# Overlooking Roots: A Framework for Making Nondeferred Reference-Counting Garbage Collection Fast

Pramod G. Joisha
Microsoft Research
pjoisha@microsoft.com

## Abstract

Numerous optimizations exist for improving the performance of nondeferred reference-counting (RC) garbage collection. Their designs are ad hoc, intended to exploit different count removal opportunities. This paper shows that many of these optimizations can be unified using a notion called overlooking roots. The paper also shows how the notion enables more powerful versions of past optimizations and makes new optimizations possible.

While recent static analyses have dramatically improved nondeferred RC performance, margins relative to the deferred variant were still significant in the worst case. With the optimizations made possible by overlooking roots, we show that these margins can be reduced to within 4% on nearly all programs in a test suite, at even large heap sizes, and to within 23% in the worst case.

***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—Memory Management (Garbage Collection), Optimization, Compilers

***General Terms*** Algorithms, Languages, Performance

***Keywords*** Reference Counting, Static Analysis

## 1. Introduction

This paper demonstrates how a concept called overlooking roots unifies numerous past optimizations for the nondeferred reference-counting (RC) method. Nondeferred reference counting is the body of RC techniques that have three invariants: (1) all live data have positive reference counts; (2) the reference count is zero when the last reference disappears; and (3) a zero reference count implies dead data.[1] Classic reference counting [6, 4] belongs to this class. Deferred reference counting [5] does not, because a live object can have a zero reference count between collections.

Collectors in the nondeferred RC class offer unique advantages, such as immediate finalization and low memory footprint. They can present simpler designs—for example, at least for single-thread applications, components crucial to other collectors, like stack scanners and garbage collection (GC) maps, need not be present. But because implementations typically preserve the three invariants by counting all references to data, their throughputs can suffer.

It has long been believed that to get reference counting operating at an acceptable level of performance, one needs to use the deferred variant. Deferred reference counting provides the throughput boost by giving up on some of the ideal features of the nondeferred version (e.g., immediacy of reclamation). This work shows that such a compromise is not necessary in important cases.

Past efforts have addressed the throughput problem by a medley of optimizations, such as identifying permanently live data [1, 10], removing RC updates on lifetime-subsumed references [18, 20, 10], and statically coalescing RC updates [1, 10]. These optimizations eliminate RC updates, leaving the reclamation characteristics of the collection scheme intact. This paper's main contribution is the insight that these optimizations can be expressed as applications of the overlooking root information. This systematizes what was previously done in an ad hoc fashion.

The article also shows how overlooking roots enable more powerful versions of past optimizations, and how new optimizations can be devised using them. In particular, it shows a new and aggressive form of lifetime subsumption called ORCS. Relative to a recent form of lifetime subsumption proposed by us [10], it demonstrates that ORCS can improve throughput by up to a factor of 4.

Informally, a root $x$ *overlooks* a root $y$ if whatever is reachable from $y$ is also reachable from $x$. The paper presents an analysis for computing this information. The

[1] Note that the count can drop to zero even before the last reference disappears, for example, immediately after the last use of the last reference.

analysis is intraprocedural, and relies on simple mutation summaries to handle interprocedural calls. This makes it suitable to optimizing large programs in a piecemeal fashion. What makes the analysis novel is that it combines a standard data-flow approach with the weak updating of a central structure called a "tie function" to obtain reasonably good overlooking root information. If it were not for the tie function, the intraprocedural analysis would have been stymied by procedure invocations and other potentially heap-mutating statements.

While the analysis and optimizations account for multithread influences, our implementation is for single-thread programs. The design of an efficient nondeferred RC collector for multithread applications is a separate, open problem. As is, our results have utility beyond single-thread application domains. They could be useful in areas where reference counting is hand-implemented, such as in COM [19], device drivers, and microcontroller programs.

Our previous work reduced the throughput gap between the two forms of RC collection in the single-thread case. But the worst-case difference was still significant, and occurred on the `cmp` benchmark [10]. At the same heap size, optimized nondeferred RC collection can be up to four times slower on `cmp`. This paper shows that with ORCS, it can come within 4% of the deferred version. Furthermore, by utilizing just three of the five overlooking-root-based optimizations in this article (referred to as "OR optimizations"), we show that the nondeferred throughput is at worst within 23% of the deferred, even on large, cycle-intensive programs.

The 23% is on a compiler compiling itself, at heap sizes that are over 75% of the minimum required by the nondeferred collector to operate. At smaller sizes, the nondeferred collector can outperform the deferred on the same benchmark, by as much as 26%. On all other benchmarks, OR-optimized nondeferred RC collection is at worst within 4% of the deferred, even at large heap sizes.

The rest of this paper is organized as follows. Limitations of a recent form of lifetime subsumption are explained in Section 2. Section 3 presents the overlooking roots binary relation. It shows how the relation enables ORCS, and four other optimizations: (1) a coalescing of RC updates more general than done in the past; (2) the superseding of analyses for permanently live data; (3) omitting the buffering of roots in the trial deletion algorithm; and (4) the specialization of RC updates on non-null references. Section 4 describes an overlooking roots analysis. Measurements on OR-optimized nondeferred RC collection, and performance comparisons with deferred RC collection, are reported in Section 5. Section 6 discusses related work, and Section 7 concludes.

## 2. Lifetime Subsumption in the Past

We recently defined a local reference $y$ as being always *RC subsumed* by a local reference $x$, if

A1. every live range of $y$ is contained in a live range of $x$;

A2. neither $x$ nor $y$ can be redefined when $y$ is live; and

A3. the set $\mathscr{R}(y)$ of objects reachable from $y$ is always a subset of the set $\mathscr{R}(x)$ of objects reachable from $x$ [10].

The definition's aim was the efficient identification of lifetime-subsumed references, beyond those detected by past work [18, 20]. These references are valuable to nondeferred RC collectors, because they do not *have* to be counted.

This definition, which will be referred to as *enveloping RC subsumption* (ERCS) in this paper, led to an optimization that was shown to be effective on many programs [10]. Nonetheless, it covered only a limited set of scenarios. Moreover, the algorithm for finding ERCS references (roots that fulfill the ERCS definition) was a conservative one [10]. A conservative algorithm for an already conservative definition resulted in missed opportunities in the `cmp` outlier reported in our previous work [10]. Sections 2.1 and 2.2 elaborate on these matters further.

