Object Equivalence: Revisiting Object Equality Profiling

An experience report

Alejandro Infante
Pleiad Lab, DCC, Universidad de Chile
Chile
ainfante@dcc.uchile.cl

Alexandre Bergel
Pleiad Lab, DCC, Universidad de Chile
Chile
abergel@dcc.uchile.cl

Abstract

Modern object-oriented programming languages greatly alleviate 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.

CCS Concepts •**Software and its engineering → Software performance;** *Object oriented development;*

Keywords object equivalence, cache, memory bloat, profiling

ACM Reference format:
Alejandro Infante and Alexandre Bergel. 2017. Object Equivalence: Revisiting Object Equality Profiling
An experience report. In *Proceedings of 13th ACM SIGPLAN International Symposium on Dynamic Languages, Vancouver, Canada, October 25–27, 2017 (DLS'17),* 12 pages.
DOI: 10.1145/3133841.3133844

DLS'17, Vancouver, Canada

DOI: 10.1145/3133841.3133844

1 Introduction

Memory consumption is a major concern for most non trivial software [12]. Memory is used to store the resources used in software executions, which are constantly allocated and released.

Management of memory resources constitutes a fundamental task in software development. In object-oriented languages, memory is usually consumed by objects, which are created and destroyed continuously during the execution of a program. Manually handling the memory is a complex and prone-to-error activity.

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 unnecessary created [2, 5].

In 2003, Marinov and O'Callahan [7] proposed a profiling technique called *object equality profiling*. This technique identifies redundant objects by inferring groups of objects qualified as *mergeable*. These groups of objects have the property of being shareable between them, *i.e.,* a single object of the group may be used to replace any other object of the group. Identifying groups are then important, because reducing the size of mergeable objects significantly reduce applications memory consumption.

Marinov *et al.* applied their technique to the SpecJVM98 benchmark, which identified a significant amount of mergeable objects. Although appealing, the proposed technique has not been used in an industrial environment. We believe there are two reasons for its low acceptance:

- *Unknown mergeability site* - Marinov and O'Callahan's model ignores object mergeability sites, *i.e.,* the exact location in the source code where two or more objects become mergeable. Therefore identifying optimization sites is left to the programmer. Our experience shows that addressing excessive object creation without knowing their object creation contexts is challenging.

- *Not reproducible results* - Marinov and O'Callahan describe with great care many aspects of object equality profiling and their experiment. However, some of analyzes exercised on the graphs of objects are not sufficiently detailed, which prevent us from replicating their experiment.

This paper revisits Marinov's original contribution. We present and carefully analyze *object equivalence*, a technique to identify redundant objects and efficiently avoid memory waste in object oriented languages with classes. Our technique features the followings:

- We propose the object equivalence definition, which requires a relaxed notion of immutability we define as *unchanged*. In contrast, the approach of Marinov does not require immutability, but introduces the notion of *mergeability at time t*. Time t represents a checkpoint in the execution when a pair of objects become mergeable, then these objects may only mutate prior to t. Our approach has the advantage of simplifying memory analysis without losing optimization opportunities.
- Each object creation is associated to its exact location in the source code and a portion of the runtime stack. This is a relevant when removing an unnecessary object creation.
- Our model supports specific notion of equivalence, tailored to identify redundancy in numerical values, string, points, and collections. Though tailoring the definition for specific classes increase the complexity of the analysis, a reduced set of tailored classes has a significant impact on identifying memory consumption.

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 Pharo (http://pharo.org/), which offers a simple object model and runtime. In total, we have carried out an analysis over six large Pharo applications.

Contributions. The paper makes the following contributions:

- We carefully analyzed the object productions of six large Pharo application executions and found that, on average, 46.3% of the objects are equivalent, *i.e.,* objects that may be removed without affecting the application behavior. These objects represent 45% of the memory consumption of the application.
- We have manually implemented 14 optimizations (mostly caches) and we have reduced memory consumption in more than 40% on half of the studied applications.
- We partially replicate Marinov *et al.*'s experiment. We discuss thoroughly their approach similarities and differences from our approach. For example, we found that the class `String` and `Point` are indeed a cause of redundant objects.
- We introduce the concept of *kernel objects*, which treats objects from known core libraries in a different way for achieving better performance or results.

