Resurrector: A Tunable Object Lifetime Profiling Technique for Optimizing Real-World Programs

Guoqing Xu
University of California, Irvine
guoqingx@ics.uci.edu

Abstract

Modern object-oriented applications commonly suffer from severe performance problems that need to be optimized away for increased efficiency and user satisfaction. Many existing optimization techniques (such as object pooling and pretenuring) require precise identification of object lifetimes. However, it is particularly challenging to obtain object lifetimes both precisely and efficiently: precise profiling techniques such as Merlin introduce several hundred times slowdown even for small programs while efficient approximation techniques often sacrifice precision and produce less useful lifetime information. This paper presents a tunable profiling technique, called *Resurrector*, that explores the middle ground between high precision and high efficiency to find the precision-efficiency sweetspot for various liveness-based optimization techniques. Our evaluation shows that Resurrector is both more precise and more efficient than the GC-based approximation, and it is orders-of-magnitude faster than Merlin. To demonstrate Resurrector's usefulness, we have developed client analyses to find allocation sites that create large data structures with disjoint lifetimes. By inspecting program source code and reusing data structures created from these allocation sites, we have achieved significant performance gains. We have also improved the precision of an existing optimization technique using the lifetime information collected by Resurrector.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management, optimization, run-time environments; F.3.2 [*Logics and Meaning of Programs*]: Semantics of Programming Languages—Program analysis

OOPSLA '13, October 29–31, 2013, Indianapolis, Indiana, USA.
Copyright © 2013 ACM 978-1-4503-2374-1/13/10…$15.00.
http://dx.doi.org/10.1145/2509136.2509512

General Terms Language, Measurements, Performance

Keywords object lifetime information, memory management, performance optimization

1. Introduction

Large-scale object-oriented programs commonly suffer from systemic performance problems that have significant impact on their scalability and real-world usefulness. Evidence suggests that these problems stem from a combination of the performance-oblivious design/implementation principles as well as large amounts of input data that need to be quickly processed [37, 55]. *Runtime bloat* exists throughout the execution, making it difficult for the compiler to find and remove its performance penalty. To attack the problem, various techniques [7, 7, 13, 15, 16, 21, 23, 25, 32, 35, 36, 43, 44, 48, 50, 58] have been developed to help programmers detect and fix performance problems.

Motivation One major category of these techniques (e.g., [7, 13, 16, 21, 25, 32, 48, 50, 58]) concerns the reduction of objects and data structures. Because object-oriented applications typically create and destroy large numbers of objects, creating, initializing, and destroying them consumes much execution time and memory space. For example, in the white paper "WebSphere Application Server Development Best Practices for Performance and Scalability" [1], four of the eighteen best practices are instructions to avoid repeated creation of identical objects. One important observation that motivates these techniques is that in large-scale applications, objects created for different tasks (e.g., iterations of an event loop, database transactions, etc.) often have disjoint lifetimes and can never be used simultaneously. Reusing existing objects and data structures in the heap can often lead to substantial reductions of time and memory footprint.

Key to any technique that attempts to reduce numbers of objects and their related computation is the *precise understanding of object lifetimes*. For example, object equality profiling (OEP) [32] needs object lifetimes to find mergeable objects and our recent work from [50] can reuse data structures only if their lifetimes do not overlap. Of particular interest is to *know precisely the lifetimes of objects created by the same allocation site*, because optimizations are often performed at the allocation site level. For example, if the lifetimes of the objects created by an allocation site are completely disjoint, one single object will be sufficient throughout the execution; if the allocation site is in a loop, it may be hoistable [58], or at least a singleton pattern can be applied to improve performance [7, 50]. In addition, existing pretenuring techniques [10, 12, 14, 33] often rely on precise allocation-site-based lifetime profiles to allocate likely long-lived objects into an infrequently or never collected region. Recent work on Headroom-based pretenuring [42] has found that pretenuring can benefit more from finer-grained lifetime information than from the simple estimation of "long-lived" objects.

Problems While such precise information about object lifetimes is highly desirable, there does not exist any algorithm that can compute it efficiently. A number of static analyses (such as [13, 16, 21, 22, 25, 48, 58]) have been developed to approximate object lifetimes. However, static analysis usually does not perform well in the presence of large numbers of heap accesses that commonly exist in large-scale programs. In addition, it often falls short in answering queries that require dynamic information, such as how many run-time objects can be simultaneously live for an allocation site, which is important for many optimization techniques.

Merlin [26] is by far the most precise dynamic object lifetime profiling (OLP) technique. It generates an object event trace and uses a backward pass to transitively recover object death points. While Merlin can precisely compute object lifetime information, it suffers from tremendous run-time overhead that significantly reduces its real-world practicality. For example, our experiments show that *Elephant Tracks*—a recent implementation of the Merlin algorithm [41]—incurs an average 752.4× slowdown during the execution of the DaCapo benchmarks [9] under small workloads. To improve scalability, approximations have been employed to estimate object lifetimes. A common approximation is to use the GC point at which an object is collected as its death point. However, GC may occur far behind the actual point at which the object becomes unreachable. Hence, significant imprecision may result, leading to reduced usefulness of the client analysis. Other work such as [50] develops metrics to approximate lifetime information, which can also cause rather imprecise handling. For example, all false positives reported by the reusable object analysis in [50] are due to the imprecise approximation of object lifetimes.

Our Proposal In this paper, we present a novel dynamic technique, called Resurrector, that explores the middle ground between high precision and high efficiency to find the precision-scalability sweetspot for various optimization techniques. Resurrector's design is motivated by an important observation that the scalability bottleneck of a traditional OLP algorithm (such as Merlin) lies in the need to compute transitive closures on the dead objects (e.g., Merlin's backward pass). Resurrector improves efficiency by completely eliminating this need. Similarly to Merlin, Resurrector first identifies the root dead objects whose reference counts are zero. Instead of computing transitive closures from them, Resurrector exploits *object caching and reusing* to find dead objects (transitively reachable from the roots) that have non-zero reference counts.

Resurrector caches and reuses objects based on *their allocation sites*. Upon the execution of an allocation site, the Resurrector allocator first attempts to find an object that was previously created from the allocation site and that is currently unreachable. If the attempt fails, the regular object allocator is invoked to create a new object and Resurrector caches it into a list for this allocation site. If such a dead object o is found, Resurrector records its death and returns it to the application. Resurrecting o will cause it to be reinitialized (by the object initializer), which automatically forces the dead objects pointed to by o to lose references and become unreachable. Hence, their death points can be detected by Resurrector as if they were root dead objects. These objects will be subsequently resurrected when their allocation sites are executed. The resurrection process is repeated until all transitively dead objects under o in the object graph have zero references. In other words, Resurrector resurrects dead objects and leverages the mutator execution to compute transitive closures, leading to elimination of an additional reachability analysis to identify dead objects. Information regarding the maximal number of objects whose lifetimes can overlap for an allocation site, as required by many of the aforementioned techniques, can be closely approximated by identifying the maximal length of the cache list for the allocation site during execution.

The Resurrector algorithm requires precise identification of the point at which an object becomes unreachable in order to determine whether this object is resurrectable. Counting only heap references does not suffice, because the object may still be referenced by stack variables even if the number of its heap references is zero. Tracking every stack and register update can be very expensive. To solve the problem, we develop an efficient timestamp-based algorithm that enables Resurrector to identify a dead object soon after it becomes unreachable without relying on GC.

Precision and Scalability Comparison A graphical comparison among the precision and efficiency of Merlin, Resurrector, and the GC-based approximation is shown in Figure 1. In general, Resurrector is much more precise than the GC-based approximation. For both Resurrector and the GC-based approximation, there is a delay between the death point of an object and the point at which its death is detected. The delay incurred by Resurrector is often much shorter—the death of the object is detected within a very small number of executions of its allocation site, while, using the GC-based approximation, the object's death cannot be detected until the next GC. In a large-scale application, the execution frequency of an allocation site can be orders

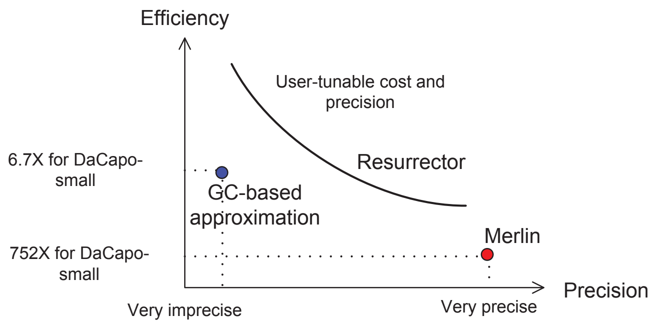


Figure 1. A precision-efficiency comparison among Merlin, Resurrector, and the GC-based approximation.

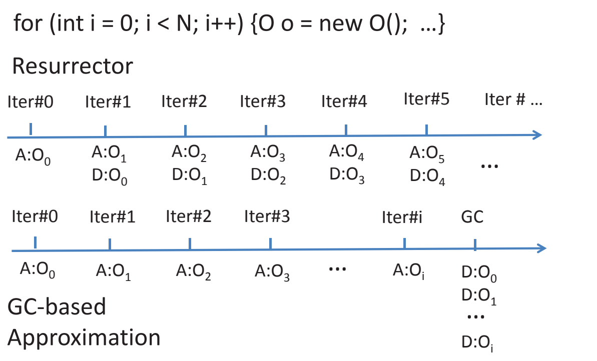


Figure 2. A graphical comparison between the lifetime information collected by Resurrector and by the GC-based approach.

of magnitude higher than that of GC. Figure 2 uses a simple example to compare the object lifetimes collected by Resurrector and by the GC-based approach. We use O_i to denote the i-th object created by the allocation site, and $A : O_i$ and $D : O_i$ to denote the creation and death events of the object. For Resurrector, the death of object O_i (created in the i-th iteration of the loop) is detected upon the next execution of the allocation site (i.e., in the $(i + 1)$-th iteration), while the GC-based approach has to wait until the next GC occurs.

During our experiments, we found that *Resurrector is much more efficient than Merlin and, in most cases, even more efficient than the GC-based approach*. Resurrector does not need the heap reachability analysis used by Merlin to identify dead objects; this task is being performed along with the mutator execution. It is less obvious to see why Resurrector has better performance than the GC-based approach—the primary reason is that the GC-based approach needs to find all *unreachable* objects from the heap and record their death during each GC, which incurs a heavy running time overhead; Resurrector, however, identifies death points for most objects during the Mutator execution, and does not rely on GC to find unreachable objects and report death events.

To further improve Resurrector's practicality, we use *the number of objects cached for each allocation site as a tuning parameter* to adjust Resurrector's precision and scalability. The more objects an allocation site caches, the more precise lifetime information Resurrector may produce and the higher overhead Resurrector may incur. We demonstrate, using experiments on real-world applications, that Resurrector is precise in finding optimization opportunities even when the threshold parameter is small, and yet it is efficient enough to be able to scale to large applications.

Resurrector has immediate benefit for all the existing optimization techniques that require precise understandings of object lifetimes. To demonstrate this benefit, we have implemented a client analysis that uses the Resurrector-collected lifetime information to find *large data structures that have disjoint lifetimes*. By inspecting and reusing the reported data structures, we have achieved significant performance gains (e.g., speed up a large program by 5×). This experience is described in the 4 case studies in Section 5. Resurrector has also been used to improve the optimization technique in our previous work [50]. The results show that the Resurrector profiles can significantly reduce numbers of false positives.