### 2.1 Conservatism in the ERCS Definition

A shortcoming of ERCS is that for a variable to be nontrivially subsumed, it must always be reachable from a *particular* variable different from itself. For example, consider the code fragment on the left in Figure 1. This consists of mutator code, and the RC updates that a classic RC collection scheme would require. (RC updates are the increment and decrement operations $RC_+(r)$ and $RC_-(r)$, where $r$ is a reference.) If $x$, $y$ and $z$ are defined for the first time on Lines 2, 4 and 6, and used last on Lines 14, 13 and 12 respectively (ignoring the RC update usages), then even if we assume that the fields $\mathtt{f}_1$ and $\mathtt{f}_2$ reside in thread-local objects, the object targeted by $z$ will not always be reachable from $x$ or $y$ alone. Thus, despite $z$ satisfying Provision A1, it is not an ERCS reference since Provision A3 does not hold relative to $x$ or $y$ alone. And yet, intuitively, $z$ should be subsumed since it will always be reachable from *either* $x$ or $y$.

Another example is the code fragment on the right in Figure 1. Neither Provisions A3 nor A1 hold for any of the variables in it. Hence, there would be no subsumption by ERCS, even though the RC updates for $w$ can be avoided if the coverage *jointly* provided by $u$ and $v$ were considered.

Another drawback is that of the two clauses in Provision A2, the second is constraining. The first is that $y$ should never be live through a redefinition of itself. It is needed to prevent a dangling reference problem (see Figure 2 in [10]). The second is that $y$ should never be live through a redefinition of $x$. It exists to eliminate the possibility of $y$'s target becoming unreachable from $x$ due to indirect writes of $x$ through pointers. It is unnecessary if Provision A3 can be computed more precisely.

### 2.2 Limitations of Previous Algorithm for ERCS

ERCS references were computed by finding local references that "overlook" the object targeted by a live local reference $y$, from just before a statement $s$ until their death or possible

```
1   RC+(e1)
2   x := e1
3   RC+(e2)
4   y := e2
5   RC+(x.f1)
6   z := x.f1
      ⋮
7   RC+(z)
8   RC-(y.f2)
9   y.f2 := z
10  RC-(x.f1)
11  x.f1 := null
12  ··· z ···
13  ··· y ···
14  ··· x ···
15  RC-(z)
16  RC-(y)
17  RC-(x)
```

```
1   RC+(e1)
2   u := e1
3   RC+(u.f1)
4   w := u.f1
5   RC+(e2)
6   v := e2
      ⋮
7   RC+(u)
8   RC-(v.f2)
9   v.f2 := u
10  RC-(u)
11  u := null
12  ··· w ···
13  RC-(w)
14  w := null
15  ··· v ···
16  RC-(v)
```

**Figure 1.** Two examples illustrating the deficiencies of ERCS. Both indicate a shortcoming in Provision A3. The example on the right also shows a deficiency in Provision A1.

redefinition. This *overlooking root set* was defined as

$$\mathbb{R}(s, y) = \{u \mid u \in R \wedge y \in live_{out}(s) \wedge y \overset{s_{out}}{\to} \omega \wedge \omega \in \mathscr{R}(u) \text{ on all paths from } s_{in} \text{ until } u \text{ dies or could be redefined}\}, \quad (1)$$

where $R$ is the set of local references, $live_{out}(s)$ is the set of local references that are live just after $s$, $y \overset{P}{\to} \omega$ means $y$ points to the object $\omega$ at program point $P$, and $s_{in}$ and $s_{out}$ are program points just before and just after $s$ [10].

Because a straight calculation of $\mathbb{R}(s, y)$ by Equation (1) is computationally expensive, the solution was to approximate it by a peephole examination of a small context around $s$. For example, the article showed that an approximate $\mathbb{R}(s, y)$ for the statement

$$y := x.\mathtt{f}$$

is $\{x\}$, if $x$ is known to only target thread-local objects, and if $x.\mathtt{f}$ is not written into before $x$ dies.

Although a peephole examination is sufficient for a number of important $s$, opportunities can be missed. For $y := x.\mathtt{f}$, since ascertaining whether $x.\mathtt{f}$ is written into before $x$ dies might require an inspection of all basic blocks reachable from the basic block $B$ in which $s$ occurs, the opportunity was conservatively identified by restricting the inspection to $B$. The approach was to *not* consider $y$ for subsumption, if $x$ did not die before the end of $B$.

Thus, subsumption previously was limited both in the opportunities it recognized and the flow-insensitive manner in which it was calculated. While this article gives a more aggressive definition of subsumption called ORCS, ERCS might still be an apposite choice in more than a few situations. For instance, the data structure complexity of the ERCS analysis in [10] is $O(|V|^2)$, where $|V|$ is the total number of root references. On the other hand, the data structure complexity of the ORCS analysis in this paper is $O(|V|^2|B|)$, where $|B|$ is the number of basic blocks in a procedure.

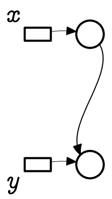


**Figure 2.** The root $x$ overlooks the root $y$. Boxes depict roots, and circles objects in the heap. Arcs represent points-to relations from roots to heap objects, as well as heap paths.

## 3. Overlooking Roots, and Their Applications

This section shows that a generalization of the formulation in Equation (1) leads to a concept that enables the conflation of numerous root-based RC optimizations. (The term "root" in this paper means a local or static reference.)

The generalized definition is as follows: A root $x$ overlooks a root $y$ at a program point $P$ if whatever is the object reachable from $y$ at $P$ is also reachable from $x$ at $P$ without going through $y$. This *overlooking roots* binary relation is irreflexive and transitive at any $P$. Any set of ordered pairs that fulfills the relation is denoted as $olook(P)$. When $(x, y) \in olook(P)$, we call $x$ the *overlooker* or the *overlooking root*, and $y$ the *overlookee* or the *overlooked root*. $(x, y)$ is an *overlooking pair*. Figure 2 displays the relation.