Findings. Our findings:

- We found that a reduced set of classes in the core libraries of the language contribute to the largest memory saving opportunities.
- Strongly connected components do not contain equivalent objects.

Outline. The paper is structured as follows. Section 2 presents object equivalence, our technique to identify redundant memory consumption. Section 3 describes the experiment we conducted to assess the relevance of object equivalence in relevant software executions. Section 4 runs our experiments and presents our results. Section 5 highlights some relevant aspects of our implementation. Section 6 positions our work against Marinov's experiment. Section 7 describes the work related to our effort. Section 8 concludes out work.

2 Object Equivalence

2.1 Intuition

We define object equivalence, $\approx$, a condition between objects describing redundancy. We say that two objects are equivalent, $o_1 \approx o_2$, if making all objects pointing to o1 to point to o2 preserves the program invariant, *i.e.,* does not affect the program behavior.

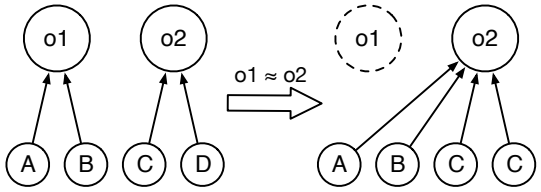

Figure 1. Object equivalence relation

Figure 1 illustrates this idea: if o_1 is equivalent to o_2 during the whole execution of the application, the creation of o_1 may be considered as unnecessary and may be avoided without impacting the invariant of the program. Intuitively, two objects are equivalent if:

- both objects are instances of the same class;
- both objects instance variables 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.

2.2 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. Unchanged property is required for ensuring that equivalence is preserved during the execution.

Definition 1. Let $t, t' \in T$, being T the timestamps of all operations performed over the object o. An object o is unchanged, written *unchanged(o)* iff:

$$\forall f \in instVars(o)$$
$$\nexists t < t'$$
$$o.f_t \neq NULL \quad and \quad o.f_t \neq o.f_{t'}$$

Where $o.f_t$ represents the value of the instance variable f of the object o at instant t of the execution. For instance, this definition forbids an object instance variable $o.f$ from having a different value after being initialized.

Unchanged property is a more permissive property than object immutability, enabling objects to be initialized outside objects constructor and to have lazy initialized instance variables.

Identity operations. The object identity allows the program to distinguishing two objects that may be shared. A known example of an identity operation is the identity test called using == in Java and Pharo. As soon as the object identity is used, then the object is not equivalent to any other object.

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 our experiment setting, the operations we consider are (i) object reference comparisons (use of ==) and (ii) using default hash implementation (use of `identityHash` in the Pharo programming language).

Object cycle. When configuring the equivalence of two objects to depend on the equivalence of their instance variables, we are defining a recurrence relation. 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:

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

Object Equivalence. Consequently we define object equivalence, $o_1 \approx o_2$ in two different cases:

Definition 2. If both o_1 and o_2 do not belong to a cycle, then $o_1 \approx o_2$ iff:

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

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

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

Where $scc(o_1) = (e_1^1, ..., e_n^1)$ and $scc(o_2) = (e_1^2, ..., e_n^2)$, 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 $e_1^2 = o_2$.

By definition, the nodes in a graph belonging to the same cycle are members of the same strongly connected component. A strongly connected component (SCC) is a set of nodes that for every node a path to every other nodes exists. 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].

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.

2.3 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.
- *Kernel objects* are a subset of the external objects for which it is known that they are prone to be redundant (*e.g.*, `String`, `Point`). A kernel 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. Kernel objects are subject to particular object equivalence relations, as described below.

