Hinting Refactorings to Reduce Object Creation In Java

Dries Buytaert*, Kristof Beyls*, Koen De Bosschere*

* *ELIS, Ghent University, Sint-Pietersnieuwstraat 41, 9000 Gent, Belgium*

ABSTRACT

In this paper, we study refactorings that reduce the number of objects created in Java programs. More specifically, we present a tool that automatically identifies the locations in Java applications where such refactorings are useful. Initial experiments show that the refactorings can result in significant savings in terms of memory usage and execution time.

KEYWORDS: program analysis, software refactoring, program optimization, performance debugging, object-oriented programming, garbage collection, Java

1 Introduction and Motivation

It has long been observed that many objects in languages with garbage collection have a short life-time [Hirz02, Lieb83, Stef94]. In other words, many objects become unreachable and garbage shortly after they have been created. In this paper, we investigate if we can prevent these short-lived objects from being created in the first place. We conjecture that most of these objects are created merely to communicate data from one location in the program to another. When there is a short time between the object's creation and its last use, we expect that a simple program refactoring might be possible to establish the communication between the two program locations without creating an object. Here, we understand refactoring to mean "changing the internal structure of the software, without changing its functionality" [Fowl00]. Whereas most refactorings focus on optimizing the design of the software, we focus on reducing the rate at which it creates new objects.

By reducing the number of allocated objects, less garbage is created. This results in less work for the garbage collector and a better temporal locality, potentially leading to significant speedups. Furthermore, in a number of cases, time-consuming code executed in object constructors can be avoided, leading to further speedups.

This paper presents a tool which helps to automatically find the locations in the source code where many short-lived objects are created, the locations where they are used for the last time, and the methods between which these objects are communicated. By intelligently

[1]E-mail: {Dries.Buytaert,Kristof.Beyls,Koen.DeBosschere}@elis.UGent.be

clustering all the objects observed during a profiling run, our tool groups together objects that can be optimized by the same refactoring. The main contributions of this paper are:

- A novel way to model the locations where objects are created, where they are used for the last time and the methods that communicate these objects.
- A simple clustering method based on that model. Each cluster corresponds to many objects created at run-time that can likely be eliminated by the same refactoring.
- We used our tool to optimize a number of programs from well-known benchmark suites. The optimizations typically require less than an hour of programmer effort, which is measured as the total time required to analyze the output of our tool, understand the source code constructs responsible for generating many short-lived objects, coming up with a suitable refactoring, and applying the required source code changes. In most cases, only a limited number of code lines need to be changed, without breaking the object-oriented design of the program. After refactoring, between 1.5 and 14 times fewer bytes are allocated, resulting in speedups between about 1.1 and 15.

In Section 2, the model of program execution is introduced. Section 3 presents how our tool measures the model data at run-time, and how it displays the measured results to the user. Section 4 discusses the results of using the tool, followed by concluding remarks.

2 Program Model

Programs typically generate millions of objects. Our tool aims at grouping these millions of objects into a few categories, such that each category can be optimized by a single refactoring. To make such categorization feasible, program execution is modeled as presented below.

Definitions *In Java, an object is created by using the keyword* `new`*. The method in which an object is created is called the object's* allocation site*. Likewise, the method where the object's address or any of its fields is last used, is called the* last use site*. The* Nearest Common Ancestor Method (NCA) *is the nearest common ancestor of the allocation site and the last use site in the dynamic call tree of the program, see Figure 1.* □

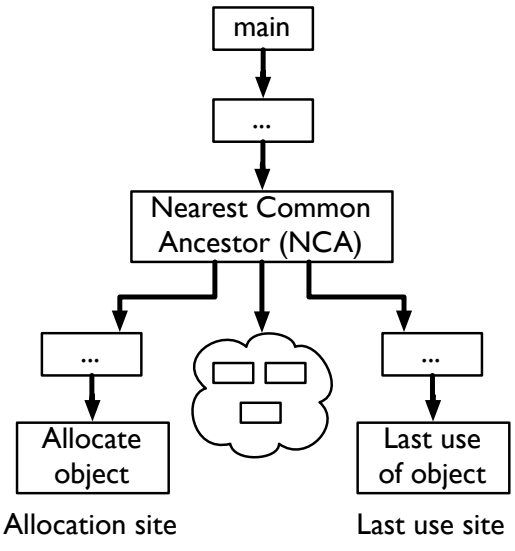
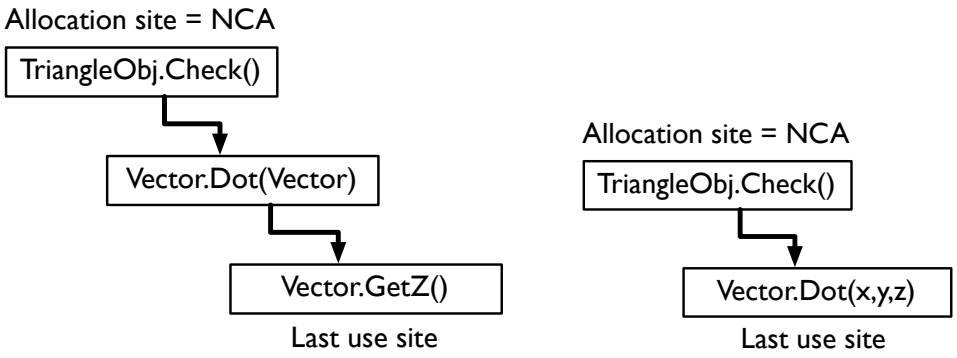


