Compile-Time Deallocation of Individual Objects [*]

Sigmund Cherem and Radu Rugina

Computer Science Department
Cornell University
Ithaca, NY 14853
{siggi, rugina}@cs.cornell.edu

Abstract

This paper presents a static analysis and transformation system that enables the deallocation of individual objects in Java programs. Given an input program, the compiler automatically inserts *free* statements to deallocate individual objects. This transformation is enabled by an inter-procedural, context-sensitive dataflow analysis that tracks the state of one object instance at a time, from the point where it is allocated, and up to the point where the object instance becomes unreachable and can be freed.

For the SPECjvm98 benchmarks, free-instrumented programs generated by our compiler and executed in a virtual machine with explicit memory deallocation reclaim, on average, more than 50% of the total memory allocated by the program, and have a low run-time overhead of 1%. For several benchmarks, the analysis can free more than 85% of the total memory.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers, Memory management; F.3.2 [*Semantics of Prog. Languages*]: Program Analysis

General Terms Languages, Performance

Keywords Compile-time memory management, individual object deallocation, shape analysis, program transformations.

1. Introduction

Compile-time memory management refers to compiler transformations that automatically augment programs with explicit memory reclamation code. To enable such transformations, the compiler statically approximates object lifetimes and identifies program points where memory reclamation can be safely performed. Unlike dynamic memory reclamation via garbage collection, static memory management has lower run-time overhead because most of the work of identifying object lifetimes is done at compile-time. Unlike manual memory deallocation, compiler-enabled memory reclamation is safe. The price that static approaches must pay for safety and run-time efficiency is that they must be conservative and over-approximate object lifetimes, and thus are likely to reclaim less memory.

[*] This work was supported in part by National Science Foundation grants CCF-0541217 and CNS-0406345.

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.
Copyright © 2006 ACM 1-59593-221-6/06/0006. . . $5.00.

Two main directions have been explored in the area of compile-time memory management: stack allocation via escape analysis, and region allocation via region inference. The first approach, escape analysis, identifies objects that do not escape their static scopes, and then allocates such objects on the run-time stack. These objects are automatically deallocated when methods return. Existing work in this area has targeted both functional languages [3, 4], and imperative languages [9, 24, 6]. The proposed techniques for Java typically combine escape analysis with some form of pointer analysis. The second approach, region inference, determines how to group heap objects together into regions, and when regions can be deallocated. Alternatively, the user can specify region deallocation points using a region type system. Such techniques have been initially developed for functional languages [21, 1]; more recently, they have been applied to imperative languages [10, 11, 7, 8]. Regions can improve data locality, and can allow the efficient deallocation of entire collections of objects.

This paper addresses the static memory management problem for imperative languages using *individual object deallocation*. The goal of the compiler is to automatically insert *free* statements that reclaim single object instances. Compared to the other two static memory management techniques, this approach has several appealing properties:

- *Flexibility*: Individual object deallocation allows flexible, unrestricted object lifetimes. Unlike stack allocation or lexically-scoped region systems, object lifetimes are not required to follow a stack discipline and can span across multiple procedures. More important, objects allocated at the same site can have different lifetimes and can be deallocated at different points in the program. In other words, object lifetimes are not tied to the allocation points;
- *Simplicity and efficiency*: Individual object deallocation requires minimal program changes. When used as the only memory management mechanism, individual object deallocation doesn't require specialized run-time support, but just the standard malloc-free mechanism. In contrast, region-based systems require more program changes for manipulating region handles (allocating handles, using them, and passing handles as parameters), and are likely to impose higher run-time costs, both in time and in space.

The key aspect that distinguishes individual object deallocation from the stack- and region-allocation techniques is that the analysis must reason about object *instances* and their lifetimes, not about collections of objects and conservative approximations of their lifetimes. Hence, standard abstractions, such as allocation-site summaries used in points-to analyses, are not a good match for this problem. Shape analysis abstractions and techniques are more appropriate, because they can compute precise heap reference counts, even in the presence of destructive heap mutations [18].

This paper addresses the problem of individual object deallocation using a shape analysis framework that we recently proposed [13], where the compiler analyzes one single heap cell at a time. We develop a new analysis for Java in this framework, and show how to use the analysis for the compile-time deallocation of individual objects. For each dynamic allocation site, the analysis tracks the reference counts of one representative object instance allocated at this site through the program, up to the point where it becomes unreachable. At that point, the compiler inserts a free statement. We believe that this approach is a good match to the problem of individual object deallocation, since tracking one single object instance at a time makes it easier to determine when that particular instance can be freed.

An issue that arises in the context of individual object deallocation concerns multi-level heap structures that die at once, when the program writes the last reference to the head of the structure. In such cases, the compiler must "traverse" the structure and deallocate the structure objects individually. Our compiler addresses this issue by re-analyzing the program when new free statements are inserted, trying to identify more objects that can be deallocated as a result of those frees.

We have implemented a prototype compilation system JFREE that inserts free statements in Java bytecode programs. We have extended Jikes RVM [2] to run the free-instrumented programs generated by our compiler. On average for the SPECjvm98 benchmarks, the transformed programs reclaim more than 50% of their total allocated memory, at a low run-time cost of just 1% more than a system with no memory reclamation. For several benchmarks, the analysis can free more than 85% of the total memory. We have also experimented with a combined system that supports explicit reclamation via free and runs mark-and-sweep collection at the same time. In this setting, our compiler improves the performance of the collector, reducing the application running time by up to 31% for small heaps. This numbers are consistent with a recent study that compares explicit memory reclamation and garbage collection [15].

The rest of the paper is organized as follows. Section 2 presents a simple example program. Section 3 presents the main structure of our algorithm, we discuss the analysis details in Section 4 and 5. Section 6 presents experimental results. Finally, we discuss related work in Section 7 and conclude in Section 8.

2. Example

Figure 1 shows an example program that illustrates some of the difficulties of individual object deallocation. A similar heap manipulation occurs in *compress*, one of the SPECjvm98 benchmark programs.

The program uses two classes `A` and `B` and manipulates array objects in the loop of method `run`. At each iteration, the loop creates a new array, initializes it, and then performs a computation using the arrays created in the last two iterations. At line `7` the program allocates an object `B`, and stores in it a reference to the old array. Then, at line `8` method `fill` allocates the new array and stores a reference to it in field `a`. Finally, at line `9` the program calls `compute` to perform a computation using the two arrays.

For each object instance, the analysis tries to determine a program point where the object can be safely deallocated. There are two allocation sites: one at line `13` in method `fill` for the array object; and another at line `7` in `run` for the object `B`.

2.1 Deallocating an Object Instance

Identifying the lifetime of each `B` object is relatively easy. The reference to this object is placed into several local variables: variable `x` and the receiver `this` of two methods, `compute` and `B`'s constructor. However, references to `B` objects are never written into the heap. The task of the analysis is to determine, at each program point, the

```
 1  class A {
 2      int[] a;
 3
 4      void run(int n) {
 5          B x = null;
 6          while(true) {
 7              x = new B(this.a);
 8              this.fill(n);
 9              x.compute(this.a);
10          }
11      }
12      void fill(int n) {
13          this.a = new int[n];
14          while(n > 0) this.a[--n] = ...;
15      }
16  }
17
18  class B {
19      int[] b;
20      B(int[] c) { this.b = c;  }
21      void compute(int[] d) { ... }
22  }
```

Figure 1. (a) Example program

set of all of the variables that point to the same instance of an object `B`; when this set becomes empty, all of the references to the object are lost and the object can be freed.

Our analysis determines these facts by tracking the state of one *instance* of an object `B` allocated at line `7` through the program. The compiler performs a flow analysis starting from this allocation site, and tracks the references to this object through the loop body, through the call to `compute` (where the object temporarily has a reference count of 2), and then back to the top of the loop. At that point, the analysis determines that `x` holds the last reference to this object instance, and that `x` is being written with a different value. The analysis can therefore insert a deallocation statement `free(x)` before line `7`. Similarly, a `free(x)` is inserted when local variable `x` goes out of scope at line `11`, when method `run` returns.

2.2 Structure Deallocation

Determining the lifetime of the array allocated at line `13` is significantly more complicated. This array escapes the method `fill` that has allocates it. Even worse, during the call to `compute` there are two object instances created at the same site, that are live at the same time.

Again, our analysis can successfully deallocate the array object in this example. As before, the analysis tracks the state of this object through the program, from its allocation site at line `13`, to its caller `run`, through function `compute`, then to the top of the loop, and so on. The analysis computes a set of possible states for the array at each program point. In particular, after line `7`, there are two possible states for this object: a state where the array has a reference count of 2, from `this.a` and `x.b` (this is the array allocated in the previous iteration); and one state where the array has a reference count of 1, from an object `B` (this is the array allocated two or more iterations before). The update of field `a` performed by `fill` decreases the reference count in the first state, but not in the second. In fact, the reference count of the object never goes down to zero, because it is always referenced by an object `B`.