2.4 Kernel Objects

Kernel objects are defined by the language or libraries and have a known invariant. We use this in

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. 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 kernel 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; `Dictionary` is an expandable unordered key-value collection. We have the following equivalence relations:

- An array is equivalent to another array if its size and content are the same and both meet the *unchanged* property.
- An ordered collection is equivalent to another ordered collection under our approach if their sizes at the end of the execution are the same, their contents are identical, and the collections have been filled only appending elements at the end, without removing elements.
- A set is equivalent to other sets if their size and contents are the same, 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 and mutations are infrequent.

2.5 Equivalency 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; }
```

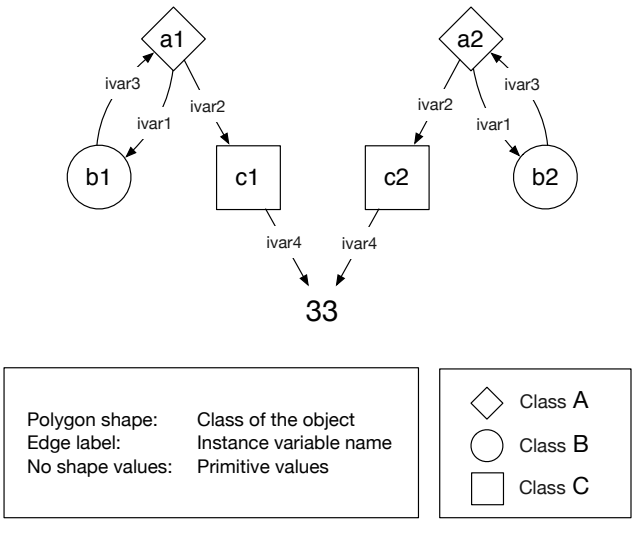

Figure 2. Object graph of example which shows that $a1 \approx a2$, $b1 \approx b2$, and $c1 \approx c2$.

The snippet of code generates the object graph shown in Figure 2. 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 $a2.ivar2 = c2$. 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 $c1 \approx c2$.

2.6 Equivalent Object Group

Object equivalence is an equivalence relation. It is then possible to group objects into equivalence groups.

Definition 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:

Be O all objects in the execution.

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

The main property of these groups is that only a single element of the group is required in the execution. The rest of the elements of the group are redundant.

The object groups are defined by the equivalence relation definition. Therefore, do not hold a relation with the place or moment of the computation where they were created. Despite that, identifying the allocation or creation context is fundamental in order to reduce the bloat, which is performed by allowing the garbage collection of redundant objects or avoiding their creation.

3 Design of the Experiment

This section describes the experiment we performed to assess our technique to identify equivalent objects.

3.1 The 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.

3.2 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. Measure the overall memory redundancy bloat of a set of representative software executions.
2. Measure the memory consumption of redundant kernel objects.
3. Evaluate the impact of memory optimizations proposed for 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 enable 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.

3.3 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 1 lists our metrics.

Related to the Number of Objects. As described in Section 2.3, objects created during a program execution belong to one of three distinct groups. We therefore provide the metrics **NDO**, **NKO**, **NEO** describing the number of domain, kernel, and external objects, respectively. In addition, we measure 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. Our approach therefore consists in reducing the **NOP** value for our benchmark.

Related to Equivalent Objects. We define 5 metrics for measuring the quantity of equivalent objects on the execution. We then define **NEqD** and **NEqK** as the number of equivalent domain and kernel 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 describes the number of equivalence groups using **NEqDG** and **NEqKG**, being the number of domain groups and kernel 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 memory