The new notion is related to Equation (1), because if $x \in \mathbb{R}(s, y)$ and if $x$ is not defined in $s$, then there exists an $olook(s_{out})$ such that $(x, y) \in olook(s_{out})$.

The following sections show how five different RC optimizations are expressible using the new notion.

### 3.1 Overlooking RC Subsumption

*Overlooking RC subsumption* (ORCS) is a kind of subsumption more general than ERCS. A live range $l$ of a local reference $y$ is considered subsumed according to ORCS, if

B1. $y$ is overlooked by live roots at every point in $l$;[2] and

B2. $y$ cannot be redefined in $l$.

$l$ is then referred to as an *ORCS live range*. Hence, the live ranges of $y$ need not be wholly contained in the live ranges of other variables for subsumption to occur. In addition, the object targeted by $y$ need not always be reachable from the same root. Thus, ORCS accommodates a more dynamic

[2] Because the overlooking roots relation is irreflexive, the object targeted by $y$ must always be reachable in $l$ from *another* live root.

view of reachability than ERCS. In fact, it is straightforward to show that ERCS is just a special case of ORCS.

Provision B2 exists for the same reason as the first clause in Provision A2—to prevent dangling references.

Although there is no subsumption in Figure 1 according to ERCS, $z$ and $w$ will be subsumed according to ORCS.

The *olook* sets contain path-insensitive information. If $(u, v) \in olook(P)$, then $u$ overlooks $v$ at $P$, irrespective of the path taken to reach $P$. This is stronger information than needed for the ORCS optimization. What is required is a set of live roots at each point, of which at least one is guaranteed to be an overlooker of the root in question. Section B explains an analysis for deriving this information using the overlooking roots analysis as a client service.

## 3.2 Superseding Custom Immortal Analyses

An object is *immortal* if it lasts, once created, until program termination. RC updates on these objects—examples of which include string literals and GC tables—are not needed as they live forever. Unlike for subsumption, the RC updates do not have to be "matched" for elimination; even removing an isolated RC update on an immortal object will not compromise program correctness, or risk a memory leak.

Past work presented a tailored data-flow analysis for finding sets of *immortal target variables* (local references to immortal objects) [10]. With overlooking roots, such a custom analysis becomes unnecessary.

### 3.2.1 Directly Overlooking Immortal Roots

Observe that a special virtual root can be thought to always target an immortal object. This *immortal root* is immutable, and "materializes" when the target immortal object is allocated (which might be at the very beginning of program execution, as is the case for statically allocated data).

Now consider the *directly overlooking roots* binary relation: A root $x$ *directly overlooks* a root $y$ at a point $P$ if whatever is the object targeted by $y$ at $P$ is also targeted by $x$ at $P$, and $x$ and $y$ are distinct. In most situations, this is simply an aliasing relation. The exception is when virtual roots target multiple objects at the same time (see Figure 11).

Let $dolook(P)$ be any set of ordered pairs that honor the directly overlooking roots relation. Then there is always an $olook(P)$ for a given $dolook(P)$ such that $dolook(P) \subseteq olook(P)$. Hence, if the direct overlookers among the overlookers of a root $r$ include an immortal root, then $r$ must be an immortal target variable. In this way, immortal analyses can be superseded by an analysis for overlooking roots.

### 3.2.2 Directly Overlooking Pristine Roots

Overlooking roots make it possible to go further—they permit the detection of immortal overlookers when roots are loaded off *pristine fields*. A field `f` is in the pristine state from the moment its containing object is allocated, up to the moment it is assigned a nonzero value. According to the allocation semantics of virtual execution environments like Java [14] and .NET [2], the value of `f` in the pristine state is assured to be an appropriately casted zero. Therefore, if a reference field in the pristine state is loaded into a root $y$, $y$ will be directly overlooked by an immortal root. An RC update against $y$ can then be omitted.

This definition can be slightly generalized. Rather than up to the moment at which it is assigned a nonzero value, a reference field can be considered pristine up to the moment at which it points to an object that is not immortal.

The pristine field mechanics can be captured in the framework of overlooking roots by introducing another set of virtual roots called the *pristine roots*. Consider the fragment in Figure 3, which corresponds to an initialization sequence. *allocobj*$(T)$ returns a reference to a newly allocated, unconstructed object of type $T$.[3] If $T$ has, say two fields $\mathtt{f}_1$ and $\mathtt{f}_2$, then after Line 1, $r$ can be regarded as being directly overlooked by two pristine roots, denoted by the pairs $(r, \mathtt{f}_1)$ and $(r, \mathtt{f}_2)$. This is shown on the right in Figure 3. (In our display convention, hatched boxes designate virtual roots.) Thus, Line 3 can be optimized out because the temporary that $r.\mathtt{f}_1$ is loaded into will be directly overlooked by an immortal root. When $r.\mathtt{f}_1$ is overwritten on Line 4, $(r, \mathtt{f}_1)$ should be removed from the set of $r$'s overlookers, as part of the "kill" calculation for that statement. Because $(r, \mathtt{f}_2)$ will remain a pristine overlooker of $r$, this allows Line 6 to also be optimized out.

```
1  $r := allocobj(T)$
2  $RC_+(x)$
3  $RC_-(r.\mathtt{f}_1)$
4  $r.\mathtt{f}_1 := x$
5  $RC_+(y)$
6  $RC_-(r.\mathtt{f}_2)$
7  $r.\mathtt{f}_2 := y$
```

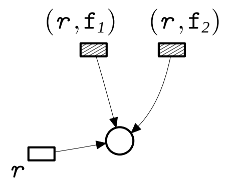


**Figure 3.** Code fragment illustrating pristine roots. After Line 1, $r$ is directly overlooked by the pristine roots $(r, \mathtt{f}_1)$ and $(r, \mathtt{f}_2)$; this is shown by the object graph on the right.