The key observation is that all of the "old" references come from instances of objects `B` that became themselves not reachable. To avoid counting references from unreachable objects `B`, our compiler re-analyzes the array object after inserting the deallocation statement `free(x)` described above. In the subsequent analysis,

```
                               Iter 1   Iter 2      Iter 3
 6: top of loop body:                   (t1.a)      (x.b)
 6: free(x)                             (t1.a)      ()
 7: x = new B(t1.a)                     (t1.a,x.b)
 8: t1.fill(n)
12: -> t2.fill(n)                       (t2.a)+
13:    t2.a = new int[n];      (t2.a)   ()+
14:    while(...) ...;         (t2.a)   ()+
 8  <-
 9: x.compute(t1.a)            (t1.a)   (x.b)
```

Figure 2. Tracking the array allocated at line 13.

the algorithm treats this free statement as a nullification of all of the fields of `x`.

Figure 2 shows the analysis result for the array object in our example, after the insertion of `free`. This is the fixed point result for the loop body. The first column shows a trace of the code. The arrows `->` and `<-` indicate method call and method return points. We use different names for the receiver `this` in different methods: `t1` and `t2` denote the receiver `this` of methods `run` and `fill`, respectively. The columns in the right of the figure show the possible states for the tracked object. Each state is denoted using parentheses $(\ldots)$ and indicates the references to the array object. States describe "must reference" relationships; and include all of the incoming references to the array. The analysis records only references in the current scope; it explicitly marks with "+" those states where the callers hold additional references to the object, to forbid the callee from deallocating such objects. The caller references are recovered when the callee returns.

The analysis begins with the allocation site at line 13. Right after this point, there is a new state (t2.a) describing the newly allocated object. The object allocated in the previous iteration is described using (t2.a)+ before the allocation; and using ()+ after the allocation.

There are two possible states of the array at the top of the loop body: (t1.a) and (x.b). Since the `free(x)` is treated as `x.b = null`, the second state describes an object that becomes unreachable after `free`. The compiler can now successfully deallocate the array by freeing `x.b`, but only after checking that `x` is not null. The transformed program will contain the following statements at the top of the loop body, before line 7:

```
if (x != null) free(x.b);
free(x);
```

Essentially, these lines of code deallocate a structure of two objects that simultaneously become unreachable. In general, larger structures, with multiple levels of indirection, can become unreachable all at once. To deallocate such structures, our analysis iterates several times, freeing objects deeper in the structure at each iteration.

3. Main Algorithm

Given an input program, the compiler analyzes all of the allocation sites and automatically inserts code containing explicit `free` statements to deallocate individual objects. The analysis consists of the following main phases:

- *Tracked Object Analysis*, a dataflow analysis that computes heap reference counts at each program point, and identifies program points where particular object instances become unreachable;
- *Free Transformation*, an analysis and transformation that examines the results of the previous phase and inserts the appropriate deallocation code in the program.

The second phase can lead to opportunities for more object deallocations, since freeing an object might remove the last reference for other heap objects. Our analysis iterates between the two phases until no free statements are inserted. At each iteration, the compiler invalidates the dataflow information that is affected by the newly generated free statements and then runs the tracked object analysis incrementally. The following sections describe these phases in more detail.

4. Tracked Object Analysis

This section presents the tracked object analysis algorithm. This analysis builds on the heap reference counting analysis framework that we developed in our earlier work [13]. First, we give an overview of the analysis framework. Then, we present a new formulation for Java programs.

4.1 Overview of Reference Counting Shape Analysis

The goal of the analysis is to compute accurate reference counts for heap cells. The analysis requires a coarse-grain partitioning of the program memory (stack and heap) into a finite set R of disjoint regions, and points-to information between these regions. For C programs, such a partitioning can be obtained using a standard points-to analysis. The analysis framework then uses the region partitioning to build a non-standard heap abstraction with reference counting information. The heap abstraction consists of a finite set of *configurations*, each of which models a possible state of one single heap cell. If multiple heap cells have the same state, the configuration essentially describes one representative among those equivalent cells. The key aspect is that configurations are independent, so that the algorithm can analyze them in isolation.

A configuration describes a heap cell using a triple (rc, H, M), where $rc : R \to \{0, 1, .., k, \infty\}$ indicate precise reference counts from each region to the cell in question; H is a set of program expressions known to reference the cell (hit expressions); and M is a set of expressions that do not reference the cell (miss expressions). The maximum number of references per region is bounded by a fixed value k (usually 2). Hit and miss expressions are needed in order to accurately restore reference counts after destructive updating operations. For instance, an acyclic list structure pointed to by variable x can be represented using two independent configurations: $(X^1, \{x\}, \varnothing)$ and $(L^1, \varnothing, \varnothing)$, where X is the region that contains the memory location of variable x, L is the heap region that contains all list elements, and superscripts indicate reference counts. The configuration $(X^1, \{x\}, \varnothing)$ describes the first element of the list, and the configuration $(L^1, \varnothing, \varnothing)$ describes a representative cell from the tail of the list. Because the two configurations are independent, the compiler can analyze them separately. The framework then uses a dataflow analysis to determine how configurations change through the program.

The transfer function for a statement maps a configuration before the statement to the set of possible configurations after the statement. To be able to maintain precise reference counts, the analysis must determine whether the left and right hand-side expressions reference the cell. When these facts cannot be determined from the state described by the configuration, the analysis bifurcates into multiple possible states that clarify this uncertainty. It then analyzes each of the resulting cases in turn.

The starting points of the analysis are the allocation sites. At each site, the analysis manufactures a configuration that describe one representative heap cell created at that site. It then tracks this configuration through the flow of the program. An interprocedural component propagates configurations across procedures in a context-sensitive manner.

Program statements

$$\begin{aligned} st ::= \;& x = y \mid x = \mathsf{null} \mid x = y.f \mid \\ & x.f = y \mid x.f = \mathsf{null} \mid x = \mathsf{new} \mid \\ & x = y_0.m(y_1, ..., y_n) \mid \mathsf{return\ ret} \end{aligned}$$

Program expressions

$$e ::= \quad \mathsf{null} \mid x.\overline{f} \qquad \text{where} \quad \overline{f} ::= \varepsilon \mid \overline{f}.f_k$$

Figure 3. Program representation

4.2 Tracked Object Analysis for Java

This section presents reference counting analysis with tracked objects for Java. This formulation differs from the above in two respects. First, the analysis relies on type safety to partition the memory using types, eliminating the need for pointer analysis. Furthermore, because pointers to local variables are not allowed, only heap regions need to be considered. Intuitively, each variable has its own region, denoted by the variable itself. Second, we have augmented the abstraction with expression equivalences that indicate program expressions having the same value. The analysis uses this information to derive more accurate knowledge about expressions that reference the tracked object.

We formalize the analysis as follows. A tracked object is represented by a tuple of the form $L = (rc, S, h, M, a)$, where:

- rc : $\mathsf{Field} \rightarrow \{0, 1, ..., k, \infty\}$ describes heap reference counts. Here, Field is the set of class fields. $rc(f)$ is the number of references to the tracked object from other objects via field f.
- S is a partitioning of expressions into equivalence classes. Two expressions are in the same equivalence class if they have the same value. Expressions in different classes may or may not have the same value.
- $h \in S$ is a hit class. It describes the expressions that reference the object. Notice that rc must include all heap references mentioned in h, i.e., if $e.f \in h$ then $rc(f) > 0$.
- $M \subseteq S$ is a set of miss classes. These expressions do not reference the tracked object.
- a is a boolean flag indicating whether the tracked object has external references in the callers of the current method.

Figure 3 presents the program representation used in our formalization. Statements are represented in standard three-address form, and include assignments, allocations, and method calls. We assume the program contains explicit null assignments $x = \mathsf{null}$ when variables go out of scope. The return value of a method is stored in a special variable ret. Program expressions e can be either a null constant or variables followed by a possibly empty sequence of field accesses (denoted by $\overline{f}$). Other language constructs, such as integer expressions and array accesses, are handled in a similar fashion, but omit them to simplify the presentation.