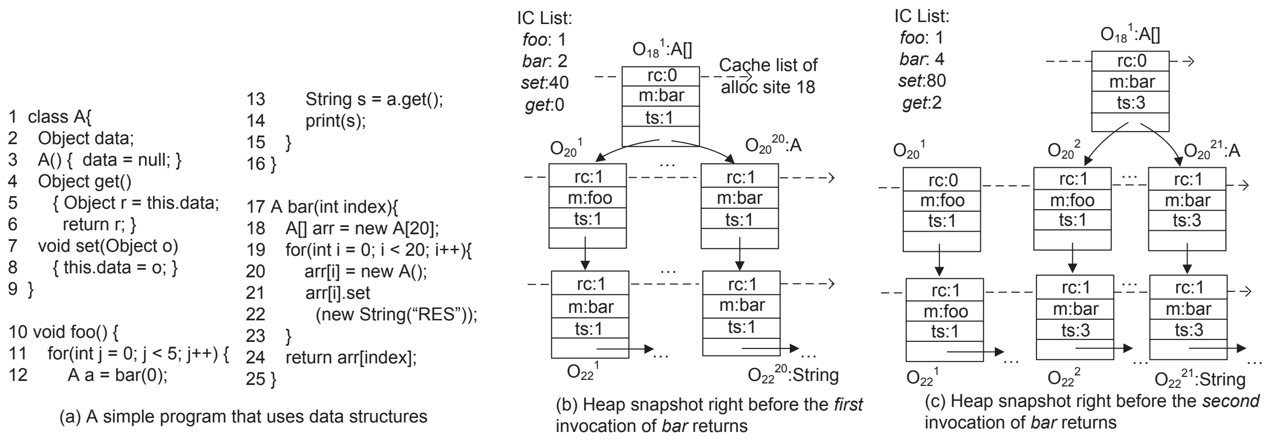
We have implemented Resurrector in Jikes RVM 3.1.3, a high performance research virtual machine written in Java, and successfully applied it to real-world applications. An evaluation of Resurrector on a set of 10 DaCapo programs (i.e., in its 2006 release) is described in Section 5. Varying the tuning parameter in the range [1, 500] results in an overall 1.3×–6.0× space overhead and 2.3×–13× time overhead (under the DaCapo large workloads). Resurrector is publicly available at the Jikes RVM Research Archive (`http://jikesrvm.org/Research+Archive`).

The contributions of this paper are:

- A novel lifetime profiling algorithm that supports user-tunable precision and scalability;
- an implementation of this algorithm in the Jikes RVM that provides compiler and runtime system support to reuse objects and collect their lifetimes;
- two client analyses that use this technique to identify lifetime-disjoint data structures;
- an evaluation on a set of large-scale, real-world applications showing that the precision of Resurrector is close to that of Merlin while it is orders of magnitude more efficient than Merlin;
- four case studies demonstrating that the Resurrector lifetime information can help programmers quickly identify reuse opportunities and fix problems for large performance gains.

2. Overview

This section presents an overview of the Resurrector OLP technique. Figure 3 (a) shows a simple program that keeps



```
1  class A{
2    Object data;
3    A() {  data = null; }
4    Object get()
5       { Object r = this.data;
6         return r; }
7    void set(Object o)
8       { this.data = o; }
9  }

10 void foo() {
11    for(int j = 0; j < 5; j++) {
12       A a = bar(0);
13       String s = a.get();
14       print(s);
15    }
16 }

17 A bar(int index){
18    A[] arr = new A[20];
19    for(int i = 0; i < 20; i++){
20       arr[i] = new A();
21       arr[i].set
22          (new String("RES"));
23    }
24    return arr[index];
25 }
```

(a) A simple program that uses data structures

(b) Heap snapshot right before the *first* invocation of *bar* returns

(c) Heap snapshot right before the *second* invocation of *bar* returns

Figure 3. A simple program and graphical illustrations of its run-time heap.

creating and using data structures. Despite the simplicity of the program, these data structures have multiple layers of objects; profiling their lifetimes using a traditional approach (such as Merlin) would require a reachability analysis that transitively identifies the death point for each object from the root (e.g., the array of type `A` created at line 18). In this section, we show how Resurrector eliminates the need to perform such a reachability analysis. As our focus is on the illustration of the basic idea, we assume the number of objects an allocation site can cache is unbounded.

Figure 3 (b) and (c) illustrate the two heap snapshots right before the first and the second invocation of method *bar* finishes, respectively. Each run-time object is represented by O_i^j, where i is the ID of its allocation site and j indicates that the object is the j-th object created by allocation site i. In this example, the line number of an allocation site is used as its ID. Each solid arrow represents a reference edge and each dashed line represents an object cache list for an allocation site. For simplicity, we assume this example has only one thread running, and thus, there is one cache list per allocation site.

Tracking Information As discussed in Section 1, when an allocation site is executed, Resurrector needs to know whether there exists a cached object that is dead and resurrectable. To precisely track the death point for each object, we associate with the object four pieces of tracking information: its allocation site ID ($alloc$), its reference count (rc), the ID of the method where the object may be captured (m), and a timestamp that records the invocation count (which will be discussed shortly) of m at the moment the object flows into m (ts). Following the escape analysis terminology [13, 16, 21, 48], an object is captured in a method if it does not escape the method boundary. Reference count rc is used to identify *when the object becomes unreachable in the heap*, while the method-related information (i.e., m and ts) is used to find *when the object loses all its stack references*. Because tracking the updates of stack variables can be very expensive, we take an efficient approach—Resurrector considers the return point of the object's capturing method as the point at which the object loses all its stack references. While the object may actually become unreachable from the stack earlier (e.g., when the execution leaves the control flow region in which its pointer variable is declared), this return point is often not far away and yet is easy to detect. Future work may consider to perform a precise static liveness analysis to identify the liveness scope for each object, which can then be fed to the runtime system for improved lifetime precision.

Determining Object Lifetimes rc is incremented every time a reference of the object is assigned to a heap location and is decremented every time its reference is removed from a heap location. This can be easily done by instrumenting heap loads and stores. To correctly identify the return point of the object's capturing method, Resurrector maintains an *invocation count* (IC) for each method in the program. In particular, the IC of a method is incremented twice for each invocation of the method, once at the beginning of the invocation and once at the end. Each method has its own counting system, and ICs of different methods are counted independently of each other. For example, within the first execution of method *bar*, its IC is 1; during the period after this invocation finishes and before the next invocation starts, its IC is 2. In this section, we focus our discussion on *recursion-free programs*; the handling of recursion and other advanced language features will be discussed shortly in Section 3. Clearly, the IC of a method is always an odd number if it is on the call stack, and it is always an even number otherwise.

Object o may flow to a method on the stack and be captured there in the following three cases. First, o is created and its reference is immediately written into a stack variable. In this case, o's m and ts are assigned the ID of its creating method and the current IC of the method, respectively. Second, o is returned from a callee to a caller. In this case, m

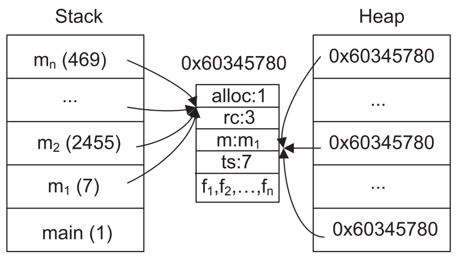


Figure 4. A Resurrector object in the heap.

and ts *may be* updated with the method and IC information of the caller. Finally, a reference of the object is assigned to a stack variable through a heap load. In this case, m and ts may be updated with the corresponding information of the method invocation in which this load occurs. Note that o can be captured in a method only if none of the callers of the method on the stack have a reference to it. Therefore, our goal is to keep track of the *"lowest" method on the call stack* where the object is referenced. To do this, before Resurrector updates $o.m$ and $o.ts$, it always verifies whether the (old) method recorded in $o.m$ has returned. For a recursion-free program, this verification can be easily done *by checking whether $o.m$'s current IC is greater than $o.ts$*. If $o.m$'s current IC is equal to $o.ts$ (indicating that $o.m$ is still on the call stack), $o.m$ and $o.ts$ do not need to be updated, because $o.m$ must be "lower" than (i.e., a caller of) the current method.

Note that parameter passing is not tracked in Resurrector, because an object passed from a caller to a callee can never be captured in the callee. Figure 4 illustrates a typical object in the Resurrector execution. For each method on the stack, its current (per-method) IC is shown in the parentheses. The object is referenced by three heap locations, and thus, its rc is 3. While it is referenced in four methods, its m and ts record the invocation information of m_1, which is the lowest method on the stack among them[1]. After the current invocation of m_1 returns, m_1's IC must be at least 8, which will be greater than the object's ts (7). It is easy to see that *an object o is ready to be resurrected if $o.m$'s IC is greater than $o.ts$ and $o.rc$ is 0.*

Example To illustrate, consider the first invocation of method *bar*. The initial m and ts for all objects created in *bar* are *bar* and 1, respectively. Allocation site 20 is executed 20 times. At each time, the Resurrector allocator attempts to find a dead object from the cache list (for allocation site 20), that is, an object such that its reference count is 0 and its capturing method m's current IC is greater than its ts. Obviously, such an attempt fails because all the existing `A` objects created at line 20 have non-zero reference counts and their capturing method (*bar*)'s IC is equal to their ts, indicating that the method has not returned yet. Hence, a fresh `A` object is created and stored in line 20's cache list. At the end of this invocation, the cache lists for both line 20 and line 22 have 20 objects, as shown in Figure 3 (a). Note that although O^1_{22} is passed into *set*, its m and ts are not updated because *bar* is lower than *set*.

Right before the the first invocation of *bar* finishes, *bar*'s IC becomes 2. Resurrector finds the return object (O^1_{20}) and attempts to update its m and ts with, respectively, the ID of the caller (*foo*) and the current IC of the caller (1). Before the update, it checks whether the method recorded in $O^1_{20}.m$ has finished. The attempt succeeds because *bar*'s IC (2) is now greater than $O^1_{20}.ts$ (1). Note that the objects reachable from O^1_{20} (e.g., O^1_{22}) are not updated at this moment because they do not flow to the stack. O^1_{22} is updated later when line 13 is invoked—its m is changed to *foo* (due to the return at line 6) and its ts is changed to 1 (i.e., *foo*'s IC), as shown in Figure 3 (b).

During *bar*'s second invocation, the array (O^1_{18}) created in the previous invocation of *bar* is resurrected by Resurrector—it is the only object on line 18's cache list, its rc is 0, and its capturing method (*bar*)'s current IC is 3, which is greater than its ts (1). Resurrector determines it is a dead object, and resurrects it by cleaning up its contents and returning it to the application. This resurrection point is recognized as the death point of the old O^1_{18} and the creation point of the new O^1_{18}. The two events are recorded in a trace if Resurrector's trace generation option is enabled. This point can be much earlier than the GC point at which the old O^1_{18} is collected (in a non-Resurrector execution), which explains Resurrector's high precision over the GC-based approach in most cases.

Cleaning up O^1_{18}'s contents includes three steps: decrementing rc of each object it directly references, zeroing the array, and reinitializing its tracking information. Its m and ts now become *bar* and 3, respectively. After O^1_{18} is resurrected, rc of all the 20 cached `A` objects (O^1_{20}—O^{20}_{20}) becomes 0. Upon the execution of line 20, Resurrector inspects object O^1_{20}, which is the first cached object on the list. O^1_{20} is not resurrectable—although its rc is 0, its capturing method (*foo*)'s current IC is equal to its ts. However, O^2_{20} is resurrectable, because it was captured in a finished invocation of *bar*. This can be seen from the fact that *bar*'s current IC (3) is greater than O^2_{20}'s ts (1).

Subsequently, the resurrection of O^2_{20} decrements the reference count of O^2_{22}, making it resurrectable. It is resurrected when line 22 is executed. In the second invocation of *bar*, when the loop at line 19 terminates, 19 `A` objects created in the previous invocation of *bar* have been reused and one new `A` object (O^{21}_{20}) has been created. Hence, line 20's cache list now contains 21 objects, as shown in Figure 3 (b). Before *bar*'s second invocation finishes, O^2_{20} is about to be returned, and its m and ts are updated with *foo* and 1, respectively (similarly to the update of O^1_{20} in the previous invocation).

[1] For clarity of presentation, we use the name of a method as its ID in the paper.

It is easy to calculate that at the moment the loop at line 11 terminates, there are totally 24 objects on line 20's cache list. This is very close to the actual number of objects that are needed simultaneously for allocation site 20 (which is 20). The difference is due to the use of the return point of an object's capturing method as the point at which the object loses all stack references. For example, even if object O_{20}^{1} escapes to *foo*, it will never be used after the first iteration of the loop (at line 11) finishes. Hence, it is actually resurrectable in the second invocation of *bar*. Understanding that O_{20}^{1}'s lifetime is within the first iteration of the loop requires a precise static liveness analysis, such as the one proposed in [28]. It is worth investigating, in the future work, how this technique can be incorporated into Resurrector and how much precision improvement may result from it. Note that if the GC-based approximation is used to compute this number, it is very likely that we will get 100 (if GC occurs after method *foo* returns).