## 3.3 RC Chaining

Figure 4 displays two live ranges, one of $x$ and another of $y$. $x$ is defined at Points 1 and 2, and dies at Points 5, 6 and 7. $y$ is defined at Points 3 and 4, and dies at Points 8 and 9. The live ranges overlap in the shaded area of the figure. In unoptimized nondeferred reference counting, there would be increments against $x$ at Points 1 and 2, and decrements against it after Points 5, 6 and 7. Likewise, there would be increments against $y$ at Points 3 and 4, and decrements against it after Points 8 and 9. Now, if $x$ and $y$ target the same object in the overlapping portion of their live ranges, then the increment against $y$ at Point 3 can be coalesced with the decrement against $x$ at Point 7.

[3] Only the vtable and RC fields are set up in such an object.

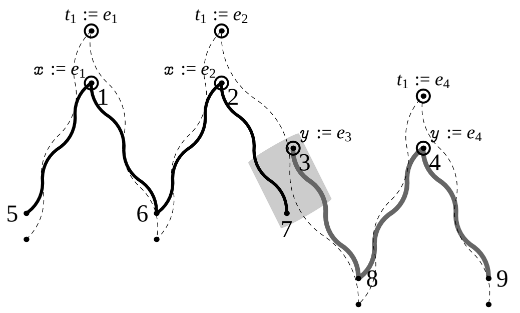


**Figure 4.** The live ranges of $x$ and $y$ overlap in the shaded area, in which they directly overlook each other. The RC chaining transformation creates the dashed live range for $t_1$.

In the past, compile-time RC update coalescing was restricted to within contiguous RC update sequences occurring within basic blocks [10]. This section shows that after a transformation called *RC chaining*, a more general coalescing effect can be achieved, simply by applying an ORCS optimization on the transformed code. Notice that, as is, the RC updates for Figure 4 cannot be optimized using ORCS.

Let $l_u$ be a live range of the root $u$, and $l_v$ a live range of the root $v$. We say that $l_u$ and $l_v$ are *RC chained*, if

C1. there are points at which $l_u$ and $l_v$ *interfere* (i.e., a definition point of $l_u$ lies within $l_v$, or vice versa [16]);

C2. $u$ and $v$ are not redefined in $l_u$ and $l_v$ respectively; and

C3. $u$ and $v$ directly overlook each other at the intersection points of $l_u$ and $l_v$.

An *RC chaining graph* $G_C = (V, E_C)$ is an undirected graph in which nodes stand for live ranges and edges for the RC chaining relationship. The graph can be set up by first computing the interference graph $G_I = (V, E_I)$ for the procedure [16]. If a variable could be redefined in its live range $l$, then edges in $G_I$ incident on $l$ are pruned out. The remaining edges represent live range pairs that satisfy Provisions C1 and C2. Let the resulting graph be $G'_I$.

Next, at each point $P$ in the program, the set

$$\begin{aligned}\Delta(P) = \{(l_u, l_v) \mid u \in \mathit{live}(P) \wedge v \in \mathit{live}(P) \\ \wedge\, ((u,v) \notin \mathit{dolook}(P) \\ \vee\, (v,u) \notin \mathit{dolook}(P))\}\end{aligned} \tag{2}$$

is calculated, where $\mathit{live}(P)$ is the set of live roots at $P$, and where $l_u$ and $l_v$ are the live ranges corresponding to $u$ and $v$ at $P$. If $(l_m, l_n)$ occurs in $\Delta(P)$, then that edge is deleted from $G'_I$. After all the program points are processed in this manner, it can be shown that the resulting graph is $G_C$.

Every connected component in the RC chaining graph represents a set of live ranges across which RC updates can be coalesced. To realize coalescence, the RC chaining transformation generates a temporary $t_c$ against each connected component $c$ in $G_C$. Assignments are made against $t_c$, and fake uses of it introduced, so that its live range tightly spans

1. Create a temporary $t_c$ for a connected component $c$ in $G_C$.
2. For every live range $l_u$ in $c$, find $D(l_u)$, the set of its definition points at which no other $l_v$ in $c$ is also active.
3. Precede every definition $u := e$ that corresponds to a definition point in $D(l_u)$ by the definition $t_c := e$.
4. Introduce a fake use of $t_c$ after every last use of $u$ in $l_u$.

**Figure 5.** The RC chaining transformation. It is based on $G_C$, the RC chaining graph for the procedure.

all the live ranges in $c$. The assignments are such that $t_c$ aliases at every point the variables corresponding to the live ranges in $c$ that are also active at that point.

If an ORCS optimization is applied on the transformed code, all the live ranges in $c$ will be subsumed by $t_c$. Only RC updates against $t_c$ will be retained. The net effect equals a coalescing of the RC updates at the overlap points in $c$.

Figure 5 gives an algorithmic description of the RC chaining transformation. When applied to the example in Figure 4, it generates the single temporary $t_1$, since its RC chaining graph has a single connected component. The assignments $t_1 := e_1$, $t_1 := e_2$ and $t_1 := e_4$ will be introduced before the existing definitions at Points 1, 2 and 4. Uses of $t_1$ will be introduced after Points 5 to 9. A definition of $t_1$ will not be introduced before Point 3 because $x$ is live there. The transformed result is a live range for $t_1$ that spans the two live ranges in the figure, and in which $t_1$ directly overlooks $x$ or $y$ or both. If an ORCS optimization is now applied, the RC updates against $x$ and $y$ will be eliminated. The optimized result is as if the increment at Point 3 was cancelled with the decrement at Point 7 in the pre-transformed code.

If the general overlooking roots relation is used instead in Provision C3, either the reclamation characteristics of the original collection scheme could be affected or dangling references could be created. For instance, suppose that $x$ overlooks $y$ but does not alias it in the overlap region of Figure 4. Then from Point 7 to Point 8, $t_1$ would hold on to the object targeted by a (dead) $x$.[4] If it was the converse, i.e., $y$ overlooks $x$ but does not alias it in the overlap region, then $y$ could become a dangling reference after Point 3.

The creation of a new spanning live range can sometimes be obviated by copy propagation. As an example, suppose that the live range of $y$ in Figure 4 only stretched from Point 3 to Point 8, and did not include the portion between Points 4, 8 and 9. Then a copy propagator might be able to extend the live range of $x$ from Point 7 to Point 8, hence rendering the creation of $t_1$ unnecessary. This may be beneficial since spanning live ranges like that of $t_1$ can increase the maximum number of mutually interferring live ranges, and therefore, can increase a procedure's register pressure.