4.2.1 Analysis Initialization

The algorithm uses a worklist algorithm to propagate the state of the tracked object through the program. The analysis starts by introducing a configuration $L = (rc, S, h, M, a)$ for each allocation site $x = \mathsf{new}$, at the program point after the allocation. The configuration is initialized as follows:

$$\begin{aligned} rc &= \{f \mapsto 0 \mid f \in \mathsf{Field}\} \\ h &= \{x\} \\ M &= \{\{x.f_1, ..., x.f_n, \mathsf{null}\}\} \\ S &= \{h\} \cup M \\ a &= \mathsf{false} \end{aligned}$$

The configuration describes an object instance right after the allocation point: the new object is referenced just by the variable x, and all the object fields are null. The analysis then tracks this configuration though the control-flow of the program.

$$\begin{aligned} [\![x = \mathsf{new}]\!]_{\mathsf{s}} s \;&=\; (s - \{x.\overline{f} \mid \text{for any } \overline{f}\}) \,\cup\, \{x.f \mid \mathsf{null} \in s\} \\ [\![x = e]\!]_{\mathsf{s}} s \;&=\; (s - \{x.\overline{f} \mid \text{for any } \overline{f}\}) \,\cup\, \{x.\overline{f} \mid e.\overline{f} \in s\} \\ [\![x.f = e]\!]_{\mathsf{s}} s \;&=\; (s - \{e'.f.\overline{g} \mid e' =_? x\}) \\ &\qquad \cup\, \{e'.f.\overline{g} \mid e.\overline{g} \in s \,\wedge\, e' =_v x\} \end{aligned}$$

$$\begin{aligned} [\![x = e]\!]_{\mathsf{r}} rc \;&=\; rc \\ [\![x.f = e]\!]_{\mathsf{r}} rc \;&=\; rc[f \mapsto i], \text{ where} \end{aligned}$$

$$i = \begin{cases} rc(f) + 1 & \text{if } e \in h \,\wedge\, x.f \notin h \\ rc(f) - 1 & \text{if } e \notin h \,\wedge\, x.f \in h \\ rc(f) & \text{otherwise} \end{cases}$$

Figure 4. Operations used by transfer functions. The relation $=_v$ denotes must-equality, and the relation $=_?$ denotes may-aliasing.

4.2.2 Intraprocedural Analysis

Let $L = (rc, S, h, M, a)$ be an input configuration describing the state of the tracked object before an assignment statement st of the form $e_1 = e_2$. For each of the two expressions e_i, with $i \in \{1, 2\}$, the analysis tries to determine whether e_i references the tracked object or not. If $e_i \in h$, then e_i definitively hits the tracked object. If $e_i \notin h$, then e_i misses the object if: 1) e_i belongs to a miss class in M; or 2) $e_i = x$ is a variable that doesn't belong to the hit set: $x \notin h$; or 3) $e_i = x.f$ is a field expression such that $rc(f) = 0$. If the analysis cannot determine whether e_i hits or misses the tracked object, it creates two new configurations, one where the e_i hits (L_H), and one where it misses (L_M):

$$\begin{aligned} L_H &= (rc, S - h - s_e \cup \{h \cup s_e\}, h \cup s_e, M, a) \\ L_M &= (rc, S, h, M \cup \{s_e\}, a) \end{aligned}$$

where s_e is the equivalence class of e_i in S, or, if no such class exists, then $s_e = \{e_i\}$. After this process, we say that the resulting configurations are focused with respect to e_i. The analysis focuses the initial configuration with respect to each of the expressions e_1 and e_2.

Next, the analysis applies the transfer functions to each of the focused configurations. For each configuration L before the current statement st, the transfer function computes a configuration L' after st. Figure 4 presents two operations $[\![\cdot]\!]_{\mathsf{s}}$ and $[\![\cdot]\!]_{\mathsf{r}}$ used to define all transfer functions. The operation $[\![\cdot]\!]_{\mathsf{r}}$ updates the reference counts. For store statements, reference counts are incremented or decremented appropriately. For assignments to variables, the reference counts remain unchanged. The operation $[\![\cdot]\!]_{\mathsf{s}}$ updates one equivalence class $s \in S$. The operation kills equivalences that are no longer valid, either because a variable is being updated (in copy and load assignments) or because a field is being written (store assignments). Store assignments $x.f = y$ conservatively kill expressions of the form $e.f$, whenever e may be aliased with x. To determine whether two expressions may be aliased, the analysis uses the information in L, as follows. Two expressions a and b are aliased, written $a =_v b$, if they belong to the same equivalence class:

$$\exists s \in S \,.\, \{a, b\} \subseteq s \Rightarrow a =_v b$$

The expressions are not aliased if one of them hits the tracked object and the other misses it:

$$(\{a, b\} \cap h \neq \varnothing \;\wedge\; \exists m \in M \,.\, \{a, b\} \cap m \neq \varnothing) \;\Rightarrow\; a \neq_v b$$

If $\neg(a \neq_v b)$, we say that the two expressions may be aliased: $(a =_? b)$.

With the definitions from Figure 4, the transfer functions are:

$$[\![\mathsf{st}]\!](rc, S, h, M, a) = ([\![\mathsf{st}]\!]_{\mathsf{r}} rc, [\![\mathsf{st}]\!]_{\mathsf{s}} S, [\![\mathsf{st}]\!]_{\mathsf{s}} h, [\![\mathsf{st}]\!]_{\mathsf{s}} M, a)$$

where $[\![\mathsf{st}]\!]_{\mathsf{s}} S$ is a shorthand notation for applying the operation to each element in the set S.

To define the merge operation, we separate the information of each configuration in two pieces: an index and the secondary information. Our analysis maintains the invariant that, at each program point, there is at most one configuration with a given index. Given two sets of configurations, the merge operation compares and combines configurations with the same index.

We define the index of a configuration as the heap reference count and all hit expressions shorter than a fixed length r:

$$I(L, r) = (rc, \{x.\overline{f} \mid x.\overline{f} \in h, |\overline{f}| \leq r\})$$

where $|\overline{f}|$ denotes the length $\overline{f}$. We call r the radius of the index. For $r = 0$, the index includes the set of all variables that reference the location. For $r = 1$, the index includes the set of all variables x and field expressions $x.f$ that reference the object.

Given two configurations $L_1 = (rc_1, S_1, h_1, M_1, a_1)$ and $L_2 = (rc_2, S_2, h_2, M_2, a_2)$ with the same index, $I(L_1, r) = I(L_2, r)$, the merge operation $L' = L_1 \sqcup_r L_2$ is:

$$\begin{array}{rcl} L' & = & (rc', S', h', M', a'), \text{ where} \\ rc' & = & rc_1 \\ S' & = & \{s_1 \cap s_2 \mid s_1 \in S_1 \ \wedge\ s_2 \in S_2\} \\ M' & = & \{s_1 \cap s_2 \mid s_1 \in M_1 \ \wedge\ s_2 \in M_2\} \\ h' & = & h_1 \cap h_2 \\ a' & = & a_1 \vee a_2 \end{array}$$

The result of the merge operation is essentially the intersection of the equivalence and non-equivalence relations from the input configurations.

This approach ensures that the computed abstraction cannot grow arbitrarily large, provided that the index domain is finite. For this, the analysis bounds the number of references possible for each field: it requires that the object has at most k references from the same field. If the analysis reaches a point where this restriction is not met (either because of imprecision in the analysis or because the program actually creates that many references), it stops tracking the object through the program, and the compiler will not attempt to deallocate this object.

4.2.3 Interprocedural Analysis

The interprocedural analysis is formulated at the granularity of single heap cells. Given a configuration that describes the state of one heap cell before a method call, the analysis computes a set of possible configurations that describe the state of that cell after the call. Procedure summaries are used to cache this information. The analysis uses a standard worklist algorithm similar to the one of Sharir and Pnueli[20]. The analysis supports two additional mechanisms:

- *Filtering*. The algorithm filters out the information that is not relevant to the callee. This reduces the size of the configuration that is passed to the callee. More importantly, it allows for more reuse of method summaries.
- *Extended parameters*. Because of filtering, the analysis might lose expression equivalence information at the call. The compiler introduces additional variables called *extended parameters* at call sites to avoid losing precision in such cases. The extended parameters are used to combine the result of the method call with the filtered portion. This allows the analysis to reconstruct equivalences that existed before the call.