Table 1. Metric definitions

Short	Metric
	Number of Objects by category
NOP	Number of Non-Primitive Objects
NDO	Number of Domain Objects
NKO	Number of Kernel Objects
NEO	Number of External Objects (Excluding NKO)
	Number of Equivalent Objects by category
NEqO	Equivalent Objects
NEqD	Equivalent Domain Objects
NEqDG	Domain Objects Equivalence Groups
NEqK	Equivalent Kernel Objects
NEqKG	Kernel Objects Equivalence Groups
	Memory Consumption by category (KB)
MOP	Non-Primitive Objects
MDO	Domain Objects
MKO	Kernel Objects
MEqO	Equivalent Objects
MEqD	Equivalent Domain Objects
MEqK	Equivalent Kernel Objects

consumption corresponds to 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 **MKO** describe the amount of KB consumed by domain objects and kernel objects respectively; **MOP** is the total memory consumption of the application as the addition of the previous two metrics; **MEqD** and **MEqK** describe the amount of KB consumed by equivalent domain and equivalent kernel 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.

3.4 Benchmarks

We have considered six applications in our benchmark: Roassal2 (a visualization engine), Nautilus (a source 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.

We have a representative execution using a large input provided by the author of each application.

4 Results

We present the measurement obtained from computing the metrics during the execution of our set of applications (Section 4.1). We then discuss the optimization we manually implemented and their impact on the memory footprint (Section 4.2).

4.1 Metrics Analysis

Number of objects. We have run our analysis on each application composing our benchmark. Table 2 gives the metric values for our benchmark. The NOP column sums the columns NDO, NKO, 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 kernel 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 kernel objects.*

Equivalent objects. Table 3 gives the number of equivalent objects measured in our benchmark. Half of our benchmarks report that more than 65% of the objects created are equivalent (NEqO column), being the average ratio of equivalent objects 46.3%. We also report that 38.6% of the total number of objects corresponds to equivalent kernel objects (NEqK column).

Figure 3 shows that NEqK is in high proportion, representing over 40% of the objects for all applications but one. NEqD tops at 28% for the PetitParser application, but we found equivalent domain objects only on 2 of the 6 benchmarks. We have the relation NEqO = NEqD + NEqK: the number of equivalent objects is equal to the number of equivalent domain objects summed up with the number of equivalent kernel objects. These figures indicate that *a significant portion of kernel 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.

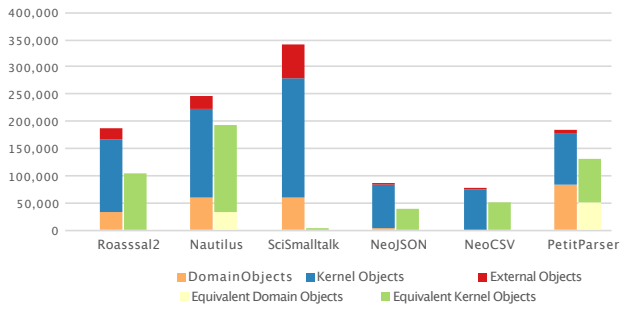

Figure 3. Distribution of objects and equivalent objects in benchmarks.

Memory footprint. Table 4 gives the memory footprint of objects produced during the benchmark execution. The column MOP gives the memory used by non-primitive objects,

Table 2. Results in number of objects

	NOP	NDO	NKO	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 3. Number of equivalent objects

	NOP	NEqO	NEqD	NEqDG	NEqK	NEqKG
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
SciSmalltalk2	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 4. Results in memory usage (KB)