[4] This poses an interesting question beyond the paper's scope: How to use overlooking roots to trade garbage drag with the degree of coalescing?

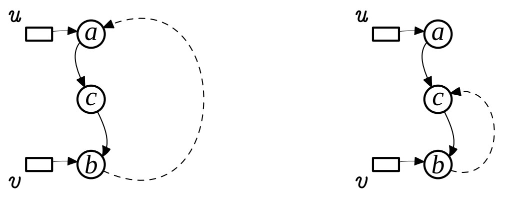


**Figure 6.** There may be a cycle-forming heap path from $b$ to $a$, or from $b$ to $c$. Regardless, if $v$ is overlooked by $u$ when it dies, it does not have to be put on the PLC list.

### 3.4 Optimizing the Buffering of Roots in Trial Deletion

Reference counting, by itself, cannot detect when a cyclic structure becomes unreachable. To get around this, a technique called *trial deletion* has been used, which avoids a full heap traversal to capture garbage cycles [15]. Trial deletion is based on the following observation: When a reference is swung away from an object whose reference count is at least 2, that object may become unreachable. If such references are stashed away in a "potentially leaked cycles" (PLC) list, they can be processed later for reclaiming leaked cycles.

Overlooking roots make the following optimization possible: If a root $v$ is overlooked by some other live root, say $u$, at the time $v$ is swung away from an object $b$, then $v$ does not have to be put on the PLC list. This is because $b$ will still be reachable from $u$ at that time. Thus, the decrement against $v$ can be as if it pointed to an acyclic object. Figure 6 illustrates this optimization.

### 3.5 Tracking Root States with Virtual Overlookers

Overlooking relations involving virtual roots are a way to track interesting states of a concrete root. As an example, consider the state of a concrete root that is non-null. This is useful information because an RC update on a non-null reference can be substituted by a specialized version that elides the initial null check [10].

If a concrete root $x$ is directly overlooked by a non-immortal root, then $x$ is naturally non-null.[5] But there may be cases where $x$ is non-null, even though it may not be overlooked by any of the roots discussed so far. An example is in the code fragment on the left in Figure 7. At the beginning of Line 2, $x$ will be non-null, due to the exception semantics of the statement on Line 1; this is independent of whether $x$ has a non-immortal overlooker before Line 1. Another example is in the code fragment on the right in Figure 7. $x$ will be non-null before Line 3, regardless of whether any non-immortal root directly overlooks it at that point.

It may therefore be profitable to track a concrete root's non-null state with a separate *non-null root*. A statement's "gen" calculation would have to suitably include it in the computed *olook* set. In both the examples of Figure 7, the calculation would add it to the overlookers of $x$ after Line 1.

[5] If it is overlooked by an immortal root instead, it *could* be null.

```
1   y := x.f         |   1   x := allocobj(T)
2   RC+(x)           |   2   ···x···
3   RC−(z)           |   3   RC−(x)
4   z := x           |   4   x := null
```

**Figure 7.** Two code fragments with specializable RC updates. In both, the non-null root overlooks $x$ after Line 1.

Virtual roots could also be used to track the aliasing of local and actual references. The rest of this paper assumes only four kinds of virtual roots—undefined, immortal, pristine, and actual parameter roots—to keep the exposition simple.

For the most part, virtual roots are not distinguished from concrete roots when producing and consuming overlooking root information. The calculation in Equation (11) is an example. But distinctions are sometimes needed. For instance, in ORCS and the trial deletion buffering optimizations, the live overlooker should either be immortal, actual or concrete.

## 4. Overlooking Roots Analysis

The analysis obtains $olook(P)$ at any point $P$ in a procedure. This is "must" information. For efficiency reasons, the analysis maintains the information per basic block, and computes it on demand per statement. At the cost of a constant amount of extra storage, the analysis also provides $dolook(P)$, the set of direct overlookers at $P$.

The analysis is intraprocedural. It handles invoked procedures using *reference mutation summaries*. A reference mutation summary for a procedure $F$ is the set of reference fields and reference array types that *could* be mutated by $F$.

The analysis's approach is to calculate $olook_{out}(s)$ at each statement $s$, using the *gen*, *kill* and $olook_{in}$ sets for $s$. $olook_{in}(s)$ and $olook_{out}(s)$ are the *olook* sets just before and just after $s$. There are three possibilities for $olook_{in}(s)$. If $s$ is preceded by a statement $s'$ in its basic block $B$, then $olook_{in}(s)$ equals $olook_{out}(s')$. Otherwise, $olook_{in}(s)$ equals the *meet* of the $olook_{out}$ sets for the last statements in the predecessor basic blocks of $B$. If $B$ has no predecessors, such as the entry basic block of the control-flow graph (CFG), $olook_{in}(s)$ equals a special initializing set called $olook_{\top}$.

The elements in $olook_{\top}$ are the overlooking root pairs on entry to a procedure. For every local reference $r$ that is not a formal reference, it contains the pair $(\top, r)$, where $\top$ denotes the "undefined" virtual root. This is because these references have initially undefined values. For every formal reference $z$, $olook_{\top}$ contains the pair $(\hat{z}, z)$, where $\hat{z}$ is a virtual root that models the actual parameter corresponding to $z$. There are no other pairs in $olook_{\top}$ because all the remaining roots, such as the static and virtual roots (including $\top$), do not have initially known overlookers.[6]

[6] There is no gain in including pairs of the form $(z, \hat{z})$. But there should not be pairs of the form $(r, \top)$ since $\top$ could target multiple objects.

## 4.1 The Problem of Mutations in the Heap

$olook_{out}(s)$ is given by a standard data-flow equation [16]:

$$olook_{out}(s) = (olook_{in}(s) - kill(s)) \cup gen(s). \tag{3}$$

The equation by itself is inadequate because $olook_{out}$ becomes vacuous in the face of called procedures that may mutate the heap. For example, consider the call statement

$$y := F(),$$

where $F$ is known to mutate a reference field `f`. Because `f` may lie on the heap path by which one root overlooks another, the *kill* set for the statement, in the absence of more knowledge, would have to be at least $olook_{in}(s)$. But then, all incoming information for the statement would be lost.

