization. Postal code objects satisfy all conditions for that: (i) Objects are instances of the same class `PostalCode`, (ii) their instance variables hold the same

object `-1`, (iii) they do not mutate and (iv) no identity operations are used on them, like `==` or `hashCode()`.

Marinov and O'Callahan applied a technique to the SpecJVM98 benchmark, a heavily tested benchmark suite of 7 applications created by SPEC.org for measuring java virtual machines performance, which identified redundant objects. Although appealing, the proposed technique did not have an impact beyond the carried out case studies. We believe there are three reasons for this rather low acceptance among practitioners:

- ***Unknown mergeability site.*** OEP captures the allocation site of all objects. Even though that information is useful, allocation sites are not necessarily related with *mergeability sites*. We understand *mergeability site* as the exact location in the source code and the moment in the execution where the objects become mergeable. This happens because each pair of mergeable objects have their own particular time t when they become mergeable and the optimizations can only then be applied from this point. Unfortunately, the task of mergeability site searching is left to the practitioners.
- ***Not reproducible results.*** Marinov and O'Callahan describe with great care many aspects of object equality profiling and their experiments. However, some of the analyses exercised on the graphs of objects are not sufficiently detailed, which prevents us from replicating their experiments.
- ***Need a customized Virtual Machine.*** Most of the implementations of analysis techniques to profile object equality are carried out in the virtual machine. As a consequence, a dedicated virtual machine has to be used to benefit from the proposed technique.

We adapt Marinov and O'Callahan's technique to facilitate the proposal of fixes for the memory bloat. The approach has a lower recall than OEP, but as shown in this research, a significant amount of objects and memory has been removed from the analyzed applications.

This thesis revisits Marinov and O'Callahan's original contribution. We present and carefully analyze *object shareability* and *object equivalence*, techniques to identify redundant objects and efficiently avoid memory waste. Our technique features the following:

- We propose the *object equivalence* definition, which requires immutable objects. In contrast, the approach of Marinov and O'Callahan does not require immutability, but introduce the notion of *mergeability at time t*, which forbids mutations after time t. This has the advantage of greatly simplifying the memory analysis without discarding some relevant optimization opportunities.
- Each object creation is associated with its exact location in the source code and a reduced copy of the runtime stack. This is relevant when removing an unnecessary object creation. In our case study, we were able to optimize six applications without prior knowledge.
- Object equivalence is determined without modifying the virtual machine. Bytecode instrumentation and reflection are used instead. Performing our analysis on a standard and unmodified virtual machine has the advantage increasing adoption by practitioners. Because not requiring the set up of a complex virtual machine environment for

benchmarking reduces entry barriers.

- Our model supports specific notion of equivalence, tailored to identify redundancy in numerical values, string, points, and collections.

To evaluate our technique we designed an experiment that (i) measures the memory bloat our profiler is able to detect and (ii) measures the impact of the memory optimizations we are able to implement using the feedback of our profiler. We execute our experiments using the Pharo programming language, who offers a simple object model and runtime. Pharo is a pure object-oriented programming language inspired in Smalltalk. It is characterized for being an image based system[1] and and be a live and immersive environment.[2] Furthermore, we have access to Pharo community, which provided feedback about our optimizations. In total, we have carried out an analysis over six large Pharo applications.

Contributions. This thesis makes the following contributions:

- We carefully analyzed the object productions of six large Pharo application executions and found that we have 46.3% of equivalent objects on average, *i.e.,* objects that may be removed without affecting the application behavior. These objects represent 45% of the memory consumption of the application. Note that not all equivalent objects are redundant since at least one object per group has to be in memory.
- We have manually implemented 14 optimizations (mostly caches) and we have reduced the memory consumption, in average, by 25.1%. Moreover, the memory consumption reduction is higher than 40% in half of the benchmarks.
- We partially replicate Marinov and O'Callahan's experiment. We discuss thoroughly their approach similarities and differences from our approach. For example, we found that several classes defined as core of the language or libraries, like `Point` and `String`, are prone to produce redundant objects. We also addressed Marinov and O'Callahan's concern about strongly connected components are rarely redundant.

Findings. Our findings:

- We found that the following classes and their associated hierarchy contribute to the largest share of memory: `Number`, `String`, `Collection`, `Point`, `Rectangle`, `Association`.
- We found that strongly connected components, therefore cycles, can not be optimized by analyzing equivalent objects.

These findings will guide future efforts on memory profiling. In particular, we will focus on techniques for improving the way the runtime is used, instead of less profitable approaches like analyzing object cycles and strongly connected components.

Outline. The thesis is structured as follows:

[1] Image based system does not differentiate application state and source code, both are stored in a single file that is run by a virtual machine.

[2] The application state evolves and is saved within the source code. The execution may then be changed in run-time. Also programming tools such as the debugger, IDE, source code versioning, test runner, and others live in the same object space than the application.

- Section 2 presents an overview of shareable objects, its definition and properties.
- Section 3 presents our technique of object equivalence and the required formal definitions.
- Section 4 describes the design of an experiment we conducted to assess the relevance of object equivalence to identify memory bloat.
- Section 5 presents the results of the experiment.
- Section 7 describes a small experiment we conducted with an expert of Roassal2 and the feedback received by another expert about the optimizations proposed for NeoJSON and NeoCSV.
- Section 8 highlights some relevant aspects of our implementation.
- Section 9 discusses the impact of the research and presents an exhaustive comparison with Marinov and O'Callahan research.
- Section 10 describes the work related to our effort.

Chapter 2

Object Shareability

2.1 Overview

The scope of this research is to find objects that are not required in the execution because other objects already present in the execution may fulfill their behavior. There are two fundamental characteristics to accomplish that: (1) The callers or users of the object should not be able to distinguish if they are calling the original object or the candidate. (2) The objects calling the candidate should not distinguish that the candidate is being shared with other objects by replacing the original object.

Example. Consider the following example found in one of our case studies:

```
Builder>>createElement
  ^ GraphicalElement new
          color: self defaultColor;
          position: 0 @ 0

Builder>>defaultColor
   "Gray color"
  ^ Color r: 0.5 g: 0.5 b: 0.5
```

This example written in Pharo describes a class `Builder` that builds graphic-related objects. The method `createElement` returns an instance of a class `GraphicalElement`, initialized with a default color and an initial position. The default color is obtained by the method `defaultColor`, which returns a new instance of the class `Color`. Each time `createElement` is then called, a new `Color` object is created.

The application from which `Builder` comes employs this class to create large data visualizations composed of hundreds of thousands of colored elements. Many of these elements will have the same colors, and many of them will remain with their default color. Unfortunately, the class `Builder` does not consider the possibility of reusing object instances.

Our profiler identified this particular problem of the class `Builder`. The bloat problem

has been solved by creating a new class variable `DefaultColor` and inserting a cache in the method `defaultColor`.

```
Builder>>defaultColor
   "Gray color"
  DefaultColor = nil
      ifTrue: [ defaultColor := Color r: 0.5 g: 0.5 b: 0.5 ].
  ^ DefaultColor
```

This simple modification avoids the creation of a large number of objects, while preserving the behavior of the application. This improvement, by adding a cache, is relatively simple because:

- The example has a single production site of `Color` objects.
- The control flow going from the method `createElement` and `defaultColor` is very short.
- Instances of `Color` are not mutable.

However, a real-world application may have very large and deep control flow producing thousands of objects. Furthermore, lack of documentation about object invariants and mutability also hamper the proposal of improvements.

The followings sections of this chapter will set the foundation of our profiling technique to identify the creation of redundant objects.

2.2 Shareability relation

In order to characterize when it is possible to share an object and therefore use it to reduce the memory consumption of the application, we formalize a relation of shareability representing the property above.

2.2.1 Definition

We present the Object Shareability definition, $\rightsquigarrow$, a relation that describes if an object may be replaced by another object.

Definition 2.1 *Be o_1 and o_2 two objects in the execution, $o_1 \rightsquigarrow o_2$ iff modifying all references to o_1 to reference o_2 does not cause a change in the observable behavior of the program.*

Figure 2.1 shows a pictorial representation of the relation. The object o_1 is referenced by two objects: A and B, while o_2 is referenced by C and D. If o_1 is shareable to o_2, *i.e.,* $o_1 \rightsquigarrow o_2$, it is then possible to make A and B reference o_2 instead of o_1. Here we see that o_1 would not have any other object referencing it, and therefore, it can be garbage collected to reduce the memory consumption of the application. It is then possible to say that o_1 is *redundant* because of o_2.

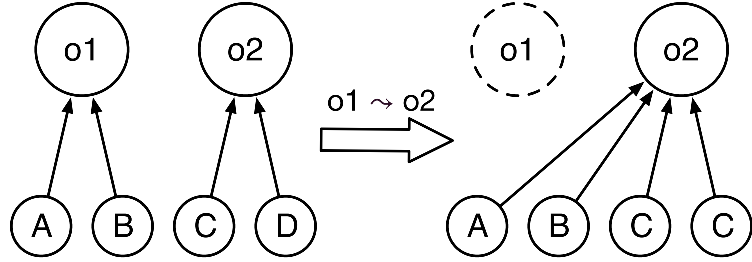


Figure 2.1: Object Shareability Relation

Revising the example in Section 2.1, all `Color` instances that hold the same RGB value are shareable between them. All objects using different instances of `Color`, which hold the same RGB value, may instead share a `Color` instance, hence reducing the number of objects used by the application.

2.2.2 Properties

We have analyzed some properties of the Shareability relation ($\rightsquigarrow$). We have found that $\rightsquigarrow$ is a reflexive and transitive relation.

Reflectivity ($o_1 \rightsquigarrow o_1$)

The relation is trivially reflexive because $o_1 \rightsquigarrow o_1$ means that there would be no change in the references, and therefore, no change in the behavior of the program.

Transitivity ($o_1 \rightsquigarrow o_2 \wedge o_2 \rightsquigarrow o_3 \implies o_1 \rightsquigarrow o_3$)

If the shareability relation is transitive, meaning that all shareable objects with an object a, which at the same time is shareable with an object b, then all the references to the objects can be changed to reference b, causing no change in the observable behavior of the program.

Let o_1, o_2, o_3 be objects in the execution that satisfy $o_1 \rightsquigarrow o_2$ and $o_2 \rightsquigarrow o_3$. Be $\{a_1, a_2, ..., a_k\}$ all the objects that reference o_1. Because of $o_1 \rightsquigarrow o_2$, it is possible to make $\{a_1, a_2, ..., a_k\}$ reference o_2 instead of o_1 without impacting the program behavior. Finally, because $o_2 \rightsquigarrow o_3$, it is possible to make $\{a_1, a_2, ..., a_k\}$ reference o_3 instead of o_2, causing no change in the observable behavior of the program. We conclude that we can make all the objects referencing o_1 to reference o_3 without impacting the program behavior, which corresponds to $o_1 \rightsquigarrow o_3$. The $\rightsquigarrow$ relation is then transitive.

No Symmetry ($\neg(o_1 \rightsquigarrow o_2 \implies o_2 \rightsquigarrow o_1)$)

That shareability is not a symmetric relation means that there are cases where $o_1 \rightsquigarrow o_2$ is true, but the reverse, $o_2 \rightsquigarrow o_1$, is not. To demonstrate this, we present a case where an object is replaceable by another one, but not viceversa.

The case is an ordered collection with an unordered collection. An immutable unordered collection can be replaced by an immutable ordered collection with the same elements without impacting the application behavior. In contrast, replacing an ordered collection with an unordered collection may break the application, because the objects using the collection expect the collection to iterate the elements in the same sequence, which is not ensured in an unordered collection.

We conclude that the relation is not symmetrical.

2.3 Redundant objects and groups

Redundant objects are objects in the execution that are not needed. It is possible to avoid their creation or garbage collect them earlier, without impacting the program behavior. Objects may be redundant for several reasons, *e.g.,* objects that are initialized without side-effects, but are not used.

It is then possible to characterize *redundant objects* using the *shareability relation.* An object is *redundant* if there exists another object to which it is shareable to.

Definition 2.2 Let O be the set of all objects allocated by a program. Let a be then an object in O. a is *redundant* iff:

$$\exists o \in O,\ a \rightsquigarrow o$$

In an execution, an object may be the cause of redundancy of many objects. Then we propose a definition for *redundant groups*, where all the objects in the group are redundant because of the same object.

Definition 2.3 *Let O be the set of all objects allocated by a program. We define $[a]_{\rightsquigarrow}$, the redundant group of a in the execution, as:*

$$[a]_{\rightsquigarrow} = \{o \in O | o \rightsquigarrow a\}$$

Object a can then be used as representative of all the objects in the redundant group $[a]_r$, reducing the memory consumption of the execution while causing no change in the observable behavior of the application.

Revising the example in Section 2.1, the redundant group of a `Color` corresponds to the set of colors with the same RGB values. More formally:

$$[Color(r, g, b)]_{\rightsquigarrow} = \{Color(r', g', b') \in O | r = r' \wedge g = g' \wedge b = b'\}$$

2.4 Shareability Oracle

The *shareability relation* is built to represent the capability of an object to be replaced by another one. Identifying the objects that are related by *shareability* is not trivial, because it involves ensuring the observable behavior of the application is not impacted. Therefore, building an oracle for *shareability* capable of computing the relation for all pair of objects may not be possible.

Several approaches are presented in the literature attempting to create oracles capable of identifying the largest possible subset of objects in the *shareability relation* [9, 12, 7]. All of them reduces the present relation, which is not previously formalized, in order to create a practical tool which work for large executions.

We have found in the literature that authors privilege the precision of the analysis, *i.e.,* reduced amount of false positive, instead of the recall, *i.e.,* reduced amount of false negatives. We have not found an approach that sacrifices the safety of the tool to increase the recall. Also, we have not found a reduction of the shareability relation that use a non-symmetrical reduction of the relation.

Chapter 3

Object Equivalence

We propose a novel reduction of the *object shareability* property named *object equivalence*. This new construction is a relation between objects that corresponds to a subset of the *object shareability relation*. This approach sacrifices the recall of the *object shareability* to boost performance and achieve a practical analysis that enables practitioners to reduce memory bloat on real industry applications.

3.1 Intuition

We say that two objects are equivalent, $o_1 \approx o_2$, if in the application they are indistinguishable from each other. Object equivalence then means that the application is not able to decide if it is referencing o_1 or o_2. Trivially, we conclude that two equivalent objects are also shareable objects, because making all objects that reference to o_1 to reference o_2 would be a technique for distinguish between o_1 and o_2.

In order for two objects o_1 and o_2 to be indistinguishable, then equivalent objects during the whole execution, they have to satisfy the following:

- both objects are instances of the same class;
- both objects instance variables do reference the same objects or equivalent ones;
- the identity hash value is never retrieved and object pointers are never compared for both objects;
- both objects remain unchanged, beyond their initialization. This means that instance variables values do not change after the object construction and initialization.

Our object equivalence technique monitors the life of each object during the application execution. After the execution, our technique automatically identifies groups of objects that are equivalent. The reminder of this section details the notion of object equivalence.

3.2 Object Equivalence

Unchanging objects. Identifying whether an object has a varying state during the application execution is a critical aspect of our approach. In order to reduce memory bloat from equivalent objects, both objects are required to remain unchanged during the rest of the execution. The unchanged property is required for ensuring that equivalence is preserved during the execution.

An object for which each of its instance variables receives one value at most during the lifetime is qualified as unchanged.

Definition 3.1 *Let $t, t' \in T$, being T the timestamps of all the operations performed over the object o. An object o is unchanged, written* unchanged(o) *iff:*

$$\begin{gathered} \forall f \in instVars(o) \\ \forall t \mid o.f_t \neq NULL, \\ \nexists t' > t \mid o.f_{t'} \neq o.f_t \end{gathered}$$

Where $o.f_t$ represents the value of the instance variable f of the object o at the instant in the execution defined by the timestamp t. For instance, this definition forbids an object instance variable $o.f$ from having a different value at two different times if it was previously initialized.