3. The Resurrector Profiling Technique

This section presents the core technique of Resurrector. Resurrector supports precise handling of all important features of Java, including multithreading, reflection, recursion, and exception handling. We first describe our baseline algorithm (for programs without recursive methods and exceptions), and then we gradually modify the baseline algorithm to consider more advanced language features.

3.1 Preliminaries

We begin by formalizing notions of operations, events, and execution traces. A program consists of a number of concurrently executing threads, each with a thread identifier $t \in \mathit{Tid}$, and these threads call methods $m \in \mathit{Mid}$, which manipulate (reference-typed) variables $x \in \mathit{Var}$. A method contains allocation sites $a \in \mathit{Alloc}$, which create run-time objects $o \in \mathit{Obj}$. An object o has a set of fields $f \in F$, each of which specifies the offset of a location in o. An environment $\rho \in \mathit{Var} \rightarrow \mathit{Obj} \cup \{\mathit{null}\}$ maps each variable to the object it points to or a null value. Each o is associated with a 6-tuple tracking information $\langle \mathit{gid}, \mathit{rc}, m, t, \mathit{ts}, \mathit{dm}\rangle$: a global object ID $\mathit{gid} \in \mathit{Nat}$, a heap reference count $\mathit{rc} \in \mathit{Nat}$, a method ID $m \in \mathit{Mid}$, a thread ID $t \in \mathit{Tid}$, a timestamp $\mathit{ts} \in \mathit{Nat}$, and a death mark $\mathit{dm} \in \{T, F\}$. gid uniquely identifies the object during the whole execution. In a multithreaded program, we use m and t to identify the capturing method of the object. The death mark dm is a boolean flag, indicating whether the death point of the object has already been recorded. Note that this tuple includes all the information needed by a client analysis. Various optimizations can be performed to reduce the unnecessary information to save space.

Each method m has an invocation count vector $IV : \mathit{Tid} \rightarrow \mathit{Nat}$, which records m's invocation count in each thread in the system. The set of operations that a thread t can perform includes:

- *new* (t, a), which creates an object at allocation site a;
- *store*$(t,\ x_1.f,\ x_2)$ and *load*$(t,\ x_2,\ x_1.f)$, which, respectively, writes the (reference) value of x_2 into and reads it from field f of the object referenced by x_1;
- *enter*(t, m) and *exit*(t, m), which enters and exits method m;
- *return* (t, x), which returns the object pointed to by x to the caller;

Note that an *exit* operation must be followed immediately by a *return* operation, which may or may not return any value. A thread t has a method stack $S : \mathit{Nat} \rightarrow \mathit{Mid}$. A method m is pushed onto the stack when thread t *enters* it and is popped out when t *exits* it. Each allocation site a has an object cache list $L : \mathit{Nat} \rightarrow \mathit{Obj}$ per thread t. In other words, a maintains a cache list vector $LV : \mathit{Tid} \rightarrow L$. Caching objects on a per-thread basis allows the concurrent execution of the dead object retrieval and resurrection; otherwise, heavy synchronization has to be performed at each allocation site to guarantee thread safety. An object event $e \in \mathit{Eve}$ is generated when an object is created or when Resurrector determines the object is dead. In Resurrector, e is a triple $\langle \mathit{gts}, \mathit{gid}, 'A'|'D'\rangle$, where $\mathit{gts} \in \mathit{Nat}$ is a global timestamp indicating when this event occurs. In the memory management literature, *bytes of allocation* is often used as a metric to measure time. gid is the global ID of the object and $'A'$ ($'D'$) indicates that the event is an allocation (death) event. Resurrector eventually collects a trace α of events, in which the timestamp distance between the $'D'$ event and the $'A'$ event of the same object determines the lifetime of the object.

3.2 The Baseline Algorithm

This subsection describes the core Resurrector OLP algorithm. Resurrector's analysis state includes:

Invocation count map	I	:	$\mathit{Mid} \rightarrow IV$
Method stack map	T	:	$\mathit{Tid} \rightarrow S$
Alloc site cache map	C	:	$\mathit{Alloc} \rightarrow LV$
Object event trace	α	:	$e, \alpha \mid \epsilon$

Resurrector's instrumentation semantics at the seven operations are shown in Algorithm 1–7, each of which reads and/or updates the analysis state. Algorithm 1 (that handles allocation sites) first retrieves the current method on the call stack of thread t (line 2), which will be used later to update m and ts of the newly created object. As described in Section 2, Resurrector iterates through a's cache list in t (denoted as $C(a)(t)$) to find a dead object created previously by a (line 3–22). The condition in line 6 shows Resurrector's criterion of selecting dead objects—an object o is dead and resurrectable if $o.rc$ is 0 and the invocation count of its capturing method $o.m$ in thread $o.t$ (denoted as $I(o.m)(o.t)$) is

Algorithm 1: The modified semantics of *new* (t, a).

```
Input: Thread $t$, Allocation site $a$
Output: New object $ret$
1  Object $ret \leftarrow null$
2  Method $currMet \leftarrow T(t)_0$
3  foreach index $i \in [0, |C(a)(t)|)$ do
4      /*Iterate through $a$'s cache list in $t$*/
5      Object $o \leftarrow C(a)(t)_i$
6      if $o.rc = 0 \wedge I(o.m)(o.t) > o.ts \wedge (\neg isArray(o) \vee length(o) = requestedLength(a))$ then
7          /*Found a dead object*/
8          $ret \leftarrow o$
9          if $o.dm$ = $F$ then
10             $\alpha \leftarrow append(\alpha, \langle globalTS(), o.gid, 'D' \rangle)$
11         else
12             $o.dm \leftarrow F$
13         $o.gid \leftarrow newGID()$
14         $o.rc \leftarrow 0$
15         $o.m \leftarrow currMet$
16         $o.t \leftarrow t$
17         $o.ts \leftarrow I(currMet)(t)$
18         foreach reference-typed field $f$ in $o$ do
19             Object $o' \leftarrow o.f$
20             if $o' \neq null$ then
21                 $o'.rc \leftarrow o'.rc - 1$
22         break
23 if $ret = null$ /*No object has been resurrected*/ then
24     $ret \leftarrow allocNew()$
25     $ret.gid \leftarrow newGID()$
26     $ret.rc \leftarrow 0$
27     $ret.dm \leftarrow F$
28     $ret.m \leftarrow currMet$
29     $ret.t \leftarrow t$
30     $ret.ts \leftarrow I(currMet)(t)$
31     Cache list $l \leftarrow C(a)(t)$
32     $l \leftarrow append(l, ret)$
33 else
34     /*move object $ret$ to the end of the list $C(a)(t)$*/
35 $\alpha \leftarrow append(\alpha, \langle globalTS(), ret.gid, 'A' \rangle)$
36 $zeroMemory(ret)$
37 return $ret$
```

Algorithm 2: Instrumentation before *store* $(t, x_1.f, x_2)$.

```
Input: Thread $t$, Field dereference expression $x_1.f$, variable $x_2$
1  Object $o_1 \leftarrow \rho(x_1)$
2  Object $o_2 \leftarrow \rho(x_2)$
3  Object $o' \leftarrow o_1.f$
4  if $o' \neq null$ /*Update rc of $o'$*/ then
5      $o'.rc \leftarrow o'.rc - 1$
6  if $o_2 \neq null$ /*Update rc of $o_2$*/ then
7      $o_2.rc \leftarrow o_2.rc + 1$
```

Algorithm 3: Instrumentation before *load* $(t, x_2, x_1.f)$.

```
Input: Thread $t$, Field dereference expression $x_1.f$, variable $x_2$
1  Object $o_1 \leftarrow \rho(x_1)$
2  Object $o_2 \leftarrow o_1.f$
3  if $o_2 \neq null$ then
4      if $o_2.t \neq t \wedge I(o_2.m)(o_2.t) = o_2.ts$ then
5          $untrack(o_2)$
6          return
7      if $I(o_2.m)(t) > o_2.ts$ then
8          Method $currMet \leftarrow T(t)_0$
9          $o_2.t \leftarrow t$
10         $o_2.m \leftarrow currMet$
11         $o_2.ts \leftarrow I(currMet)(t)$
```

Algorithm 4: Instrumentation before *enter*(t, m).

```
Input: Thread $t$, Method $m$
1  Stack $s \leftarrow T(t)$
2  $s \leftarrow push(s, m)$
3  $I(m)(t) \leftarrow I(m)(t) + 1$
4  /*Invariant: $I(m)(t)$ must be an odd number*/
```

Algorithm 5: Instrumentation before *exit*(t, m).

```
Input: Thread $t$, Method $m$
1  Stack $s \leftarrow T(t)$
2  $s \leftarrow pop(s)$
3  $I(m)(t) \leftarrow I(m)(t) + 1$
4  /*Invariant: $I(m)(t)$ must be an even number*/
```

Algorithm 6: Instrumentation before *return*(t, x).

```
Input: Thread $t$, Return variable $x$
1  Object $o \leftarrow \rho(x)$
2  if $o \neq null \wedge I(o.m)(o.t) > o.ts$ then
3      /*Update $o$'s method info with the caller's info*/
4      Method $caller \leftarrow T(t)_0$
5      $o.ts \leftarrow I(caller)(t)$
6      $o.m \leftarrow caller$
```

greater than $o.ts$. In addition, if the allocation site creates an array, the requested array length must be equal to o's length. Other than what has been described in Section 2, this algorithm records an 'A' event (line 35), indicating that a new object is created (regardless of whether it is a fresh new object or it is a resurrected object). If a dead object is found and its death has not yet been recorded, a 'D' event is appended to the trace (line 10). Resurrector allows the user to specify whether a trace is needed, because many client analyses (such as those discussed in Section 4) can be implemented without a trace. If the trace generation option is not selected, these events do not need to be recorded. After an object is resurrected, we move it to the end of the cache list (lines 34) to improve the performance of the next search. This is based on the observation that objects' death order is often consistent with their creation order.

Resurrector's instrumentation at each store (shown in Algorithm 2) updates reference counts in expected ways. At each load (shown in Algorithm 3), Resurrector first checks whether the retrieved object has stack references in different threads (line 4–6). The death point for an object referenced simultaneously from stacks of multiple threads cannot be precisely determined unless the object maintains per-thread tracking information, which can be prohibitively expensive in large systems. Hence, Resurrector stops tracking the object (i.e., removes its tracking information) and re-

Algorithm 7: The modified garbage collection semantics.

```
Input: Heap $h$
foreach Object $o \in h$ do
    if $\neg$reachable(o) $\vee$ ($o.rc = 0 \wedge I(o.m)(o.t) > o.ts$) then
        /*Record death for $o$*/
        if $o.dm = F$ then
            $\alpha \leftarrow append(\alpha, \langle globalTS(), o.gid, 'D' \rangle)$
        if reachable(o) then
            $o.dm \leftarrow T$
```

lies on GC to detect its death point. Note that this handling does not lead to significant precision loss because (1) most heap objects are thread-local [19], and (2) none of the existing object-reduction-based optimization techniques (such as [7, 32, 50]) can optimize shared objects. If o_2 is not referenced by multiple thread stacks, Resurrector updates its m and ts with the information of the current method invocation (line 8–11).

Algorithm 4 and Algorithm 5 pushes and pops method m onto and out of t's method stack, respectively. Both algorithms increment m's IC. Algorithm 6 inspects the return object, and updates its m and ts with the corresponding information of the caller (if its old m has returned). Note that because the *return* of a method always comes after the *exit* of the method (that increments the method's IC and pops the stack, as shown in Algorithm 5), when the return object is inspected by Algorithm 6, the callee is treated as "already returned" (i.e., that is why the caller is retrieved by $T(t)_0$).

Algorithm 7 shows our modification of a tracing collector. It is important to note that the modified collector records death points not only for unreachable objects but also for *cached objects that are ready to reuse*. As such, an object's death point can be recorded either (1) when it is reused, or (2) when a GC occurs and the object is ready to reuse, or (3) when a GC occurs and the object is unreachable, whichever event comes first. This guarantees that the Resurrector-collected lifetime information can never be worse than the lifetime information obtained from a GC. When the object's death point is recorded, its death mark is set to T, which will prevent Resurrector from recording its death again (upon its reuse). Its death mark is set back to F after it is reused (line 11 in Algorithm 1).