	MOP	MDO	MKO	MEqO	MEqD	MEqK
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%)
SciSmalltalk2	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%)

Table 5. Reduction of memory footprint due to our optimizations

	NOP Before	NOP After	Reduction	MOP Before (KB)	MOP After (KB)	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%
SciSmalltalk2	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%

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 4.4 discusses that topic further.

In the benchmarks, memory consumption of equivalent objects MEqO represented more than 50% of the total memory consumption in half of the benchmarks. 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 kernel objects, 25% and 75% respectively. Equivalent objects represent 45% of the memory consumed by the benchmark execution. Equivalent specific objects represent 38.3% of the whole memory.

4.2 Avoiding equivalent object creation

Table 3 indicates that 46.3% 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 the creation of equivalent objects. Note that we cannot remove all equivalent objects since at least one object per group of equivalent objects has to remain. In several cases, completely avoiding the creation of equivalent

objects is not worth the effort, *e.g.,* if the equivalence object 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 the equivalent object groups and implemented 14 optimizations across five applications in our benchmark suite.

Despite of these optimizations were relatively simple to implement, they have a significant impact on the measurements. Table 5 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 represents a reduction in memory consumption of more than 40% on half of the applications in the benchmark suite.

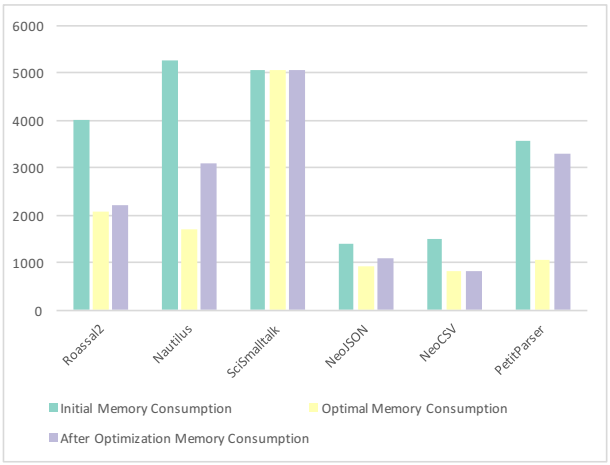

Figure 4. Comparison of initial memory consumption, optimal consumption and after optimizations consumption.

4.3 Optimizations implemented

4.3.1 Strategies for proposing optimizations

The profiling tool we designed provides an analysis environment enabling the practitioner to inspect the object graph of the execution. Our tool supports an expressive graph API for finding memoization opportunities. The strategies are the following:

Large equivalent object groups. 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.

An example of this is presented on the *single object cache* description in Section 4.3.2.

Equivalent objects 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. Even if the size of the object group is small, the impact in memory consumption can be significant if the number of equivalence groups is big enough. In this case, several objects needs to be cached to remove the memory bloat. One object per each equivalence group involved.

An example of this is presented on the *map of objects cache* description in Section 4.3.2.

4.3.2 Classification of Optimizations

Using our profiler we have proposed and implemented 14 optimizations in 5 applications of the benchmark suite. We spent 14 hours in total for proposing and implementing the optimizations.

We classified the 14 optimizations we found in four categories:

Single object cache (5 implementations). The single object cache is an optimization that targets a single equivalence object group. It consists in implementing 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.

We found in Nautilus that 22% of the equivalent domain objects are instances of `MethodSelected` created using `null` as model. The following pseudo-code illustrates our solution:

