Compiler-Cooperative Memory Management in Java (Extended Abstract)

Vitaly V. Mikheev, Stanislav A. Fedoseev

A.P. Ershov Institute of Informatics Systems,
Excelsior, LLC
Novosibirsk, Russia
{vmikheev,sfedoseev}@excelsior-usa.com

Abstract. Dynamic memory management is a known performance bottleneck of Java applications. The problem arises out of the Java memory model in which all objects (non-primitive type instances) are allocated on the heap and reclaimed by garbage collector when they are no longer needed. This paper presents a simple and fast algorithm for inference of object lifetimes. Given the analysis results, a Java compiler is able to generate faster code, reducing the performance overhead. Besides, the obtained information may be then used by garbage collector to perform more effective resource clean-up. Thus, we consider this technique as "compile-time garbage collection" in Java.
Keywords: Java, escape analysis, garbage collection, finalization, performance

1 Introduction

Java and other object-oriented programming languages with garbage collection are widely recognized as a mainstream in the modern programming world. They allow programmers to embody problem domain concepts in a natural coding manner without paying attention to low-level implementaion details. The other side of the coin is often a poor performance of applications written in the languages. The problem has challenged compiler and run-time environment designers to propose more effective architectural decisions to reach an acceptable performance level.

A known disadvantage of Java applications is exhaustive dynamic memory consumption. For the lack of stack objects — class instances put on the stack frame, all objects have to be allocated on the heap by the *new* operator. Presence of object-oriented class libraries makes the situation much worse because any service provided by some class, prerequires the respective object allocation. Another problem inherent to Java is a so-called *pending object reclamation* [1] that does not allow garbage collector to immediately utilize some objects even though they were detected as unreachable and finalized. The Java Language Specification imposes the restriction on an implementation due to the latent

caveat: if an object has a non-trivial finalizer (the **Object.finalize()** method overriden) to perform some post-mortem clean-up, the finalizer can resurrect its object "from the dead", just storing it, for instance, to a static field. Pending object reclamation reduces memory resources available to a running application.

Generally, performance issues can be addressed in either compiler or run-time environment. Most Java implementations (e.g. [2] [3]) tend to improve memory management by implementing more sophisticated algorithms for garbage collection [4]. We strongly believe that the mentioned problems should be covered in both compile-time analysis and garbage collection to use all possible opportunities for performance enhancement.

Proposition 1. *Not to junk too much is better than to permanently collect garbage*

We propose a scalable algorithm for object lifetime analysis that can be used in production compilers. We implemented the system in **JET**, Java to native code compiler and run-time environment based on the Excelsior's compiler construction framework [5].

The rest of the paper is organized as follows: Section 2 describes the program analysis and transformation for allocating objects on the stack rather than on the heap, Section 3 describes our improvements of the Java finalization mechanism. The obtained results are presented in Section 4, Section 5 highlights related works and, finally, Section 6 summarizes the paper.

2 Stack Allocating Objects

In Java programs, the lifetimes of some objects are often obvious whereas the lifetimes of others are more uncertain. Consider a simple method, getting the current date:

```
int foo() {
  Date d  = new Date();
  return d.getDate();
}
```

At the first glance, the lifetime of the object d is resctricted to that of method *foo*'s stack frame. That is an opportunity for a compiler to remove the *new* operator and allocate the object on the stack. However, we have to guarantee that no d aliases *escape* from the stack frame, that is, no aliased references to d are stored anywhere else. Otherwise, such program transformation would not preserve the original Java semantics. In the above example, the method *getDate* is a possible "escape direction".

Escape analysis dating back to the middle 1970s [6], addresses the problem. Many algorithms proposed vary in their application domains and time and spatial complexity. We designed a simple and fast version of escape analysis specially adapted to Java. Despite its simplicity, the algorithm shows promising results of benchmarking against widespread Java applications.