Each operation $\mathsf{op} \in \{\mathsf{map}, \mathsf{extend}, \mathsf{split}, \mathsf{unmap}\}$ is defined using op_{r}, op_{s} and op_{a}, as follows:

$$\mathsf{op}(L) = (\mathsf{op}_{\mathsf{r}}(rc), \mathsf{op}_{\mathsf{s}}(S), \mathsf{op}_{\mathsf{s}}(h), \mathsf{op}_{\mathsf{s}}(M), \mathsf{op}_{\mathsf{a}}(rc, a))$$

Map : $\mathsf{map}(L) = L_m$

$$\begin{array}{rcl} \mathsf{map}_{\mathsf{s}}(s) & = & [\![p_i = y_i]\!]_{\mathsf{s}} s \\ \mathsf{map}_{\mathsf{r}}(rc) & = & rc \\ \mathsf{map}_{\mathsf{a}}(rc, a) & = & a \end{array}$$

Extend : $\mathsf{extend}(L_m) = L_e$

$$\begin{array}{rcl} \mathsf{extend}_{\mathsf{s}}(s) & = & s \cup \{q_{p.\overline{f}} \mid p.\overline{f} \in s \ \wedge\ |\overline{f}| > 0\} \\ \mathsf{extend}_{\mathsf{r}}(rc) & = & rc \\ \mathsf{extend}_{\mathsf{a}}(rc, a) & = & a \end{array}$$

Split (non-accessed/accessed) : $\mathsf{split}(L_e) = (L_n, L_a)$

$$\begin{array}{rcl} \mathsf{n\text{-}split}_{\mathsf{s}}(s) & = & \{x.\overline{f} \in s \mid \overline{f} \cap \mathsf{eff}(m) = \varnothing\} \cup \mathsf{EPar}(s) \\ \mathsf{n\text{-}split}_{\mathsf{r}}(rc) & = & rc[f \mapsto 0 \mid f \in \mathsf{eff}(m)] \\ \mathsf{n\text{-}split}_{\mathsf{a}}(rc, a) & = & a \\ & & \\ \mathsf{a\text{-}split}_{\mathsf{s}}(s) & = & \{x.\overline{f} \in s \mid \overline{f} \subseteq \mathsf{eff}(m)\} \cup \mathsf{EPar}(s) \\ \mathsf{a\text{-}split}_{\mathsf{r}}(rc) & = & rc[f \mapsto 0 \mid f \notin \mathsf{eff}(m)] \\ \mathsf{a\text{-}split}_{\mathsf{a}}(rc, a) & = & (\mathsf{a\text{-}split}_{\mathsf{r}}(rc) \neq rc) \ \vee\ p_i \in h \ \vee\ a \end{array}$$

where $\mathsf{eff}(m)$ = fields accessed by m, and

$$\mathsf{EPar}(s) \ = \ \{p_i \mid p_i \in s\} \cup \{q_{p.\overline{f}} \mid q_{p.\overline{f}} \in s\}$$

Combine : $\mathsf{combine}(L_n, L'_a) = L_c$

$$\begin{array}{rcl} \mathsf{combine}_{\mathsf{s}}(s_n, s'_a) & = & \{e \mid e \in s_n \cup s'_a \ \wedge\ s_n \cap s'_a \neq \varnothing\} \\ \mathsf{combine}_{\mathsf{r}}(rc_n, rc'_a) & = & \{f \mapsto i \mid i = rc_n(f) + rc'_a(f)\} \\ \mathsf{combine}(L_n, L'_a) & = & (\ \mathsf{combine}_{\mathsf{r}}(rc_n, rc'_a), \\ & & \ \ \mathsf{combine}_{\mathsf{s}}(S_n, S'_a),\ h_n \cup h'_a, \\ & & \ \ \mathsf{combine}_{\mathsf{s}}(M_n, M'_a),\ a_n\ ) \end{array}$$

Unmap : $\mathsf{unmap}(L_c) = L'$

$$\begin{array}{rcl} \mathsf{unmap}_{\mathsf{s}}(s) & = & ([\![x = \mathsf{ret}]\!]_{\mathsf{s}} s) - (\mathsf{EPar}(s) \cup \{\mathsf{ret}\}) \\ \mathsf{unmap}_{\mathsf{r}}(rc) & = & rc \\ \mathsf{unmap}_{\mathsf{a}}(rc, a) & = & a \end{array}$$

Figure 5. Interprocedural analysis steps.

The analysis of one configuration through a call site involves the following 6 steps:

1. *Map*: The algorithm first takes a configuration and assigns actual values at the call site to the formal parameters of the callee.
2. *Extend*: Extended parameters are added.
3. *Split*: The analysis splits the configuration in two new configurations: one containing only the portion that can be accessed by the callee, and the other containing the non-accessed portion. Formal and extended parameters are kept in both halves.
4. *Analyze*: The accessed half is then tracked through the body of the callee. The analysis stores methods summaries, thus, the compiler never analyzes a method for the same input twice.
5. *Combine*: The analysis recombines the resulting configuration with the non-accessed portion of the original configuration.
6. *Unmap*: The analysis finishes by assigning the return value at the call site and removing all formal and extended parameters.

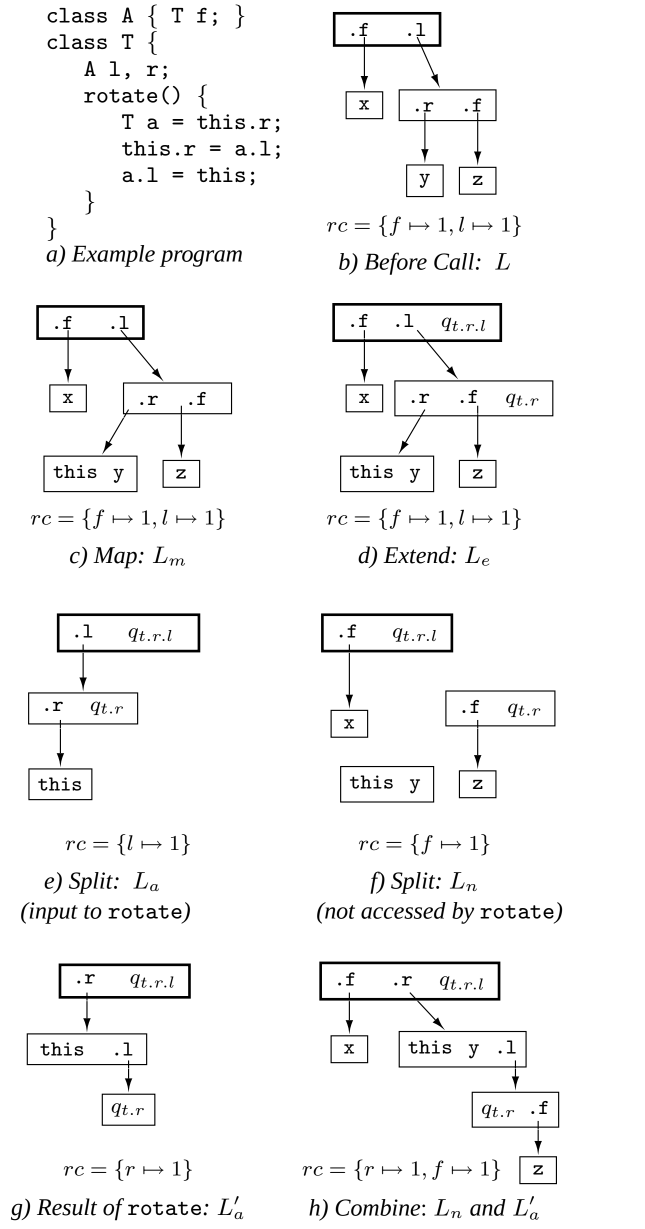
When the accessed half produced by the split operation is empty, then the call is irrelevant to the tracked object, and the call is essentially ignored by the analysis.

Figure 5 shows a formal description of these steps. Starting with a configuration L before the call site, the result of *map* is the configuration $L_m = \mathsf{map}(L)$. Next, L_m is augmented with extended parameters to produce $L_e = \mathsf{extend}(L_m)$. The analysis introduces an extended parameter $q_{p.\overline{f}}$ for each subexpression $p.\overline{f}$ mentioned in the mapped configuration, where p is a formal parameter. In order to determine if the callee can change the information stored in the configuration, the analysis first computes a set of access effects $\mathsf{eff}(m) \subseteq \mathsf{Field}$. The set $\mathsf{eff}(m)$ contains all the fields f read or written by the callee method m, and all the methods that m transitively invokes. The split step breaks the configuration L_e in two halves based on these effects: one half that might be accessed by the callee ($L_a = \mathsf{a\text{-}split}(L_e)$), and the other that is definitively not modified during the call ($L_n = \mathsf{n\text{-}split}(L_e)$). Reference counts are updated during splitting: fields not accessed in the callee are set to zero in the accessed half, and fields accessed by the callee are set to zero in the non-accessed half. Note that the caller might have references to the tracked object that are not included in the accessed half, in that case the flag ($\mathsf{a\text{-}split_a}(rc, a)$) becomes true. The analysis then analyzes the configuration L_a through the callee, and computes a resulting configuration L'_a. Further, L'_a is combined with the non-accessed half at the call L_n to produce $L'_c = \mathsf{combine}(L_n, L'_a)$. Finally, in the *unmap* step, the analysis computes a configuration $L' = \mathsf{unmap}(L'_c)$ after the call site.

