ICSE: U: Identifiying caching opportunities by revisiting Object Equality Profiling

Alejandro Infante
Department of Computer Science (DCC), University of Chile
alejandroinfante91@gmail.com

ABSTRACT

A memory profiler is a tool used by software developers to understand how an application uses the physical memory during its execution. Most programming environments are shipped with an accurate profiler. In case of an excessive memory consumption, a software developer may reduce the application memory footprint by examining a profiler report. Unfortunately, optimization opportunities are not apparent in such a report due to the limitations of traditional profilers. Optimizing an application requires deep and extended knowledge about the analyzed application.

This paper presents and evaluates a new approach to identify redundant objects. Once their origin is precisely located, such objects may be shared to reduce the memory consumed by the application. This paper reproduces and evaluates a memory situation identified by Marinov and Callahan. Our approach improves their work to address the needs met by practitioners.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging aids*

General Terms

Performance

Keywords

Memory, profiling, object shareability

1. INTRODUCTION

Most of the programming languages used nowadays are object-oriented. The last decade has experienced tremendous advances in virtual machines and garbage collectors to reduce the cost due to handling large number of objects. In addition, the software engineering community has produced numerous tools to accurately monitor the memory consumption and to produce highly detailed reports.

Most of widespread memory profilers are good at detailing the overall consumption. However, understanding which portion of the memory is *redundant* is unfortunately entirely left to the programer. Consider the following contrived example in Java:

```
class A {
  String hello = "Hello World";
  void sayHello() { System.out.println(hello); }
  public static void main(String[] arg) {
    new A().sayHello();
    new A().sayHello();
  }
}
```

The example unnecessarily creates two objects from the class `A`. Leading profilers in Java, such as JProfiler[1], YourKit[2], and Mat[3] will indicate that `A` has produced two objects. However, none will actually suggest that only one instance of `A` is sufficient as in the following:

```
  ...
  public static void main(String[] arg) {
    A a = new A();
    a.sayHello();
    a.sayHello();
  }
  ...
```

The code given above is far from the complexity met in large applications, however it does illustrate a recurrent situation in object-oriented programs. It has been empiricaly shown [3, 8, 11, 9, 7, 6] that a significant portion of objects in object-oriented programs are redundant and could be simply avoided to save memory. For example, Marinov and Callahan [7] report that the portion of redundant memory ranges from 9% to 59% for a set of selected Java applications. Although memory has become relatively cheap, the memory situation described above is prominent in real-time applications (*e.g.,* garbage collection creates non-predictable system pauses and sluggish application behavior) and constrained system (*e.g.,* excessive garbage collection has a negative effect on the battery energy consumption [12, 10, 13]).

Research baseline. Identifying and avoiding the creation of redundant objects has been the focus of Marinov and Callahan [7]. They propose an innovative dynamic analysis technique to identify *mergeable* objects: two redundant

[1] `https://www.ej-technologies.com/products/jprofiler/overview.html`
[2] `https://www.yourkit.com`
[3] `https://eclipse.org/mat/`

objects (as in the example given above) may be merged under well described situations to avoid their creation. Their analyzes instruments the Java bytecodes to extract the graph of object dependencies from an application execution. Afterwards the graph is partitioned into independent clusters. Monitoring the occurrence of side-effects qualify an object cluster as mergeable or not.

Although Marinov's approach is effective, it has a significant cost in practice: an application under analysis is about 133 times slower and the graph partitioning, which is carried out offline, is 19 times the execution time. We believe this significant overhead contributed to mitigating the adoption of Marinov's approach among software engineers.

Our overall effort. Our long-term effort goal is to provide techniques and tools to software engineers to easily identify and avoid unnecessary resource consumption. In this paper, we narrow the scope of our effort by revisiting a solution proposed by Marinov *et al.*. In particular, the two research questions we are focusing on are:

Q1- *Can the Marinov's problem identification be replicated?*

Q2- *Can the Marinov's technique be improved to meet practical needs?*

Before addressing the problem formulated above, it is important to verify if it is actually a problem. Marinov found that the proportion of redundant objects ranges from 1% to 59%. We would like to replicate the experience of Marinov, which is the goal of Q1. Q2 is about implementing the technique variations described below and measuring the memory saving gain and the result precision.