Identity operations. The object identity is used in two cases: (i) when the object hash value is obtained or (ii) when the object reference is compared, using `==`. As soon as object identity is used, then the object is not equivalent to any other object. As a consequence, as soon as an object is used as a key in a `Dictionary` or added into a `Set`, then it is likely to be excluded from our analysis. The object may still be considered in case that its class provides a customized hash and equals methods.

An object o does not expose its identity, *noIdExpose(o)* iff there is no identity based operation applied to o during the program execution. In the specific case of Pharo this means to the == and *identityHash* messages.

Object cycle. When configuring the equivalence of two objects to depend on the equivalence of their instance variables, we are defining a recurrence relation. In Pharo, as in many languages, wide and cyclic object webs are easy to produce. Cyclic structures have to be treated in a particular way to compare them. Therefore, we differentiate whether the relation is used with objects in a cycle or not. An object o belongs to a cycle iff:

$$\exists f_1, f_2, ..., f_n | o.f_1.f_2....f_n = o$$

Strongly connected components (SCC). A strongly connected component is a set of nodes that for every node exists a path to every other node. By definition, the nodes in a graph belonging to the same cycle configure a SCC that includes, at least, all of the nodes

in the cycle.An essential property of SCC is that there must exist at least one cycle that involves all the nodes of the same SCC. Furthermore, if o_1 and o_2 belong to different strongly connected components then there is no cycle that contains o_1 and o_2 [3].

Object Equivalence. Consequently we define object equivalence, $o_1 \approx o_2$ in the case both objects do not belong to any cycle and the case both objects belong to a cycle. According with our definition, an object that is not in a cycle can not be equivalent with an object that does belong to a cycle.

Definition 3.2 *If both o_1 and o_2 do not belong to a cycle, then $o_1 \approx o_2$ iff:*

$$o_1 = o_2 \qquad or \qquad \begin{array}{c} class(o_1) \; = \; class(o_2) \\ \forall f \in instVars(o_1) \mid o_1.f \approx o_2.f \\ unchanged(o_1) \text{ and } unchanged(o_2) \\ noIdExpose(o_1) \text{ and } noIdExpose(o_2) \end{array}$$

The condition that all the instance variable pairs must be equivalent, ensures that all the possible paths from the objects must reference equivalent objects. Therefore, an object that does not belong to a cycle can not be equivalent to an object that does belong to one, and viceversa.

Definition 3.3 *If both o_1 and o_2 belong to a cycle, then $o_1 \approx o_2$ iff $o_1 = o_2$ or:*

$$\left(\begin{array}{c} \forall k \in \; \{1, ..., n\} \mid \\ class(e^1_k) \; = \; class(e^2_k) \\ \forall f \in instVars(e^1_k) \mid \\ \begin{cases} e^1_k.f = e^1_t \implies e^2_k.f = e^2_t & e^1_k.f \in scc(o_1) \\ e^1_k.f \approx e^2_k.f & e^1_k.f \notin scc(o_1) \end{cases} \\ unchanged(e^1_k) \text{ and } unchanged(e^2_k) \\ noIdExpose(e^1_k) \text{ and } noIdExpose(e^2_k) \end{array} \right)$$

Where $scc(o_1) = (e^1_1, ..., e^1_n)$ and $scc(o_2) = (e^2_1, ..., e^2_n)$, with scc a function that returns the list of objects belonging to the strongly connected component, using a depth-first search. We have $e^1_1 = o_1$ and $e^2_1 = o_2$.

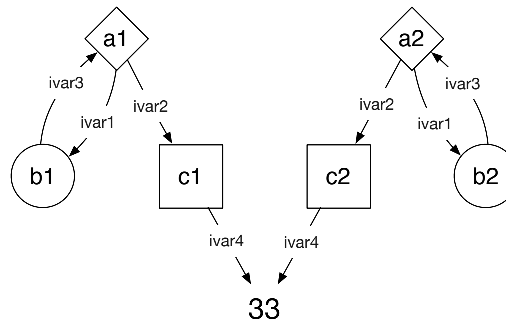
The definition above indicates that two objects are equivalent iff they are instances of the same class and the object graphs reachable from the objects are equivalent. In the case that o_1 does not belong to a cycle the recurrence is straightforward, since a recursive definition always finds a base case. For the case that o_1 does belong to a cycle, we have to analyze: (1) the topology of the strongly connected component and the relative position of each member and (2) all the references from the strongly connected component members to other parts of the object graph.

3.3 Equivalence example

Consider the following contrived example using a Java syntax:

```
class A {
   B ivar1;
   C ivar2;
}
class B {
   A ivar3;
}
class C {
   int ivar4;
}
```

```
main() {
   A a1 = new A();
   B b1 = new B();
   C c1 = new C();
   A a2 = new A();
   B b2 = new B();
   C c2 = new C();
   a1.ivar1 = b1;
   a1.ivar2 = c1;
   b1.ivar3 = a1;
   c1.ivar4 = 33;
   a2.ivar1 = b2;
   a2.ivar2 = c2;
   b2.ivar3 = a2;
   c2.ivar4 = 33; }
```



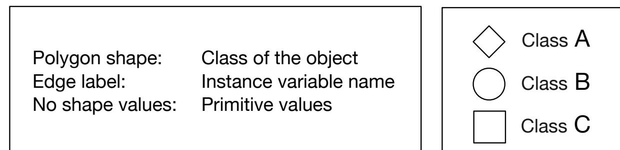


Figure 3.1: Object graph of example which shows that $a1 \approx a2$, $b1 \approx b2$, and $c1 \approx c2$.

The snippet of code generates the object graph shown in Figure 3.1. We can identify two portions of the graph that are equivalent between them. We deduce that $c1 \approx c2$ because both objects reference the same object, the integer `33`.

In order to evaluate if a_1 is equivalent to a_2, we analyze the possibilities of equivalence between the cycles $\{a1, b1\}$ and $\{a2, b2\}$. Firstly $scc(a1) = (a1, b1)$ and $scc(a2) = (a2, b2)$, then $a1.ivar1 = b1$ holds the same position in the strongly connected component as $a2.ivar1 = b2$. For $a1.ivar2 = c1$ is equivalent to $a1.ivar2 = c2$. Finally $b1.ivar3 = a1$ which is matched by $b2.ivar3 = a2$. With all these, and ensuring that no mutations and identity based operations are performed, we conclude that $a1 \approx a2$, $b1 \approx b2$ and $c1 \approx c2$.

3.4 Properties

Object equivalence is reflexive ($o_1 \approx o_1$), symmetrical ($o_1 \approx o_2 \implies o_2 \approx o_1$) and transitive ($o_1 \approx o_2 \; and \; o_2 \approx o_3 \implies o_1 \approx o_3$). We then conclude that *object equivalence* is an equivalence relation.

A property of our definition while handling cycles is that if a member of a cycle is equivalent to a member of another cycle, then both complete cycles are equivalent between them. This is concluded from the recurrence definition of *object equivalence*. If two objects belonging to cycles are equivalent, then each pair of instances variables are equivalent between them. Using induction over the instance variables belonging to the cycle we conclude that for each member of the cycle there is an equivalent object in the other cycle holding the same relative position in the cycle.

3.5 Equivalent object group

It is possible to group the objects using the equivalence class. From now on, we will refer the equivalence class of the equivalence relation defined in this thesis as *object equivalence groups*.

An *object equivalence group* is a set of objects where all elements are equivalent between them. For this reason, we only need one object per object equivalence group to act as a representative of all the objects in the equivalence group. The rest of the elements of the group are redundant.

Definition 3.4 Object equivalence group:
Let O be the set of all objects allocated by a program. We define the object equivalence group of the object a using the notation $[a]$ as:

$$[a] = \{o \in O | o \approx a\}$$

The object groups are defined by the equivalence relation definition. Therefore, they are equivalent since they were created and during the whole execution. Despite that, gathering information about the allocation or creation context is fundamental in order to reduce the memory bloat caused by redundant objects. This is performed by allowing the garbage collection of redundant objects or avoiding their creation.

Chapter 4

Experiment design

This section describes the experiment we performed to assess our technique to identify equivalent objects. Section 4.1, Section 4.2 and Section 4.3 present the background required for the experiment, such as an introduction to the Pharo object model and the object characterization used in this case study. Section 4.4 presents the experiment methodology. Section 4.5 describes the metrics to be collected in the experiment. Section 4.6 briefly presents the benchmark suite to be used in this case study.

4.1 Pharo object model

We use the Pharo programming language in our experiment as the execution and analysis platform. We chose Pharo for the simplicity of both its object model and its runtime. Pharo's object model is uniform: Pharo does not contains primitive types, which means that all the computation happens by sending messages. Even the object reference comparison is performed using the message ==. Pharo offers a sophisticated reflective API, which allows us to easily introspect the memory and intercept the messages sent between objects.

4.2 Object characterization

For the analysis we characterize the objects produced during the execution of a program P (*i.e.,* set of classes composing the program) and using a runtime R (*i.e.,* set of classes used by P but defined in system libraries) in three distinct groups:

- *Domain objects* corresponds to instances of the classes that belong to P. A domain object typically describes an entity created and modeled by P. Equivalent domain objects are described by the $\approx$ relation, previously given.
- *External objects* refer to instances of classes that are not defined in P. External objects are created by libraries used by P or directly generated from R. We consider these

objects as non-equivalent.

- *System specific objects* are a subset of the external objects for which it is known that they are prone to be redundant (*e.g.,* `String`, `Point`, `Collections`). A system specific object is an instance of a class defined in R.

The object equivalence definition given previously is used to identify redundant objects characterized as domain objects. System specific objects are subject to particular object equivalence relations, as described below.

4.3 System specific objects

System specific objects require a customized equivalence definition motivated by their object invariant, *e.g.,* unordered collections should not be distinguished by the order they iterate on their elements.

Numbers. Numbers are part of the core of the system and are widely used in software systems. In Pharo, a number is represented as an immutable instance of a class belonging to the `Number` class hierarchy. Even though the virtual machine optimizes small integers, other numbers do consume memory as they are class instances. As in most programming languages, the numbers library provides different representations for the same value. For example, the value `4` can be represented as the integer `4`, the float `4.0`, the fraction `8/2`, or as a large integer. All numbers representing the same value are equivalent.

String. Strings are part of the core of the system and their manipulation may lead to unnecessary intermediate objects. Because Strings are immutable, two strings having the same ordered set of characters are then equivalent objects.

Collections. Collections are intensively used. Our equivalence relation for system specific objects considers the most four frequently used collections: `Array` is a fixed-size collection; `OrderedCollection` is an expandable sequential collection; `Set` describes an unordered expandable collection without duplicates; `Dictionary` is an expandable unordered key-value collection. We have the following equivalence relations:

- An array is equivalent to another array if its sizes are the same, the contents are equivalent and both meet the *unchanged* property.
- An ordered collection may be equivalent to another ordered collection if both sizes are the same at the end of the execution, their contents equivalent, and the collection has been filled only appending elements at the end without removing elements.
- A set is equivalent to other sets if their sizes are the same and the contents are equivalent, while forbidding elements removal.
- A dictionary is equivalent to other dictionaries if their sizes are the same, they have the equivalent keys associated with equivalent values and that no association $\{key, value\}$ is redefined in the dictionary during the execution.

These relations ignore the inner state of collections such as collection capacity or the inner

hash table.

Point and Rectangle. Graphical applications do heavily rely on the classes `Point` and `Rectangle`. In Pharo, these classes are immutable as value objects, which implies that their equivalence is trivially determined.

Association. Associations are simple key / value pairs and are widely used in the implementation of Dictionaries and other structures. Two associations are equivalent if both their keys and values are equivalent.

Pharo does not strictly prohibit Numbers, String, Point and Rectangle mutation, but their immutability is a known invariant by the community and mutations are infrequent. Point and Rectangle mutators are marked as private and only called when initializing a new object. Float does not have mutator and only can be mutated by the use of reflection. For Strings, there exists mutators, but we have not found cases in which String objects were mutated.

We have found a single case where this invariant is not followed in Athens project, which mutates Float values. This is fortunately an infrequent behavior and is documented by the authors to prevent practitioners of unexpected errors.

4.4 Experiment

To evaluate our technique and compare it with the related work, we introduce an experiment that uses our profiler on a representative benchmark suite. The data is then analyzed to achieve the following specific objectives:

1. Assess that system specific objects have a large impact on memory consumption.
2. Assess if most applications have a large amount of memory bloat or not.
3. Evaluate the impact of memory optimizations proposed by reducing memory bloat.

To achieve the proposed objectives we use the following methodology for the experiment:

1. Build a representative benchmark suite for which to apply our technique.
2. Propose a set of metrics that allow us to evaluate the objectives of the experiment.
3. Run the profiler on the benchmarks and obtain the profiler report.
4. Compute metric values of execution before implementing optimizations from the profiler report.
5. Analyze profiler report and implement optimization for all applications in the benchmark suite.
6. Rerun the profiler on the benchmark suite with the optimizations and obtain the report of the profiler.
7. Compute metric values of execution after implementing optimizations.

4.5 Metrics

Equivalent objects indicate an opportunity to improve the memory management. Avoiding their creation reduces the memory footprint without changing the application behavior. To measure the amount of redundant objects and measure the effect of the memory footprint reduction, we provide a set of 15 metrics to cover the different aspects of memory management related to equivalent objects. Table 4.1 lists our metrics.

Related to the Number of Objects. As described in Section 4.2, objects created during a program execution belong to one of three distinct groups. We therefore provide the metrics **NDO**, **NSO**, **NEO** describing the number of domain, specific, and external objects, respectively. The addition of **NDO**, **NSO** and **NEO** represent the number of non-primitive objects, named **NOP**, which represents the total number of objects with non-zero memory consumption. Both `SmallInteger` and the special value `nil` are encoded into their reference, without consuming space in the heap, then they are not counted on **NOP** metric. Our approach therefore consists in reducing the **NOP** value for our benchmarks.

Related to Equivalent Objects. We define 5 metrics for measuring the quantity of equivalent objects on the execution. We then define **NEqD** and **NEqS** as the number of equivalent domain and system specific objects, respectively. Furthermore we define **NEqO**, which corresponds to the total number of equivalent objects and this metric is the sum of the two previous metrics. Lastly we describe the number of equivalence groups using **NEqDG** and **NEqSG**, being the number of domain groups and system specific groups, respectively.

The number of equivalent object groups represents the quantity of objects that are needed to fulfill the behavior of all the objects of the category. For example, **NEqDG** is the minimum number of objects needed to represent all of the **NEqD** objects. The ideal case for reducing the memory consumption would be a low amount of groups and a high amount of equivalent objects.

Related to Memory Consumption. We provide 6 metrics to detail the memory consumption: **MDO** and **MSO** describes the amount of KB consumed by domain objects and system specific objects respectively; **MOP** is the total memory consumption of the application as the addition of the previous two metrics; **MEqD** and **MEqS** describes the amount of KB consumed by equivalent domain and equivalent system specific objects respectively; **MEqO** is the amount of memory consumed by all the equivalent objects, which corresponds to the addition of the previous two metrics.

Expected variations. These metrics allows us to measure the impact of equivalent objects in the memory consumed by our benchmark. An improvement on memory consumption is reflected by a reduction on the number of objects (**NOP**) and total memory consumption (**MOP**). Furthermore, this is related to a reduction in the number and memory used by equivalent objects (**NEqO** and **MEqO**), which are the target object for optimizations in this research.