4.2.4 Example

Figure 6 presents an example illustrating the interprocedural analysis. In the figure, we use a graphical notation to represent configurations. This notation is the actual data structure used in our implementation. We call this structure a *Tracked Object Structure* or TOS. A TOS can compactly represent equivalent expressions and it is useful to implement the transfer functions efficiently. A TOS is a graph where each node describes an equivalence class from S. A node contains a set of variables and fields. Each field f denotes a field access expression $e.f$. There is an edge from the field f to the node that represents the sub-expression e on which the field is accessed. The hit class h is shown using a bold node; and there are no miss classes in this example. For instance, the graph in Figure 6(b) represents a configuration where $h = \{x.f, y.r.l, z.f.l\}$, $S = \{h, \{y.r, z.f\}\}$ and $M = \emptyset$.

Figure 6(a) shows a method `rotate` that performs a tree rotation. Class `T` represents tree nodes with child fields `l` and `r`. The program also uses a class `A` with a field `f` of type `T`. Assume that some method with local variables x, y, and z calls y.`rotate`(). The example shows how the analysis tracks one of the grandchildren of y through this call. The state of this object before the call is shown in Figure 6(b). The extension step in Figure 6(d) introduces two extended parameters $q_{t.r}$ and $q_{t.r.l}$ representing the expressions *this*.r and *this*.$r.l$, respectively. The method `rotate` accesses fields r and l, but not f, thus eff(`rotate`) = $\{r, l\}$. The subexpressions containing only the field f are considered as non-accessed and are filtered out by the analysis. Finally, the extended parameters are used in the *combine* step in Figure 6(h) to reconstruct expression equivalences. Without the extended parameters, the analysis would have lost the information that y is equivalent to $z.f.l$, and that $z.f.l.r$ references the tracked object after the call.



```
class A { T f; }
class T {
   A l, r;
   rotate() {
      T a = this.r;
      this.r = a.l;
      a.l = this;
   }
}
```

a) Example program
$rc = \{f \mapsto 1, l \mapsto 1\}$
b) Before Call: L
$rc = \{f \mapsto 1, l \mapsto 1\}$
c) Map: L_m
$rc = \{f \mapsto 1, l \mapsto 1\}$
d) Extend: L_e
$rc = \{l \mapsto 1\}$
e) Split: L_a (input to `rotate`*)*
$rc = \{f \mapsto 1\}$
f) Split: L_n (not accessed by `rotate`*)*
$rc = \{r \mapsto 1\}$
g) Result of `rotate`*: L'_a*
$rc = \{r \mapsto 1, f \mapsto 1\}$
h) Combine: L_n and L'_a

Figure 6. Analysis of method call `y.rotate()`.

5. Free Transformation

After the tracked object analysis phase, the next step of the algorithm is to insert explicit object deallocation statements in the program. The compiler examines each assignment in the program ($e_1 = e_2$ or $e_1 = \mathsf{new}$) and searches for configurations L that become unreachable right after the assignment, hence e_1 is the last reference to the corresponding object. Once such a configuration is found, the compiler tries to deallocate the object by adding the statement `free`(e_1). The process of inserting the actual free is complicated by several factors. First, it is not guaranteed that an object that matches L always exists in the heap when the program reaches this point. Second, the compiler must avoid deallocating an object that is read in the right-hand side of the same assignment that causes the deallocation. Finally, the same expression e_1 might hit another configuration L' that does not become unreachable after the assignment. Hence, although the tracked object analysis analyzes one allocation site at a time, the free analysis needs information about all of the configurations at a program point before it can decide whether freeing an object at that point is safe. To illustrate this last issue, consider the following program fragment:

```
x = new C();
if (...) {
     y = new C();
     x = y;
}
x = null;
```

At the program point before `x = null`, there are two configurations: one with hit set (`x`), and one with hit set (`x, y`). Only the first one becomes unreachable after `x = null`. Inserting `free(x)` before this assignment is unsafe, because `x` also hits the second configuration. We must identify program expressions that allow the compiler to distinguish between different configurations. We call these expressions *discriminants*. In this example, expression `y` is a discriminant. Therefore, the compiler can introduce the following guarded deallocation statement before `x = null`:

```
if (x != y) free(x)
```

Thus, we only insert a free statement $\texttt{free}(e_1)$ if all configurations $L = (rc, S, h, M, a)$ that become unreachable satisfy the following conditions: 1) L models all references to the tracked object (i.e., the flag a is true); 2) no subexpression of e_2 (if any) can hit the configuration L; and 3) we can compute a set of discriminant expressions to distinguish L from other configurations that don't become unreachable.

The compiler computes discriminant expressions as follows. It examines the tracked object analysis result before the assignment, and identifies all configurations $L' \neq L$, such that e_1 might reference L' and L' doesn't become unreachable. For each such configuration $L' = (rc', S', h', M', a')$, it computes the set of expressions $H = h' - h$ that reference L', but not L. If H is empty, the analysis cannot distinguish L from L' and hence, it cannot free the tracked object. Otherwise, the analysis selects an expression from H and adds it to the set of discriminant expressions.

Let D be the computed set of discriminant expressions, the transformation builds a deallocation statement that frees e_1 and nullifies all of its fields. If the discriminant set is not empty: $D = \{d_1, ..., d_m\}$, the compiler adds a guard to the deallocation:

$$\texttt{if}\ (e_1 \neq d_1\ \wedge\ ...\ \wedge\ e_1 \neq d_m)\ \texttt{free}(e_1)$$

When such guards are inserted, the analysis must ensure that evaluating expressions d_1 through d_n does not cause null pointer dereferences. We currently add explicit null pointer checks. Alternatively, simple dataflow analyses could identify non-null references and eliminate such checks.

After inserting a free statement before an assignment, the analysis places all of the configurations before the assignment into a worklist. The compiler will subsequently re-analyze them starting at this point. In the subsequent analysis, each $\texttt{free}(e)$ statement is treated as a nullification of all of e's fields:

$$e.f_1 = \texttt{null};\ \ \dots\ ;\ e.f_n = \texttt{null};$$

These nullifications are just considered during the analysis, but are not generated in the compiled programs.

5.1 Partial Recomputation

The analysis alternates between a tracked analysis phase and a subsequent transformation. As mentioned above, the free transformation places configurations that must be reanalyzed into a worklist. The compiler then performs a cleanup phase that starts from the configurations in this worklist and invalidates all of the information reachable from these configurations. This algorithm relies on the fact that the tracked object analysis also constructs a graph of configurations that describe the correspondence between configurations at different program points. The cleanup phase inspects merge points in this graph. For each configuration L that is not invalidated, but has successor configurations being invalidated, it adds L to the worklist of the tracked object analysis. This approach allows the compiler to reuse analysis results from previous iterations, and only recompute portions of the program that are affected by the newly added `free` statements.

5.2 Recursive and Aggregate Structures

When entire recursive structures (such as lists or trees) become unreachable because of one single assignment, the approach presented so far will not terminate. It will attempt to generate code that deallocates all of the elements in the structure. The main loop of the algorithm will trigger one more object at each iteration. To ensure that the iterative process terminates, we bound the number of iterations with a constant value, for instance, the maximum depth of a non-recursive structure in the program.

We emphasize, however, that our approach can successfully deallocate individual object instances from arrays or from recursive structures, such as linked lists or trees, when the objects become unreachable one at a time. For instance, when the program removes one single element from a list or from an array, and the analysis successfully determines that the object loses its last reference, the compiler can free the removed object.

5.3 Analysis Enhancements

We briefly discuss several analyses that enhance the ability of the analysis to deallocate more objects, or to deallocate them earlier.

- *Variable equality analysis*. Because the analysis of an object starts at the allocation site, it cannot capture expression equivalences that have been created before that point. We use a simple dataflow analysis that computes, at each program point, sets of variables that have identical values. The tracked object analysis then uses this information after each application of a transfer function.
- *Variable liveness analysis*. We perform the standard live variable analysis and insert nullifications in the program as soon as variables become dead. This enables to compiler to free objects earlier in the program.
- *Variable must-unalias analysis*. The tracked object analysis can trace each configuration in isolation, but the transformation needs knowledge about all configurations at each deallocation point, as described earlier. We perform an intra-procedural dataflow analysis to detect when a variable is not aliased. In such cases, the transformation can reason in isolation about the object referenced by this variable, and deallocate it when the corresponding configuration becomes unreachable.
- *Escape Analysis*. We also use an flow-insensitive, unification-based escape analysis to improve the field effects information. The compiler excludes from the effect set of each method those fields that are read, but do not escape the method. Reducing the effect sets allows the analysis to skip more method calls that are irrelevant to the tracked object.