The problem outlined here is not specific to callees that may mutate the heap. It exists for any instruction that may mutate the heap. Its ramifications in the procedure invocation case, however, are extreme. In the case of a statement like

$$y.\mathtt{f} := x,$$

not all of the incoming information has to be killed, because at the very least, $y$ will overlook $x$ after the statement (assuming that `f` is a thread-safe or read-only field [7]).[7]

### 4.1.1 Tie Fields, Tie Array Types, and the Tie Function

The heap mutation problem can be solved using the concept of *tie fields* and *tie array types*. A reference field `f` *ties* a root $u$ to a root $v$ if there exists a point $P$ in the procedure at which `f` occurs on *every* heap path by which $u$ overlooks $v$ at $P$. Similarly, a reference array type $A$ is said to tie $u$ to $v$ if an instance of $A$ occurs on every heap path by which $u$ overlooks $v$ somewhere in the procedure. A *tie function* $\mathscr{T}$ can then be constructed such that given $u$ and $v$, $\mathscr{T}(u, v)$ is the set of all fields and array types that *may* tie $u$ to $v$.

Intuitively, tie fields and tie array types are links in the heap that when severed cause the overlooking relation between a pair of roots to be broken. Hence, if a field or an instance of an array type in $\mathscr{T}(u, v)$ is updated, then the overlooking relation between $u$ and $v$ *could* be broken.

There is an analogy between ties, which exist in the heap, and *dominator nodes* [16], which exist in a CFG. Just as a field or array type that ties the root $u$ to the root $v$ occurs on every heap path by which $u$ overlooks $v$, a node $x$ that dominates a node $y$ in the CFG occurs on every control-flow path from the entry node to $y$.

The overlooking roots analysis keeps one tie function per procedure. $\mathscr{T}$ is initialized to map every pair of roots to the empty set $\emptyset$. As new overlooking pairs are generated with every application of Equation (3), the tie function is updated to include new fields and array types. Once added to an image of $\mathscr{T}$, these fields and array types are never removed; this conforms to a *weak update* policy [3].

[7] Section A.4 shows other pairs that do not have to be killed in this case.

A disadvantage of weak updating is that it can rapidly dilute the usefulness of the gathered information [3]. The tie function could be updated for each pair in $gen(s)$, but that may needlessly dilute the tie information. In particular, if $(u, v) \in olook_{in}(s)$ and $(u, v) \notin kill(s)$, then there is no need to update $\mathscr{T}(u, v)$, because whatever ties $u$ to $v$ after $s$ already exists in $\mathscr{T}(u, v)$ before $s$. Hence, $\mathscr{T}$ is updated at $(u, v) \in gen^*(s)$ as

$$\mathscr{T}(u, v) \leftarrow \mathscr{T}(u, v) \cup \begin{cases} \{\mathtt{f}\} & \text{if } \mathtt{f} \text{ may tie } u \text{ to } v, \\ \{A\} & \text{if } A \text{ may tie } u \text{ to } v, \end{cases} \tag{4}$$

where

$$gen^*(s) = gen(s) - olook^*_{in}(s), \tag{5}$$

$$olook^*_{in}(s) = olook_{in}(s) - kill(s). \tag{6}$$

### 4.1.2 Utilizing the Tie Function

The main benefit of the tie function is that it enables the *kill* set to be more specialized. For instance, Section 4.2.1 shows that for procedure calls, the *kill* set need only be a subset of

$$\{(u, v) \mid \mathscr{T}(u, v) \neq \emptyset\}.$$

Another use is in determining the *dolook* set at any point in a procedure. This is shown in Equation (7), which gives the *dolook* set just before a statement $s$, after a fixedpoint is reached in the computation of the *olook* sets:

$$\begin{aligned} dolook_{in}(s) = \{(u, v) \mid (u, v) \in olook_{in}(s) \\ \wedge \mathscr{T}(u, v) = \emptyset\}. \end{aligned} \tag{7}$$

As an implementation aside, it helps to keep two maps for $\mathscr{T}$, to speed up tie function lookups. The first is for querying all the ties for a root pair, as in Equation (4). The second is for determining all the pairs tied by a field or type, as in Equation (22). Both must be updated with every logical update of $\mathscr{T}$.

## 4.2 Transfer Functions

The analysis iterates over all the statements in the CFG, applying Equations (3) and (4) in succession on relevant statements, and computing the meet of the $olook_{out}$ sets at confluence points. This is repeatedly performed until the *olook* sets reach a fixedpoint.

Statements are deemed "relevant" if they can alter the *olook* sets. Irrelevant statements, such as those that only side-effect arithmetic variables, propagate their $olook_{in}$ sets to their $olook_{out}$ sets.

In the interests of clarity, the analysis is described assuming all roots to be only references. This is in accordance with the Java programming model. In .NET, roots can also be interior pointers to objects [2]. It is straightforward to extend the description in this paper to statements involving them. In fact, our implementation for .NET does handle them.

Assuming $x$, $y$, $u$ and $v$ are local or static references, relevant statements can be divided into five categories:

- *simple assignments*—these are $y := x$ and $y := c$, where $c$ is a constant reference, like null or a string literal;
- *allocations*—this is $y := allocobj(T)$, where $T$ is the type of the allocated instance;
- *heap loads*—these are $y := x.\mathtt{f}$ and $y := x[e]$;
- *heap stores*—these are $y.\mathtt{f} := x$ and $y[e] := x$; and
- *procedure invocations*—this is $y := F(\cdots)$.

In all cases, the basic steps of the analysis are: (1) the statement's *kill* set is established, consulting at most its $olook_{in}$ set and $\mathscr{T}$; (2) its *gen* set is computed, at most utilizing its $olook^*_{in}$ set; and (3) $\mathscr{T}$ might be updated, at only pairs in $gen^*(s)$. The following explains how statements in the procedure-invocation category are analyzed. For the other categories, the reader is referred to the appendix.