Table 4.1: Metric definitions

Short	Metric
	Related to the Number of Objects
NOP	Number of Non-Primitive Objects
NDO	Number of Domain Objects
NSO	Number of System Specific Objects
NEO	Number of External Objects (Excluding NSO)
	Related to Equivalent Objects
NEqO	Number of Equivalent Objects
NEqD	Number of Equivalent Domain Objects
NEqDG	Number of Domain Objects Equivalence Groups
NEqS	Number of Equivalent System Specific Objects
NEqSG	Number of System Specific Objects Equivalence Groups
	Related to Memory Consumption
MOP	Memory used by Non-Primitive Objects (KB)
MDO	Memory used by Domain Objects (KB)
MSO	Memory used by System Specific Objects (KB)
MEqO	Memory used by Equivalent Objects (KB)
MEqD	Memory used by Equivalent Domain Objects (KB)
MEqS	Memory used by Equivalent System Specific Objects (KB)

Table 4.2: Applications metrics

Applications	Numbers of Classes	Number of Methods	Lines of Code
Nautilus	123	1917	11411
SciSmalltalk	277	2569	17208
Roassal	819	9019	83075
NeoJSON	23	278	1669
NeoCSV	8	185	1118
PetitParser	340	3233	15692

4.6 Benchmarks

We have considered six applications in our benchmark: Roassal2 (a visualization engine), Nautilus (a code browser), SciSmalltalk (a scientific library), NeoJSON (JSON parser), NeoCSV (CSV parser), and PetitParser (a parser framework). These applications are heavily maintained by the Pharo community and represent valuable assets.

These applications are typically used with a large amount of input data, and reducing their memory footprint is valuable their developers and the Pharo community. For each application, we have a representative execution using a large input data provided by the author of that application.

We provide some metrics of the applications being used for our case study in Table 4.2.

4.7 Summary

This section presents a detailed characterization of objects in the execution in order to allow the use of the most appropriate analysis technique to each of the objects. This introduces the notion of: *Domain Objects*, objects defined by the project to be analyzed; *System Specific Objects*, objects of the runtime or libraries that are prone to be redundant and have a known invariant; *External Objects*, objects out of the scope of the analysis and to be ignored by the tool.

The experiment methodology is then presented, where we explain our case of study and present the three objectives of the experiment: (1) Assess that system specific objects have a large impact on memory consumption. (2) Assess if most applications have a large amount of memory bloat or not. (3) Evaluate the impact of memory optimizations proposed by reducing memory bloat.

A set of metrics is presented to reflect and allow analysis of potential memory consumption opportunities and the impact of a solution. Then a high number of equivalent objects (NEqO) reflects a potential improvement on memory consumption and the reduction on the total number of objects and memory consumption (NOP and MOP) reflects the impact of the optimizations.

Lastly we briefly present the 6 applications included in the benchmark suite. All applications maintained by the Pharo community are open source and are widely used in industry.

Chapter 5

Results

This chapter presents the measurement obtained from running the benchmarks using the profiler (Section 5.1) and then exposes the impact on the memory footprint of the optimizations we manually implemented (Section 5.2). Lastly, it presents a summary of the results (Section 5.3).

5.1 Metrics analysis

Number of objects. We have run our analysis on each application composing our benchmark. Table 5.1 gives the metric values for our benchmark. The NOP column sums the columns NDO, NSO, and NEO. The complete benchmark execution produces over 1.1M objects, for which 22% are domain objects (*i.e.,* object instances of the class defined by the applications), 68% are system specific objects (*e.g.,* numbers, collections, strings), and only 10% are external objects. This measurements highlight that *a significant portion of created objects during an execution are system specific objects.*

Equivalent objects. Table 5.2 gives the number of equivalent objects measured in our benchmark. In total, our benchmark produces 46.3% of objects that are equivalent objects (NEqO column). We also report that 38.6% of the created objects are equivalent system

Table 5.1: Results in number of objects

	NOP	NDO	NSO	NEO
Roassal2	188,901	34,931 (18%)	132,950 (70%)	21,010 (11%)
Nautilus	245,602	59,468 (24%)	162,611 (66%)	23,523 (10%)
SciSmalltalk	342,455	60,947 (18%)	218,253 (64%)	63,255 (18%)
NeoJSON	85,017	5,003 (6%)	80,009 (94%)	5 (0%)
NeoCSV	75,935	2 (0%)	75,862 (100%)	71 (0%)
PetitParser	184,649	83,544 (45%)	95,190 (52%)	5,915 (3%)
Total	1,122,559	243,895 (22%)	764,875 (68%)	113,779 (10%)

Table 5.2: Number of equivalent objects

	NOP	NEqO	NEqD	NEqDG	NEqS	NEqSG
Roassal2	188,901	104,915 (55.5%)	21 (0.0%)	5	104,894 (55.5%)	758
Nautilus	245,602	191,872 (78.1%)	34,449 (14.0%)	582	157,423 (64.1%)	1,357
SciSmalltalk	342,455	1,773 (0.5%)	0 (0.0%)	0	1,773 (0.5%)	767
NeoJSON	85,017	40,000 (47.0%)	0 (0.0%)	0	40,000 (47.0%)	13
NeoCSV	75,935	50,923 (67.1%)	0 (0.0%)	0	50,923 (67.1%)	4,746
PetitParser	184,649	129,738 (70.3%)	51,955 (28.1%)	5,168	77,783 (42.1%)	2,041
Total	1,122,559	519,221 (46.3%)	86,425 (7.7%)	959	432,796 (38.6%)	1,614

Table 5.3: Results in memory usage (KB)

	MOP	MDO	MSO	MEqO	MEqD	MEqS
Roassal2	4,018	1,228 (31%)	2,791 (69%)	1,963 (49%)	0 (0.0%)	1,963 (48.8%)
Nautilus	5,268	1,916 (36%)	3,352 (64%)	3,613 (69%)	478 (9.1%)	3,135 (59.5%)
SciSmalltalk	5,066	1,659 (33%)	3,407 (67%)	19 (0%)	0 (0.0%)	19 (0.4%)
NeoJSON	1,380	78 (6%)	1,302 (94%)	456 (33%)	0 (0.0%)	456 (33.0%)
NeoCSV	1,486	0 (0%)	1,486 (100%)	740 (50%)	0 (0.0%)	740 (49.8%)
PetitParser	3,552	1,621 (46%)	1,931 (54%)	2,655 (75%)	1,006 (28.3%)	1,649 (46.4%)
Total	20,771	6,502 (31%)	14,269 (69%)	9,446 (45%)	1,484 (7.1%)	7,962 (38.3%)

specific objects (NEqS) and only 7% are domain objects (NEqD).

Figure 5.1 shows that NEqS is in high proportion, representing over 40% of the objects for all applications but one. NEqD tops at 28% for the PetitParser application, but no equivalent domain objects were found on 4 of the 6 benchmarks. We have the relation NEqO = NEqD + NEqS: the number of equivalent objects is equal to the number of equivalent domain objects summed up with the number of equivalent system specific objects. These figures indicate that *a significant portion of system specific objects are redundant during a program execution.*

Also we distinguish that SciSmalltalk benchmark is an outliner because only 1,773 objects were found to be equivalent, representing 0.5% of the total number of objects. This is discussed further in Section 9.

Memory footprint. Table 5.3 gives the memory footprint of objects produced during the benchmark execution. The column MOP gives the memory used by non-primitive objects, *i.e.,* objects that are not nil or small-integers. It indicates that the 1.1M objects produced during the benchmark execution consumes 20,771KB. This amount is the total memory consumed within the heap. In practice, this 20MB have a larger footprint due to the memory management. Section 9.1 discusses that topic further.

In the benchmarks the memory consumption of equivalent objects MEqO represented more than 50% of the total memory consumption in half of the benchmarks. Then in our set, half of the applications have the potential to reduce memory consumption by more than 50%.

We also notice that domain objects consume less memory than system specific objects, 31% and 69% respectively. Equivalent objects represent 45% of the memory consumed by

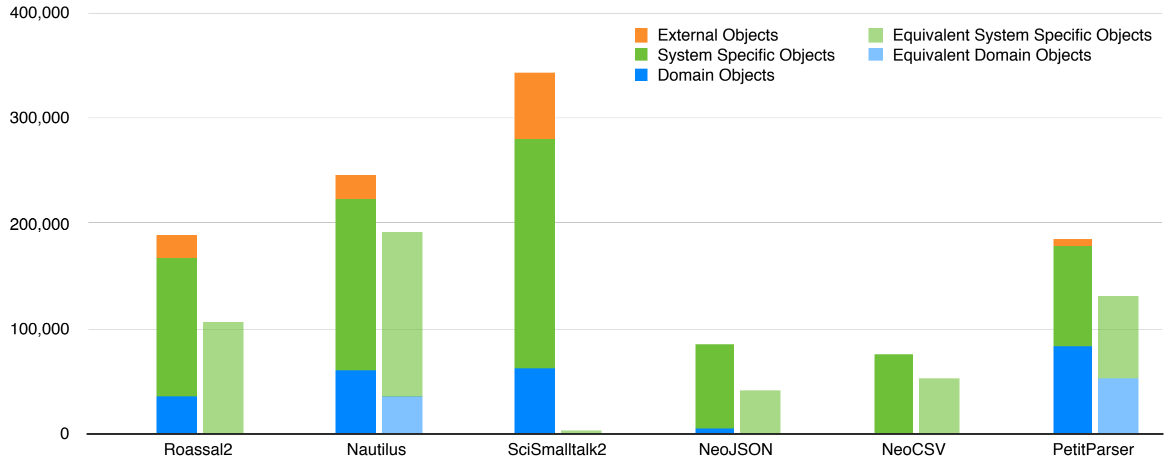


Figure 5.1: Distribution of objects and equivalent objects in benchmarks.

Table 5.4: Reduction of memory footprint due to our optimizations

	NOP Before	NOP After	Reduction	MOP (KB) Before	MOP (KB) After	Reduction
Roassal2	188,901	126,349	33.1%	4,018	2,223	44.7%
Nautilus	245,602	96,948	60.5%	5,268	3,072	41.7%
SciSmalltalk	342,455	342,455	0.0%	5,066	5,066	0.0%
NeoJSON	85,017	55,025	35.3%	1,380	1,081	21.7%
NeoCSV	75,935	29,796	60.8%	1,486	817	45.1%
PetitParser	184,649	175,526	4.9%	3,552	3,303	7.0%
Total	1,122,559	826,099	26.4%	20,771	15,560	25.1%

the benchmark execution. Equivalent system specific objects represent 38.3% of the whole memory.

5.2 Avoiding equivalent object creation

Table 5.2 indicates that 53.1% of the objects created during the benchmark executions are equivalent. This value represents therefore the maximum amount of object reduction we can obtain by avoiding creation of equivalent objects. Note that we cannot remove all equivalent objects since at least one object has to remain per object equivalence group. In several cases, completely avoiding the creation of equivalent objects is not worth the effort, *e.g.,* if object equivalence groups are very small and with a low memory footprint.

We use the profiling information and the object creation context given by our profiler to avoid the creation of the largest groups of equivalent objects. We manually revised all

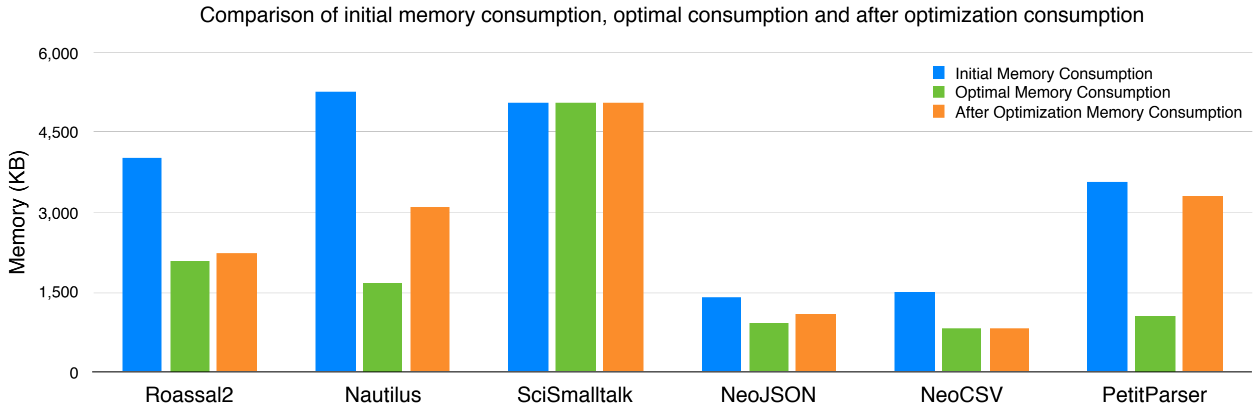


Figure 5.2: Comparison of initial memory consumption, optimal consumption and after optimizations consumption.

the equivalent object groups and implemented 14 optimizations across the six applications composing our benchmark. The implementation details are discussed on Section 6 and are distributed per application in the following way:

- Roassal2 - 2 optimizations
- Nautilus - 9 optimizations
- SciSmalltalk - 0 optimizations
- NeoJSON - 1 optimizations
- NeoCSV - 1 optimization
- PetitParser - 1 optimization

Despite that these optimizations were relatively simple to implement, they have a significant impact on the measurements. Table 5.4 gives the metric values and their variations. In total our optimizations have reduced the number of object creations by 26.4%, approximately representing 2/3 of the total number of equivalent objects (NEqO). This diminution of objects results in a reduction of 25.1% of the consumed memory.

Figure 5.2 shows the memory consumption of the applications before any optimization, the optimal memory consumption and the memory consumption after our proposed optimizations.

The reduction of memory consumption is lower than the reduction of number of objects (25.1% vs 26.4%). This indicates that equivalent objects have usually smaller size than average objects.

Impact on Number of Domain Objects.

Figure 5.3 shows that 2 of the 6 projects, Nautilus and PetitParser, have a significant amount of redundant domain objects, *i.e.,* domain objects that are not needed in the execution. This is represented as the difference between the number of initial domain objects and the

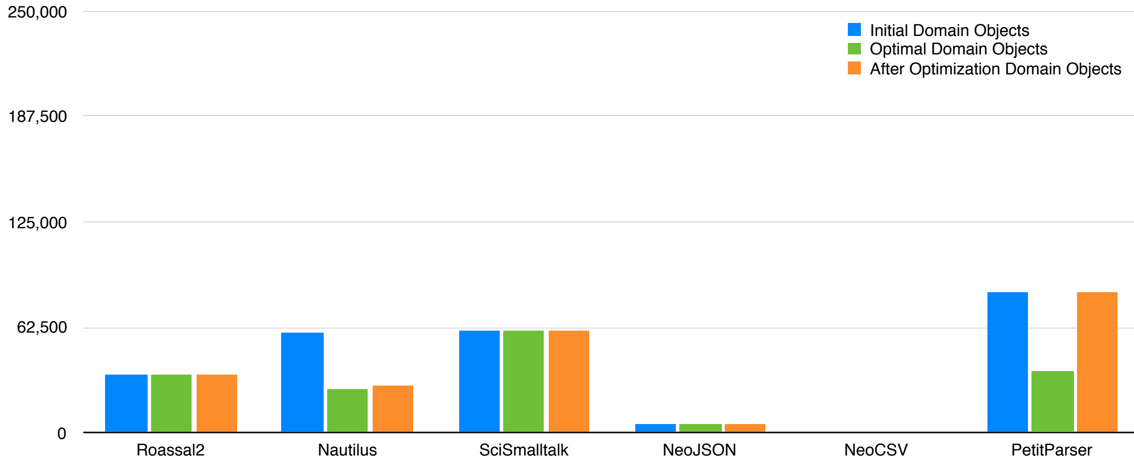


Figure 5.3: Comparison between initial number of domain objects, optimal number of domain objects and after optimizations number of domain objects.

optimal number of domain objects.

Nautilus is the only benchmark with different values for the number of domain objects after the optimizations. This is because only in 2 projects we found redundant domain objects and we were not able to reduce the number of redundant domain objects for PetitParser.