6. Experimental Results

This section presents results collected from an implementation of the analysis described on this paper, including the enhancements discussed in Section 5.3. Our compilation system JFREE has been built in the Soot infrastructure [23]. We have also extended the Jikes virtual machine (version 2.3.5) [2] to support explicit reclamation of objects via free. The transformed programs generated by our compiler were then executed on this extended virtual machine. We evaluated programs from the SPECjvm98 benchmark suite [22]. Experiments we conducted on a 2 GHz Pentium machine with 1GB of memory and running Linux.

Program	Methods Analyzed	Alloc. Sites	Analysis Time	Frees added Iteration 1	2	3
compress	304	112	24	72	17	3
jess	689	535	71	162	8	0
raytrace	392	224	61	211	6	0
db	394	134	43	129	11	0
javac	1335	980	358	310	73	2
mpegaudio	340	1123	89	62	1	0
mtrt	389	234	55	160	8	0
jack	533	416	82	266	45	0

Table 1. Analysis statistics, using $k = 1$, $r = 0$ and $t = 30s$.

Program	Objects Freed	Memory Freed	Runtime Overhead
compress	3%	78%	2.86%
jess	23%	19%	-5.21%
raytrace	85%	81%	4.21%
db	90%	86%	-0.13%
javac	19%	14%	0.27%
mpegaudio	0%	0%	0.68%
mtrt	81%	77%	0.41%
jack	29%	24%	3.98%

Table 2. Runtime statistics, using $k = 1$, $r = 0$ and $t = 30s$.

6.1 Methodology for Evaluating the Analysis

The JFREE compiler performs a bytecode-level transformation. It reads standard Java bytecodes at the input, and generates output bytecodes augmented with one special *free* instruction, which frees the local variable specified as argument. Our analyses and transformations have been implemented in the three-address code representation of Soot.

For each benchmark the compiler analyzes all of the allocation sites in the application and in the library code, but transforms only the application code. Hence, objects that die in the library will not be freed by the analysis. However, objects that are allocated in the library code, but die in the application code can be freed. The compiler tracks objects through the entire program, including the library code, regardless of where they have been allocated. We use the call graph provided by Soot to disambiguate virtual calls and calls by reflection. We do not attempt to track objects that are shared between multiple threads, or objects that may be thrown. These constructs have complex flow of control and a precise treatment of these features would significantly complicate our flow analysis. Instead, the analysis stops tracking objects as soon as they are thrown or passed to other threads.

The analysis can become expensive for long-lived objects, because they need to be tracked throughout most of the program. To make things worse, the benefits of eventually deallocating such objects are very small. To avoid these situations, the compiler uses a cutoff parameter t to limit the amount of exploration during the tracked object analysis. The analysis limits the amount of time per iteration to a fixed amount of t seconds. This time is evenly distributed among all of the allocation sites being analyzed in that iteration. When tracking an object exceeds the timeout, the analysis no longer attempts to free the object.

We have experimented with different values of the timeout t and different values of the analysis parameters k and r. We observed that a good balance between the analysis time and the amount of memory freed is achieved for $r = 0$, $k = 1$, and $t = 30$ sec. Hence, our system uses these values as defaults. Additional memory can, however, be freed with other values of these parameters, at the expense of making the analysis more costly. In Section 6.2.1 we present results collected using different values of these parameters, and discuss their impact on the amount of memory freed and on the analysis time.

6.1.1 Analysis Statistics

Table 1 presents analysis statistics for the default values of the parameters. The first two columns show the size of the applications in terms of number of methods and number of allocation sites being analyzed. The next column shows the time required by all of our analyses. Our analyses account, on average, for 69% of the total compilation time. In addition to the analysis time, the total compilation time includes several tasks performed by Soot: reading and writing files to disk, running a points-to analysis, and using its results to construct the call graph. Most of the analysis time (i.e., more than 90% of the analysis time) is spent in the tracked object analysis, except for *javac*, where about 70% of the analysis time is spent in the escape analysis.

The last three columns show how many free statements are added in each of the iterations of the tracked object analysis. None of the applications required more than 3 iterations for the default parameter values. On average, we observed that the first iteration takes most of the tracked object analysis time, 66%. Without the incremental approach for the subsequent iterations, the overall tracked object analysis time would double.

6.2 Methodology for Evaluating Generated Programs

We evaluate the generated programs in four main settings: 1) no memory reclamation (NoFree); 2) deallocation via free statements (Free); 3) mark and sweep garbage collection (MS); and 4) a hybrid systems that supports both explicit freeing and dynamic collection (Free+MS).

All settings use the same run-time memory management structure: the *segregated freelist* provided by the Memory Management Toolkit available in Jikes RVM [5]. This structure is organized as an array of buckets of fixed sizes, with one freelist per bucket, plus a separate freelist for large objects. In the Free and Free+MS settings, objects carry information about the bucket they belong to in their header. When the execution reaches a free instruction, the VM looks up the header and places the object back into the freelist of the appropriate bucket. In the other two settings (NoFree and MS), free instructions are treated as no-ops.

We first evaluate Free as the only means of memory management (Section 6.2.1). Then we discuss the impact of our approach when used in combination with garbage collection (Section 6.2.2).

6.2.1 Free Only

We instrumented the memory allocator in Jikes RVM and collected the number of bytes allocated and freed by the transformed programs. We then compared these numbers with the total amount of allocated memory. Since our compiler only transforms the application code, we were interested in the memory allocated by the application only. We therefore excluded the memory internally used by the virtual machine, and the memory used by the JIT compiler. The former was approximated as the memory allocated by an empty program when executed without JIT compilation. The latter was approximated by comparing the difference between two runs of each application, one with and one without JIT optimizations (BaseBase and FastAdaptive configurations in Jikes RVM). The amount of excluded memory ranges from 7% of the total memory for *jess*, to 83% for *mpegaudio*. On average, the excluded memory represents 24% of the total memory.

Program		$k=0$					$k=1$				
		$t=5s$	$t=15s$	$t=30s$	$t=1m$	$t=5m$	$t=5s$	$t=15s$	$t=30s$	$t=1m$	$t=5m$
compress	Time (s)	4	6	7	9	14	12	21	21	30	47
($r=1$)	Frees	38	50	63	65	69	60	86	98	99	128
	Objects	1%	1%	1%	1%	1%	1%	4%	4%	4%	4%
	Memory	0%	0%	0%	0%	0%	9%	91%	91%	91%	91%
compress	Time (s)	4	6	7	9	14	12	21	22	27	43
($r=0$)	Frees	38	50	63	65	69	55	85	92	94	123
	Objects	1%	1%	1%	1%	1%	1%	3%	3%	3%	3%
	Memory	0%	0%	0%	0%	0%	0%	78%	78%	78%	78%
jess	Time (s)	14	23	33	48	69	19	38	67	118	528
($r=0$)	Frees	30	133	158	208	253	104	140	170	203	284
	Objects	0%	23%	41%	41%	41%	0%	0%	23%	41%	41%
	Memory	0%	19%	26%	26%	26%	0%	0%	19%	26%	26%
raytrace	Time (s)	10	23	32	32	43	16	34	60	106	458
($r=0$)	Frees	116	170	207	221	230	102	172	217	237	364
	Objects	83%	87%	87%	87%	87%	67%	83%	85%	87%	92%
	Memory	80%	83%	83%	83%	84%	64%	79%	81%	83%	89%
db	Time (s)	6	9	11	12	18	15	34	42	46	86
($r=0$)	Frees	80	110	130	138	148	92	113	140	148	204
	Objects	0%	0%	90%	90%	90%	0%	0%	90%	90%	90%
	Memory	0%	1%	59%	59%	59%	0%	28%	86%	86%	86%
javac	Time (s)	26	41	61	79	150	28	61	104	172	1046
($r=0$)	Frees	83	250	309	366	555	86	248	385	406	578
	Objects	1%	16%	18%	18%	19%	1%	13%	19%	19%	19%
	Memory	1%	13%	14%	14%	14%	1%	11%	14%	14%	15%
mpegaudio	Time (s)	4	7	10	12	18	23	48	87	143	569
($r=0$)	Frees	10	43	61	72	86	10	49	63	95	136
	Objects	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
	Memory	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
mtrt	Time (s)	11	14	18	26	39	17	32	54	92	680
($r=0$)	Frees	102	155	184	217	228	105	143	168	218	372
	Objects	63%	79%	81%	83%	84%	46%	79%	81%	83%	88%
	Memory	60%	75%	77%	79%	80%	44%	75%	77%	79%	85%
jack	Time (s)	14	25	33	39	55	21	45	79	129	352
($r=0$)	Frees	119	210	274	314	390	93	216	311	347	516
	Objects	8%	28%	29%	29%	31%	13%	13%	29%	29%	31%
	Memory	4%	22%	24%	24%	25%	8%	8%	24%	24%	26%
Avg. ($r=0$)	Memory	18%	27%	36%	36%	36%	15%	35%	47%	49%	51%

Table 3. Impact of parameters k, r and t on analysis time, number of *free* statements added, objects reclaimed and memory reclaimed.

