Identifying Caching Opportunities, Effortlessly

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

ABSTRACT

Memory consumption is a great concern for most non trivial software. In this paper we introduce a dedicated code profiler that identifies opportunities to reduce memory consumption by introducing caches.

Categories and Subject Descriptors

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

General Terms

Performance

Keywords

Memory, profiling

1. INTRODUCTION

In this paper we introduce a dedicated code profiler that identifies opportunities to reduce memory consumption by introducing caches.

Memory consumption is a great concern for most non trivial software [9]. Minimizing the memory consumption can be achieved in multiple different ways, including using a smart memory allocator [1] or garbage collector [4]. A number of programming patterns have been adopted to cope with excessive memory consumption. For example, memoization [10], and its generalization, the flyweight pattern [5], are used to share and reuse objects thus reducing the memory load.

Unfortunately, identifying the exact location in the code to apply a memoization or a flyweight pattern is not trivial. Understanding the propagation of side effects along the call-flow and monitoring the provenance of instances is usually necessary, even for trivial scenarios.

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

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

Builder>>defaultColor
  ^ Color r: 0.5 g: 0.5 b: 0.5
```

This example is written in the Pharo programming language[1] and 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.

Since Pharo do not offer any advanced memory profiler, the complaints of users are often what trigger a reaction from the developers. Excessive memory consumption is usually accompanied by high activity of the garbage collector, leading to lags and sluggish application behavior. This particular problem of the class `Builder` has been solved by the authors of the class by inserting a cache in the method `defaultColor`:

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

Behind the apparent simplicity of caching `defaultColor`, a number of questions have to be answered.

- *Is a color object mutable?* If an instance of the class `Color` may have one of its component color values (red, green or blue) modified during the execution of the application, then sharing a unique color with all the objects may seriously alter the behavior of the application, thus defeating our objective which is to reduce the memory consumption without impacting its behavior.
- *Why cache `defaultColor` and not `createElement`?* Caching colors seems the natural option in this case since the programmers are aware that elements are likely to have different positions and colors.

[1] `http://pharo-project.org`

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2768-8/14/05...$15.00
http://dx.doi.org/10.1145/2591062.2591198

Answering these two questions is not easy in most cases. Determining if a color is mutable or not requires checking for the presence of variable modifiers after initialization, which is in fact tricky to detect. Furthermore, instances may even be incrementally initialized, or use private modifiers during initialization, all of this incrementing the complexity of the analysis.

Deciding to insert the cache in the method `defaultColor` and not in `createElement` requires having knowledge about which objects are likely to be instantiated many times without being modified.

The research question investigated in this paper is: *Can the location to insert caches in the source code be determined by monitoring the program execution?* This paper (partially) addresses this question by sketching the description of a dedicated code execution profiler to help practitioners identify methods in which it is beneficial to insert caches.

Related work. Analyzing the presence of side effects in code and determining which objects are sharable have many applications, ranging from security to just-in-time compilation. The research community has proposed various techniques related to our research question. In particular, type ownership [3], sharable objects [2], and side-effect free methods ensured by a type system [8] are effective and expressive techniques to understand how caches may be designed to minimize the number of created objects. Unfortunately, such techniques make strong assumptions on the programming language, such as the presence of a specialized type system. Such assumptions do not hold in dynamically-typed languages [11].

To overcome this issue we propose a multi-stage profiling technique that uses dynamically collected data to reduce the profiling overhead, as presented in Section 2.

2. PROFILING TECHNIQUE

We have implemented a code profiler for Pharo that monitors instance creations and the mutations of these instances. Our profiler associates to each method of the application the instances its has created. It further groups objects based on their mutual *equivalence*. We propose the notion of equivalence as follows: two objects, o_1 and o_2, are equivalent if all variables pointing to o_1 can instead point to o_2 without affecting the behavior of the application. More specifically, two objects as equivalent if o_1 and o_2 are produced by the same class; if o_1 and o_2 have equivalent instance variables; both o_1 and o_2 are immutable after their construction phase (but mutation during their construction is allowed); neither o_1 and o_2 have their identity compared (either using their identity hash values or the identity test).

Our profiler outputs to the software developer details for each group of equivalent object instances, such as from which methods the group has been produced and its size. The notion of equivalence has already been presented in our previous work [6]. The equivalence is inspired from the work of Darko and O'Callahan [7].

Long execution and scalability. The first version of our profiler tracked all the instance creations for all the classes that define an application. Even if our profiler extracts a numerical blueprint of each objects, keeping tracks of a large number of objects results in excessive memory consumption.

The memory profiler now supports multi-stage profiling. A first profiling would identify what are the interesting classes to get more detail from (*e.g.,* the ones with a large number of instances, likely to be responsible to high memory consumption). A second profiling of the benchmarking execution reveals full detail about the mutation.Multi-stage profiling is effective at analyzing long program executions.

Multi-stage profiling assumes that a problematic scenario is easily reproducible. This is an assumption that does not fit all the situations (*e.g.,* long running and uninterruptible server), however it addresses the practical situations we are facing.

Relaxed equivalence. Another limitation we found in our first version of the profiler is the strong restriction imposed on mutability. The equivalence relation given in our previous work [6] rejects two objects as being equivalent if they are mutated after construction. It often happens in practice that objects are modified after initialization but may still be considered equivalent. For example, consider the following code excerpt:

```
p1 := Point x: 5 y: 10.
p2 := Point new.
p2 setX: 5 setY: 10.
```

If the two points `p1` and `p2` are not mutated during the remaining of the program execution and no identity check is carried out, then one of the two points is redundant. The object `p2` is apparently initialized after it has been constructed, therefore initialized with default values.

The real problem behind this example is the fact the boundary between the construction and the mutation is not clear. A first heuristic to set an end point to the construction of the objects is that the creation is finished at the end of the last factory method involved. But, as every heuristic, is not flawless. In the last example, the last factory method for `p2` would be `new`, so the extended construction, using the setters after calling `new`, would be considered as a mutation.

As another possible heuristic is to define the end of construction as the first access to the state of the object. Certainly we do not have the answer for which heuristic is the best, but defining this boundary is essential.

3. CONCLUSION & 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 methods 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 immutability – 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 after 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 benchmark made of representative applications to assess our profiling technique in diverse situations.

Acknowledgment. This work has been partially funded by FONDECYT project 1120094.

4. REFERENCES

[1] Suparna Bhattacharya, Mangala Gowri Nanda, K. Gopinath, and Manish Gupta. Reuse, recycle to de-bloat software. In *Proceedings of the 25th European conference on Object-oriented programming*, ECOOP'11, pages 408–432, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 2–27, London, UK, UK, 2001. Springer-Verlag.

[3] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings OOPSLA '98*, pages 48–64. ACM Press, 1998.

[4] Tamar Domani, Gal Goldshtein, Elliot K. Kolodner, Ethan Lewis, Erez Petrank, and Dafna Sheinwald. Thread-local heaps for java. In *Proceedings of the 3rd international symposium on Memory management*, ISMM '02, pages 76–87, New York, NY, USA, 2002. ACM.

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

[6] Alejandro Infante, Juan Pablo Sandoval, and Alexandre Bergel. Identifying equivalent objects to reduce memory consumption. In *Proceedings of 8th International Workshop on Smalltalk Technologies (IWST'13)*. ACM Digital Library, 2013.

[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] David J. Pearce. JPure:: A modular purity system for java. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 104–123, Berlin, Heidelberg, 2011. Springer-Verlag.

[9] 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.

[10] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE '07)*, pages 75–82, New York, NY, USA, 2007. ACM.

[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.