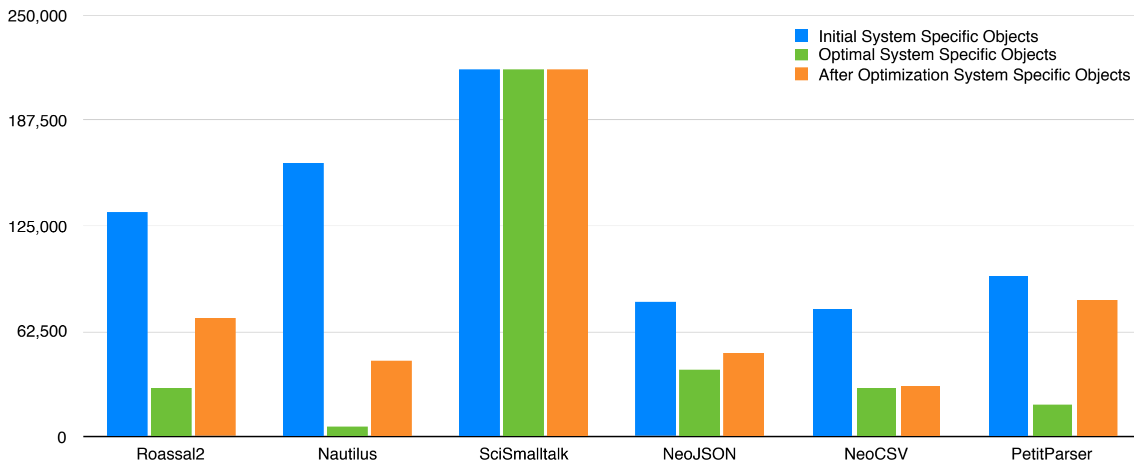


Figure 5.4: Comparison between initial number of system specific objects, optimal number of system specific objects and after optimization number of system specific objects.

Impact on Number of System Specific Objects.

Figure 5.4 shows that all projects except for SciSmalltalk, present a significant amount of redundant system specific objects.

We have proposed optimizations for reducing the amount of system specific objects for all

the benchmarks that presented a significant amount of redundant system specific objects. For NeoJSON and NeoCSV the achieved number of system specific objects after the optimizations is similar to the optimal number. For Roassal, Nautilus and PetitParser the chart shows that the optimal number of system specific objects is still far from the achieved one after the optimizations. This is because we were not able to propose optimizations for all cases.

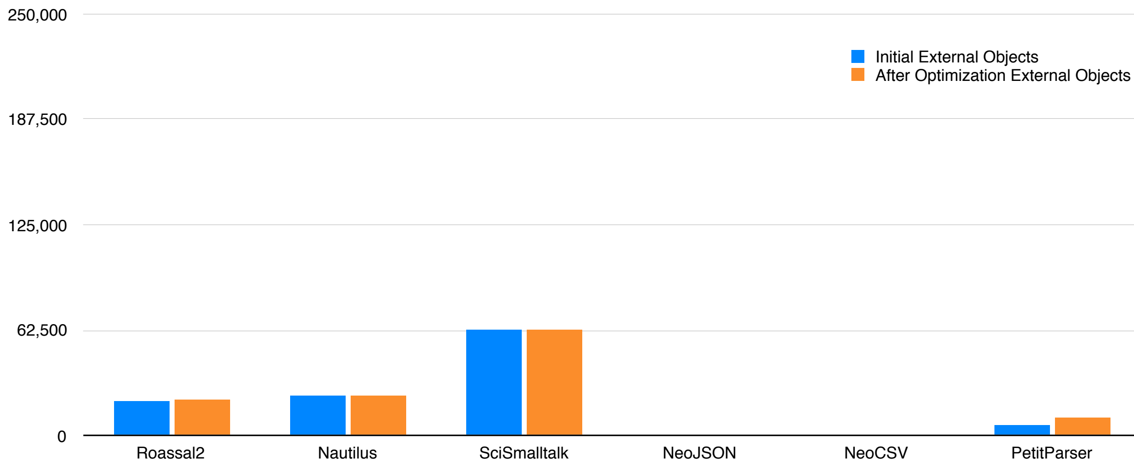


Figure 5.5: Comparison between initial number of external objects and after optimizations number of external objects.

Impact on Number of External Objects.

Figure 5.5 shows that all projects maintain the number of external objects after the optimizations expect for PetitParser. This is expected because the optimizations are not supposed to modify the behavior of the application, *i.e.,* they should not have unexpected behavior.

For PetitParser the increase of number of external objects after the optimizations is explained because of the use of SmallDictionary instead of Dictionary as part of the fix. SmallDictionary instances are not part of the set of system specific objects and are treated as external objects. Further details about the optimization for PetitParser are available at Section 6.2.4.

5.3 Summary

This section presented results obtained from running benchmarks on 6 different industrial Pharo projects. We have found that half of the projects have a potential memory consumption reduction of 50% and just one of the 6 applications did not present a potential memory consumption reduction by means of this analysis.

For the 5 projects in which we found potential memory consumption reduction, we implemented a set of optimizations in the source code and measured the impact of our

solutions. In three of the five projects the solution achieved more than 40% of memory consumption reduction.

Chapter 6

Solving object redundancy

This chapter presents our finding when solving the memory bloat found by the profiler. The profiler output offers the practitioner a large amount of data to analyze. This chapter also serves as a guideline for practitioners to analyze the gathered data and propose their fixes.

The chapter is divided into three main sections. Section 6.1 describes in detail the report handled by the tool, how to analyze it and offer high-level guidelines for discovering significant optimization opportunities. Section 6.2 presents a classification of the implemented optimizations for our benchmark. For each of the categories it describes the case of use of the optimization, a high-level description of its implementation and gives a real example of it proposed for our our case of study. Section 6.3 presents the number of optimizations performed on each application per category and the time required for it.

6.1 Tool report and use strategy

The profiling tool provides an analysis environment enabling the practitioner to inspect the object graph of the execution. The tool supports an expressive graph API with pre-computed data for finding memoization opportunities.

Regarding the analysis of the data, the most used resource provided by the tool is the collection of equivalent object groups, which are pre-computed by the environment. Using this information the practitioner can inspect if it is a worth optimizing an object group or allocation site.

The available data offered by the profiler are nodes and edges of the object graph and the collection of equivalent object groups. Additional information is also offered for the analysis on each node:

- object class,
- references from object,
- connected component it belongs to,

- if the object is unchanged or not,
- allocation and garbage collection time,
- allocation context of reduced size,[1]
- soHash (the hashing function to compute equivalent objects is defined at Section 8.3)
- equivalence object group it belongs to, if applicable.

This information is gathered because it is required for computing the object equivalence groups or provides valuable information to propose a cache implementation for a group of equivalent objects.

Reflecting on the approach we followed to implement the optimizations proposed, we distinguish two main approaches:

6.1.1 Large equivalent object groups

Equivalent object groups are computed by grouping objects by their `soHash` value and then filtering out all groups with only one object. The analysis tool already provides the behavior to compute these groups. This enables the practitioner to process the data and focus the effort on known redundant objects.

Equivalence groups are computed by:

```
SOObjectGraph>>computeShareableNodes
  ^ (self nodes groupedBy: #soHash) reject: [ :n | n size = 1 ]
```

Largest groups correspond to immediate memory saving opportunities by introducing a cache. A single object is needed for each equivalence group, but the use of several cache implementations may be needed to reduce the bloat, one for each allocation context.

By definition, all objects in a group hold the same or equivalent state. This information is crucial for implementing the cache, because the implementation has to decide on returning a new object or a cached object. The condition used relies on the state of the object, and if avoiding its creation, the parameters given to the factory method.

6.1.2 Shareable object groups with shared allocation context

Some allocation contexts are responsible for the creation of many equivalent objects, but not all of them belong to the same equivalence group. The impact of storing a cache for each of the groups depends on the size of the objects. We have seen that this strategy may have a large impact on memory consumption, even when applied on small object groups, but that share allocation context with many groups.

[1] For the experiments, we found that using size 10 is enough for detecting objects that shares allocation context, but this parameter can be modified by the practitioner if needed.

In order to compute the objects that are created by the same context we take frames at the top of the stack and check for equality between the methods being executed. The number of frames works as a threshold, being size 1 the most permissive threshold, and infinite size the most restrictive one. For our analysis we used size 10 as our threshold, which allowed use to effectively find caching opportunities for several applications.

The objects grouped by their allocation context, using last n context frames only, is computed by:

```
SOAnalysis>>equivalentObjectsContextsOfSize: int
  ^ self equivalentObjects groupedBy: [ :o | o contextOfSize: int ]
```

In this case, one value must be cached per equivalence group to act as representative of the group. The implementation has to be able to dynamically decide which of the representative objects it should return. This task is usually achieved by analyzing the parameters given to the factory associated with and the contexts calling the factory that effectively return equivalent objects.

A good example of optimizations in this category is the use of hash-consing for String, implemented in `intern` method in Java[2] and `asSymbol` method in Pharo. String interning consists in having a pool of objects and avoids duplication by searching for previously created objects before attempting to create the new one. If the requested object is found to be present in the pool, the object in the pool is returned instead of creating the new object. All new objects which were duplicated in the pool are then garbage collected.

6.2 Classification of optimizations

This section presents the optimizations designed and implemented for this research. We classify the optimizations proposed in 4 categories. For each category, we offer a discussion about the usage situations for the fixes, a high-level description and an example of the optimization implemented for the experiment.

6.2.1 Single object cache (5 implementations)

Usage

The single object cache consists in a optimization that targets a single equivalence object group. The best use case for this optimization is a single object production site which only produces equivalent objects belonging to the target equivalent group.

Some other good use cases for this optimization are:

[2] `https://docs.oracle.com/javase/7/docs/api/java/lang/String.html`

- Having multiple allocation sites that only produce equivalent objects belonging to the same target equivalent object group.
- Having a reliable method to predict if the object to create belongs to the target equivalent object group.

We have implemented 5 caches of this category, belonging to Roassal2 and Nautilus applications.

High-level description

Consists in the implementation of a cache on the factory method that is responsible for creating equivalent objects. The cached object is stored in a variable, so it can be reused for later calls. If required, an *if* condition is implemented to test a property on the argument for deciding if returning the cached value or creating a new object is the desired action.

The following code is the basic structure of the Single object cache:

```
Factory>>factory: anArgument
  anArgument = targetArgument
    ifTrue: [ ^ cachedValue ]
    ifFalse: [ ^ self createNewObject: anArgument ]
```

Example (Nautilus)

Analyzing profiler output. Upon analyzing the data, we have found that the largest object group of domain objects hold `NautilusMethodSelection` instances. This class defines a single instance variable called `class` and all the objects in the value hold `nil` for the instance variable. The total number of equivalent objects belonging to this group is 7,672, which represents 3% of the total number of objects of the Nautilus benchmark, and 22% of the total number of equivalent domain objects.

The objects are allocated from 4 different contexts, while 2 of them are responsible for 99.9% of the objects. Both contexts share the factory method `NautilusMethodSelection class>>#method:`, and therefore, this will be the spot to implement the cache in.

```
NautilusMethodSelected class>>method: aMethod
  ^ self new method: aMethod
```

Proposal. To reuse instances that have `nil` as an instance variable, a cache is placed in the factory method. It includes a test for `nil` value to cache only those objects, ignoring the non-equivalent ones.

The changes are the following:

```
Announcement subclass: #NautilusMethodSelected
  instanceVariableNames: 'method'
  classVariableNames: 'NilMethod'
```

```
    poolDictionaries: ''
    category: 'NautilusCommon−Announcements'
```

```
NautilusMethodSelected class>>method: aMethod
    aMethod ifNil: [ ^ self nilMethod ].
    ^ self new method: aMethod
```

```
NautilusMethodSelected>>nilMethod
    ^ NilMethod ifNil: [ NilMethod := self new method: nil ]
```

This implementation returns the cached value each time a new instance with `nil` is attempted to be created. If an instance with another object is attempted to be created, the cache performs the original behavior of the application, then returning a new instance of the object.

6.2.2 Map of objects cache (2 implementations)

Usage

The map of objects cache is a strategy that targets multiple equivalence object groups. In order to be applicable, the equivalent objects must be permanent in the execution and be created in the same object production site.

The best use case for this strategy are in situations where the number of equivalent object groups is statically bounded, *i.e.,* the maximum number of groups does not depend on the execution. For example, the number of equivalent groups is determined by the number of classes of the application.

We implemented 2 caches belonging to this category. Both of the cases were found in Nautilus.

High-level description

Consist on the implementation a cache using a hash map. The map `key` corresponds to the lookup value, which is usually the argument of the object factory and the `value` corresponds to the cached object.

The following code is the basic structure of the Map of objects cache:

```
Factory>>cacheInitialize
    cacheMap := Dictionary new.
Factory>>factory: anArgument
    ^ cacheMap at: anArgument ifAbsentPut: [ self createNewObject: anArgument ]
```

Example (Nautilus)

Analyzing profiler output. Upon grouping the equivalence object group by class, we noticed that only 3 groups of `NautilusChanged` were present. The total number of equivalent objects on the 3 groups is 5,148 objects. Through analysis of the allocation context we found that all objects were allocated by calling `NautilusChanged class>>symbol:`.

```
NautilusChanged class>>symbol:
  ^ self new
    symbol: symbol;
    yourself
```

The 3 different groups were generated by giving different arguments to the method, more concretely `#sourceCodeFrom:`, `#getHistoryList` and `#currentHistoryIndex`. Using static analysis we found that those symbols are hard-coded on Nautilus and represent concrete message names understood by `NautilusUI` instances. The size of the cache is then statically bound, ensuring a safe maximum size for the cache.

The method responsible for creating `NautilusChanged` instances and also send the message to `NautilusUI` is `NautilusUI>>changed:`:

```
NautilusUI>>changed: aSymbol
  super changed: aSymbol.
  self announce: (NautilusChanged symbol: aSymbol)
```

Also some senders of `changed:` message are:

```
AbstractNautilusUI>>#classRenamed: anAnnouncement
  ...
  self changed: #sourceCodeFrom:
  ...
```

```
NautilusUI>>historyChanged
  ...
  self changed: #getHistoryList.
  self changed: #currentHistoryIndex
```

Proposal. To reuse `NautilusChanged` instances we implement a cache on `NautilusChanged class>>symbol:` that use a dictionary to store pairs `Symbol -> NautilusChanged`. The cache is stored in a class variable called `pool` and is accessible through an accessor by the whole object space.

```
Announcement subclass: #NautilusChanged
  instanceVariableNames: 'symbol'
  classVariableNames: 'NautilusChangedPool'
  poolDictionaries: ''
  category: 'NautilusCommon-Announcements'
```

```
NautilusChanged>>pool
  ^ NautilusChangedPool ifNil: [ NautilusChangedPool := Dictionary new ]
```

```
NautilusChanged>>symbol: symbol
  ^ self pool at: symbol ifAbsentPut: [
    self new
      symbol: symbol;
      yourself ]
```

6.2.3 Weak map of objects cache (5 implementations)

Usage

The weak map of objects is used to cache objects that are not permanent in the execution or have a short life. Even though they have lower memory consumption impact than permanent objects, they may still cause significant impact on memory consumption or time performance, caused by excessive object creation and excessive garbage collection.

This cache is used when the map of objects cache is not applicable. Storing cached objects in a map prevents their garbage collection, which may cause severe performance problems when used improperly. To provide a memory-leak safe approach, the weak map of objects cache uses weak references, which do not prevent object garbage collection.

This optimization is used to implement unique `String` values, named `Symbols` in Pharo.

High-level description

Similarly to the map of objects cache, it consists of implementing a cache on the factory method that is responsible for creating equivalent objects. The objects are cached in a weak value map, which holds arguments of the factory as keys, and weakly holds the cached object as values.

The following code is the basic structure of the Weak map of objects cache:

```
Factory>>cacheInitialize
  cacheMap := WeakValueDictionary new.
Factory>>factory: anArgument
  ^ cacheMap at: anArgument ifAbsentPut: [ self createNewObject: anArgument ]
```

Example (NeoCSV)

Analyzing profiler output. Upon analyzing the data, we have found that most equivalent objects are instances of `String`. The total number of equivalent `String` objects is 50,856, distributed in 4,717 equivalence groups. Moreover, all equivalent `String` instances are created by the same context.

`String` equivalent object instances represent 67.0% of the total number of objects in NeoCSV.