```
class MethodSelected {
  private model;
  static create(model){
    return new MethodSelected(model);
    }
  ...
}
  // Original code
```

```
class MethodSelected {
  private model;
  static nilModel =
    new MethodSelected(null);
  static create(model){
    if(model == null)
      return nilModel;
    return new MethodSelected(model);}
} //Modified code
```

Map of objects cache (2 implementations). 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.

It consists in implementing 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 map *value* corresponds to the cached object.

We found in Nautilus that 15% of the equivalent domain objects are instances of `Changed`. They are created using the strings #sourceCodeFrom:, #getHistoryList or #currentHistoryIndex as possible commands. The following pseudo-code illustrates this cache:

```
class Changed {
  private command;
  static create(command){
    return new Changed(command);
    }
  ...
}
  // Original code
```

```
class Changed {
  private command;
  static cache = new HashMap();
  static create(command){
    if(not cache.containsKey(command))
      cache.put(model,
        new Changed(command));
    return cache.get(command);}
} //Modified code
```

Weak map of objects cache (5 implementations). 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.

Others (2 implementations). This category holds implementations that do not belong to the previous categories and their proposal is performed by manual inspection of the profiler output and the application source code.

An example of this is the case of PetitParser. We found that `PPContext` instances uses `Dictionary` instances of capacity 5, but only hold 1 element. This class is heavily used and we achieved a reduction in memory consumption by using the implementation optimized for small collections `SmallDictionary`. 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.

4.4 Memory and run time impact on the system

The measurements reported in Table 4 and Table 5 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 with the size of each instance variable.

From the operating system point of view, 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 memory allocation for the heap.

We use Pharo as our runtime. The size of a clean Pharo process for this experiments is 62.9MB. Upon executing the Roassal benchmark, the size of the process raises to 69MB, but when executing the benchmark with the optimizations the size of the process raised only to 65.2MB. We attribute this decrease on memory consumption to the optimizations proposed in this research.

Accurately measuring the memory variation from the operating system point of view is difficult and not easily reproducible: two runs of the very same process do not produce identical memory footprint. However, we estimated that the reduction of 1,122,559 - 826,099 = 296,460 object results in a memory reduction of 19Mb, from the operating system point of views.

We have measured the execution time of the benchmarks before and after the optimizations by executing them 10 times. We computed the average and standard deviations of the results and we found that four benchmarks suffered a non statistically significant speed-up and one suffered a non statistically significant slowdown.

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.

5 Implementation

This section highlights some aspects of our implementation. Our profiler is available under the MIT license (Available in ainfante/ShareableObjects on http://smalltalkhub.com).

Profiling technique. We used the Spy2 profiling framework [1] to build our profiler, not depending on a customized virtual machine. To capture the necessary information from the execution we have chosen the following strategies: we capture all the instances creations and hash messages sent using method wrappers and bytecode instrumentation; monitor instance variable uses using Slots [11]; keep track of the exact time objects are garbage collected. All this data is stored to be analyzed after the benchmark execution.

Equivalence analysis. In order to compute all the object equivalence groups of the execution we model the object graph as a directed vertex and edge labeled graph. In the graph model we use the nodes to represent objects created during a program execution and vertex labels represent the classes of the objects. The edges represent the instance variables and their values and the edge labels represent the instance variables names.

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. We precompute a hash value for each node we call *soHash*. This value is built to have 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 it by using MD5 collision resistant function. Assuming that object a is unchanged:

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

Else, in case a is mutable:

$$soHash(a) = MD5(uniqueNodeValue(a))$$

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

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

Performance. All the experiments execution were performed in less than 3 hours, indicating the execution time of the profiler is not an impediment for its use. But, we have not found a direct relation between the profiler execution time and the benchmark original execution time or the number of objects created by the benchmark.

We have not attempted to research further on this topic because the execution time has not been a problem for executing the experiments.

Marinov *et al.*'s approach reported a median overhead of 169.9x for their tool, which included profiling and analysis time [7]. Other analysis techniques, such as Nguyen *et al.*'s approach, incurred in an average overhead of 201.96x [8]. Despite the large overheads, they have reported that the large overhead was not a problem to use the tool and gather the data for their work.

Implementation in other languages. We have identified some constraints when porting 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-oriented 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. Also, new equivalence definition for kernel objects are required since it depends on the language core and libraries.

We are currently working on a port of this profiling technique to Java using DiSL [6] and to VisualWorks Smalltalk using Spy. We believe it is possible to port it also to Ruby and Python using equivalent instrumentation frameworks or libraries.

6 Partial replication of Marinov et al. experiment