### 4.2.1 Procedure Invocations

$\underline{y := F(\cdots)}$ This statement's analysis uses the reference mutation summary $\mu(F)$ for the callee $F$, if one is available. The summary is transitive—the summaries of the callees are included in the caller's summary. If no summary is available, as may happen under separate compilation, all tied pairs have to be killed. Otherwise, only those tied by the fields or array types in $\mu(F)$ have to be killed:

$$kill(s) = \begin{cases} \{(u,v) \mid u = y \vee v = y \\ \quad \vee \mathscr{T}(u,v) \neq \emptyset\} & \text{if unknown } \mu(F), \\ \{(u,v) \mid u = y \vee v = y \\ \quad \vee \mathscr{T}(u,v) \cap \mu(F) \neq \emptyset\} & \text{otherwise.} \end{cases} \tag{8}$$

The *gen* set is normally $\emptyset$. However, if the program call graph is available, as may happen under whole-program compilation, better *gen* information can be produced.

The *x-projection* on a root $v$ of a set $S$ of ordered pairs is the set of first elements in pairs of the form $(u,v)$ in $S$. This is expressed as $xproj(S,v)$. Consider a return point $Q$ in the function $F$, at which a local reference $r$ is returned. We call $xproj(olook(Q), r)$ the *return overlooker set* of $F$ at $Q$. Then the intersection of the return overlooker sets across all the return points of $F$ gives the set of roots that always overlook $F$'s returned value. Let this be $olook_{ret}(F)$.

Now consider a call statement that invokes $F$:

$$y := F(z_1, z_2, \ldots, z_n).$$

On return, $\mathfrak{I}$ overlooks $y$ if $\mathfrak{I} \in olook_{ret}(F)$. $y$ is also overlooked by $z_i$ ($1 \leq i \leq n$), if $\hat{z}_i \in olook_{ret}(F)$, where $\hat{z}_i$ is the actual parameter virtual root corresponding to $z_i$. This gives

$$gen(s) = \{(\mathfrak{I}, y) \mid \mathfrak{I} \in olook_{ret}(F)\} \\ \bigcup \{(z_i, y) \mid \hat{z}_i \in olook_{ret}(F)\}. \tag{9}$$

For Equation (9) to be efficacious, the analysis should be first performed on the callees of a procedure, before being performed on the procedure itself. This can be done by processing the procedures in a postorder traversal of the call graph. The $olook_{ret}$ sets for leaf procedures, and procedures at the ends of back edges in the call graph, can be set to $\emptyset$.

The extension makes it possible to derive valuable overlooking information across procedure boundaries, without resorting to a full interprocedural analysis. For instance, we can determine that the reference returned by the function

```
public IrType getType() { return this.type; }
```

is always overlooked by the actual parameter, without propagating information into the function through its argument.

## 5. Performance Measurements

We implemented three of the five OR optimizations discussed in this paper. They were ORCS, the immortal OR optimization (which elides RC updates on immortal objects) and the trial deletion OR optimization (which avoids putting roots on the PLC list). This section reports performance improvements due to them, and where these improvements stand in relation to our past work. The improvements were determined by measuring execution times and count profiles across the eight C# programs shown in Table 1. The test bed was an HP XW8000 workstation running in hyperthread mode on an Intel Xeon 2.8GHz CPU, with Windows XP Version 2002 (Service Pack 2). Its memory was a 2GB RAM, an 8KB primary cache, and a 512KB secondary cache.

The OR optimizations were coded into Bartok, an ahead-of-time optimizing compiler that converts MSIL (the Microsoft Intermediate Language [2]) into x86 binaries. MSIL equivalents of the C# benchmarks were obtained using version 7.10 of `csc`, which is the .NET C# compiler. The three OR optimizations were compared against ERCS. In all cases, all other optimizations provided by Bartok were turned on, including previously proposed optimizations for nondeferred RC collectors [10]. The specific previous RC optimizations were a custom optimization for immortal objects, a type-based optimization for trial deletion, a custom specialization of RC updates with non-null operands, a simple intra-block coalescing optimization, and the inlining of lightweight RC updates [10]. Thus, improvements due to the immortal OR optimization are over that achieved with the previous custom immortal optimization.

There were no RC optimizations for heap references, except those pointing to immortal data and statically inferred acyclic types. The results thus show the improvements achievable by just looking at program roots. If the static OR optimizations were combined with dynamic techniques, such as the run-time coalescing of RC updates against heap references [13], further improvements should be possible.

### 5.1 Run-time System Description

Both the RC collectors are presently nongenerational. They support only single-thread execution, although the optimizations by design are applicable to multithread settings. They share major parts of a run-time system. This includes the

| Benchmark | Description | Cyclic Garbage | | Objects | Memory |
|---|---|---|---|---|---|
| | | Maximum (KB) | Total (KB) | Allocated ($\times 10^6$) | Allocated (MB) |
| `cmp` | File comparison tool, run on two 1006KB files. | 0 | 0 | $6.62 \times 10^{-3}$ | 1.72 |
| `xlisp` | Xlisp interpreter executing `au`, `boyer`, `browse`, etc., as part of a workload of 21 Lisp programs. SPEC CINT95 program port. | 0 | 0 | 125.49 | 1965.56 |
| `othello` | Othello (aka Reversi) strategic board game, played on an $8 \times 8$ grid. | 0 | 0 | 0.64 | 15.44 |
| `go` | Game of Life, played on a $40 \times 19$ board. SPEC CINT95 program port. | 0 | 0 | 0.64 | 15.44 |
| `satsol` | Boolean formula satisfiability solver. Available from `www.research.microsoft.com/~zorn/benchmarks`. | 0 | 0 | 8.16 | 167.74 |
| `chess` | Chess-playing program. SPEC CINT2000 program port. | 0.83 | 0.83 | 1.79 | 212.70 |
| `ahcbench` | The Adaptive Huffman Compression algorithm applied on files. Available from `www.research.microsoft.com/~zorn/benchmarks`. | 7.73 | 7.73 | 33.20 | 642.39 |
| `bartok` | MSIL to x86 ahead-of-time optimizing compiler, compiling itself to use generational copying collection. | 55051.96 | 537937.69 | 434.40 | 11073.55 |

**Table 1.** Descriptions and characteristics of the C# programs on which RC collection performance was evaluated.

cycle reclaimer, the delayed deallocator, and the segregated free-list allocator. The cycle reclaimer gathers lost cycles using the trial deletion algorithm. The delayed deallocator maintains a list of zombie objects, which are objects registered as garbage but that are yet to be reclaimed. When an object's reference count drops to zero, it is put on the zombie list. In the nondeferred case, it happens as soon as the last reference to the object dies or is overwritten. In the deferred case, it happens as a result of processing the zero-count table (ZCT) [5]. "Processing the zombie list" means traversing it in a last-in-first-out fashion, decrementing the reference counts of the immediate descendents of each of its entries. (This, in turn, can cause the list to grow.) After a constant number $d$ of decrements are applied, fully processed zombies are returned *en masse* to the allocator. (A zombie is fully processed after decrements are applied on all its immediate descendents.) The motivation for fixing $d$ is to bound pause times [22]. It was set to $2^{20}$ in both the collectors.