The following method is responsible for introducing the redundant arrays:

```
readFieldAndSeparator
  | field |
  field := self readField.
  self readSeparator.
  ^ field
```

Proposal. In order to reduce the number of duplicated `String` instances we will use `Symbol`, a unique string value. `Symbol` class holds an implementation of the weak map object cache, which stores unique string values.

The unique value may be obtained by sending the message `#asSymbol` to a string object. The implementation of it in NeoCSV is the following:

```
readFieldAndSeparator
  | field |
  field := self readField asSymbol.
  self readSeparator.
  ^ field
```

After the optimization, the method `#readFieldAndSeparator` returns unique string values, removing `String` duplication.

6.2.4 Others (2 implementations)

Prototype in Roassal2

Upon analyzing the profiler report, we found a large equivalence group of 6,947 instances of `Rectangle`. The rectangle definitions are (-2.5 @ -2.5) corner: (2.5 @ 2.5).

All these rectangles are defined within the same context:

```
TRAbstractBoxShape>>#fromRectangle:<66>
TRAbstractBoxShape>>#fromRectangle:color:<24>
RTEllipse>>#updateFor:trachelShape:<88>
RTShape>>#trachelShapeFor:<29>
...
RTShapeBuilder>>#elementsOn:<43>
```

Looking at those five methods, one can conclude that the rectangles correspond to the computation of the encompassing rectangle of a default element. Default width and height defined for `RTEllipse` is set to 5, which corresponds to the size of the equivalent rectangle.

Unfortunately, computing the encompassing rectangle is performed on every single `RTEllipse` based on the current status of the shape, preventing the value from being cache. This value

is then written on another object `TREllipseShape`.

```
RTEllipse>>updateFor: anElement trachelShape: trachelShape
   | ex rec |
   trachelShape isNil ifTrue: [ ^ self ].
   ex := (self widthFor: anElement) @ (self heightFor: anElement).
   rec := (anElement position − (ex / 2)) extent: ex.

   trachelShape fromRectangle: rec color: (self colorFor: anElement).
   trachelShape strokeWidth: (self borderWidthFor: anElement).
   trachelShape strokePaint: (self borderColorFor: anElement).
   ^ trachelShape
```

To avoid computing the same value, producing duplicated objects, we use a prototype of the `TrachelShape`, *i.e.,* copies of the whole object with the precomputed values. The value needs to be recomputed only if parameters of height, width or position, are different from those used in the prototype.

```
updateFor: anElement trachelShape: trachelShape usingPrototype: trShapePrototype
   | ex rec newWidth newHeight newPosition|
   trachelShape isNil ifTrue: [ ^ self ].
   newWidth := self widthFor: anElement.
   newHeight := self heightFor: anElement.
   newPosition := anElement position.

   ((self widthFor: trShapePrototype) = newWidth and: [
      (self heightFor: trShapePrototype) = newHeight and: [
         trShapePrototype position = newPosition ] ])
      ifFalse: [
         ex := newWidth @ newHeight.
         rec := (anElement position − (ex / 2)) extent: ex.
         trachelShape fromRectangle: rec.].

   trachelShape color: (self colorFor: anElement)  .
   trachelShape strokeWidth: (self borderWidthFor: anElement).
   trachelShape strokePaint: (self borderColorFor: anElement).
   ^ trachelShape
```

This approach, under the right circumstances, avoids the recomputation of geometric values by using a prototype object. Then, a large amount of `Rectangle`, `Points` and `Float` objects are no longer recomputed, and hence, duplicated.

Collection use in PetitParser

Upon analyzing the profiler report, we found that the second largest group of equivalent objects were instances of Dictionary. There were 3,000 instances, each of them hold an `Array` instance of size 5, and each array holds a single `Association` instance.

Furthermore, all Dictionaries were created by the same context and the responsible method is the following:

```
PPContext>>propertyAt: aKey put: anObject
```

```
  "Set the property at aKey to be anObject. If aKey is not found, create a new entry for aKey and set is value
    to anObject. Answer anObject."
  ^ (properties ifNil: [ properties := Dictionary new: 1 ])
    at: aKey put: anObject
```

Unfortunately, we were not able to cache the dictionary instances. Even though the benchmark continued working, the tests of PetitParser began to fail. Despite that, we were able to optimize the application by reducing empty space in the inner capacity of the Dictionary and using a Collection more suitable for small capacity, such as SmallDictionary.

The changes are the following:

```
PPContext>>propertyAt: aKey put: anObject
  "Set the property at aKey to be anObject. If aKey is not found, create a new entry for aKey and set is value
    to anObject. Answer anObject."
  ^ (properties ifNil: [ properties := SmallDictionary new ])
    at: aKey put: anObject
```

This approach does not reduce the number of Dictionaries, instead it reduces empty space on inner arrays. The impact of this change is a reduction in memory consumption of 7.0% in our benchmark.

6.3 Optimizations performed

As part of the experiment we analyzed the profiler output for the 6 application benchmarks. Using this data, we proposed and then implemented 14 different optimizations for 5 of the applications. The optimizations were proposed without previous knowledge of the applications, but using only the profiler output, the benchmark code and the tests for checking soundness of the changes.

Table 6.1 shows the distribution of optimizations per application and per category. The application with the highest amount of implemented optimization is Nautilus, with 9 implementations. In turn, the categories with the highest number of optimizations are single object cache and weak map of objects cache, each of them with 5 implementations.

Table 6.2 presents the time spent on obtaining the optimizations for the applications. In total we spent 14 hours and 4 minutes to obtain all optimizations. Analyzing the data and identifying concrete optimizations opportunities are the most time consuming tasks, consuming in total 12 hours. In contrast, profiling and implementation time are responsible for less than 20% of the total time spent for producing the optimizations.

The applications that consumed most of the analysis time were Roassal2 and PetitParser. This is explained by the fact that both projects have memory bloat problems that we were not able to classify into the previous memory optimization strategies. Because of that, each case has to be analyzed from scratch and a new solution has to be designed, which leads to a higher analysis time.

Table 6.1: Classification of implemented optimizations for the applications in benchmark suite.

	Single object cache	Map of objects cache	Weak map of objects cache	Others	Total
Roassal2	1	-	-	1	2
Nautilus	4	2	3	-	9
SciSmalltalk2	-	-	-	-	0
NeoJSON	-	-	1	-	1
NeoCSV	-	-	1	-	1
PetitParser	-	-	-	1	1
Total	5	2	5	2	14

Table 6.2: Time spent for producing optimizations for the applications in benchmark suite.

	Profiling Time	Analysis Time	Implementation Time	Total
Roassal2	3m	4h	40m	4h 43m
Nautilus	2m	2h 30m	45m	3h 17m
SciSmalltalk2	19m	-	-	-
NeoJSON	1m	45m	<5m	51m
NeoCSV	5m	45m	<5m	55m
PetitParser	13m	4h	<5m	4h 18m
Total	0h 43m	12h	1h 40m	14h 4m

Chapter 7

Feedback from developers

In our effort to evaluate the optimizations obtained by the use of our tool, we approached two developers to elucidate our concerns. We performed an interview with one of the Roassal developers where we compared the optimization designed by us with the optimization designed by an expert. Also, we asked contacted the developer of NeoJSON and NeoCSV for feedback about our proposed optimizations in both applications.

7.1 Roassal Case

We choose the Roassal visualization engine to be part of our benchmark. As mentioned in Table 5.2, 55.5% of the objects created by Roassal are equivalent. From the 14 optimizations we implemented in our benchmark, two are in Roassal and reduce the number of objects by 33.1%. These two optimizations were implemented without prior knowledge about the internals of Roassal. Therefore they are implemented solely using the indications provided by our profiler.

To verify whether the optimizations are (i) easy to implement and (ii) relevant for the Roassal application, we interviewed one of its main developers, Milton Mamani. Note that the developer was not aware of our effort on object equivalence when we conducted the interview.

The methodology for the interview is to ask the developer to identify and remove the most important causes of memory bloat. When the developer arrives to a dead-end or get stuck, a new piece of information provided by the profiler is then handed to the developer. This way, we gradually disclose information to the developer to evaluate the relevance and effectivity of each logic step of the report provided by the tool.

Beginning of the experiment. We provided the developer with the exact execution of Roassal we use in our benchmark. The developer was then asked to spot the greatest sources of memory redundancy. Pharo has execution and memory profilers and the developer is well aware of them. Still, he could not identify which classes produce redundant objects.

First optimization. We then disclosed the largest source of equivalent objects, without disclosing any additional information about the objects creation. We simply indicated that a large amount of arrays containing four float numbers were heavily duplicated in pairs in the execution. In other words, for every array of such characteristics there was a second one exactly duplicated. Because the creation of array is within a highly used method, these arrays have a significant impact on the memory consumption. The developer narrowed his search compared with the previous phase of the interview, but he was still not able to spot the precise location where the bloat was produced.

Finally, we disclosed the production site, *i.e.*, concrete method, of the duplicated arrays.

```
TRShape>>transformedEncompassingRectangle
  "Not all shapes are alredy using matrix.
  For that reason we have this method."
  | basicRectangle rotatedRectangle topLeft topRight bottomRight bottomLeft p1 p2 p3 p4 |
  basicRectangle := self basicEncompassingRectangle.
  topLeft := matrix transform: basicRectangle topLeft.
  topRight := matrix transform: basicRectangle topRight.
  bottomRight := matrix transform: basicRectangle bottomRight.
  bottomLeft := matrix transform: basicRectangle bottomLeft.

  p1 := OrderedCollection new add: topLeft x; add: topRight x; add: bottomRight x; add: bottomLeft x;
    yourself.
  p2 := OrderedCollection new add: topLeft y; add: topRight y; add: bottomRight y; add: bottomLeft y;
    yourself.
  p3 := OrderedCollection new add: topLeft x; add: topRight x; add: bottomRight x; add: bottomLeft x;
    yourself. "p3 duplicate p1"
  p4 := OrderedCollection new add: topLeft y; add: topRight y; add: bottomRight y; add: bottomLeft y;
    yourself. "p4 duplicate p2"

  rotatedRectangle :=
    (p1 min @ p2 min) corner:
    (p3 max @ p4 max).
  ^ rotatedRectangle
```

Using this new piece of information, the developer produced an optimization in less than 5 minutes. We proposed the same optimization than the developer a few days in advance of the interview without having extensive knowledge of Roassal internal behavior.

The changes performed are the following:

```
TRShape>>transformedEncompassingRectangle
  "Not all shapes are alredy using matrix.
  For that reason we have this method."
  | basicRectangle rotatedRectangle topLeft topRight bottomRight bottomLeft p1 p2 |
  basicRectangle := self basicEncompassingRectangle.
  topLeft := matrix transform: basicRectangle topLeft.
  topRight := matrix transform: basicRectangle topRight.
  bottomRight := matrix transform: basicRectangle bottomRight.
  bottomLeft := matrix transform: basicRectangle bottomLeft.

  p1 := Array with: topLeft x with: topRight x with: bottomRight x with: bottomLeft x.
  p2 := Array with: topLeft y with: topRight y with: bottomRight y with: bottomLeft y.
```

```
    rotatedRectangle :=
        (p1 min @ p2 min) corner:
        (p1 max @ p2 max).
    ^ rotatedRectangle
```

This piece of code was written 2 years before this interview and, in the opinion of the expert, the author focused on reusing the `min` and `max` messages of `SequenceableCollection`. We then believe that he missed that he had copies of *p1* and *p2*.

Second optimization. We repeated the interview with the second optimization. The second bloat memory problem consists in having a large amount of duplicated `Float` values. These objects are stored in a `TransformMatrix` container, but the values are created in a different method defined in another class. This other method is not apparent at first sight, *i.e.,* the method creating the matrix has many incoming dependencies. Therefore, identifying the responsible method that created the float values is not a trivial task.

Our expert did not identify the source of memory bloat, even after indicating which method is producing the large amount of matrices. We therefore provided a copy of the stack trace that triggers the memory bloat, which happens to be key for the expert to design an optimization. He proposed a high-level solution to the bloat, but he was not specific enough to compare it with the solution we implemented.

However, the expert upon analyzing our proposed solution, was able to conclude that it solved the bloat, but was not optimal. We did not spot this ourselves. To spot this, the expert relied on the knowledge of being working on a pure method, *i.e.,* a method that do not cause side effects and always return the same value when called with the same arguments. This knowledge is not gathered by the profiler.

The bloat problem and the solution proposed is described in detail in Section 6.2.4.

7.2 NeoJSON and NeoCSV Case

NeoJSON and NeoCSV are two applications selected to be part of our benchmark. As mentioned in Table 5.2, the number of equivalent objects present in the benchmarks for NeoJSON and NeoCSV are 47.0% and 67.1%, respectively.

As part of our research we analyzed a representative execution of each application with our profiling technique. Using the results, we proposed one optimization for each of the applications resulting in a reduction in memory consumption of 21.7% for NeoJSON and 45.1% for NeoCSV.

With this result, we contacted the person responsible for both applications Sven Van Caekenberghe, through email, where we gave to him:

- Achieved memory consumption reduction with the benchmark.

- Source code of the optimizations performed in `mcz` files.[1]
- High-level description of the fixes.

The main objectives are to know whether the author was aware about the memory bloat, received feedback about the soundness of our changes and to validate that our profiler output enabled a developer without prior knowledge in the application to suggest optimizations.

NeoJSON. The application is a parser of JSON data, which is responsible for transforming the data to native Smalltalk objects, such as Dates, Numbers, Arrays, Dictionaries and others.

Sven's feedback about this application discusses three ideas: (1) The representativity of the benchmark. (2) That the solution holds a good best case scenario, but has bad worst case scenario.(3) The increase of complexity caused by our solution.

Our proposed solution consists in using unique string values in JSON entries, then avoiding string duplication.

About the first idea, our experiment consists of a single benchmark for NeoJSON. Though our benchmark represents a usual use case, it does not represent all possible situations.

The second concern, while related to the first one, focuses on the idea that using unique values may be costly if all JSON string entries are different. This case could happen for special types of entries, such as addresses or timestamps. However, Sven addresses this issue by implementing selective usage of unique values, *i.e.,* for certain JSON keys the string value should be parsed in a different way, such as when using an Address object, Timestamp objects, or non-unique String objects, instead of using unique Strings for all values. This can be addressed by implementing `#neoJsonMapping:` in the domain objects, specifying then which values should be cached and how they should be manipulated.

The third concern is related to software complexity. Our proposed solution relies on a flag for using unique values or not. Sven proposed using the existing API `#neoJsonMapping:` to define selective behavior of the parser for JSON values, without adding complexity to the application. This avoid adding flags and instance variables in the code for each new case we want to cache and also allow the developer to introduce different strategies for the same type of objects *e.g.,* caching first names, but not last names.

Though our solutions are not optimal from a design perspective, they do not introduce bugs and effectively reduce memory bloat. We were also able to validate that for some inputs the applications show room for improvements, but also learnt that for these kind of applications flexibility and extensibility is a requirement for optimizations.

NeoCSV. The application is a parser of CSV data, which is responsible for transforming it into native Smalltalk objects.

Our proposed solution is similar to the NeoJSON solution and use unique values on string cells. Instead of storing Strings we store the unique Symbol value.

[1]Mcz files are standard source code format of Monticello, the most used application for version control systems in Pharo.

Sven's comments focus on the idea that the user is the one responsible for deciding if using unique values or not. Even though String instances are immutable objects, the use of unique values requires the system to hold a hash table with all unique Symbols. This approach is memory efficient when a high amount of duplicated Strings are used, but performance issues arise when most of the objects are unique.

Sven detailed that NeoCSV API currently supports per-column use of unique values for Strings. The method `#addSymbolField:` use unique values on the records under the column specified as the argument. This method mimic the behavior of our optimization proposal, then this memory bloat case was already addressed by Sven, but is not used as default and its use was not part of our benchmark.

Similarly to NeoJSON, we were able to validate the existence of memory bloat. Despite that, the profiler was not useful for realizing if the the functionality was already implemented, but not used.

7.3 Summary

We approached one of the developers of Roassal2 and the developer of NeoJSON and NeoCSV to validate our proposed optimizations.