3.3 Algorithm Correctness

In this subsection, we demonstrate the correctness of the Resurrector algorithm by showing that it preserves the semantics of the original program. We first show that any object resurrected by Resurrector is guaranteed to be a dead object that will no longer be used in the forward execution.

PROPOSITION 1. (Heap Unreachability). *If rc of an object is 0, there must not exist any heap location that contains a reference to the object.*

This proposition is straightforward to see.

PROPOSITION 2. (Correctness of the Return Point Detection). *For each object o tracked by Resurrector, if $I(o.m)(o.t) > o.ts$, the method invocation $o.m$ in which o is referenced must have returned.*

This is because $o.ts$ captures the IC of the method $o.m$ at the moment o is referenced in $o.m$. Because IC can change only at the entry or the exit, $I(o.m)(o.t) > o.ts$ implies that at least an *exit* operation has occurred on $o.m$ in $o.t$ (if $o.m$ is not a recursive method).

PROPOSITION 3. (Lowest Method on the Stack). *For each object o tracked by Resurrector, there must not exist any method that is lower than $o.m$ on $o.t$'s call stack and that contains a variable referencing o.*

We prove this by contradiction. Suppose there exists a method n that is a caller of $o.m$ and that has a reference to o. The reference of o can flow to n only in three cases: (1) o is created in n, (2) o is returned to n from a callee (such as $o.m$), or (3) a load retrieves o from a heap location. In case (1), $o.m$ should be initialized with n. Since n has not returned, $o.m$ can never be updated to a callee of n; in case (2) and (3), similarly, $o.m$ would have already been updated to n (at the *return* operation and the *load* operation, respectively).

LEMMA 1. (Resurrection Soundness). *An object chosen by Algorithm 1 for reuse must be a dead object.*

To prove the lemma, we consider heap and stack reachability separately. At the moment object o is chosen by Algorithm 1, $o.rc$ must be 0, which implies that there does not exist any heap location that contains a reference to o (Proposition 1). On the other hand, condition $I(o.m)(o.t) > o.ts$ implies that the lowest method invocation (on the call stack) in which o is referenced has finished (Propositions 2 and 3). o must be captured in this method invocation because it escapes neither to the heap (otherwise $o.rc$ would not have been 0) nor to the caller of this method (otherwise this lowest method would have been the caller).

THEOREM 1. (Semantics Preservation). *The Resurrector execution preserves the semantics of a program.*

From Lemma 1, we know that an object resurrected by Resurrector will never be used in the forward execution. If the allocation site creates an array, Resurrector returns a dead array with exactly the same length. Since the contents of a dead object are removed before it is reused (line 32 in Algorithm 1), the application sees the object as if it were a freshly created object.

3.4 Defining a Tradeoff Framework

The size of the cache list for each allocation site can be naturally used as a tuning parameter to define a tradeoff frame-

Algorithm 8: Instrumentation before *enter*$(t,\ m)$ that handles recursion.

```
Input: Thread $t$, Method $m$
1 Stack $s \leftarrow T(t)$
2 $s \leftarrow push(s, m)$
3 $depth \leftarrow D(m)(t)$
4 if $depth = 0$ then
5     $I(m)(t) \leftarrow I(m)(t) + 1$
6 $D(m)(t) \leftarrow depth + 1$
7 /*Invariant: $I(m)(t)$ must be an odd number*/
```

Algorithm 9: Instrumentation before *exit*$(t,\ m)$ that handles recursion.

```
Input: Thread $t$, Method $m$
1 Stack $s \leftarrow T(t)$
2 $s \leftarrow pop(s)$
3 $D(m)(t) \leftarrow D(m)(t) - 1$
4 if $D(m)(t) = 0$ then
5     $I(m)(t) \leftarrow I(m)(t) + 1$
6 /*Invariant: $I(m)(t)$ must be an even number*/
```

work between precision and scalability. This framework allows for the tuning of Resurrector for different client analyses that have different precision/scalability requirements. For example, supporting longer cache lists at allocation sites would allow Resurrector to find more reusable objects and thus produce more precise lifetime information. However, certain allocation sites can create great numbers of lifetime-overlapping objects; caching all of them may lead to significant time and space overheads. Furthermore, these allocation sites are not likely to be optimizable; precisely tracking lifetimes for all of their objects may not produce much benefit for a client optimization technique.

When the number of objects on a cache list exceeds a user-specified threshold parameter, we release the whole list so that the cached objects can be garbage collected. Their collection points will be recognized as their death points. Note that if the threshold is 0, the lifetime information computed by Resurrector is exactly the same as that approximated using GC. On the other hand, if the length of a cache list is unbounded, all objects created by the allocation site would be cached and thus, the precision of our lifetime information can be very close to (but still lower than) that of Merlin's lifetime information. Of course, doing this suffers from the same scalability problem as Merlin does and thus would not work for real-world programs.

Because most optimization techniques target allocation sites that create small numbers of lifetime-overlapping objects, running Resurrector with a small threshold can often provide sufficiently precise information for these techniques. During our experiments, we have evaluated Resurrector with several different threshold parameters; our results demonstrate that large optimization opportunities can be found even when the threshold value is very small.

3.5 Handling of Advanced Language Features

This subsection discusses how we revise our baseline algorithm to deal with advanced language features.

3.5.1 Recursion

The current IC-based scheme does not function properly in the presence of recursion. Naively incrementing the IC of a method at its entry and exit can inappropriately direct Resurrector to resurrect live objects created by a recursive method invocation. To solve the problem, we maintain a recursion depth vector $DV\ :\ Tid\ \rightarrow\ Nat$ for each method, which records the recursion depth (RD) for the method in each thread. Suppose D maps each method m to its DV. Algorithms 8 and 9 describe, respectively, our modified algorithms at the method entry and exit to properly handle recursive invocations. Every time an *enter*$(t,\ m)$ operation is encountered, Resurrector first checks if m's recursion depth count (denoted as $D(m)(t)$) is 0. m's IC (i.e., $I(m)(t)$) is incremented only if this is the first time m is pushed onto t's stack. Similarly, at an *exit*, m's IC is incremented only if the current invocation is the last invocation of m on t's stack. All the recursive invocations of the same method share the same IC. $D(m)(t)$ is incremented at each *enter* and decremented at each *exit*.

Note that the other four algorithms are still correct after the instrumentation at *enter* and *exit* is modified. This is because an object created in any invocation of a recursive method cannot be resurrected until the first invocation of the method returns. However, this would increase the delay (between an object's actual death point and the point at which Resurrector detects its death) and prevent Resurrector from quickly reusing a dead object. To solve the problem, we add one additional field rd in the tracking data of each object to record the recursion depth of its capturing method m. Whenever an object's method-related information (i.e., m, t, and ts) needs to be updated, its rd is updated (to $D(m)(t)$) as well. Hence, whether its m has returned can be verified by checking the following relaxed condition C_1:

$$I(o.m)(o.t) > o.ts \vee D(o.m)(o.t) < o.rd \qquad (C_1)$$

This condition would allow Resurrector to quickly identify a dead object created in a finished recursive invocation.

3.5.2 Exception handling

Resurrector modifies an existing exception delivery algorithm in Jikes RVM to appropriately update the IC and RD of each method involved in the exception delivery. Particularly, when an exception is thrown, this existing algorithm walks the call stack to find the most recent method invocation that has an handler for the exception. When this algorithm is about to exit an invocation on the stack, Resurrector treats it as an *exit* event and performs the updates as shown in Algorithm 9. In the method that catches the exception, Resurrector performs a *return* operation (Algorithm 6) with the exception object being treated as the returned object.

3.5.3 Object cloning

In Java, the `clone` method can be invoked on an object o to create a new object o' of the same type and perform a *shallow copy* of o's data fields into o'. Resurrector treats a call site that invokes `clone` as a special allocation site. At the end of the cloning process, Resurrector inspects each reference-typed field in o'; if the field contains a non-null reference value, Resurrector retrieves the referenced object and increments its reference count.

3.5.4 Handling of multi-dimensional arrays

Different JVMs may have different implementations of multi-dimensional arrays. In Jikes RVM, each i-dimensional array of the form `A[]...[]` is implemented by maintaining and connecting a number of one-dimensional arrays of type `java.lang.Object` and type `A`. Resurrector assigns the same allocation site ID to all of the arrays upon their creation and caches them into the same cache line. Their reference counts are initialized appropriately based on the reference relationships among them. When the allocation site is executed again to create a new multi-dimensional array, Resurrector inspects all the cached one-dimensional arrays to find dead ones in order to assemble this new array.

3.5.5 Handling of `arraycopy`

Since method `arraycopy` is implemented via native code, Resurrector needs to explicitly account for its side effects. When objects in array a are copied into array b using `arraycopy`, Resurrector increments their reference counts so that these objects cannot be mistakenly resurrected. However, a user-defined native method may also write or read heap references; failing to model these effects may crash a program or introduce arbitrary behaviors. Future work may address this issue by developing low-level support (e.g., at the virtual memory level) to detect reads and writes performed in native code.

3.5.6 Reflection and recursive data structures

Resurrector detects calls to method `newInstance` during compilation and treats them as allocation sites.

As with any other reference-counting-based algorithm, Resurrector cannot appropriately handle recursive data structures. Their reference counts are always non-zero, preventing Resurrector from correctly identifying their death points. In the current implementation, Resurrector leaves them to the garbage collector. Because they are always unreusable, their allocation sites' cache lists will keep growing. When the number of objects on a cache list exceeds the threshold parameter, the whole list will be released and garbage collected. Future work needs to investigate how to incorporate techniques such as trial deletion [4, 6, 46] into Resurrector to precisely handle recursive data structures. It is also interesting to measure, in the future work, how much precision loss is due to Resurrector's inability of handling recursive data structures; in the current implementation, we do not have any information regarding whether or not the discarding of a cache list is caused by cycles.

4. Implementation and Clients

We have implemented Resurrector on Jikes RVM 3.1.3 [27], a Java-in-Java Virtual Machine. To demonstrate its usefulness for optimizing programs, we have implemented two client analyses: one that finds allocation sites creating very few lifetime-overlapping objects and second that approximates the sizes of run-time data structures. These two analyses are then used to find optimization opportunities in large-scale, real-world programs.

4.1 Implementation and Optimization

For each tracked object o in Resurrector, we create an additional metadata object o' to store its tracking information. o' is *implicitly referenced* by o (in its header space) while o is *explicitly referenced* by o' (in a field). Resurrector caches o by adding o' onto the cache list. This will prevent o from being garbage collected (after it becomes unreachable). Yet, if o' is removed from the list (e.g., when Resurrector decides to untrack o) and o becomes unreachable, both o and o' will be appropriately garbage collected.

We have modified both the baseline compiler and the optimizing compiler of the Jikes RVM to perform the instrumentation. Instrumentation code is added at each allocation site, each store, each load, and the entry, exit and return point of each method. The invocation count vector and the recursion depth vector for each method are created as two new fields of `RVMMethod`. The method stack S on each thread is implemented as an integer array and kept in a field of `RVMThread`. Cache lists for allocation sites are implemented as a list array and kept in a field of `RVMThread`.

As described in Algorithm 7, the garbage collector is modified in such a way that it records a death event when (1) a resurrectable (live) object is traversed during the reachability analysis or (2) an untracked object is collected. While this modification has been done only in the Mark-and-Sweep GC, the algorithm can be easily implemented in all the other GCs.

Optimizations Two major optimizations have been performed to reduce the time and space costs. First, since each thread may actually execute only a small portion of allocation sites, it is not necessary to create cache lists for all allocation sites in each thread. To optimize this case, Resurrector creates a cache list for an allocation site in a thread only when the allocation site is executed by the thread. Resurrector maintains a global index array for each allocation site a that records, for each thread, the index of a's cache list in the thread. This optimization significantly reduces the number of cache lists that need to be created. Second, for each object, its tracking fields t and m can be combined into one single field $addr_i$, that directly records the address of $I(m)(t)$. Similarly, we use another field $addr_d$ to record the address of $D(m)(t)$. As $I(m)(t)$ and $D(m)(t)$ are the most frequently accessed tracking data, this combination leads to more efficient retrieval of IC and RD.