2.1 Definitions

All variables and formal parameters in the below definitions are supposed to be of Java reference types. By definition, formal parameters of a method also include the implicit "this" parameter (method receiver).

Definition 1 (Alias). *An expression expr is an alias of a variable v at a particular execution point, if v == expr (both v and expr refer to the same Java object)*

Definition 2 (Safe method). *A method is safe w.r.t its formal parameter, if any call to the method does not create new aliases for the parameter except, may be, a return value*

Definition 3 (Safe variable). *A local frame variable is safe, if no its aliases are available after method exit*

Definition 4 (Stackable type). *A reference type is stackable, if it has only a trivial finalizer*

Definition 5 (A-stackable variable). *A safe variable v is A-stackable, if a definition of v has the form of v = new T(..) for some stackable type*

Definition 6 (Stackable variable). *An A-stackable variable is stackable, if no local aliases of the variable exist before a repetitive execution of the variable definition in a loop, if any*

The stackable type definition is used to hinder a possible reference escape during finalizer invocation. The A-stackable to stackable variable refinement is introduced to preserve the semantics of the *new* operator: being executed in a loop, it creates different class instances so the analysis has to guarantee that previously cretead instances are unavailable.

2.2 Program Analysis and Transformation

To detect if a variable is not safe, we distinguish two cases of escape:

1. *explicit*: **return v**, **throw v** or **w.field = v** (an assignment to a static or instance field)
2. *implicit*: **foo (...,v, ..)** invocation of a method non-safe w.r.t **v**

Operators like **v = v1** are subject for a flow-insensitive analysis of local reference aliases (LRA) [10]. In order to meet the requirement for loop-carried variable definitions, the algorithm performs a separate LRA-analysis within loop body. Determining of safe methods is proceeded recursively as a detection of their formal parameter safety except the *return* operator. In such case, the return argument becomes involved into local reference aliasing of the calling method. We implemented our algorithm as a backward inter-procedural static analysis on call graph like algorithms described in related works [7],[9]. We omit the common

analysis scheme due to its similarity to those of related works and focus on some important differencies further.

Once stackable variables have been detected, the respective *v = new T(..)* operators are replaced with the *v = stacknew T(..)* ones from internal program representation. Besides, the operators like *v = new T[expr]*, allocating variable length arrays are marked with a tag provided for subseqent code generation. That makes sense because our compiler is able to produce code for run-time stack allocation.

2.3 Implementation Notes

The Excelsior's compiler construction framework features a statistics back-end component [5] making it a suitable tool of statistic gathering and processing for any supported input language. Also, we had a memory allocation profiler in the run-time component so we were able to analyze a number of Java applications. We found that the algorithms described in related works may be somewhat simplified without sacrificing effectiveness. Moreover, the simplification often leads to better charateristics such as compilation time and resulting code size.

Type inference. So far, we (implicitly) supposed that all called methods are available for analysis. However, Java being an object-oriented language, supports virtual method invocation — run-time method dispatching via Virtual Method Tables that hinders any static analysis. Type inference [13] is often used to avoid the problem to some extent. Our algorithm employs a context-sensitive local type inference: it starts from the known local types sourcing from local *new T(..)* operators and propagates the type information to called method context. We used a modified version of the rapid type inference pursued in [12]. Another opportunity which helps to bypass the virtual method problem is global type inference based on the class hierarchy analysis [11]. We implemented a similar algorithm but its applicability is often restricted because of the Java dynamic class loading. We did not consider polyvariant type inference (analysis of different branches at polymorphic call sites) due to its little profit in exchange for the exponential complexity.

Inline substitution. Local analysis in optimizing compilers is traditionally stronger than inter-procedural because, as a rule, it requires less resources. This is why inline substitution not only removes call overhead but also often improves code optimization. Escape analysis is not an exception from the rule: local variables that were not stackable in the called method may become so in the calling one, for instance, if references to them escaped via the *return* operator. Escape analysis in Marmot [7] specially treats called methods having that property to allocate stack variables on the frame of calling method. In the case, called method should be duplicated and specialized to add an extra reference parameter (Java supports metaprogramming so the original method signature may not be changed). In our opinion, that complicates analysis with no profit:

the same problem may be solved by an ordinary inline substitution without the unnecessary code growth.