Our contributions. This paper is about revisiting Marinov's approach to make it practicable. We took Marinov as a starting point and modified the notion of mergeability to trade precision for performance and practicability. In particular, (i) we discard object cycles from our analysis since they rarely lead to an optimization opportunity and (ii) we allow for an object to mutate only during its initialization (*e.g.,* while still being in its constructor). These two changes from Marinov's approach lead to significant performance improvement without loosing much precision.

In addition to identifying the exact source of memory overhead, we also employ the resultant object graph to support easy-to-implement source code refactorings to save memory. In particular our technique help practitioners to insert object caching to successfully improve memory consumption.

We implemented our tool in the Pharo programming language and we applied our technique over a reduced set of open source Pharo applications with use cases provided by the developers of the applications.

Paper outline. Section 2 presents our definition of shareable objects, sketches out our implementation, and describes the application benchmarks we are here considering. Section 3 details the result of profiling our benchmarks. The section shows that our approach improves the state of the Art in the field of automatic memory analysis to identify caching opportunities. Section 4 concludes, presents our gained experience, and describe our future work.

2. OBJECT SHAREABILITY

2.1 Definition

Two objects o_1 and o_2 are said to be shareable, noted $o_1 \sim o_2$, iff:

- o_1 and o_2 have the exact same class.
- each pair of corresponding fields must follow $o_1.v_i == o_2.v_i$ or $o_1.v_i \sim o_2.v_i$. We refer to v_i the name of the object variable and == to the classical object reference comparison.
- no identity based operation (*e.g.,* getting hash values or comparing object references) are performed on o_1 and o_2 except when verifying the shareability (*i.e.,* previous point).
- o_1 and o_2 are constant: object variable v_i can be assigned at the most once.
- the creation of both o_1 and o_2 do not perform a side effect outside the objects themselves.

In the code example given in Section 1 the two objects o_1 =`new A()` and o_2 =`new A()` are indeed shareable since both objects are from the same class, the class `A` has no fields, the operator `==` is not used and the message `hash()` is not sent, and no side effects are involved.

In our setting, we characterize an object as constant if all its instance variables are written once at most (*i.e.,* a variable of an object cannot be assigned more than once). This is similar to the restriction of a `final` variable in Java.

The definition of shareability that we propose is inspired in the definition of mergeability proposed by Marinov. However, it excludes particular cases that have relevance in our results, as described below.

2.2 Profiling Shareability

Dedicated profiler. We created a code execution profiler using the Spy profiling framework [1]. For an application execution, our profiler outputs the complete graph of objects involved in the computation (such objects may be created or not during the execution). Each object is annotated to indicate whether it is constant.

During a program execution, our profiler captures all the variable accesses (read and write) in addition to keeping track of a reduced set of messages exchanged between objects (to verify shareability). Each object receives a unique id, used in a weak large hashtable. By being weak, the hashtable does not disrupt the garbage collection activity.

After the execution, an offline analysis recreates the graph of objects and identify objects cycles and assesses the shareability, according to the definition previously given, for each objects not present in a cycle. Note that an intermediate graph presentation is built where all strongly connected components are condensed, thus discarding cycles.

Spy is implemented in the Pharo programming language. Pharo[4] is an emerging language programming environment which is dynamically typed, Smalltalk-inspired, and has a syntax close to that of Ruby and Objective-C. Pharo offers the necessary tooling, hooks, and an expressive reflective API, to accurately measure time memory consumption of a

[4] `http://pharo.org`

significant benchmark. In addition, the Pharo community is friendly and easily reachable, which is crucial when authors of the application composing our benchmark have to be contacted.

Our benchmark. A benchmark is a repeatable and measurable application execution. A benchmark is likely to describe a common and relevant usage scenario of an application. The Pharo ecosystem does not contain widely accepted benchmarks, such as DaCaPo and SpecCPU [2, 5]. We therefore have to constitute a reliable benchmark.

We have selected five applications: Roassal, Morphic, Famix, Moose, and Glamour. The applications we have selected are open-source[5], considered as significant for the Pharo community, and have been under steady improvement over a long period of time. Each application comes with a set of benchmark. We consider one data-intensive benchmark per application. We have applied our profiler to monitor the memory activity for each benchmark execution.

2.3 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` to build 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`.

The application from which `Builder` comes from 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. Unfortunately, the class `Builder` does not consider the possibility to reuse object instances.