Figure 1: Allocation and last-use in a dynamic call-graph tree of a program. Every rectangle represents one method invocation at run-time. The lines connect the calling and the called methods.

Pinpointing the nearest common ancestor method is interesting, since it is the method with the smallest run-time scope where the object does not escape. As a result, the memory occupied by the object can be reused in different invocations of the NCA, e.g. by allocating the object in the stack frame of the NCA. When the NCA method finishes, the object



(a) (c)

(b)

Type	Objects	Short-lived	Overlap	Immutable	Size
`Vector`	1065903 (18%)	yes	no	yes	51 MB (17%)

Figure 2: Example output for `_227_mtrt`. (a) displays a cluster creating many short-lived objects. (b) shows the additional information recorded by our tool. The objects don't overlap, so at most one object is live at any given time. Objects of type `Vector` contain three `double` fields: `x`, `y` and `z`. We eliminated the objects by storing these three fields as local variables in the method `Check()`. We added a method `Dot(x,y,z)` to class `Vector` that performs the same computation as `Dot(Vector)`, and call that method instead, which eliminates the need for creating a `Vector` object.

memory is automatically freed by adjusting the stack pointer, and hence there's no garbage collection overhead. Another possibility is to use a software cache that recycles objects on the heap between different invocations of the NCA.

Our tool groups objects according to the tuple (NCA, allocation site, last use site), since these objects need to communicate data from one given allocation site, to a specific last use site, via the indicated NCA function. As such, we expect that optimizations might change code in any of these three methods.

3 Finding Useful Refactorings

To gather the profile information, the Jikes Research Virtual Machine (RVM) was modified to capture the following data for each object: (i) class type, (ii) size in bytes, (iii) allocation site, (iv) last use site, (v) time of allocation, (vi) time of last use, (vi) number of read operations, and (vii) number of write operations. This data is collected during program execution and stored in a database when the object is garbage collected.

To determine whether an optimization opportunity exists, the data is analyzed and quantified. First, the objects are clustered by the tuple (NCA, allocation site, last use site), as discussed in Section 2. Second, the clusters are sorted according to the number of bytes the objects in the cluster occupy. As such, the programmer can focus his efforts on the clusters which generate most garbage collection overhead.

After our tool identifies *where* to look for optimizations, it provides hints as *how* to optimize a particular cluster of objects. To assist the programmer in identifying viable refactorings, our tool presents additional information for each of the clusters: (i) whether the objects

Application	Refactoring	Programmer time	Lines changed
`ps`	Eliminate exception	$< \frac{1}{2}$ hour	22
`health`	Introduce software cache	< 1 hour	81
`_227_mtrt`	Communicate data through parameters instead of through object	< 1 hour	125