4.2 Client Analyses

Detecting Allocation Site Optimizability One common piece of information required by many optimizations is the maximal number of lifetime-overlapping objects needed by an allocation site. We develop a client analysis that computes this information by keeping track of the maximal length of each cache list. Specifically, we maintain a max-length count for each allocation site; this count is checked every time a cache list for this allocation site (in a thread) grows and it is updated when its value is smaller than the current list length. For an allocation site whose cache list has been released (because its size has exceeded the threshold), its max-length is assigned a very large number (e.g., the maximal value of integer). While the max-length for an allocation site is an (over-)approximation of its maximal number of overlapping objects, these two numbers can be very close if the allocation site is frequently executed.

Data Structure Size Approximation An important optimizability measurement used in many optimization techniques is the size of a data structure (i.e., the number of objects reachable from its root). For example, if an allocation site always has one live object and the data structure rooted at the object has a large size, significant performance gain may result from optimizing this allocation site (e.g., cache the whole data structure so that the effort of recomputing its contents can be saved). We have implemented another client analysis in Resurrector to approximate the size of the data structure rooted at an object when the object is allocated/resurrected. As it is a common practice that most objects in a data structure are created and initialized in the constructor of its root object, we augment Resurrector to instrument the entry and exit of each constructor to calculate the number of objects created in the constructor. Specifically, we maintain an object stack in each thread; an object on the stack indicates that its constructor is currently being executed. Upon the creation/resurrection of an object o, we first retrieve each object r already on the object stack and increment its data structure size. In other words, o is considered to be part of the data structure rooted at each such r. Next, we push o onto the stack. Upon the exit of its constructor, o is popped out of the stack.

The reports of these two client analyses are generated and used to understand the precision and usefulness of the information collected by Resurrector. In particular, allocation sites are first sorted in ascending order of their max-length counts. The top 100 allocation sites on this list are re-sorted in descending order of the multiplication of their frequencies and their (approximated) data structure sizes. We have manually inspected the final list to find optimization opportunities.

5. Evaluation

Our benchmarks (shown in Table 1) include 10 programs from the 2006 release of the DaCapo benchmark set. Resurrector encounters an error when running eclipse, which is thus not included in our subjects. All experiments are executed on a quadcore machine with an Intel Xeon E5620 2.40GHz processor, running Linux 2.6.18. The maximal heap size specified for each program run is 2GB. The Jikes RVM configuration for the experiments is FastAdaptive-MarkSweep, which specifies the use of the optimizing compiler and the Mark-and-Sweep garbage collector. Our implementation distinguishes objects created in the VM code and those created in the regular Java code; objects created by the VM are not tracked.

5.1 Resurrector Performance

Time Overhead on DaCapo-small Table 1 reports various running time statistics collected from the executions of DaCapo small workloads. To compare Resurrector with existing techniques, we have obtained and run Elephant Tracks [40, 41], which is the only publicly available tool implementing the Merlin algorithm [26]. We have also implemented a GC-based lifetime approximation technique on Jikes RVM 3.1.3. Instead of finding every single dead object from the heap and reporting its death during each GC (which may incur a significant time overhead), our implementation of the GC-based approximation records the address of each tracked object upon its creation in a list, and traverses the list to check the reachability of each object at the end of each GC. Hence, our implementation would incur higher space overhead but lower time overhead than the naive implementation (that needs to find all dead objects from heap blocks).

We choose to report the running times of the small workloads in order to enable the comparison among Resurrector, Elephant Tracks, and the GC-based approximation. The Merlin algorithm implemented in Elephant Tracks is very slow and generates extremely large trace files (e.g., at the scale of dozens to hundreds of Gigabytes). We could not finish running most of the benchmarks for even default workloads within a reasonable amount of time and space. Sections (a), (b), and (c) in Table 1 report, respectively, the execution time information of Elephant Tracks, Jikes RVM without instrumentation, and the GC-based approximation. Elephant Tracks is implemented based on JVMTI, a native interface currently not supported by Jikes RVM. Hence, we have to run Elephant Tracks on HotSpot (version 1.6.0_27); the running times of the original HotSpot and Elephant Tracks are shown in columns T_H and T_{ET}, respectively. Elephant Tracks crashes during the running of `bloat` and `chart`; the numbers annotated with $*$ are obtained from the experimental results in [40].

Bench	(a) ET (HotSpot)		(b) R	(c) GCA	(d) Resurrector ($ml = 1, 10, 100, 200, 500, \text{and } \infty$)					
	T_H	T_{ET}	T_R	T_G	T_1	T_{10}	T_{100}	T_{200}	T_{500}	T_∞
antlr	0.48	169.8 (352.3)	0.41	1.79 (4.4)	1.15 (2.9)	1.45 (3.6)	1.26 (3.1)	1.08 (2.7)	1.16 (2.9)	1.83 (4.5)
bloat	2.87*	17336.2 (6036)*	1.03	12.33 (12.0)	3.53 (3.4)	4.46 (4.3)	3.60 (3.5)	3.45 (3.3)	3.63 (3.5)	98.3 (95.3
chart	2.96*	19425.9 (6565)*	2.12	6.90 (3.3)	6.96 (3.3)	7.60 (3.6)	7.06 (3.3)	8.72 (4.1)	7.8 (3.7)	137.3 (64.8)
fop	1.81	182.2 (100.6)	1.04	4.97 (4.8)	2.13 (2.0)	2.08 (2.0)	1.94 (1.9)	2.00 (1.9)	2.34 (2.3)	12.35 (11.9)
hsqldb	0.87	261.9 (300.3)	0.66	3.51 (5.3)	2.09 (3.2)	2.61 (3.9)	2.20 (3.3)	2.14 (3.2)	2.16 (3.3)	106.2 (160.4)
jython	0.32	1283.1 (4061.1)	0.70	9.23 (13.2)	4.96 (7.1)	4.86 (7.0)	6.04 (8.7)	6.03 (8.7)	6.07 (8.7)	271.5 (389.5)
luindex	0.75	367.8 (487.2)	0.55	2.23 (4.1)	1.19 (2.2)	2.73 (5.0)	2.46 (4.5)	2.67 (4.9)	2.69 (4.9)	22.59 (41.3)
lusearch	0.74	1334.1(1805.1)	0.79	12.3 (15.4)	4.03 (5.1)	4.46 (5.6)	4.40 (5.5)	4.16 (5.2)	3.82 (4.8)	32.63 (41.3)
pmd	1.19	37.9 (31.8)	0.52	2.62 (5.0)	1.15 (2.2)	2.39 (4.6)	1.07 (2.1)	1.21 (2.3)	1.20 (2.3)	4.52 (8.7)
xalan	1.3	1578.6(1214.3)	0.47	3.55 (7.5)	1.49 (3.1)	1.72 (3.6)	1.71 (3.6)	1.91 (4.0)	1.87 (4.0)	2.80 (5.9)
GeoMean		752.4×		6.7×	3.2×	4.4×	3.6×	3.7×	3.7×	40.2 ×

Table 1. Running time statistics for DaCapo-small: Section (a) reports the original HotSpot (T_H) and the Elephant Tracks (T_{ET}) running times; Section (b) reports the original Jikes RVM running time; Section (c) reports the running time of the GC-based approximation; Sections (d) reports Resurrector's running times for different cache list size thresholds; running time is measured in seconds; overhead is measured in slowdown ratios shown in parentheses. * Those numbers are taken from the experimental results reported in [40].

We have run Resurrector with six different max-length parameters, including 1, 10, 100, 200, 500, and ∞. $ml = \infty$ means that the length of a cache list is unbounded and cache lists are never discarded. The slowdown information (in parentheses) under column T_i is obtained by calculating the ratios between the numbers in T_i and their corresponding numbers in T_R. Similarly, the slowdowns reported under column T_{ET} are obtained by calculating the ratios between the numbers in T_{ET} and those in T_H. Although Elephant Tracks and Resurrector are executed on two different JVMs, it is clear to see that Elephant Tracks is orders of magnitude slower than both Resurrector and the GC-based approximation. The first five configurations of Resurrector all have better performance than the GC-based approximation. However, when the length of a cache list goes unbounded, the overhead increases dramatically.

Time and Space Overhead on DaCapo-large To have a better understanding of Resurrector's scalability, we have also conducted an experiment on the large workloads of DaCapo. The results are summarized in Figure 5, Figure 6, and Figure 7, which show, respectively, the overall running time overheads, space overheads, and GC overheads for the GC-based approximation and the first five Resurrector configurations. Note that Elephant Tracks and Resurrector with $ml = \infty$ are not included in this experiment due to the unreasonably long executions (e.g., more than five hours for most applications); they are not practical enough to be used to profile real-world long-running applications. Resurrector encounters out-of-virtual-space errors[2] when running `chart`, `lusearch`, and `xalan` with $ml = 500$; thus, the results for these three applications under $ml = 500$ are missing in the figures.

In each of the three figures, bars represent overheads in times. The space overhead is measured as the ratio between the peak heap consumption of the Resurrector execution and that of the regular execution, while the GC overhead is measured as the ratio between the total GC times of the two executions. It is clear to see from Figure 5 and Figure 7 that the GC-based approximation has the largest overall time overhead and GC time overhead. This is primarily because it needs to traverse all created objects during each GC to identify dead objects and report death events. Table 2 reports the geometric means of the time, space, and GC overheads for these different configurations. Clearly, Resurrector is more efficient than the GC-based approximation; in addition, the higher the max-length parameter is, the more costly the execution is. When we compare the performance of the five configurations, we find that, in many cases, there is no significant performance difference between them. This may be because most allocation sites in large applications have very small numbers of lifetime-overlapping objects (as demonstrated shortly in this section), and thus, increasing ml does not significantly affect the overall running time of an application.

[2] This may be because Jikes RVM cannot handle a virtual address space larger than 1.1 GB.

ml	*Time OH (times)*	*Space OH*	*GC OH*
GC-based	17.0	2.1	7.2
1	4.9	1.7	2.0
10	5.1	2.6	2.9
100	5.4	3.7	3.3
200	5.7	3.3	3.2
500*	6.5	3.9	3.4

Table 2. The geometric means of the time, space, and GC overheads for the GC-based approximation and five Resurrector configurations. * The overheads under $ml = 500$ are computed on seven applications (all except `chart`, `lusearch`, and `xalan`).

5.2 Resurrector Precision

Deallocation Difference Ratio (DDR) We first define a precision metric, called *deallocation difference ratio (DDR)*, that we have used to assess the precision of various OLP techniques. Because Merlin is the most precise OLP technique, the goal of this experiment is to understand the pre-

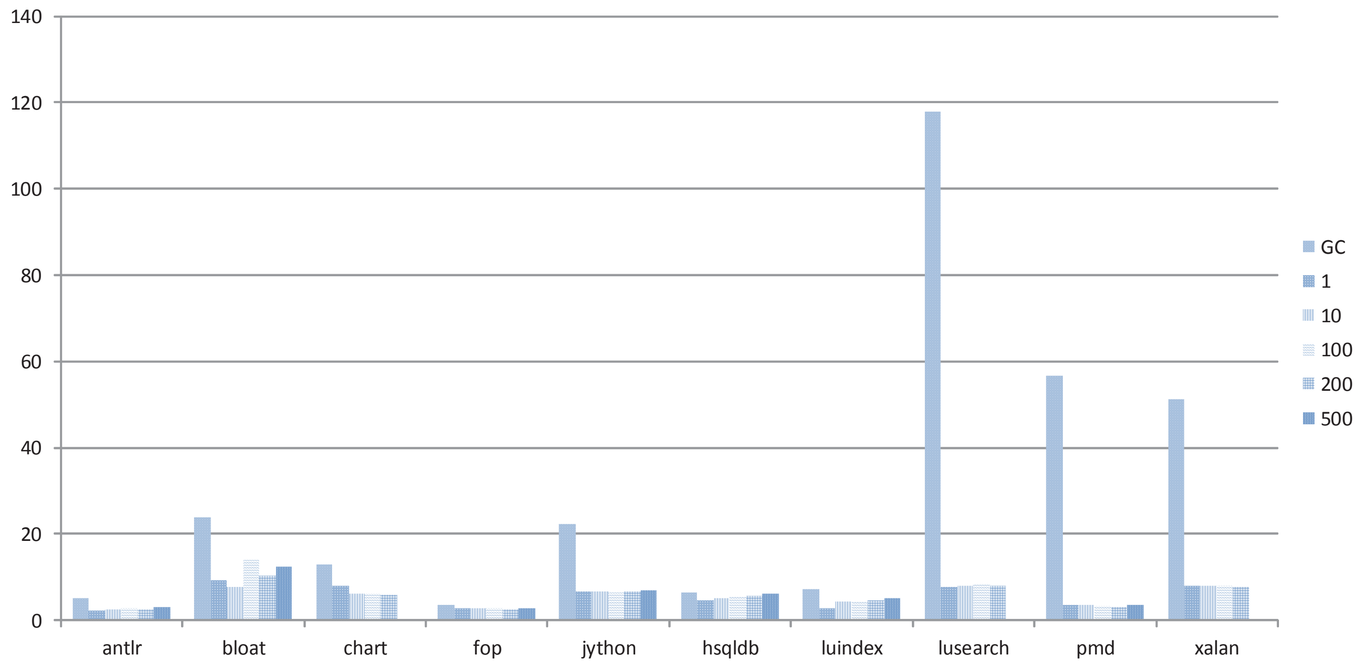


Figure 5. Resurrector overall running time overhead on DaCapo-large; the Y-axis shows overheads in times.