The small case study on Roassal2 illustrates the relevance of our code execution analyzer to identify optimization opportunities. Using the memory footprint of an execution, we were able to identify an optimization that was not immediately obvious to a knowledgeable programmer. We do not claim any generalization caused on this experiment. As future work, we will carry out a larger experiment with practitioners.

We then requested feedback from Sven Van Caekenberghe about the proposed optimizations for the applications NeoJSON and NeoCSV. Sven validated the existence of memory bloat and also recognized the value of the knowledge provided by our tool.

The main conclusions we drew from these approaches are: First, our tool shows that our analysis is very sensitive to the benchmark, such as the use of addresses or timestamps in NeoJSON. Second, that the knowledge provided by the tool is enough to propose a solution that reduces bloat, but non optimal in design, hence unnecessarily increasing software complexity.

Chapter 8

Implementation

This chapter presents relevant aspects of the tool implementation. The implementation of our technique can be summarized in 3 major steps: (1) Building of a profiler that captures necessary data for evaluating object equivalence. (2) Replication of execution object graph annotated with required data for analysis. (3) Graph analysis for evaluating object equivalency and grouping redundant objects.

Our profiler and analysis tool is available under the MIT license.[1]

8.1 Profiling Technique

The profiler is built using the Spy2 profiling framework [1] in the Pharo programming language. We choose Pharo because of the expressive reflection API which allowed us to instrument executions with an affordable overhead and without using a customized runtime or virtual machine.

The profiler captures relevant data regarding the allocation of objects, the modification to the state of the objects and the garbage collection of objects. For this the profiler uses the following capabilities of the Pharo reflective API:

- *Method instrumentation:* Inject behavior around method execution.
- *Slot instrumentation [14]:* Inject behavior around instance variable accessing.
- *Identity operations:* Call virtual machine to distinguish objects, based on object identity.
- *Garbage collection hooks:* Run arbitrary behavior after an object is garbage collected.
- *Execution context reification:* Enable reflective access to execution context, allowing the profiler to record the stack trace and sender objects.
- *Object reflective reading:* Allow profiler to access object state with no accessors and without disturbing normal execution.

[1] Available under the project ainfante/ShareableObjects on `http://smalltalkhub.com`.

- *Process Variables and Semaphores:* Usage of several mechanisms to ensure thread-safe operations, enabling the analysis of multi-threaded applications.

8.1.1 Events and strategies for data collection

This profiling technique defines events as particular execution moments that are required for this research. This section presents which events are interesting to capture, the gathered data for each event and the strategy used by the profiler to capture the data.

Allocation of Domain Objects. To capture the allocation of domain objects, the profiler use a method instrumentation technique. The profiler instruments `basicNew` methods in all domain classes to capture primitive calls to object allocation. Every time a domain object allocation is recorded, the profiler stores: allocation stack trace of reduced size, obtained by context reification; a weak reference to the object associated to a unique id; the time of the system.

Allocation of System Specific Objects. To capture system specific objects the profiler instruments all methods. For each method, the profiler replaces the classes references of system specific object classes for modified classes. (*e.g.,* Set → S2Set). This set of modified classes are thought to capture object creation and state modification of those objects. Every time a system specific object allocation is recorded, the profiler stores: allocation stack trace of reduced size, obtained by context reification; a weak reference to the object associated to a unique id; the time of the system.

First time seen Objects. Object creations that are not visible to the profiler, because they may be created before the instrumentation, received as an argument or returned as a value by an external call. This applies especially to system specific objects, which are usually created by calling external resources, *e.g.,* float values allocated because of matrix operations. Object copies also fall into this category, because `copy` method relies on a different primitive and mechanisms to generate a new object. In contrast to objects whose allocation has been observed, first time seen objects are not assumed to have `nil` references. These objects are then recorded with their initial state. When the profiler encounters objects whose allocation was not recorded, it generates a special record to marks them as "First time seen object" and stores: current state of the object when encountered; a weak reference to the object associated to its unique id; the time of the system.

Object State modification of Domain Objects. The profiler implements a Slot [14] wrapper for intercepting every instance variable writing of domain objects. Each time an instance variable is written the profiler records: the previous instance variable unique id; the new instance variable value unique id; the time of the system when the instance variable was written.

Object changes of System Specific Objects. System specific objects that allow changes are replaced by customized classes. Each of these classes include calls to the profiler each time the object change, according to the defined invariant for the object. When an object changes,

the profiler records: the kind of change performed (*e.g.,* element added to a collection); unique ids of the involved objects; the time of the system when the change occurred.

Capture of Identity Operations. To capture identity operations the profiler instrument the methods that rely on object identity (*e.g.,* `identityHash`). Each time an identity method is called the profiler records: unique id of the receiver of the identity method; time of the system when the identity method is called.

Object Garbage Collection detection. Each time an object allocation is recorded or seen for the first time, the profiler stores a weak reference in a WeakRegistry with an executor.[2] Each time an object is garbage collected in Pharo and is registered in the WeakRegistry, the executor is called to gather the required data. The profiler records: unique id of the garbage collected object; time of the system when the object is garbage collected.

8.1.2 Architecture of the profiler

The profiler architecture is based on recording events and maintaining the object unique id - definition lookup table. Each time a relevant event, described in Section 8.1.1, take place in the execution, the profiler generates an event that is pushed into an event stream to be processed. By default, the stream only stores each individual event in memory until the end of the execution.

The events captured by the profiler are represented as instances of a class hierarchy of events, one class for each kind of event. Each event, at the very least, references an object in the execution, for the which it stores the object's unique id, and the time the event takes place. Some events require additional data, like the unique ids of the previous and new values of an instance variable modification. But in no case will an event reference an object of the execution or the profiler. Therefore, events may be serialized out of the Pharo image to be analyzed by other tools. A detailed documentation of profiler events is provided at Appendix Section A.

A special record called ObjectDefinition is recorded each time a unique id is associated to an object. This is responsible of storing the class of the object and some other specific properties for System Specific Objects (*e.g.,* characters for String objects, numeric value for Float objects).

8.1.3 Unique id specification

Weak object - unique id lookup table

The unique id is an integer given by the profiler that identifies a unique object in the execution. In order to record any information of the execution, the profiler retrieves the unique id of all

[2]Object that performs an arbitrary behavior.

objects involved.

Our profiler then holds a large identity map, *i.e.,* a map that performs the lookup using identity hash and identity test instead of normal hash value and equals method. This map contains all the non-primitive objects of the execution as keys and their corresponding unique id as value.

The three main properties of this structure are the following: First, the map uses only weak references to hold objects, allowing them to be garbage collected. Second, querying the id associated to an object should be fast. A performance analysis of our profiling tool showed that the main bottleneck of the execution is querying object ids. Lastly, the table should be cautious about memory consumption, enabling shrinking when large amounts of objects are garbage collected in the execution.

Because Pharo libraries do not provide a map with all the previous requirements, we implemented our own map called S2WeakLargeIdentityDictionary. Our structure is largely inspired by FLLargeIdentityDictionary, written by Mariano Martinez for Fuel application [5]. This structure takes into account the limited 12-bit identity hash, which is responsible of causing high amounts of collisions in the map. The structure then sets up the initial hash value space to 12-bits, assigning each slot in an Array to a single hash value.

We also took inspiration from "Hash table chaining" to handle collisions. FLLargeIdentity-Dictionary implementation holds an expandable array for each hash value to store elements that have hash collisions. Unfortunately, this approach is not optimal for maps which are prone to object removal, such as the effect of the garbage collector over weak references. To address this issue, a LinkedList is used instead of an expandable array. The compaction of the map is done while looking up an element with a minimum overhead, granted by the LinkedList advantage of constant time element removal operation.

The map S2WeakLargeIdentityDictionary greatly outperforms Pharo WeakKeyIdentity-HashTable when used with large amounts of objects (>100,000 objects). We performed an experiment creating a dictionary with 200,000 elements and performing 1,000,000 queries. Our profiler outperformed standard Pharo implementation with a speedup of 1.75×.

Generation of unique ids

The unique id is an integer that is used to reference a particular object in the execution. The requirements for this value are: It must be fast to generate, fast to compare and have little impact on memory consumption.

For these reasons, the unique id is a SmallInteger. In Pharo, SmallInteger instances are 31-bit integers. They are big enough to identify all the objects that normal execution could possibly generate. The profiler uses a thread-safe counter, that is increased by 1 each time a new unique id is registered, to generate new unique ids.

SmallInteger instances are not suitable to share the same unique id space as the rest of the objects and be stored on the table explained in the previous section. This is because

SmallIntegers are used intensively and are not object pointers, then they are not garbage collected. This dramatically increases the size of the map with SmallInteger objects slowing the queries.

The unique ids space is then divided in 3 sections:

Values from -2^{30} ***to*** $2^{29}-2$. Reserved for SmallIntegers between -2^{30} to $2^{29}-2$, where the unique id of an SmallInteger is the value itself.

Value $2^{29}-1$. Reserved for special object `nil`.

Values from 2^{29} ***to*** $2^{30}-1$. Reserved for normal objects and SmallIntegers with values over $2^{29}-2$. These are assigned using the technique described in this section and the values are stored in the map described in the previous section for normal objects. SmallInteger unique ids are stored on a different map for increased performance. The structure used for SmallInteger is the default map provided by one of Pharo's core libraries, `Dictionary`.

8.2 Object graph replication

After the execution is finished, the analysis tool reifies the resulting object graph of the execution by iterating over the records. Also, all objects that have instance variables written more than once are marked as unchanged.

A directed labelled graph is implemented to represent the references of the objects of the execution. In the model, the vertexes or nodes represent objects and the edges represent instance variables. Labels on edges then represent instance variable names.

This implementation defines a concrete API to iterate over nodes and edges, and perform queries. Following the object oriented paradigm, the graph nodes are implemented using a class hierarchy matching the semantics of the object characterization described in Section 4.2, allowing developers to create visitors for more advanced and customized analysis.

8.3 Graph analysis and object equivalence evaluation

Computing the object equivalence group of an object a is reduced to compute all the isomorphic sub-graphs to the reachable sub-graph starting from a. This problem is equivalent to Subgraph Isomorphism Problem, a known NP-complete problem [15].

The main difference between our problem and the traditional Subgraph isomorphism problem is that the edges of the subgraphs have to include all the edges defined between the objects, *i.e.,* the subgraph may not ignore instance variables.

Formally this is defined as:

Let $G = (V, E)$ and $H = (V', E')$ be two graphs, such that $a \in V, V' = reach(a) \subseteq V, E' = (V' \times V') \cap E$. Is there a subgraph $H_0 = (V_0, E_0) : V_0 = reach(a_0) \subseteq V, a_0 \in V, E_0 = (V_0 \times V_0) \cap E$ that $H \cong H_0$? Letting $reach(a')$ be the enumeration of reachable elements from a' using DFS.[3]

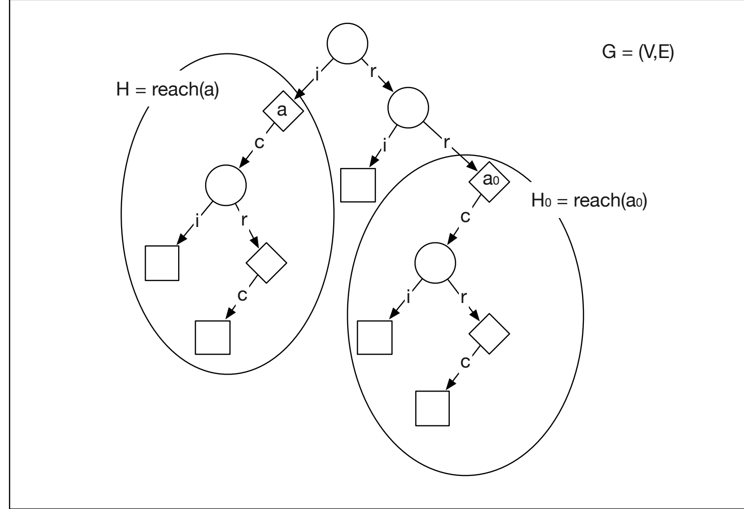


Figure 8.1: Illustration for simplified Subgraph isomorphism problem for Object Equivalence.

Figure 8.1 shows the simplified subgraph isomorphism problem for Object Equivalence, showing that the subgraphs generated from $reach(a)$ and $reach(a_0)$ are isomorphic graphs.

It is then possible to observe that a and a' satisfy the conditions regarding the shape of the object graph, and then, $a \approx a_0$ iff a and a_0 are unchanged, and a and a' identities have not been exposed.

To address this problem we introduce a hash value for each node we call *soHash*. In contrast to the *unique id* value, *soHash* identifies object equivalence groups. Formally, this value is built to satisfy the following property:

$$soHash(a) \neq soHash(b) \implies a \not\approx b$$

The *soHash* function is a recursive compression collision-resistant function. It is built by expressing the object equivalence relation property as a concatenation of Strings and then compressing them using MD5 collision resistant function.

If object a is unchanged and is not part of a cycle:

$$soHash(a) = MD5(soHash(a.1) + ... + soHash(a.n) +' \#' + className(a))$$

If the object a is unchanged, but is part of a cycle where all the objects are unchanged: Let $scc(a) = (\mathrm{e}_1, ..., \mathrm{e}_n)$ the enumeration of the reachable objects from a using DFS.

[3]Depth-first search algorithm.

```
soHash(a):
  stream = Stream new
  for e in scc(a) do:
    for i in instanceVariables(e):
      if ( e.i in scc(a) ):
        then:
          stream ← scc(a).indexOf(e.i)
          stream ← "#"
          stream ← className(e.i)
        else:
          stream ← soHash(e.i)
  return MD5(stream)
```

Otherwise, in case a is not unchanged or a identity is exposed:

$$soHash(a) = MD5(uniqueId(a))$$

Because MD5 is collision resistant, we then conclude that two objects sharing the same $soHash$ value are equivalent between one another with high probability:

$$[a] = \{o \in O | soHash(o) = soHash(a)\}$$

The amortized cost of computing *soHash* for the whole object graph using dynamic programming is linear on the number of edges of the graph, except for the strongly connected components, which raises to quadratic on the number of edges inside the strongly connected component. Fortunately, the number of strongly connected components is small and the size of strongly connected components is also small. For the cases we face on this research, the algorithm behaved as linear cost on the number of edges in time.

8.4 Implementation difficulties

The architecture of the tool evolved during the research to satisfy the requirements that arose during the research. An example of this is the hierarchy of nodes that is used for the graph structure, which forced us to implement the visitor pattern to support several operations and compute values over the graph.

Regarding the implementation of the tool, the biggest difficulties were debugging errors raised on instrumented code and increasing time performance of the analysis.

Debugging. Using a debugger is an especially challenging task when applied to profilers, because their behavior consists of modifying the behavior of another program. Sometimes, a program, trying to recover from an error caused by the profiler, may enter in an infinite loop because the code responsible for recovering is also instrumented and causing another error.

To solve this, the profiler isolates the computation performed by the profiler from the original execution. This is achieved by using recursion locks and exceptions handlers. This

behavior ensures that the profiler does not instrument itself and that any exception raised inside the profiler is properly handled to not disturb the normal execution of the profiled execution.

Performance. Regarding the performance of the analysis, three major improvements were performed.

First, the implementation of a customized weak identity dictionary, described in Section Section 8.1.3, greatly boosted profiler performance. This bottleneck was found using a sampling technique with Inti Visualization [10], manual analysis of the sampling output to aggregate the profiler hot spots and manual review.

Second, the solving of a bug in the Moose implementation of Tarjan algorithm used to compute strongly connected components for arbitrary graph structures. Our profiler uses the Moose-Algos library to find strongly connected components. The problem was that the translation of a custom graph structure to the Moose graph structure caused an increase in the algorithm implementation complexity from $\mathcal{O}(|V| + |E|)$ to $\mathcal{O}(|V||E|)$. The cost of applying a quadratic algorithm implementation over our graph is unaffordable, increasing the execution time of some benchmarks from 15 minutes to more than 10 hours. We then implemented a bugfix which allows a translation for arbitrary structures to Moose structures in $\mathcal{O}(|V| + |E|)$, then reducing the total complexity to $\mathcal{O}(|V| + |E|)$.