Memory Freed. Table 2 shows the amount of objects and memory reclaimed in the Free setting. The numbers are normalized with respect to the total amount of memory. On average, the compiler can enable the reclamation of 47% of the application's memory using the default values of the parameters. In applications several applications (e.g., *compress*, *raytrace*, *db*, and *mtrt*), the compiler enables the deallocation of more than 75% of the memory. Higher values of r, k, and t can further improve the amount of memory freed by 7% to 13% for several of these benchmarks.

For *mpegaudio*, almost all the data is live throughout the program, so the numbers for this application have little relevance. However, they are included for the sake of completeness.

The ability of our compiler to deallocate multi-level structures that become unreachable all at once is important for applications such as *compress*. Without the free's inserted in the second and third iterations, the analysis would not be able to free several large buffers used in this application, and the freed memory would drop from 78% to 0%.

For most of the applications, the percentage of objects reclaimed are correlated to the amount of memory freed. Only *compress* and *jess* present some clear difference. In *compress* few objects are being freed, but these include large array objects that represent 90% of the memory in the application. The contrary happens in *jess*, where many small object are being freed (41%), but the total memory reclaimed is much lower (26%).

Running Times. The free statements introduce very little runtime overhead in the transformed programs. The third column on Table 2 shows the normalized runtime overhead of each application, relative to the running time in the NoFree setting. On average, the runtime overhead is less than 1%. All programs have less than 5% overhead, and some actually run slightly faster than in the NoFree setting. We believe that this is due to improved data locality when heap cells are freed and reused.

Parameters Evaluation. Table 3 presents results for different values of the parameters k, r, and t. We experimented with $r \in \{0, 1\}$, $k \in \{0, 1\}$, and t ranging from 5 seconds to 5 minutes. Higher values of these parameters made the analysis more expensive, but didn't help the compiler free more memory. Each entry in the table shows four numbers: the time used by the tracked object analysis, the number of frees inserted per iteration, and the percentage of objects and memory freed.

Higher values of r and k improve accuracy and allow the compiler to free more memory, but make the analysis slower. Increasing r makes the merge operation more expensive, and increasing k causes the analysis to track objects for longer periods of time. Thus, the analysis requires more time to achieve its potential.

This effect can be observed when comparing $k = 0$ with $k = 1$. With $k = 0$, the analysis tracks only objects that don't escape into heap fields. Hence object traces are shorter and the analysis always runs a single iteration. In most of the cases, the analysis with $k = 0$ between 2 and 3 times faster. However, for *db* and *compress* it is not precise enough to free as much memory as the analysis with $k = 1$. It should be observed, however, that for runs with small timeouts, the analysis with $k = 0$ is able to reclaim more memory than the one with $k = 1$, because the analysis is faster and can trace more objects up to their deallocation point before the tracing times out. This situation can be observed for *raytrace* and *mtrt*, with $t = 5s$.

For r, the only application where a value of $r = 1$ has an impact on the results was *compress*. For this benchmark, the compiler was able to free an additional 13% of the memory with $r = 1$, as seen in the first two rows of the table. For the other benchmarks, we haven't observed additional memory deallocation improvements that would justify using $r = 1$.

The impact of the timeout t can be seen when inspecting the numbers in the table from left to right for each value of k. Allowing the analysis to spend more time per iteration makes it possible to insert more frees. However, the amount of memory freed follows the law of diminishing returns: using $t = 1m$ and $t = 5m$ gives small improvements of less than 2%, except for *jess*, *raytrace*, and *mtrt* where improvements of 7% and 8% can be achieved with higher values of t for $k = 1$.

6.2.2 Combining Free and Garbage Collection

Since our compiler frees objects before they are unreachable, the free-instrumented programs deallocate objects before any garbage collector can. Non-moving collectors that use freelists, such as mark-and-sweep collectors, can be adapted to use the results of our analysis: cells freed by the program can be reused and collections can be triggered less often.

We have implemented a hybrid collector (Free+MS) by extending a mark-and-sweep collector. The VM explicitly moves objects back into the collector's freelist when the program frees an object. Figure 8 presents a graph for each program, comparing the performance of Free+MS with respect to the default mark-and-sweep collector (MS) for different heap sizes. The x-axis is the maximum heap size, relative to the minimum size needed to run an application using the MS collector. The y-axis is the running time, normalized with respect to the minimum running time. As before, the programs were compiled with the default values of the parameters. We can observe a significant improvement in several applications. For instance, *raytrace* can run on average 7% faster in the Free+MS setting; and can run 26% faster for small heaps. Figure 7 presents the geometric mean of the values presented for each benchmark. On average, the explicit deallocation of objects improves the application running time by about 8%. Moreover, for the minimum heap size programs run 31% faster in the Free+MS setting. For many applications, such as *jess*, *raytrace*, *javac* and *jack*, the Free+MS setting was able to run in heaps smaller than MS. On average, Free+MS can reduce the minimum heap size by 5%.

7. Related Work

In previous work, Shaham et al. [19] have proposed an individual object deallocation transformation using shape analysis based on three-valued logic. Our analysis has the same goals, namely to statically compute precise heap reference counts and use this information to enable the reclamation of single objects, but is fundamentally different in its functioning. They use shape graph abstractions (modeled as 3-valued logic structures) to describe the entire heap, whereas our technique models and analyzes a single heap object at a time. Analyses that use shape graph abstractions are significantly more expensive than ours, especially at the inter-procedural level [16]. Although there has been recent effort to improve inter-procedural shape analysis [17], it has not been demonstrated yet that shape analysis via three-valued logic can scale to larger programs such as the SPECjvm98 programs. Furthermore, Shaham's analysis does not address the problem of deallocating multi-level structures.

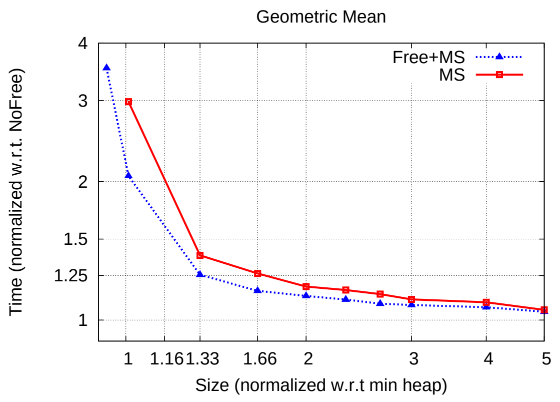


Figure 7. Performance comparison between a hybrid system (Free+MS) and a mark and sweep collector (MS)

The analysis presented in this paper builds on the shape analysis framework that we recently developed [13], where the compiler analyzes one single heap cell at a time. In our previous work, we presented such an analysis for C programs, and used it to find heap manipulation errors. In this work, we have developed a new analysis for Java and applied it to the memory management problem. The analysis presented in this paper takes advantage of Java features, such as type-safety and the lack of an address-of operator. Also, the use of equivalence classes and extended parameters makes is more precise than the analysis in [13].

In work concurrent with ours, Guyer et al. [12] also developed a compiler analysis for automatically inserting free statements in Java programs. Their compiler combines a flow-insensitive points-to analysis with an intra-procedural liveness analysis for summary nodes in the points-to abstraction. Their analysis appears to be more efficient than ours. However, it is also less precise than ours: because they use a points-to abstraction, their analysis cannot distinguish between different object instances within summary nodes when some, but not all of the nodes in the summary become dead. In particular, their analysis cannot deallocate objects that are removed from lists or arrays. Also, they don't address the problem of deallocating multi-level heap structures, illustrated in the example from Section 2. As a result, their technique cannot deallocate memory in the *compress* benchmark; in contrast, our approach can free more than 90% of the memory for this benchmark.