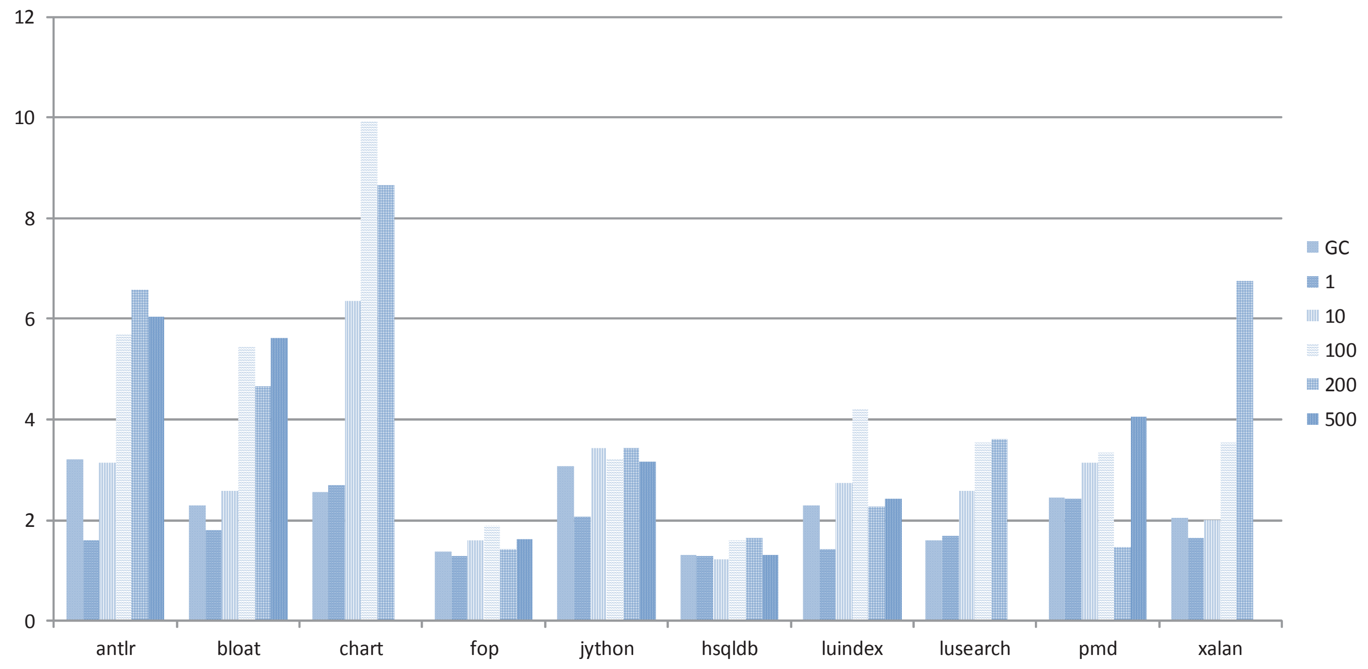


Figure 6. Resurrector space overhead on DaCapo-large.

cision gap between Resurrector and Merlin. However, Elephant Tracks, the only publicly available implementation of Merlin, does not support Jikes RVM, and thus, lifetimes collected from Elephant Tracks and from Resurrector are not directly comparable—the standard Java libraries used by HotSpot and by Jikes RVM are completely different, making a direct comparison meaningless. To mitigate the problem, we use the lifetime information collected from Resurrector's

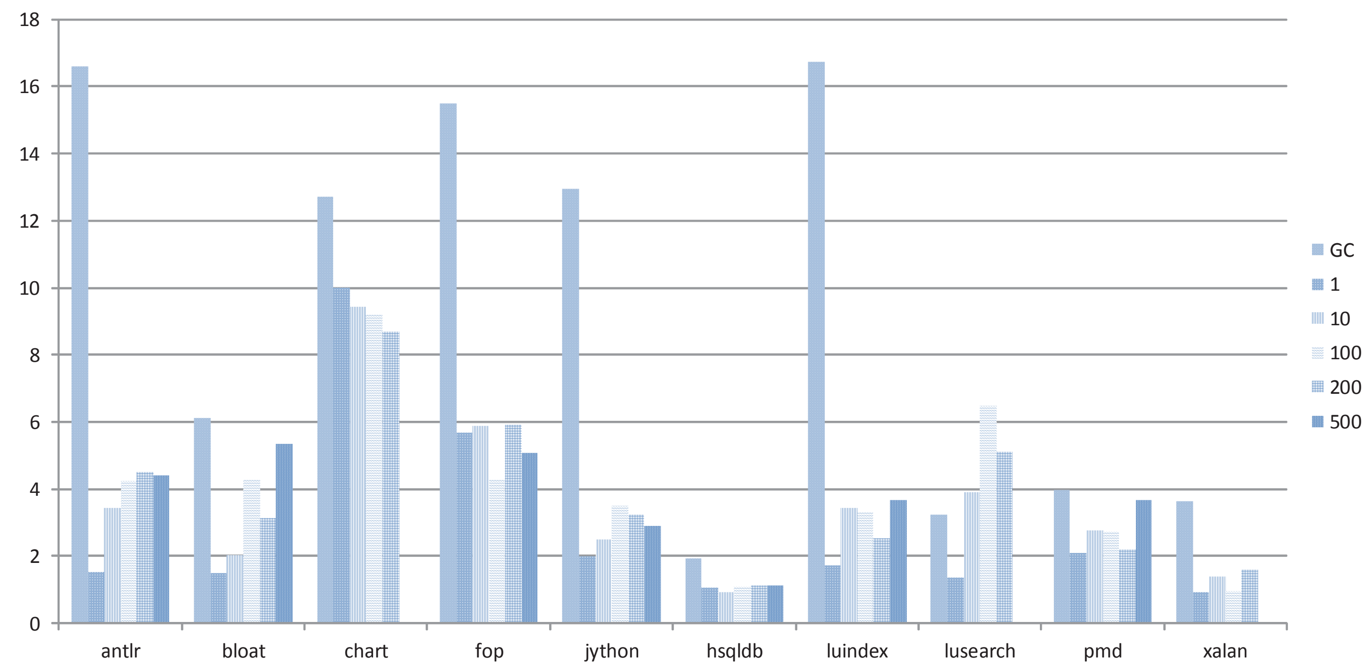


Figure 7. Resurrector GC overhead on DaCapo-large.

Bench	*DDR*$_{GC}$	*DDR*$_1$	*DDR*$_{10}$	*DDR*$_{100}$	*DDR*$_{200}$	*DDR*$_{500}$	*#Itv*
antlr	90.0	74.4	72.1	46.9	28	20.9	16
bloat	99.2	54.0	43.3	24.2	21.6	14.6	49
chart	98.5	69.9	58.0	27.7	27.9	28.2	54
fop	94.6	44.4	24.8	27.4	12.3	5.5	6
hsqldb	124.7	76.0	59.2	53.1	54.9	8.6	16
jython	96.3	72.1	70.0	60.0	56.1	43.6	62
luindex	100	93.8	93.6	93.5	93.1	92.7	17
lusearch	98.4	56.2	31.4	26.6	26.7	21.0	400
pmd	80.1	90.2	70.7	84.3	57.9	14.7	3
xalan	102.9	76.0	57.2	55.4	48.9	58.9	134
GeoMean	97.9	69.1	54.3	44.7	36.8	22.3	

Table 3. Resurrector's precision measurements; column *#Itv* reports the number of 1MB allocation intervals during the execution of each application; other columns show the deallocation difference ratios (DDRs) collected for the GC-based approximation and the five Resurrector configurations.

$ml = \infty$ configuration as an approximation of Merlin's lifetime information. Although the former is still less precise than the latter, their precision should be very close.

We run each application under its small workload (in order to obtain results for $ml = \infty$) and divide an execution into a sequence of 1MB allocation intervals. We collect the number of objects that are reported as dead in each interval for the GC-based approach and each Resurrector configuration. For a configuration $ml = c$, suppose s_c is an array such that each element $s_c[i]$ contains the number of dead objects reported in interval i; s_∞ is the corresponding array for configuration $ml = \infty$. The precision measurement *DDR*$_c$ for configuration c is defined as

$$DDR_c = \Sigma_{i\in[0,|s_\infty|)}|s_c[i] - s_\infty[i]| * 100/\Sigma_{i\in[0,|s_\infty|)}s_\infty[i]$$

In other words, *DDR*$_c$ is used to measure the overall difference in the numbers of dead objects reported between configuration c and configuration ∞. The smaller *DDR*$_c$ is, the more precise lifetime information configuration c produces. Note that *DDR*$_c$ may not necessarily be smaller than 100. We often see DDRs greater than 100 if the differences between the two reports are very big. Table 3 reports the DDRs for the GC-based approximation and the five configurations of Resurrector. In general, Resurrector is more precise than the GC-based approach even under $ml = 1$, and the greater ml is, the smaller its *DDR* is. For `luindex`, all the five Resurrector configurations cannot produce precise lifetime information. This is primarily because `luindex` is a text indexing tool. During execution, it keeps adding books into its (in-memory) dictionary and then does text indexing, which causes most of its allocation sites to have large numbers of lifetime-overlapping objects. It appears that even ml =500 is not big enough for most of the cache lists to hold objects until they can be resurrected.

Unitary Allocation Site Statistics Unitary allocation sites are those such that the lifetimes of the objects they create are completely disjoint [22, 50]. Detecting such allocation sites is important for many optimization tasks because objects created by them can be pre-allocated or easily reused. To demonstrate Resurrector's ability of finding unitary allocation sites, we compare the numbers of unitary allocation sites reported by the GC-based approach and Resurrector under configuration $ml = 1$. The results are shown in Figure 8. Resurrector has identified that an overall 72.7% of the total allocation sites executed are unitary while the corresponding percentage reported by the GC-based approach is only 12.0%.

We also use the Resurrector lifetime information to generate a histogram of allocation sites (shown in Figure 9), which gives an overview of the optimization opportunities that may exist in each program. The histogram is obtained from the configuration $ml = 100$. On average, the percentages of the allocation sites that fall into categories $ml = 1$, $ml \leq 5$, and $ml > 5$ are 72.6%, 8.8%, and 11.1%, respectively. The rest (9.5%) of the allocation sites have more than 100 lifetime-overlapping objects, and thus, their cache lists are discarded by Resurrector. These numbers clearly indicate that large opportunities exist in real-world applications; Resurrector can help either human experts or automated tools quickly find these opportunities.

5.3 Case Studies

We have inspected the reports generated by the two client analyses discussed in Section 4 for 4 DaCapo applications. In fact, we have looked at only a few top allocation sites whose *max-length* is 1 in each report, and modified the source code to cache and reuse their created objects. We add a *reset* method in some classes—when their objects are reused, *reset* (rather than the constructor) is called to reset their data content without reconstructing them. It takes us about 2 days to find the problems and implement the fixes for these applications. More opportunities could have been found if we had had more time. Performance statistics before and after the problem fixes are collected under a fast (FastAdaptiveImmix) configuration of Jikes RVM (where all compiler optimizations are enabled) with a 1G maximal heap size. Hence, the performance improvements achieved are beyond the JIT's best effort. In order to avoid the compilation cost and the execution noise, each application has been run 5 times and the median of the running times is reported.