Third, we implemented a hashing function to compute the redundant groups, explained in Section 8.3. The provided approach has a linear complexity on time that allows the tool to compute the required property in an affordable amount of time.

Validity of the measurements. A major concern about the implementation of a profiling tool is that the computation performed by the tool do not mess the measurements. This tool uses the same object space than the application being analyzed. Because of this, we followed some directives in order to ensure the correctness of the data.

First, we use implemented computation-safe calls that ensures that the computation performed inside the call is not analyzed or recorded by the profiler. All the computation performed by the profiler is then performed into a computation-safe call, avoiding the profiler tools to analyze themselves. The sole exception to this rule is the computation-safe calls implementation itself, which has been manually reviewed thoroughly.

Second, we avoid preprocessing the gathered data meanwhile the tool is gathering data from the benchmark. Even though this would reduce the memory consumption of the tool, this would introduce non-deterministic delays when a piece of data is recorded.

Third, we performed regular manual reviews on the gathered data of micro-benchmarks to ensure that no objects created by the analysis tool were recorded and that no objects of the benchmarks were ignored.

Chapter 9

Discussion

This chapter discusses our results and contributions. First, we present the real impact of our analysis in the size of a real operating system process. Second, we present our contribution as a replication of Marinov *et al.* [9], where we compared our work similarities, differences and results with their work. Lastly, we discuss some limitations on our approach, which we have encountered in our research.

9.1 Memory impact on the system

The measurements reported in Table 5.3 and Table 5.4 are given from the application point of view: the size of an object is computed by summing up the size of the object's headers (accessible from the virtual machine) with the size of each instance variable.

From the point of view of the operating system, the memory consumed by the application is essentially expressed with the heap size, stack size, native resources and the virtual machine caches. Reducing the amount of created objects is likely to reduce the internal virtual machine caches and the memory allocation for the heap.

In the case of Roassal, the size of the process increased from 62.9 MB to 69 MB in the execution without our optimization. The same execution with our optimizations decreased the memory consumed by the process from 69MB to 65.2MB.

Analyzing the size of the process reveals a larger reduction in memory consumption. This is explained by the fact that virtual machines require extra memory to handle to support their features, such as object tables and caches. The additional impact of our technique in the process size is not further researched because the results will be strongly tied to the virtual machine implementation. The conclusions about the achieved reduction on the process size may not be generalized to other Pharo VM implementations or other languages VMs.

All experiments were carried out on a Macbook Air Mid 2013 with 4GB of RAM and 1.3GHz Intel Core i5 and using a clean Pharo Image with Moose 5.1.

9.2 Partial replication of Marinov *et al.* experiment

The work presented originates from the object equality profiling technique proposed by Marinov *et al.* [9] to identify objects that are redundant in an execution. Our research introduces the *shareability* relation that states the capability of an object to take the place of another. Our claim is that Marinov et al's *mergeability* relation and our *equivalence* relation are reductions of the *shareability* relation in order to provide a foundation for dynamic analysis tools.

Our research is similar enough to be called a partial replication study of their research. This section contrasts our research similarities, differences, agreements and disagreements with Marinov's approach.

9.2.1 Similarities

Individual object monitoring. Both approaches rely on instrumentation to keep track of every single object of interest in the execution. Both projects register object allocations and changes of state of objects.

Proposal of a recurrence relation. Both approaches proposed a recurrence relation to identify redundant objects. This property is then evaluated after finishing the execution by analyzing the resulting object graphs. The redundancy relation proposed in this research is called *Object Equivalence*, while the relation proposed by Marinov is called *Mergeability*.

Measurement optimization opportunities. Both approaches measure the potential impact of memory consumption reduction by identifying objects that satisfy the proposed redundancy relation. These research opportunities are then reflected by the number of *equivalent objects* and in the research of Marinov they are reflected by the number of *mergeable objects*.

9.2.2 Differences

Mergeability time. Marinov *et al.* identifies the time when an object becomes mergeable with others. In other words, from that time the object becomes redundant and is no longer needed in the execution. Each object has their own mergeability time, which may happen way after its creation. In practice, optimizing objects that become mergeable after their creation is difficult.

Our approach reduces its target scope to objects that are redundant during their whole life. We then prefer practicality over recall. Caches for redundant objects are easier to implement, because the optimization spots are also the allocation spots, which are recorded by the profiler and are made available to the practitioner.

Regarding objects that are not mergeable from the beginning, Marinov's paper does not provide any example of optimizations for these objects. Marinov *et al.*'s empirical case of study provided optimizations only for immutable objects (`Point` and `String`), objects that were mergeable during their whole life.

Classification of objects. In contrast to Marinov's research, our research does not equally consider all objects in the execution. In particular, the object equivalence relation as described in Section 3.2 is only applied to *domain objects*. We propose ad-hoc definitions for the most used and prone to be redundant objects. We call these objects *system specific objects*.

For these objects we proposed equivalence definitions according to the invariant of the objects. This has allowed us to identify and implement optimizations that can not be identified by Marinov's technique, *e.g.,* expandable collections whose internal changes of capacity used to be marked as not mergeable.

Allocation Context. Marinov's approach captures only the allocation spot when an object is created, *i.e.,* the concrete method and the program counter where the object is allocated. In many situations, we found that the allocation spot is not enough to propose a solution for the bloat identified by this research. For this reason, our technique captures a reduced stack trace of the allocation point.

Bytecode instrumentation. Marinov relies on a customized virtual machine for instrumenting the execution. Our approach does not modify the virtual machine, instead it relies on Pharo reflective API and bytecode instrumentation. Currently the Pharo virtual machine has been through several major releases resulting in significant advantages. Our profiler is able to profit from the changes to the virtual machine without significant effort. Not relying on a customized virtual machine reduces entry barriers and simplifies the maintenance of the tool.

Analysis API and Infrastructure. Marinov's paper does not detail how the data is analyzed by a human practitioner. This is a serious downside since we are not able to fully replicate the experiment or compare our approach with Marinov's approach. We provide a graph API and infrastructure to perform queries over the redundant objects in order to group them and retrieve relevant information.

9.2.3 Results and Conclusions comparison

There are two results that are comparable between the approaches. The first result is the potential saving opportunities, called *mergeable objects* in Marinov's research and *equivalent objects* in this research. The second result is the achieved memory consumption reduction by the implementation of optimizations proposed by the use of the output of the profiler.

Potential saving opportunities. Marinov *et al.* analyzed 9 benchmarks, from which 7 are included in SpecJVM98 benchmark[1] and 2 are widely used web application servers. Table 9.1 presents the average memory consumption potential saving for each benchmark.

[1]JVM98 is a set of benchmarks built to measure the performance of Java Virtual Machines.

Table 9.1: OEP Marinov's mergeability results

	Average Base Memory Consumption (MB)	Average Merge Memory Consumption (MB)	Save
db	7.6	4.38	42%
compress	4.83	4.77	1%
raytrace	3.38	1.91	43%
mtrt	5.44	2.25	59%
jack	0.44	0.25	43%
jess	0.91	0.81	11%
javac	4.77	4.35	9%
resin	6.23	3.03	51%
tomcat	2.18	1.63	25%
Total	35.78	23.38	35%

The average memory consumption potential saving in their research is 35%, meanwhile our technique shows a potential saving of 45% in average. Moreover, Marinov's approach was able to find a memory redundancy over 50% for only one fifth of the benchmarks, compared with our results where half of the benchmarks presented a potential saving over 50% with our approach.

It is not possible to compare objects that were marked as redundant in the research of Marinov *et al.* with our categories of *domain*, *system specific* and *external objects*. Despite that, our profiling technique treated 68% of the objects as System Specific Objects. These objects have a more adequate equivalence definition, hence explaining why our tool was able to find more redundant objects than the approach of Marinov *et al.*

Impact of proposed optimizations. Both studies used their own technique to propose and implement memory optimizations on parts of the source code of the benchmark projects.

We have attempted to optimize all of our benchmarks and have optimized 5 of 6 applications. In contrast, Marinov *et al.* chose 2 from the 10 applications to perform their case study. The reasons why these two applications were chosen instead of other applications is not detailed. They chose `mtrt` and `db`, being the 1st and 4th application with most optimization potential in their benchmark suite.

Marinov implemented only 2 optimizations, targeting instances of `String` and `Point`, being both classes of immutable objects. We found that their optimizations performed (hash-consing and memoization) are a subset of the implementations we proposed for our benchmarks. We also implemented other optimizations explained in Section 6.2.

The optimizations of Marinov for `db` and `mtrt` achieved a reduction in memory consumption of 47% and 38% respectively. Even though the average memory consumption reduction for our benchmarks was 25%, half of our benchmarks had a reduction in memory consumption over 40%.

Comparison of conclusions. Both approaches agree on the existence of optimization

opportunities caused by object bloat. We also agree on the fact that a dynamic analysis technique that keeps track of the state of individual objects contributes to propose optimizations that reduce the bloat caused by redundant objects.

Our analysis suggests that most of the optimization opportunities are related to system specific objects. This is partially supported in the research of Marinov by the fact that all optimizations proposed by them are related to system specific objects.

Finally, we do agree with the work of Marinov on the statement in their discussion that cyclic structures are possibly not relevant in the analysis of redundant objects. The statement is supported by the fact that the vast majority of the mergeability opportunities they have found do not depend on cycles. From all the benchmarks we have executed, we have not found optimization opportunities depending on cycles, which is consistent with their results.

9.2.4 Summary

We presented a partial replication of Marinov's approach explaining similarities, differences and a comparison of the results and conclusions.

Both present dynamic analysis techniques that monitor individual objects and their state to identify memory consumption opportunities. Both studies agree on the fact that applications have significant memory optimization opportunities that can be addressed by the use of this technique.

We then present some contributions on the topic in comparison to the approach of Marinov and discuss how they impact the overall results. In particular, our analysis is able to identify, on average, more redundant objects than Marinov's analysis.

9.3 Threat to validity

The main threat to validity is the benchmark suite used for the experiments. According to Wohlin *et al.*'s characterization for threats to validity, this case corresponds to a threat to conclusion validity.

We use benchmarks to represent real use case scenarios for the applications. Those scenarios have been built by the developers of the application or have been previously tested and recognized by several developers as representative.

The threat is that the results depends on the fact that the benchmark scenario is a representative execution of a real life use case of the application. If the execution is not representative of the application, the expected performance improvement would not be real and the proposed fixes are prone to produce bugs. We have addressed this threat by relying in the Pharo community, where users and developers are in constant communication. Therefore, Pharo developers know how their software is being used and are capable of building reliable

and representative benchmarks.

In contrast to our approach, Marinov and O'Callahan uses SpecJVM98 benchmark, which is a benchmark suite developed by Standard Performance Evaluation Corporation (SPEC), which was built to measure the performance of Java Virtual Machine.

The threat can be reduced further by using more executions for the presented applications, but there were not more public available benchmarks for these applications. We took special care to discard benchmarks that are built specifically for analyzing time performance and use repetition of a task for the benchmarks. This impact our analysis because n repetitions will create n equal objects, amplifying our results. We have discarded some applications such as Zinc, Mustache, Graph-ET, Soup, and others because of the lack of reliable benchmarks for the experiment.

9.4 Limitations of our approach

9.4.1 Limitations of Object Equivalence definition

Our object equivalence relation is effective in identifying relevant opportunities for optimization. However, we consider our approach restrictive. Currently, two objects are equivalent if they have the same class. But, it may happen that this condition may discard some valuable optimizations. Consider the following situation: Pharo has a class `Color` to model non-translucent colors and a subclass called `AlphaColor` to model translucent colors. Having two separate classes is required to accommodate different ways to render graphics on different platforms.

Currently, our technique prevents an alpha color to be equivalent to a color. Which is too restrictive since an opaque `AlphaColor` may indeed be equivalent to a `Color` object.

Another example is present in collections. An unsorted collection may be equivalent in some cases to a sequenceable collection. However, our approach forbids these objects from being equivalent in all cases. This limitation will be considered for future work.

9.4.2 Limitations of System specific objects

The reason for introducing *system specific objects* is the high amount of objects in the execution that are not part of the domain of the application. We have found that in all our benchmarks at least 50% of the objects are not part of the domain. Furthermore, in 5 of 6 benchmarks, more than 75% of the objects are not part of the domain.

Being able to propose a technique to include these objects in the analysis is then mandatory for achieving a significant memory consumption reduction. Instead of treating them as *domain objects*, we created a new category and an ad-hoc equivalence definition for each of them. The

equivalence definition is created in base of community standards and expected invariants, *e.g.,* float numbers are immutable. Unfortunately, the previous assumption is not enforced by the language.

In the set of defined System Specific Objects this problem is recurrent in `String`, `Float`, `Fraction`, `Point` and `Rectangle`.

Even though the average amount of System Specific Objects in our benchmarks is 74%, we have not faced any difficulty or experienced any bug introduction because of this issue. But we do not discard the possibility it may happen in the future because of an application that does not follow this assumption.

9.4.3 Tool design

Section 8.1.2 presents how the profiler handles the storage of the gathered data by the profiler. The tool is designed to register all state changes of domain objects and system specific objects related to the execution. But, this may cause problems in managing all the data in the same memory heap as the application, which could lead to unexpected performance issues, such as excessive garbage collection, or space issues like a heap overflow.

Our design allows further improvements on this topic by not requiring the gathered data to keep references to the execution objects. For example, it allows a complete serialization of the data to disk storage or database. This may improve memory use of the application and performance in case large executions require it.

Despite that, we have been able to execute our benchmark suite without problems and we did not have the need to implement the behavior described above.

9.5 Extensibility to other languages

Even though the profiler is built using Pharo reflective API, we believe it is possible to replicate this profiling tool in other languages, such as Java.

We have identified some constraints for extending our approach to other languages. The first constraint is the applicability of *object equivalence*. The definition is not applicable to all languages, but only to object orientes languages with classes. For instance, we believe that the current state of the research is not directly applicable to prototype object oriented languages, such as Javascript.

In order to build the profiler to operate in another language it is required to provide a mechanism to gather enough data for computing *object equivalence* and record objects allocation sites. Marinov and O'Callahan's dynamic analysis tool captures most of the required information for replicating our approach and only minor enhancements must be performed to satisfy our requirements.

We need the profiler to record the previous and following value of a mutation, not only a timestamp, as Marinov and O'Callahan's profiler does. And second, we require the profiler to record a reduced stack trace at each allocation, instead of just the allocation spot.

Another possibility to gather the required data would be using DiSL [8], which is a dynamic analysis framework for Java that relies in bytecode instrumentation.

The post-mortem analysis can be implemented in any language, and our Pharo implementation can be easily expanded to operate with data provided by an external tool.

Chapter 10

Related work

This section covers the work related to our approach.

Object equality profiling. Our work is inspired and based on Marinov *et al.* work [9]. We gave a formal definition of objects that can be merged after their initialization, discarding the time as a variable of the analysis. Our different formal definition allowed us to simplify the post-mortem analysis while increasing the recall of feasible optimization opportunities. This is supported by the 14 different optimizations performed which reduced the memory consumption by 25.1%.

Cachetor. Nguyen *et al.* [11] describe a technique that statically binds the cost of two extremely expensive dynamic analysis techniques such as *dynamic dependence profiling* and *value profiling*. In other words, the amount of data they collect and analyze does not scale on the size of the execution and is determined by the source code. This allowed them to provide an effective tool capable of proposing caching implementations to large applications that usually are not subject to this kind of analysis due to scalability problems.

Instead of relying on *value profiling analysis* which attempts to record computed values at each instruction, we rely on instance variable instrumentation and object creation instrumentation. This way, we reduce the amount of possible optimization spots only to object creation spots, allowing us to scale our analysis.