Native method models. The Java language supports external functions called *native methods*. They are usually written in C and unavailable for static analysis. However, certain native methods are provided in standard Java classes and should be implemented in any Java run-time or even compiler, for instance the **System.arraycopy** method. Because the behaviour of such methods is strictly defined by the Java Language Specification [1], we benefit from using so-called *model methods* provided for analysis purposes only. A model native method has a fake implementation simulating the original behaviour interesting for analysis. Employing model methods improves the overall precision of escape analysis.

2.4 Complexity

In according to [14], given restrictions even weaker than ours, escape analysis can be solved in linear time. The rejection of analyzing polyvariant cases at virtual call sites and the restriction of reference aliasing to local scopes only give the complexity proportional to N (program size) + G (non-virtual call graph size). Thus, our algorithm performs in O(N+G) both time and space.

3 Finalization

The described algorithm determining *safe methods* may be used for more effective implementation of pending object reclamation in Java. As mentioned above, an object having a non-trivial finalizer is prevented from immediate discarding by a garbage collector. The main problem provoking a significant memory overhead is that all heap subgraph reachable from the object may not be reclaimed as well: finalizer may potentially "save" (via aliasing) any object from the subgraph.

To overcome the drawback, we adapted the algorithm to detect whether the finalizer is a safe method with respect to its implicit "this" parameter and other object's fields aliased from "this". The analysis results are then stored by compiler to the class object (a Java metatype instance [1]). Given that, garbage collector makes a special treatment for objects with trivial or safe finalizers. More specifically, the run-time system constructs a separate list for objects which require pending reclamation whereas other objects are processed in a simpler way. The measurement results for the optimization are listed in the next section.

4 Results

We implemented the described optimizations as a part of the JET compiler and run-time environment. We selected the Javacc parser generator, the Javac bytecode compiler from Sun SDK 1.3 and Caffein Dhrystone/Strings benchmarks to evaluate resulting performance of the escape analysis application. The

results are shown in Table 1 (the numbers were computed as *NewExecutionTime/OldExecutionTime*). The performance growth is achieved as a result of both faster object allocation and less extensive garbage collection.

These tests were choosen due to their batch nature that allows us to measure the difference in total execution time. Despite the results for the first three benchmarks are valuable, applying the optimization to the Javac compiler had only minimal effect — no silver bullet. Unfortunately, the results may not be directly compared with the results obtained by other researchers. The comparison of different algorithms may be accomplished only within the same optimization and run-time framework. For instance, a system with slower object allocation and garbage collection or better code optimization would obviously experience more significant performance improvement from the stack allocation.

Results of optimized finalization are given in Table 2. JFC samples (Rotator3D, Clipping, Transform, Lines) using Java 2D-graphics packages were chosen because of very intensive memory consumption. We measured the amount of free memory just after garbage collecting and the numbers were computed as *NewFreeMemory/OldFreeMemory*. The total amount of heap memory was the same for all tests and equal to 30MB.

Benchmark	**Execution time fraction**
Javacc	0.54
Dhrystone	0.32
Strings	0.2
Javac	0.98

Table 1. Stack allocating objects

Benchmark	**Free memory fraction**	**Memory profit, MB**
Rotator3D	1.1	+1.5
Clipping	1.15	+1.2
Transform	1.08	+0.7
Lines	1.13	+1.7

Table 2. Optimized finalization

We noted that even with the optimizations enabled, the total compilation time remains virtually unchanged. Analyzing obtained results, we draw a conclusion that the considered object-oriented optimizations may be employed by production compilers. All further information related to the JET project may be found at [18].