`The DaCapo pmd benchmark` The first (most frequently executed) allocation site in our report is located in line 73 of class *org.jaxen.expr. IdentitySet*. This class implements an IdentitySet (in which two objects a and b are considered equal only if $a == b$) by maintaining an internal Java HashSet and wrapping each element object into a wrapper object. Two wrapper objects are equal when their wrapped objects are the same. Adding wrapper objects into the HashSet prevents two different element objects from being recognized as equal even if calling their "equals" method returns true. This first allocation site creates a wrapper object in method *contains*, which is used only to check whether a given object exits in the set. Because this set is often very large, a great number of wrapper objects have to be created and garbage collected. In fact, only this one allocation site (in one of the many methods in this class) has created a total of 31,039,097 objects. We fix the problem by simply employing the JDK-provided IdentityHashMap, which uses identity equality when adding/retrieving elements. Fixing only this one problem has led to a 5.4% total time reduction (from 11189 to 10589 msec), a 19.6% reduction on the number of objects (from 778744363 to 626260549), and a 6.7% space reduction (from 153136 to 144412KB). The number of GCs has been reduced from 90 to 82.

`The DaCapo xalan benchmark` We implement singleton patterns for a few top allocation sites in xalan's reports to reuse *NumberFormatStringTokenizer*, *XPathParser*, and *Compiler* objects. These objects are *data processors* as apposed to data to be processed; it is thus unnecessary to create them every time a new data item arrives. After these fixes, the program's running time has been reduced by 8.7% (from 13073 to 11933 msec). The peak memory consumption and the number of objects have been reduced by 15.4% (from 244756 to 207064KB) and by 5.5% (from 317406559 to 299943157), respectively.

`The DaCapo luindex benchmark` The first allocation site reported is in line 165 of class *org.apache.lucene.index. SegmentTermEnum*. This allocation site takes a *TermInfo* object as input and clones a new one. Resurrector indicates that this allocation site creates a great number of objects whose lifetimes are all disjoint, we simply make all cloned objects share one single instance throughout the execution. Note that the method containing the allocation site is invoked in a variety of contexts, and it would be very difficult for a developer to obtain this information without Resurrector's precise object lifetime profiles. This fix has led to a 9.9% space reduction (from 50428 to 45888KB) and a 3.9% object reduction (from 84780111 to 81487862). The number of GC runs has been reduced from 78 to 72. No clear time reduction has been found.

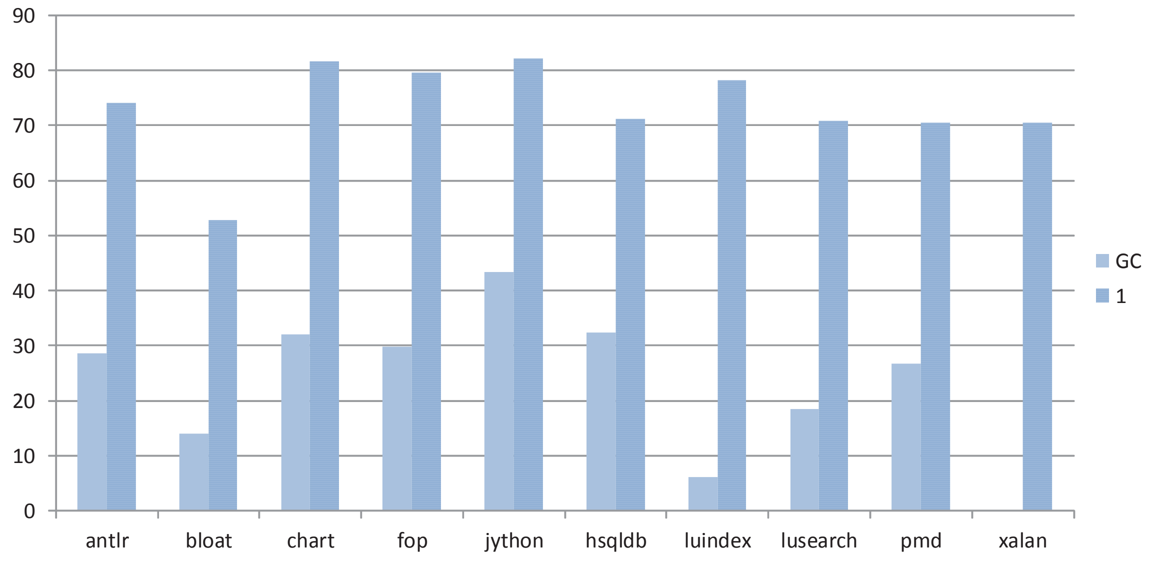


Figure 8. A comparison between the percentages of allocation sites reported as unitary by the GC-based approach and by Resurrector under $ml = 1$.

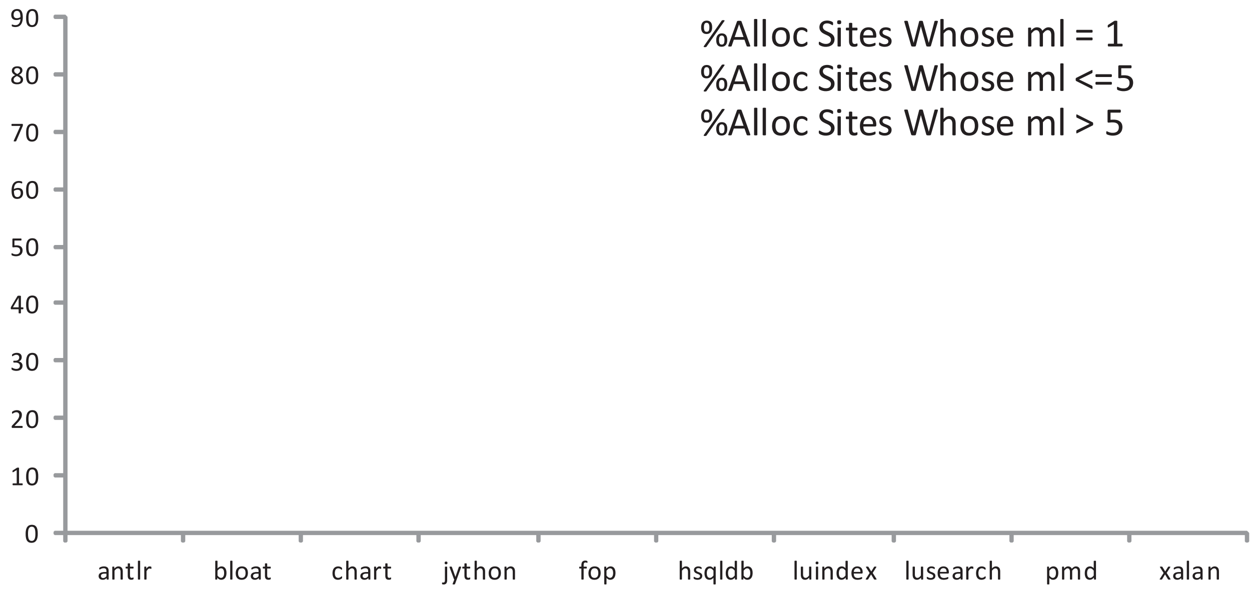


Figure 9. A histogram of allocation sites under configuration $ml = 100$. Each bar represents the percentage of allocation sites whose *max-length* falls into a category.

`The DaCapo bloat benchmark` Among the top 20 reported allocation sites, 12 create *String* and *StringBuffer* objects in the *toString* method of various classes, such as Expression, Label, Block, etc. Previous work [53] has already found that this method is invoked only to provide messages for assertions (for debugging purposes). The messages are generated regardless of whether these assertions succeed or fail. The rest of the report are allocation sites that create objects to implement visitor patterns (e.g., to traverse various graphs). We modify the source code to implement two fixes. First, *toString* is called to generate a message only when an assertion fails. Second, singleton patterns are employed to create graph visitors. These fixes have led to a 5 $\times$ (from 45099 to 8372 msec) running time reduction, a 3.9 $\times$ (from

1016816 to 257716KB) reduction on the peak memory consumption, and a 4.8 × (from 929345410 to 192934784) object reduction. This is by far the largest performance improvement that has been achieved by fixing performance problems in bloat.

Summary and Discussion The amount of effort we have spent on the report inspection and problem fixes is relatively small—the performance gains are obtained from fixing only a few allocation sites, work that just scratches the surface. During the inspection of reports and development of fixes, we have classified three categories of allocation sites that often create disjoint objects. These objects can be easily reused to improve performance. The first category of allocation sites creates *event* objects that are used only to notify listeners certain events have occurred. For example, the No.1 allocation site in the report of chart creates *SeriesChangeEvent* objects, and their lifetimes never overlap. In an event-driven system, there is often a great number of event objects and reusing them can sometimes significantly reduce the allocation and garbage collection effort. The second category contains allocation sites that create *processor* objects (as apposed to data objects to be processed). The allocation sites that create tokenizer and parser objects in xalan as well as those that create visitor objects in bloat are examples of this category. In many cases, one single processor object is sufficient to process a number of different data objects. In addition, a processor object often holds a complicated data structure to do the processing, and thus, creating and constructing the data structure many times is undesirable and can cause significant performance problems. The third category of allocation sites create *StringBuffer* objects. In Java, at each occurrence of string concatenation (e.g., using operator "+"), the compiler automatically translates it into the use of *StringBuffer* (i.e., creates StringBuffer objects and calls its *append* method). *StringBuffer* objects created at different string concatenations can never be simultaneously used. In fact, the future design of the Java compiler should consider to employ singleton patterns when it compiles string concatenations.

5.4 Improving an Existing Technique

Our previous work [50] develops a dynamic analysis to help programmers detect reusable data structures based on their allocation sites. [50] gives a three-level reusability definition: if an allocation site creates all disjoint objects, the object instances can be reused; among the allocation sites whose instances are reusable, the second reusability level concerns whether the shapes of the created data structures are always the same; the highest reusability level is data reusability that corresponds to allocation sites whose instances are disjoint, and whose shapes and data contents are the same. Key to the success of this work is the precise identification of allocation sites whose instances are reusable, which provides a basis for the other two (higher-level) reusability detectors—it is impossible to reuse data structures that have the same data contents unless their lifetimes are disjoint. [50] approximates instance reusability by computing, for each allocation site, a ratio between the number of its dead objects and the number of its live objects at each GC (i.e., DL ratio). The higher this ratio is, the more likely it is that the lifetimes of its objects are disjoint. This approximation is rather imprecise and is the cause of all false positives (reported in Section 4.1 of [50]).

bloat	3	pmd	12	luindex	8	xalan	4

Table 4. False positives of the object lifetime approximation used in [50].

We run the tool on the 4 benchmarks we have studied. For each benchmark, we carefully inspect the top 20 reported allocation sites, and Table 4 shows the number of false positives we have found. An allocation site is considered as a false positive if either it is clearly not a problem or we could not develop a solution to reuse its objects. For example, luindex creates a great number of Token objects during the parsing of expressions. These objects are linked through their next field and any regular operation of the list can break a link and make many such objects become unreachable. The allocation sites creating them often have big DL ratios while their objects are not truly reusable. We modify the tool to force it to use the Resurrector object lifetime information. This has led to the elimination of all false positives. All of the 20 allocation sites in each of the new reports are singleton allocation sites, which can be easily understood and optimized by the developer.

6. Related Work

While there exists a large body of related work in dynamic analysis and memory management, this section focuses on work that is closest to Resurrector.

Reference Counting Reference counting (RC) is used widely in the design of GC algorithms. A RC collector keeps track of the number of incoming references for each object; when its reference count becomes 0, the object can be collected [4–6, 17]. Because it is expensive to track stack and register updates, modern RC collectors introduce deferred reference counting [11, 18] that computes reference counts periodically. When a RC collector collects the heap, it must enumerate the stacks and registers. Different from all RC collectors, Resurrector takes an object-centric design, which identifies dead objects based on the method invocation information recorded in each object and, hence, Resurrector does not rely on a reachability analysis to compute object liveness.