Stack allocation via escape analysis has been extensively studied. For Java, researchers have typically proposed combined escape analysis with some form of pointer analysis [9, 24, 6]. Some analyses are formulated as dataflow analyses that compute points-to graphs and escape information simultaneously [9, 24]. Others use flow-insensitive approaches and formulate the problem of computing escape information as a set of constraints [6]. Most of these analyses also use the notion of escaping from a thread (rather than a lexical scope) and use it to eliminate synchronization. An impor-

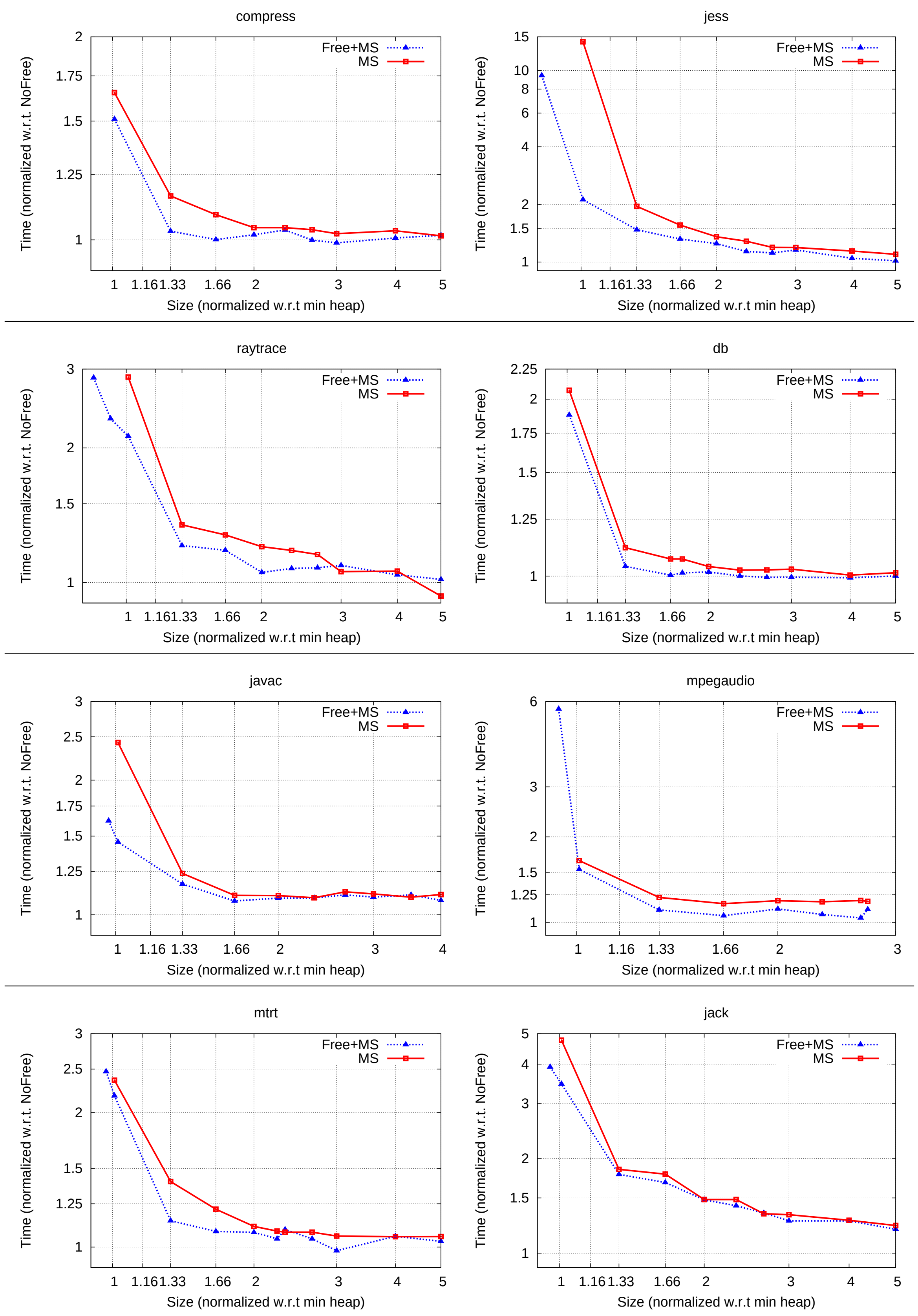


Figure 8. Performance comparison between a hybrid system (Free+MS) and a mark and sweep collector (MS).

tant limitation of all escape and stack allocation approaches is that the objects at each stack-allocated site must all be non-escaping.

Region-based memory management has become a popular approach for memory management. Tofte and Talpin propose a region inference for functional languages [21], i.e., a compiler analysis that automatically creates lexically-scoped regions, places heap objects into these regions, and deallocates the regions at the end of their scopes. Researchers have also explored hybrid systems that combine region inference with garbage collection [14]. Recently, region inference has been studied for Java programs [7, 8]. The system proposed by Chin et al. [8] infers lexically scoped regions, while the JREG system that we developed in own work [7] supports unrestricted regions, but requires a more complex analysis. Our experience is that the JFREE system presented in this paper is more flexible than JREG with respect to object lifetimes, and uses less obtrusive, easier-to-understand program changes. However, it also requires different analysis techniques based on shape analysis.

Although regions are lightweight and suitable to real-time systems, they are still more complex than stack-allocation or individual object deallocation. Also, most region-based systems are lexically scoped; and objects allocated at the same site are all deallocated at the same point. Both aspects are more restrictive than individual deallocation.

8. Conclusions

We have presented a compiler transformation for Java that automatically augments programs with *free* statements to deallocate individual objects. This transformation is enabled by the tracked object analysis, an inter-procedural dataflow analysis that analyzes a single object instance at a time, computing precise information about the references to the object at each point in the program. We have presented an experimental evaluation using a full implementation of this technique. Our results show that compile-time individual object deallocation using this approach can enable the deallocation of substantial amounts of memory, at a low runtime cost.

Acknowledgments

We want to thank Ruijie Wang for her contributions in earlier stages of this work. We also want to thank the anonymous reviewers for their comments and suggestions.

References

[1] A. Aiken, M. Fahndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *Proceedings of the SIGPLAN '95 Conference on Program Language Design and Implementation*, La Jolla, CA, June 1995.

[2] B. Alpern, D. Attanasio, A. Cochi, D. Lieber, S. Smith, T. Ngo, and J. Barton. Implementing Jalapeño in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[3] B. Goldberg and Y. Park. Higher order escape analysis: Optimizing stack allocation in functional program implementations. In *Proceedings of the 3rd European Symposium on Programming*, May 1990.

[4] B. Goldberg and Y. Park. Escape analysis on lists. In *Proceedings of the SIGPLAN '92 Conference on Program Language Design and Implementation*, July 1992.

[5] S. Blackburn, P. Cheng, and K. McKinley. Oil and water? High performance garbage collection in Java with MMTk. In *ICSE*, Edinburgh, UK, May 2004.

[6] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[7] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. In *Proceedings of the 2004 International Symposium on Memory Management*, Vancouver, Canada, October 2004.

[8] W. Chin, F. Craciun, S. Qin, and M. Rinard. Region inference for an object-oriented language. In *Proceedings of the SIGPLAN '04 Conference on Program Language Design and Implementation*, Washington, DC, June 2004.

[9] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.

[10] M. Christiansen and P. Velschow. Region-based memory management in Java. Master's thesis, DIKU, University of Copenhagen, May 1998.

[11] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[12] S. Guyer, K. McKinley, and D. Frampton. Free-Me: A static analysis for automatic individual object reclamation. In *Proceedings of the SIGPLAN '06 Conference on Program Language Design and Implementation*, Ottawa, Canada, June 2006.

[13] B. Hackett and R. Rugina. Shape analysis with tracked locations. In *Proceedings of the 32th Annual ACM Symposium on the Principles of Programming Languages*, Long Beach, CA, January 2005.

[14] N. Hallenberg, M. Elsman, and M. Tofte. Combining region inference and garbage collection. In *Proceedings of the SIGPLAN '02 Conference on Program Language Design and Implementation*, Berlin, Germany, June 2002.

[15] M. Hertz and E. Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *Proceedings of the 20th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, San Diego, CA, October 2005.

[16] B. Jeannet, A. Loginov, T. Reps, and M. Sagiv. A relational approach to interprocedural shape analysis. In *Proceedings of the International Static Analysis Symposium*, Verona, Italy, August 2004.

[17] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of the 12th International Static Analysis Symposium*, London, UK, September 2005.

[18] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[19] R. Shaham, E. Yahav, E. K. Kolodner, and M. Sagiv. Establishing local temporal heap safety properties with applications to compile-time memory management. In *Proceedings of the 10th International Static Analysis Symposium*, San Diego, CA, June 2003.

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall Inc, 1981.

[21] M. Tofte and J.-P. Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Proceedings of the 21st Annual ACM Symposium on the Principles of Programming Languages*, Portland, OR, January 1994.

[22] J. Uniejewski. SPEC Benchmark Suite: Designed for today's advanced systems. SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[23] R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - a Java optimization framework. In *CASCON '99*, Toronto, Canada, November 1999.

[24] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, November 1999.