Our profiler identified this particular problem of the class `Builder`. It has been solved by 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 avoid the creation of a large number of objects.

3. MEASUREMENTS

3.1 Object production sites

Table 1 indicates the provenance of objects, an aspect of the memory consumption of our benchmark. Column 1 lists the applications we have considered in our benchmark.

[5]Source code is available from `http://smalltalkhub.com`.

Application	All objects	Application objects	External objects
Roassal	3,481	1,481 (42.5%)	2,000 (57.5%)
Morphic	15,744	6,617 (42.0%)	9,127 (58.0%)
Famix	146,049	89,049 (61.0%)	57,000 (39.0%)
Moose	186,503	48,421 (26.0%)	138,082 (74.0%)
Glamour	40,023	21,077 (52.7%)	18,946 (47.3%)

Table 1: Object production sites in our benchmarks.

Application	Application objects	Shareable application objects
Roassal	1,481	116 (7.8%)
Morphic	6,617	840 (12.7%)
Famix	89,049	0 (0%)
Moose	48,421	0 (0%)
Glamour	21,077	1,005 (4.8%)

Table 2: Object shareability in our benchmarks.

Column 2 indicates the total amount of objects created during each application benchmarks. Column 3 indicates the number of objects produced by classes of the application. For example, Roassal's benchmark creates 3,155 different objects, for which 1,475 are instances of a Roassal class. The 1,680 remaining objects were produced by classes external to Roassal, as indicated in Column 4.

One striking fact given by Table 1 is that most of the objects created during an execution are actually created from classes that are external of the application. Consider the Moose example. Moose is a platform for software analysis[6], used in our benchmark to execute some analyses on a large application. 71.8% of the objects created by the benchmarks are actually created by classes defined outside the application. As a consequence, an excessive memory consumption may be due to underneath framework or executing platform. In our example, Moose is built on a rich stack of application layers, including parsers and several stream libraries.

3.2 Shareability

We measured the object shareability in our benchmarks. We first review shareability for objects created by application classes and then by some particular classes.

Application objects. Table 2 indicates the proportion of shareable objects that are instances of the application classes. Not all the benchmarks exhibit the same proportion of shareable object. Morphic's benchmark creates 6,617 objects, for which 840 (12.7%) are shareable. Introducing a cache, as illustrated in Section 2.3 reduces the number of objects by the same amount.

Surprisingly, the Famix and Moose benchmarks do not have shareable objects. A discussions with a group of developers involved in the maintenance of these applications reveals that Famix and Moose are used to process very large amount of data. The consumption of memory for these applications has therefore been under scrutiny and closely monitored over the last decade. Keeping the memory consumption low has been a major objective of the developers behind Famix and Moose, which explains we found no shareable objects.

[6]`http://moosetechnology.org`

Application	Strings	Shareable strings	Arrays	Shareable arrays	Points	Shareable points
Roassal	236	163 (69.1%)	3	2 (66.7%)	534	374 (70.0%)
Morphic	1,342	81 (6.0%)	1,147	33 (2.9%)	765	540 (70.6%)
Famix	4,285	118 (2.8%)	0	0 (0.0%)	0	0 (0.0%)
Moose	4,252	136 (3.2%)	23,900	0 (0.0%)	0	0 (0.0%)
Glamour	282	42 (14.9%)	1,411	487 (34.5%)	156	125 (80.1%)

Table 3: Shareability among strings, array and points.

Strings, Array, Points. It has been multiply shown that particular classes such as string character [4, 6], array, and point [7] are often involved in memory issues. We analyze the presence of instances of these classes in our benchmarks. Table 3 reports our findings in our benchmarks. Column 2 gives the number of strings created in the benchmarks. Some applications are more string intensive than other. For example, Famix and Moose created ∼20 times more strings than Roassal or Glamour. Column 3 indicates which portion of the created strings are shareable. Two object strings are shareable if they have actually the same content. This is a frequent cause of memory issue related to strings. Over 69% of the strings created by Roassal are shareable. Column 4 indicates the number of created arrays and Column 5 reports the amount of shareable arrays. Two arrays are shareable if they have the same content and their content is never updated. For example, two empty arrays are shareable. About 34% of the arrays produced by Glamour are shareable. Column 6 indicates the number points created by the benchmarks and Column 7 the shareability among them. Roassal, Morphic, and Glamour uses graphical user interfaces which are likely to use geometrical points. Points have a large proportion of shareability, over 70% for the application involving a UI. Note that the problem with points is mitigated since a point is relatively light by occupying 12 bytes (in Pharo).