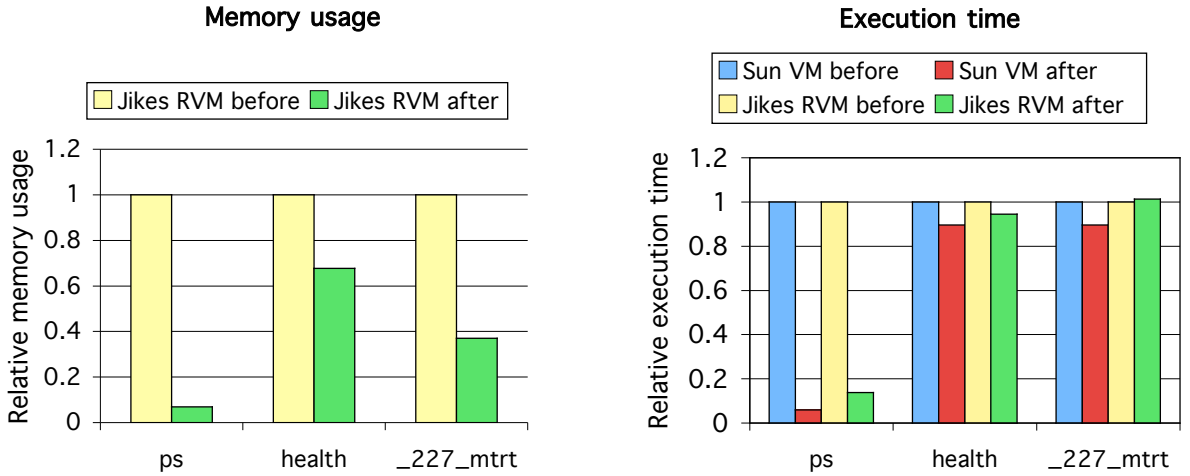


Figure 3: The results obtained by applying our tool to `ps`, `health` and `_227_mtrt`.

are long- or short-lived, (ii) whether the lifetime of the objects overlap with other objects in the cluster, (iii) whether they are immutable, (iv) their accumulated size, etc. An example is given in Figure 2, graphically showing the (NCA, allocation site, last use site) of one cluster in the mtrt program.

For example, objects whose lifetime never overlap are best eliminated using stack allocation or by refactoring the program's control flow. On the other hand, when there are many objects with overlapping lifetime, stack allocation is not an option. Instead, if many of these overlapping objects are both identical and immutable, they can be unified and represented by a single object, introducing the flyweight design pattern [Gamm96]. If they are not immutable, a third kind of refactoring can be applied to introduce a software cache, so that the heap space allocated by a previously allocated object that will never be used again, can be recycled without invoking the garbage collector.

To estimate the savings of a refactoring, the number of bytes allocated by a cluster can be used as a measure of the amount of garbage collection work saved.

4 Experimental Results

We used our tool to optimize a number of programs from well-known benchmark suites. Due to space constraints, here we focus on three Java applications; `ps` (DaCapo benchmark suite[2]), `health` (JOlden benchmark suite), and `_227_mtrt` (SPECjvm98 benchmark suite). `ps` reads and interprets a PostScript file, `health` simulates a Columbian health care system, and `_227_mtrt` is a multi-threaded raytracer.

[2]We used the beta050224 version of the DaCapo benchmark suite.

Figure 3 shows the memory and execution time savings normalized to the unmodified application. Figure 3(a) shows the memory reduction in terms of the total amount of allocated data. The reported numbers are the average of three runs. For `ps`, `health` and `_227_mtrt` the total amount of allocated memory has been reduced respectively by 93%, 33% and 63%. Similarly, Figure 3(b) depicts the impact of the refactorings on the total execution time. Because execution times depend on the heap size, we ran the applications with a range of heap sizes. The graphs show the average reduction in total execution time over all runs. Using Sun's production JVM, the execution time was reduced by 94% for `ps`, by 11% for `health` and by 11% for `_227_mtrt`. Using IBM's research JVM, the reductions are respectively 87%, 6% and -1%.

5 Conclusion and Future Work

We developed a tool that collects and clusters profile information to help the programmer find useful refactorings. In three case studies, the tool was used to identify and eliminate performance bottlenecks caused by creating many short-lived objects. Preliminary results show significant savings in terms of memory usage and execution time.

In the future, we plan the following steps. First, the tool will be used to examine a large set of programs, and distill a list of typical refactorings that are useful in reducing object creation. Second, the tool will be extended to suggest which refactorings from the list can be applied. Third, the potential profit of a proposed refactoring will be estimated by the tool, both in execution time and memory usage. Finally, we'll examine which refactorings could be automatized and implemented in a JVM.

Acknowledgments

Dries Buytaert is supported by a grant from the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT). Kristof Beyls is supported by research project GOA-12051002.

References

[Fowl00] M. FOWLER. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, 2000.

[Gamm96] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, Reading, 1996.

[Hirz02] M. HIRZEL, J. HENKEL, A. DIWAN, AND M. HIND. Understanding the Connectivity of Heap Objects. In *International Symposium on Memory Management (ISMM'02)*, pages 36–39. ACM, 2002.

[Lieb83] H. LIEBERMAN AND C. HEWITT. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, 1983.

[Stef94] D. STEFANOVIC AND J. MOSS. Characterization of object behaviour in Standard ML of New Jersey. In *LFP '94: Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 43–54. ACM, 1994.