Object-sharing refactorings. Rama *et al.*'s work [12] is also based on Marinov *et al.*'s research. For this reason there are several similarities. Isomorphic definition is similar to our equivalence definition with three main differences:

- Our unchanging state condition is per instance variable, allowing an object to be lazy initialized, instead of enforcing strict immutability. Isomorphic definition does not allow this since two isomorphic objects that have mutated are not candidates for sharing.
- Our analysis works is focused on objects and find groups of objects that can be shared and Rama *et al.*'s approach is focused on object creation sites, computing an estimation of the reduction of heap size for each site.
- The use of system specific objects allows us to simplify the analysis and identify other

opportunities.

Unfortunately, the paper does not provide enough information about the fixes and concrete refactorings performed in order to compare them.

We value the discussion proposed by the authors about long-lived and short-lived objects. In contrast to their work, we do not monitor and use the object age (*i.e.,* elapsed between their creation and their collection by the garbage collector) to rank optimization opportunities.

Maximal sharing. Steindorfer *et al.* [13] investigated the use of dynamic analyses over weak immutable objects. They define weak immutability as unchanged variables that are used in the equals method. Their main objective is to find hash-consing implementations opportunities to reduce memory consumption and increase the *equals* method performance. Using their pre-condition of weak immutability they achieved a fast analysis based on profiling object creations and *equals* method calls.

The main difference with our approach is the weak immutability pre-condition they require for their analysis. Instead, our analysis does not require it as a pre-condition and our technique is applicable for most software projects supported on the platform. But, this required us to instrument mutations and to be able to analyze object graphs with cycles, being unable to implement most of the possible optimizations they contributed. Despite that, we have been able to optimize 6 real industry applications.

MemoizeIt. Della Toffola *et al.* [4] presents a technique to identify memoization opportunities to enhance speed performance using an iterative profiling. At each run, they reduce the number of candidates for optimization, allowing them to increase the performance and precision of their analysis. They applied their tool on 11 Java applications, for which they have found several optimizations that lead to significant speedup.

In contrast to our analysis technique, they focus on all the method calls and, in order to scale their analysis, rely on iterations to not capture the whole object graph, but only one level deeper each iteration. Instead, we focus on object creations and making the analysis of the whole graph mandatory for our equivalence definition.

Finding Reusable Data Structures. The research of Xu [16] presents a technique to find allocation sites that produce expensive to compute data structures which can be reused. They implemented their tool on a modified Jikes RVM and applied it on 6 different real-world applications.

Regarding our research, it is relevant to mention the implementation Xu suggested to compute graph isomorphism. He relies on the computation of *summaries*, which are values that represent the shape of a structure, but can be compared efficiently at the same time. This value has a similar objective as our *soHash* described in Section 8.3. The main difference is that we rely on a external collision-resistant function (MD5) and that we provided it the capability to work on cycles.

Chapter 11

Conclusion

We have presented a new technique to monitoring the execution of an application improving Marinov and O'Callahan approach. We provide definitions for *shareability* and *equivalence* and use them as a base for introducing a new object characterization.

Characterizing each object into domain objects, external objects and system specific objects enabled us to increase the number of identified equivalent objects in our benchmarks. This lead us to implement 14 optimizations for 6 applications, without having a particular knowledge about their internal representation. We provided a classification of the optimizations performed and characterized the usage opportunities for each of them.

Part of our work consists of a partial replication of Marinov and O'Callahan's research. Our results enable us to validate their conclusions and allow a comparison between the studies. In their benchmark suite, they have that 35% of the memory consumed was redundant, compared to 45% in our case study.

Moreover, the memory consumption reduction achieved by the optimizations of Marinov and O'Callahan were 47% and 38%, which are similar to our median memory consumption reduction of 43.2%.

Lessons learned

We found similar results in Smalltalk to Marinov and O'Callahan's results in Java. Smalltalk applications also present a significant amount of memory bloat, which is caused by redundant equivalent objects. Moreover, the memory bloat cause by equivalent objects may be reduced by the use of a dedicated code profiler.

Analyzing the objects created by the applications, we have found that *system specific objects*, such as Numbers, Strings and Collections, are responsible for most of the applications memory consumption. Optimizing these core libraries or their use in the applications may have a major impact in memory consumption.

By revising our results, our proposed fixes and the fixes described by Marinov and O'Callahan, we conclude that the effort of finding objects that become mergeable after their initialization has little impact on the number of redundant objects found. This is supported by the optimizations described by them consist only on objects mergeable from initialization and the fact that our research partially replicates their results without relying on objects that are mergeable after initialization. The usage of time in Marinov's approach may be then removed in order to increase performance and reduce complexity of the analysis.

Future work

We have three planed extensions to this work:

Discarding cycles. In agreement with Marinov *et al.* we found that strongly connected components are not prone to be equivalent objects. Computing object equivalence on strongly connected components involves a complex analysis demanding expensive computation.

Removing strongly connected components and cycles from the analysis would not only simplify the analysis, but also increase its performance.

Controlled experiment. Section 7 describes a small experiment we carried out with an expert. We were able to identify some optimizations that were not obvious to the expert.

We plan to conduct a controlled experiment with developers using our profiling technique to assess the following:

- The relevance of the information provided by our code profiler.
- The impact of the optimizations proposed by expert developers using our code profiler.

Partial equivalence relation. We would also like to explore a new relation of partial equivalence between objects. In the present research, objects are required to have the complete state equivalent to be candidates for optimization. Manually inspecting the gathered data by the profiler we have found that objects with many instance variables, such as graphical elements, reference the same objects with many of their instance variables.

Large objects may be then split up in two objects, a non-equivalent object which holds all the non-equivalent instance variables, and an equivalent object, which holds the equivalent instance variables of the object.

Even though this approach does not reduce the number of objects, it reduces the memory consumption by reducing the object size in memory of objects that can not be shared.

Bibliography

[1] Alexandre Bergel, Felipe Bañados, Romain Robbes, and David Röthlisberger. Spy: A flexible code profiling framework. *Journal of Computer Languages, Systems and Structures*, 38(1), December 2011.

[2] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. Patterns of memory inefficiency. In *Proceedings of the 25th European Conference on Object-oriented Programming*, ECOOP'11, pages 383–407, Berlin, Heidelberg, 2011. Springer-Verlag.

[3] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[4] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. Performance problems you can fix: A dynamic analysis of memoization opportunities. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 607–622, New York, NY, USA, 2015. ACM.

[5] Martín Dias, Mariano Martinez Peck, Stéphane Ducasse, and Gabriela Arévalo. Fuel: a fast general purpose object graph serializer. *Software: Practice and Experience*, 44(4):433–453, 2014.

[6] Lu Fang, Liang Dou, and Guoqing (Harry) Xu. Perfblower: Quickly detecting memory-related performance problems via amplification. In *ECOOP*, volume 37 of *Leibniz International Proceedings in Informatics*, pages 296–320. Schloss Dagstuhl - Leibniz-Zentrum f"ur Informatik, 2015.

[7] Alejandro Infante. Identifying caching opportunities, effortlessly. In *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, pages 730–732, New York, NY, USA, 2014. ACM.

[8] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. Disl: A domain-specific language for bytecode instrumentation. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 239–250, New York, NY, USA, 2012. ACM.

[9] Darko Marinov and Robert O'Callahan. Object equality profiling. In *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems,*

languages, and applications, OOPSLA '03, pages 313–325, New York, NY, USA, 2003. ACM.

[10] Alexandre Bergel Milton Mamani, Alejandro Infante. Inti: Tracking performance issue using a compact and effective visualization. In *Proceedings of Jornadas Chilenas de Computación 2014*, JCC 2014, 2014.

[11] Khanh Nguyen and Guoqing Xu. Cachetor: Detecting cacheable data to remove bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 268–278, New York, NY, USA, 2013. ACM.

[12] Girish Maskeri Rama and Raghavan Komondoor. A dynamic analysis to support object-sharing code refactorings. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 713–724, New York, NY, USA, 2014. ACM.

[13] Michael J. Steindorfer and Jurgen J. Vinju. Performance modeling of maximal sharing. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering*, ICPE '16, pages 135–146, New York, NY, USA, 2016. ACM.

[14] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 959–972, New York, NY, USA, 2011. ACM.

[15] Ingo Wegener. *Complexity theory: exploring the limits of efficient algorithms*. Springer Science & Business Media, 2005.

[16] Guoqing Xu. Finding reusable data structures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 1017–1034, New York, NY, USA, 2012. ACM.

Appendix A

Specification of profiler events

This section present the documentation of the events generated by the profiler. The architecture of the profiler defines that all pieces of data gathered must be encapsulated in an event. For each of the defined event classes it provides a short description, superclass, defined instance variables with a short description and the visitor method to implement for its use. Figure A.1 shows the class hierarchy of events described in this section.

SOEvent

The instances of this class represent an event for ShareableObjectsProfiler to be processed later. Further events should be defined by subclassing this class and implementing the visitor method associated. Includes the unique id of the object related to the event and the time when the event took place.

- Class name: SOEvent
- Superclass: Object
- Instance Variables:
  - id : SmallInteger - Unique id given by the profiler to the object associated with the id.
  - time : LargePositiveInteger - Time in milliseconds when the event took place.
- Visitor method associated: #visitEvent:

SOAssociationSetEvent

The instances of this class represent the writing of an association in the execution. It records the id of the association object, the key and the value.

- Class name: SOAssociationSetEvent
- Superclass: SOEvent

- Instance Variables:
  - key : SmallInteger - Unique id given by the profiler to the object written as key.
  - val : SmallInteger - Unique id given by the profiler to the object written as value.
- Visitor method associated: #visitAssociationSetEvent:

SOCollectionEvent

The instances of this class represent events relation to collections. Corresponds to the root class of events related to collections.

- Class name: SOCollectionEvent
- Superclass: SOEvent
- Instance Variables: none
- Visitor method associated: none

SOAtPutEvent

The instances of this class represent the writing on a SequenceableCollection, mostly Arrays and OrderedCollections. It records the collection id, the index to be written and the id of the object that was stored.

- Class name: SOAtPutEvent
- Superclass: SOCollectionEvent
- Instance Variables:
  - atIndex : SmallInteger - Index of the collection where the object was stored.
  - obj : SmallInteger - Unique id of the stored object.
- Visitor method associated: #visitAtPutEvent:

SOChangeInnerArrayEvent

The instances of this class represent capacity changed of expandable collections. OrderedCollections, Sets and Dictionaries have an inner array to store the objects, each time the size of the collection surpasses the capacity, the inner array is replaced with another array of superior size. Records the id of the collection and the id of the previous and new inner array.

- Class name: SOChangeInnerArrayEvent
- Superclass: SOCollectionEvent
- Instance Variables:
  - fromObjId : SmallInteger - Unique id of the previous inner array.
  - toObjId : SmallInteger - Unique id of the new inner array.

- Visitor method associated: #visitChangeInnerArrayEvent:

SOIndexChanged

The instances of this class represent changes of indexes used on boundaries for inner capacity. For example, which index in the array represents the first element of the collection.

- Class name: SOIndexChanged
- Superclass: SOCollectionEvent
- Instance Variables:
  - fromObjId : SmallInteger - Previous index.
  - toObjId : SmallInteger - New index.
- Visitor method associated: none

SOFirstIndexChanged

The instances of this class represent changes of indexes used for the first element on boundaries for inner capacity.

- Class name: SOFirstIndexChanged
- Superclass: SOIndexChanged
- Instance Variables: none
- Visitor method associated: #visitFirstIndexChangedEvent:

SOLastIndexChanged

The instances of this class represent changes of indexes used for the last element on boundaries for inner capacity.

- Class name: SOLastIndexChanged
- Superclass: SOIndexChanged
- Instance Variables: none
- Visitor method associated: #visitLastIndexChangedEvent:

SOExecutionEndEvent

The instances of this class mark the end of an execution in the profiler.

- Class name: SOExecutionEndEvent

- Superclass: SOEvent
- Instance Variables: none
- Visitor method associated: #visitExecutionEndEvent:

SOIdentityHashEvent

The instances of this class represent that an identityHash message was received by an object.

- Class name: SOIdentityHashEvent
- Superclass: SOEvent
- Instance Variables: none
- Visitor method associated: #visitIdentityHashEvent:

SOObjCreationEvent

The instances of this class represent that an object allocation was recorded. Besides the object unique id, the allocation context of size 10 is also recorded.

- Class name: SOObjCreationEvent
- Superclass: SOEvent
- Instance Variables:
  - allocationContext : SOContex - Allocation context of size 10.
- Visitor method associated: #visitObjectCreationEvent:

SOObjFirstSeenEvent

The instances of this class represent that an object was found in the execution whose creation was not recorded. Records the unique id of the found object and the context of size 10 of the first encounter with it by the profiler.

- Class name: SOObjFirstSeenEvent
- Superclass: SOEvent
- Instance Variables:
  - firstSeenContext : SOContext - Context of size 10 of the first encounter of the profiler with an object.
- Visitor method associated: #visitObjectFirstSeenEvent:

SOObjGarbageColletionEvent

The instances of this class represent that an object was garbage collected in the execution.

- Class name: SOObjGarbageColletionEvent
- Superclass: SOEvent
- Instance Variables: none
- Visitor method associated: #visitObjectGarbageCollectionEvent:

SOObjWritingEvent

The instances of this class represent that an object instance variable was written. The event records the id of the object plus the instance variable name, the unique id of the previous instance variable value and the unique id of the new value.

- Class name: SOObjWritingEvent
- Superclass: SOEvent
- Instance Variables:
  - instVarName : Symbol - Name of the instance variable written.
  - oldValue : SmallInteger - Unique id of the previous value of the instance variable.
  - newValue : SmallInteger - Unique id of the previous value of the instance variable.
- Visitor method associated: #visitObjectWritingEvent:

SORectangleSetEvent

The instances of this class represent that a rectangle was written. The event record the unique id of the origin and corner new values of the rectangle.

- Class name: SORectangleSetEvent
- Superclass: SOEvent
- Instance Variables:
  - origin : SmallInteger - Unique id of the origin instance variable of Rectangle.
  - corner : SmallInteger - Unique id of the corner instance variable of Rectangle.
- Visitor method associated: #visitRectangleSetEvent:

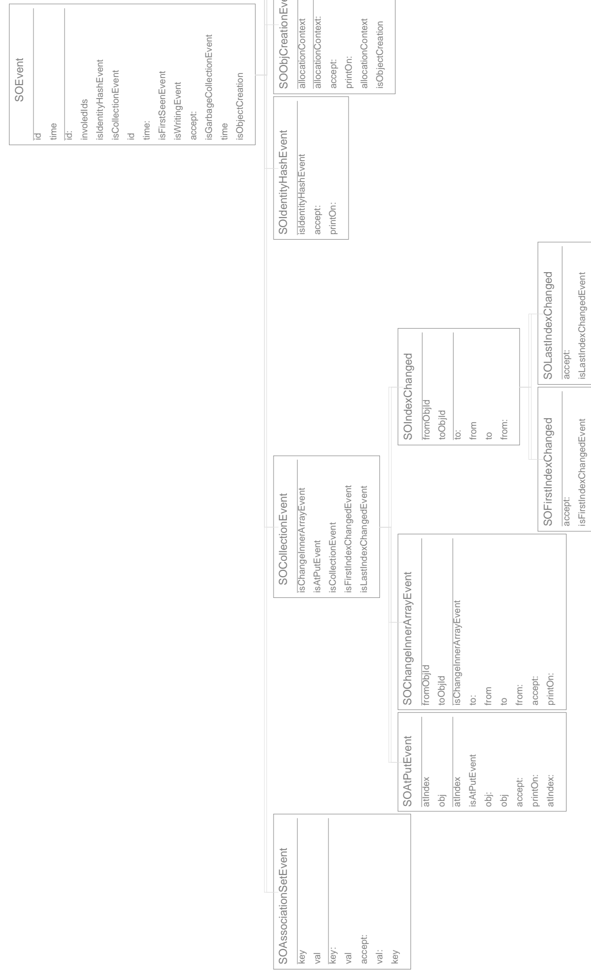


Figure A.1: SOEvent hierarchy class diagram.