Summary. The result given above report that we identified the problem described by Marinov in our benchmark. This section positively answers the Q1 research question.

3.3 Evaluating practicability

We evaluate the practicability of our profiler from two complementary perspective: usability and performance.

Usability. Our profiler has to be fed with a list of package and modules composing the application to analyze and a starting execution point, typically the `main()` method. After the execution and the object graph reconstruction, our profiler outputs a statistical report (for which Table 1 is an excerpt) and a ranked list of classes that produce shareable objects. The exact location in an application code source of shareable objects is then obtained from the class references. This relatively simple way to operate has been enough for the situation we have faced. Since the profiler used by Marinov is not publicly available, we could not compare it.

Performance. Our profiler should carry its analysis with a reduced amount of resources (time and memory) in order to meet practical needs: a profiler that requires excessively long and heavy computation is likely to be used less frequently.

As mentioned earlier, our profiler go through two steps to produce its memory consumption report: (i) monitoring an application execution and (ii) constructing a graph of objects and analyzing the shareability of each individual object. Monitoring the application execution increases the execution time by a factor ranging between 2.5 and 87.

The execution of the Roassal benchmark has been 2.5 times slower and 87 times for Morphic. Although the performance exhibited by our profiler is high, comparing the raw overhead figure make our profiler faster than the one of Marinov. In their case study, Marinov's profiler was 133 times slower while our profiler overhead ranges from 2.5 to 87 (with a median of 45).

The second step of the profiling activity, which is the graph construction and identifying the object shareability has an overhead ranging from 11 to 18 times the benchmark execution time. In the average, Marinov's profiler is 19.3 times slower than the benchmark execution time.

Summary. For the case studies considered in this paper and by Marinov, our profiler has been able to find positive results for reducing the memory consumption and also lowering the execution overhead, which positively answers the research question Q2.

4. CONCLUSION AND FUTURE WORK

Traditional code execution profilers are particularly efficient at indicating the distribution of execution time among components. However significant experience is necessary for a practitioner to clearly identify the cause of poor performance. Our profiler analyzes the memory consumption and indicates which objects are likely to be beneficial in caching. This information given to the developers constitute the delta of our work against the related work.

As future work, we plan to evaluate our heuristics about constant objects – The profiling technique described above employs heuristics that are based on our experience and intuition. An object may be considered immutable at a given point in time if it is never accessed before its last mutation. Precisely identifying when an initialization and construction really end is key to assess whether our heuristics are relevant or not.

Conduct controlled experiments on various software systems — since our current case studies are limited to a few applications only. We will set up a larger benchmark made of representative applications to assess our profiling technique in diverse situations.

5. REFERENCES

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

[2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur,

Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM.

[3] 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.

[4] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Compact and efficient strings for java. *Science of Computer Programming*, 75(11):1077 – 1094, 2010. Special Section on the Programming Languages Track at the 23rd {ACM} Symposium on Applied Computing {ACM} {SAC} 08.

[5] Tomas Kalibera and Richard Jones. Rigorous benchmarking in reasonable time. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, pages 63–74, New York, NY, USA, 2013. ACM.

[6] Kiyokuni Kawachiya, Kazunori Ogata, and Tamiya Onodera. Analysis and reduction of memory inefficiencies in java strings. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA '08, pages 385–402, New York, NY, USA, 2008. ACM.

[7] 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.

[8] Nick Mitchell, Edith Schonberg, and Gary Sevitsky. Making sense of large heaps. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 77–97, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 562–571, Piscataway, NJ, USA, 2013. IEEE Press.

[10] Amit Sinha and Anantha Chandrakasan. Software energy profiling. In Robert Graybill and Rami Melhem, editors, *Power aware computing*, pages 339–359. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[11] Xiao Xiao, Jinguo Zhou, and Charles Zhang. Tracking data structures for postmortem analysis (nier track). In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 896–899, New York, NY, USA, 2011. ACM.

[12] Olivier Zendra. Memory and compiler optimizations for low-power and -energy. In *Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOPLPS'06), co-located with ECOOP'06*, July 2006.

[13] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGOPS Oper. Syst. Rev.*, 36(5):123–132, October 2002.