The work presented in this paper originates from the object equality profiling technique proposed by Marinov *et al.* [7] to identify objects that are redundant in an execution. Our technique is similar enough to be called a partial replication study of their research. This section contrasts our research similarities, differences, agreements and disagreements Marinov's approach.

6.1 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 allocation and changes of state of objects.

Proposal of a recurrence relation. Both approaches proposed a recurrence relation to identify redundant objects, then this property is evaluated post-mortem of the execution by the analysis of the object graphs. The redundancy relation we proposed is called *Object Equivalence*, while the relation proposed by Marinov is called *Mergeability*.

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 reflected by the number of *equivalent objects* and in the research of Marinov are reflected by the number of *mergeable objects*.

6.2 Differences

Mergeability time. Marinov *et al.* identify the time where an object becomes mergeable with others. From that perspective, from that time the object becomes redundant and is no longer needed in the execution. Each object has their own mergeability, which may happen way after their 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. Then preferring practicality over recall. Furthermore, caches for redundant objects are easier to implement, because the optimization spots are also the allocation spots, which are collected 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 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 *kernel 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.

Memory Analysis. We were not able to compare how the data is analyzed in order to produce the optimizations because Marinov's paper does not explain it in detail. We believe that providing support to practitioners for analyzing the data produced by the profiler is key to achieve an important memory gain.

Table 6. OEP Marinov's mergeability results [7]

	Average Base Memory (MB)	Average Merge Memory (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%

6.3 Results and Conclusions comparison

There are two results that are comparable between the approaches. The first result is the potential saving opportunities. 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 and 2 are widely used web application servers. Table 6 presents the average memory consumption potential saving for each benchmark.

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*, *kernel* and *external objects*. Despite that, our profiling technique treated 68% of the objects as Kernel 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.

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 agrees regarding the existence of optimization opportunities caused by object bloat. We also agree over the fact that a dynamic analysis technique that keep 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 kernel objects. This is partially supported in the research of Marinov by the fact that all optimizations proposed by them are related to kernel 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 found do not depend on cycles. From all the benchmarks we executed, we have not found optimization opportunities depending on cycles, which is consistent with their results.

7 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 [7]. 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.

Cachetor. Nguyen *et al.* research [8] describes a novel technique that statically binds the cost of two extremely expensive dynamic analysis techniques, such as *dynamic dependence profiling* and *value profiling*. 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 [9] is also based on Marinov *et al.*'s research. For this reason there are several similarities between these works.

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 over the cycles of the graph avoiding transforming the object graph into an acyclic graph, losing potential optimization opportunities.
- The use of kernel 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-life and short-life 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.* [10] researched 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 found 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 [12] 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*. The main difference is that we rely on an external collision-resistant function (MD5) and that we provided it the capability to work on cycles.

8 Conclusion and Future work

We have presented a new technique to monitoring the execution of an application. Characterizing each object and identifying equivalent objects enabled us to implement 14 optimizations for 6 applications, without having a particular knowledge about their internal representation. We took the work of Marinov *et al.* as the base of our approach. We improved their original technique by considered specific objects, which have specific equivalence relations. As future work we plan to inject our technique within a virtual machine.

We gratefully thank LAM Research for its financial support. Alejandro Infante is supported by CONICYT-PCHA/MagísterNacional/2015-22150809.

References

[1] Alexandre Bergel, Felipe Ba nados, Romain Robbes, and David Röthlisberger. 2011. Spy: A flexible Code Profiling Framework. *Journal of Computer Languages, Systems and Structures* 38, 1 (Dec. 2011).

[2] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy. 2011. Patterns of Memory Inefficiency. In *Proceedings of ECOOP '11.*

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

[4] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. 2015. Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities. In *Proceedings of OOPSLA '15.*

[5] Lu Fang, Liang Dou, and Guoqing (Harry) Xu. 2015. PerfBlower: Quickly Detecting Memory-Related Performance Problems via Amplification. In *Proceedings of ECOOP '15.*

[6] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *Proceedings of AOSD '12.*

[7] Darko Marinov and Robert O'Callahan. 2003. Object equality profiling. In *Proceedings of OOPSLA '03.*

[8] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of ESEC/FSE '13.*

[9] Girish Maskeri Rama and Raghavan Komondoor. 2014. A Dynamic Analysis to Support Object-sharing Code Refactorings. In *Proceedings of ASE '14.*

[10] Michael J. Steindorfer and Jurgen J. Vinju. 2016. Performance Modeling of Maximal Sharing. In *Proceedings of ICPE '16.*

[11] Toon Verwaest, Camillo Bruni, Mircea Lungu, and Oscar Nierstrasz. 2011. Flexible object layouts: enabling lightweight language extensions by intercepting slot access. In *Proceedings of OOPSLA '11.*

[12] Guoqing Xu. 2012. Finding Reusable Data Structures. In *Proceedings of OOPSLA '12.*