5 Related Works

An number of approaches have been proposed for object lifetime analysis. Many works were dedicated to functional languages such as SML, Lisp etc. ([14], [15], [16]). The power of the escape analyses supercedes ours to a great extent, however the complexity of the algorithms is not better than polynomial. The escape analysis for Java was investigated by reseachers using static Java analyzing frameworks. Except the JET compiler, the related works were completed on the base of the TurboJ via-C translator [9], the IBM HPJ compiler [10] and the Marmot compiler project at Microsoft Research [7]. The algorithm presented is simpler but, nevertheless, quite effective and precise so it may be used even in dynamic compilers built in the most current Java Virtual Machines [2], [3]. Besides, the related works discuss only stack allocating objects whereas our approach also considers garbage collection improvement basing on the compile-time analysis.

6 Conclusion

This paper presented a technique for fast and scalable object lifetime analysis. Being used in cooperative compiler and run-time framework, the implemented optimizations profit in both execution speed and memory consumption of Java applications. The interesting area for future works is to investigate a region inference algorithms allowing compiler to approximate object lifetimes between method call boundaries. Despite the applicability of such analysis to compiler optimizations is doubt, the information may be used for more effective garbage collection in compiler-cooperative run-time environment.

References

1. J. Gosling, B. Joy and G.Steele: *The Java Language Specification.* Addison-Wesley, Reading, 1996
2. Gu et al.: *The Evolution of a High-Performing JVM.* IBM Systems Journal, Vol. 39, No. 1, 2000
3. *The Java HotSpot(tm) Virtual Machine*, Technical Whitepaper, Sun Microsystems Inc., Whitepaper, 2001.
   http://www.sun.com/solaris/java/wp-hotspot
4. D. Detlefs, T. Printezis: *A Generational Mostly-Concurrent Garbage Collector.* In Proc. ACM ISMM'00, 2000
5. V.V. Mikheev: *Design of Multilingual Retargetable Compilers: Experience of the XDS Framework Evolution* In Proc. Joint Modular Languages Conference, JMLC'2000, Volume 1897 of LNCS, Springer-Verlag, 2000
6. J.T. Shwartz: *Optimization of very high-level languages -I. Value transmission and its coloraries.* Computer Languages, 1(2), 1975
7. D. Gay, B. Steensgaard: *Stack Allocating Objects in Java.* Reasearch Report, Microsoft Research, 1999
8. J.-D. Choi et al.: *Escape Analysis for Java.* SIGPLAN Notices, Volume 34, Number 10, 1999

9. B. Blanchet: *Escape Analysis for Object Oriented Languages. Application to Java(tm).* SIGPLAN Notices, Volume 34, Number 10, 1999
10. D. Chase, M. Wegman, and F. Zadeck: *Analysis of pointers and structures.* SIGPLAN Notices, Volume 25, Number 6, 1990
11. D. Bacon: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages.* Technical Report, University of California, 1997
12. D. Bacon, P. Sweeney: *Fast static analysis of C++ virtual calls.* In Proc. OOPSLA'96, SIGPLAN Notices, Volume 31, Number 10, 1996
13. O. Agesen: *The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism.* In Proc. ECOOP'95, Aarhus, Denmark, 1995
14. A. Deutch: *On the complexity of escape analysis.* In Conference Record of POPL'97, The 24th ACM SIGPLAN-SIGAST, 1997
15. S. Hughes: *Compile-time garbage collection for higher-order functional languages.* Journal of Logic and Computation, 2(4), 1992
16. K. Inoue et al.: *Analysis of functional programs to detect run-time garbage cells.* ACM Transactions on Programming Languages and Systems, 10(4), 1988
17. Y. Park, B. Goldberg: *Escape analysis on lists* In Proceedings of PLDI'92, ACM SIGPLAN, 1992
18. *JET Deployment Environment.* Excelsior LLC, Technical Whitepaper, 2001. **http://www.excelsior-usa.com/jetwp.html**