As with other RC algorithms, Resurrector cannot find dead cycles [47]. Modern GC collectors add periodic tracing collection or perform trial deletion [4, 6, 46] to collect dead cycles. Trial deletion maintains a candidate set containing objects that lost a pointer, but whose count did not reach zero. The algorithm then iteratively performs trial deletions on the objects in this set and those reachable from them. When all the reference counts become zero, the objects form a dead cycle and can be reclaimed. The current implementation of Resurrector leaves the detection of dead cycles to GC, while it is worth investigating in the future work how to incorporate trial deletion into Resurrector to detect and reuse dead objects in cycles.

Efficient implementation of a non-deferred RC algorithm in a modern object-oriented language such as C# has been extensively studied in recent work such as [28–30]. This work relies on a dataflow analysis performed in the optimizing compiler to insert updates to reference counts, so that redundant RC updates can be eliminated and dead objects can quickly reclaimed. Unlike this line of work that uses static analysis to track stack references, Resurrector exploits a novel counting algorithm, which treats the method return as the point at which an object loses its stack references, trading off the immediacy of detecting dead objects for efficiency.

GC-Related Techniques Pretenuring long-lived and immortal objects [10, 12, 14, 33] into infrequently or never collected regions reduces garbage collection costs significantly. Other techniques such as [31, 45, 49] propose to use object lifetime profiles to improve GC performance in various aspects.

Optimization of Runtime Bloat Software bloat analysis [3, 36–39, 44, 50–58] attempts to find, remove, and prevent performance problems due to inefficiencies in the code execution and the use of memory. Prior work [38, 39] proposes metrics to provide performance assessment of use of data structures. Their observation that a large portion of the heap is not used to store data is also confirmed in our study. In addition to measure memory usage, our work proposes optimizations specifically targeting the problems we found and our experimental results show that these optimizations are very effective.

Work by Dufour *et al.* [20] uses a blended escape analysis to characterize and find excessive use of temporary data structures. By approximating object lifetimes, the analysis has been shown to be useful in classifying the usage of newly created objects in the problematic areas. Shankar *et al.* propose Jolt [44], an approach that makes aggressive method inlining decisions based on the identification of regions that make extensive use of temporary objects. Work by Xu *et al.* [53] detects memory bloat by profiling copy activities, and their later work [52] looks for high-cost-low-benefit data structures to detect execution bloat.

Escape analysis [13, 16, 21, 48] detects objects whose lifetimes are within the lifetime of the method (or its caller) that create the objects. Such objects can be allocated on the stack instead of on the heap. Gheorghioiu *et al.* propose a static analysis [22] to identify *unitary* allocation sites whose instances are completely disjoint so that these instances can be pre-allocated and reused. Free-me [25] is a static technique that identifies when objects become unreachable and inserts calls to free them. Recent work such as [7, 58] uses static analysis to identify reusable data structures created in a loop. Object equality profiling (OEP) [32] is a technique that discovers opportunities for replacing a set of equivalent object instances with a single representative object to save space. Hash consing [2, 24] and memoization [8, 34] are major techniques for reusing existing objects. They have been widely used in functional languages, such as LISP and ML. JOLT [44] is a dynamic technique that makes aggressive method inlining decisions based on the identification of regions that contain the entire lifetimes of many temporary objects (i.e., they are created and captured). Work by Dufour et al. [20] uses a blended escape analysis to characterize and find excessive use of temporary data structures. Their major motivation of designing a "blended" analysis is that a pure dynamic analysis that can detect lifetimes of short-lived temporary objects is too expensive. Resurrector solves this scalability problem by caching and reusing dead objects. Note that all these dynamic optimization techniques require precise identification of object lifetimes, and thus, all of them may benefit from the information profiled by Resurrector.

7. Conclusions

This paper presents a tunable object lifetime profiling technique, called Resurrector, that attempts to find the sweetspot between precision and scalability by detecting and reusing dead objects during the mutator execution. It is much more precise than a GC-based approximation and much more efficient than Merlin. Using the number of objects cached for each allocation site as a parameter, Resurrector defines a tradeoff framework that allows the designer of a client analysis to tune the precision of the lifetime information and the efficiency of computing it specifically for the requirement of the analysis. We have implemented Resurrector in JikesRVM and successfully run it on a set of large-scale, real-world applications. Our experimental results demonstrate that the overhead incurred by Resurrector is reasonable and large optimization opportunities can be found using the Resurrector-collected object lifetime information.

Acknowledgments

We would like to thank Brian Demsky for his helpful comments on an early draft of the paper. We also thank the anonymous reviewers for their valuable and thorough comments. This material is based upon work supported by the National Science Foundation under grant CNS-1321179.

References

[1] Websphere application server development best practices for performance and scalability. http://www-3.ibm.com/software/webservers/appserv/ws_bestpractices.pdf.

[2] J. R. Allen. *Anatomy of LISP*. McGraw-Hill, 1978.

[3] E. Altman, M. Arnold, S. Fink, and N. Mitchell. Performance analysis of idle programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 739–753, 2010.

[4] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: a nonintrusive multiprocessor garbage collector. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 92–103, 2001.

[5] D. F. Bacon, P. Cheng, and V. T. Rajan. A unified theory of garbage collection. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 50–68, 2004.

[6] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 207–235, 2001.

[7] S. Bhattacharya, M. Nanda, K. Gopinath, and M. Gupta. Reuse, recycle to de-bloat software. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 408–432, 2011.

[8] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, pages 5–42, 1987.

[9] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 169–190, 2006.

[10] S. M. Blackburn, M. Hertz, K. S. Mckinley, J. E. B. Moss, and T. Yang. Profile-based pretenuring. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

[11] S. M. Blackburn and K. S. McKinley. Ulterior reference counting: fast garbage collection without a long wait. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 344–358, 2003.

[12] S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinely, and J. E. B. Moss. Pretenuring for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 342–352, 2001.

[13] B. Blanchet. Escape analysis for object-oriented languages. Applications to Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 20–34, 1999.

[14] P. Cheng, R. Harper, and P. Lee. Generational stack collection and profile-driven pretenuring. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 162–173, 1998.

[15] A. E. Chis, N. Mitchell, E. Schonberg, G. Sevitsky, P. O'Sullivan, T. Parsons, and J. Murphy. Patterns of memory inefficiency. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 383–407, 2011.

[16] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–19, 1999.

[17] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, Dec. 1960.

[18] L. P. Deutsch and D. G. Bobrow. An efficient, incremental, automatic garbage collector. *Commun. ACM*, 19(9):522–526, Sept. 1976.

[19] T. Domani, G. Goldshtein, E. K. Kolodner, E. Lewis, E. Petrank, and D. Sheinwald. Thread-local heaps for Java. In *International Symposium on Memory Management (ISMM)*, pages 76–87, 2002.

[20] B. Dufour, B. G. Ryder, and G. Sevitsky. A scalable technique for characterizing the usage of temporaries in framework-intensive Java applications. In *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*, pages 59–70, 2008.

[21] D. Gay and B. Steensgaard. Fast escape analysis and stack allocation for object-based programs. In *International Conference on Compiler Construction (CC)*, LNCS 1781, pages 82–93, 2000.

[22] O. Gheorghioiu, A. Salcianu, and M. Rinard. Interprocedural compatibility analysis for static object preallocation. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 273–284, 2003.

[23] J. Gil and Y. Shimron. Smaller footprint for Java collections. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 356–382, 2012.

[24] E. Goto. Monocopy and associative algorithms in an extended lisp. Technical Report 74-03, Information Science Laboratory, University of Tokyo, 1974.

[25] S. Z. Guyer, K. S. McKinley, and D. Frampton. Free-Me: a static analysis for automatic individual object reclamation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 364–375, 2006.

[26] M. Hertz, S. M. Blackburn, J. E. B. Moss, K. S. McKinley, and D. Stefanović. Generating object lifetime traces with Merlin. *ACM Transactions on Programming Languages and Systems*, 28(3):476–516, 2006.

[27] *Jikes Research Virtual Machine*. `http://jikesrvm.org`.

[28] P. G. Joisha. Compiler optimizations for nondeferred reference: counting garbage collection. In *International Symposium on Memory Management (ISMM)*, pages 150–161, 2006.

[29] P. G. Joisha. Overlooking roots: a framework for making nondeferred reference-counting garbage collection fast. In *International Symposium on Memory Management (ISMM)*, pages 141–158, 2007.

[30] P. G. Joisha. A principled approach to nondeferred reference-counting garbage collection. In *VEE*, pages 131–140, 2008.

[31] R. Jones and C. Ryder. Garbage collection should be lifetime aware. In *International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS)*, 2006.

[32] D. Marinov and R. O'Callahan. Object equality profiling. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 313–325, 2003.

[33] S. Marion, R. Jones, and C. Ryder. Decrypting the Java gene pool. In *International Symposium on Memory Management (ISMM)*, pages 67–78, 2007.

[34] D. Michie. "memo" functions and machine learning. *Nature*, page 218, 1968.

[35] N. Mitchell. The runtime structure of object ownership. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 74–98, 2006.

[36] N. Mitchell, E. Schonberg, and G. Sevitsky. Making sense of large heaps. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 77–97, 2009.

[37] N. Mitchell, E. Schonberg, and G. Sevitsky. Four trends leading to Java runtime bloat. *IEEE Software*, 27(1):56–63, 2010.

[38] N. Mitchell and G. Sevitsky. The causes of bloat, the limits of health. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 245–260, 2007.

[39] N. Mitchell, G. Sevitsky, and H. Srinivasan. Modeling runtime behavior in framework-based applications. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 429–451, 2006.

[40] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: generating program traces with object death records. In *International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 139–142, 2011.

[41] N. P. Ricci, S. Z. Guyer, and J. E. B. Moss. Elephant tracks: portable production of complete and precise GC traces. In *International Symposium on Memory Management (ISMM)*, pages 109–118, 2013.

[42] A. Sewe, D. Yuan, J. Sinschek, and M. Mezini. Headroom-based pretenuring: dynamically pretenuring objects that live "long enough". In *International Conference on Principles and Practice of Programming in Java (PPPJ)*, pages 29–38, 2011.

[43] O. Shacham, M. Vechev, and E. Yahav. Chameleon: Adaptive selection of collections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 408–418, 2009.

[44] A. Shankar, M. Arnold, and R. Bodik. JOLT: Lightweight dynamic analysis and removal of object churn. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 127–142, 2008.

[45] J. Singer, G. Brown, I. Watson, and J. Cavazos. Intelligent selection of application-specific garbage collectors. In *International Symposium on Memory Management (ISMM)*, pages 91–102, 2007.

[46] S. C. Vestal. *Garbage collection: an exercise in distributed, fault-tolerant programming*. PhD thesis, Seattle, WA, USA, 1987.

[47] J. Weizenbaum. Knotted list structures. *Communications of the ACM*, 5(3):161–165, Mar. 1962.

[48] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 187–206, 1999.

[49] F. Xian, W. Srisa-an, and H. Jiang. Allocation-phase aware thread scheduling policies to improve garbage collection performance. In *International Symposium on Memory Management (ISMM)*, pages 79–90, 2007.

[50] G. Xu. Finding reusable data structures. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1017–1034, 2012.

[51] G. Xu. CoCo: Sound and adaptive replacement of Java collections. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 1–26, 2013.

[52] G. Xu, M. Arnold, N. Mitchell, A. Rountev, E. Schonberg, and G. Sevitsky. Finding low-utility data structures. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 174–186, 2010.

[53] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 419–430, 2009.

[54] G. Xu, M. D. Bond, F. Qin, and A. Rountev. LeakChaser: Helping programmers narrow down causes of memory leaks. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 270–282, 2011.

[55] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky. Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications. In *FSE/SDP Working Conference on the Future of Software Engineering Research (FoSER)*, pages 421–426, 2010.

[56] G. Xu and A. Rountev. Precise memory leak detection for Java software using container profiling. In *International Conference on Software Engineering (ICSE)*, pages 151–160, 2008.

[57] G. Xu and A. Rountev. Detecting inefficiently-used containers to avoid bloat. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 160–173, 2010.

[58] G. Xu, D. Yan, and A. Rountev. Static detection of loop-invariant data structures. In *European Conference on Object-Oriented Programming (ECOOP)*, pages 738–763, 2012.