The heap level at allocation times was the triggering factor for processing the zombie list. Processing kicks in if the level exceeds 90% of the specified maximum. (The application aborts if the specified maximum is exceeded.)

Since the nondeferred collector does not require the services of the stack-scanning module, no GC maps are generated for it. They are needed by the deferred collector to process the ZCT [5].

### 5.2 Running Times

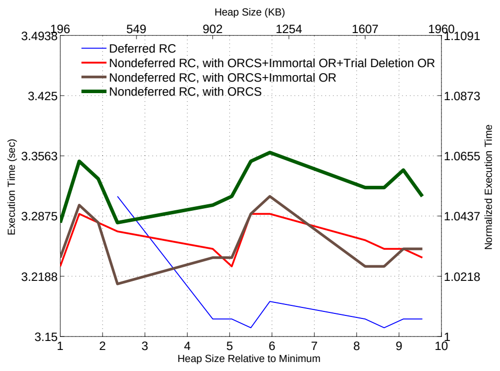
Figure 8 shows the execution times of the benchmarks with the two collectors. They are shown as a function of heap size, and in the nondeferred case, with the OR optimizations. Figure 9 shows these times normalized to the deferred collector. It also shows normalized times for the ERCS optimization; Figure 8 does not display them because they flatten the curves of programs like `cmp`.

The heap size was varied from the smallest amount at which the nondeferred collector could be executed, to an order of magnitude larger. For all the benchmarks, the deferred collector does not execute at the smallest sizes at which the nondeferred collector operates. Its curves begin between about 1.1 (in `bartok`) and 2.4 (in `cmp`) times the minimum size needed by the nondeferred collector.

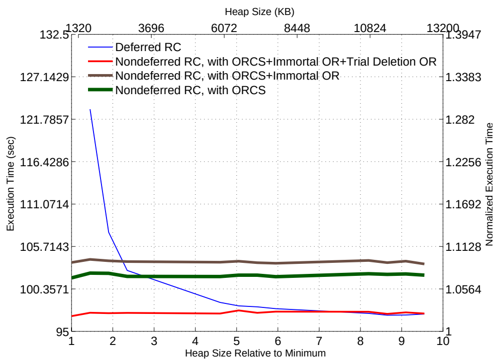
From the normalized times in Figure 9, we see that in all except one case, the ERCS-optimized nondeferred collector can match the performance of the deferred version, at least at the lower end of the heap range in which both operate. (Our past work has shown that the nondeferred collector can be an order of magnitude slower in the absence of RC optimizations [10].) For `xlisp`, the two have equal execution times at 1914KB, which is the minimum heap size required for the deferred collector to run. But as the size increases, the gap with the nondeferred collector widens. By the time it reaches 12606KB, the deferred collector is 27% faster. However, when the OR optimizations are applied, the performance of the nondeferred collector matches the deferred version even at the high end. At the low end, its performance becomes 21% better.

Worst-case performance with past optimizations is shown on the `cmp` program. Even at small heap sizes, the nondeferred collector is four times slower. But with the OR optimizations, it comes within 4% of the deferred collector.

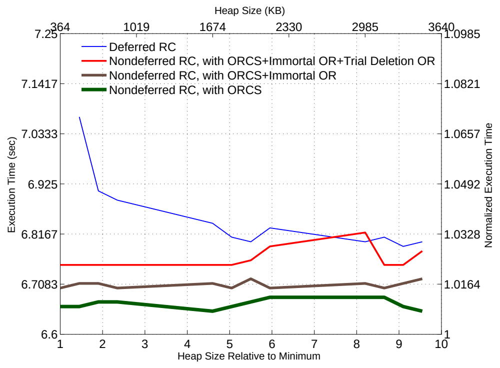
The graphs show that in all cases except `bartok`, nondeferred RC collection can reach parity with deferred RC collection. On `bartok`, the nondeferred performance at the low end is 26% better. It is 23% slower at worst, at 1.76 times the minimum heap size. There are two reasons for this. The first is that many of the reference mutation summaries for `bartok` are conservative. This is because the summaries are built using a virtual call analysis, which produces conservative results on the many virtual and interface calls in



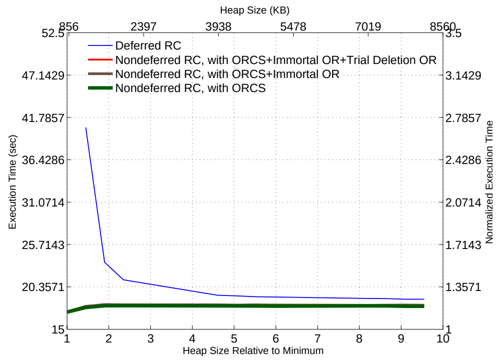
cmp



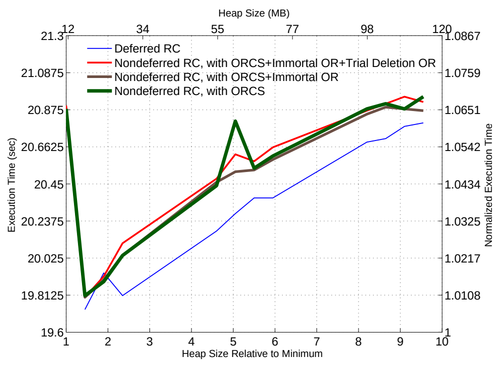
xlisp



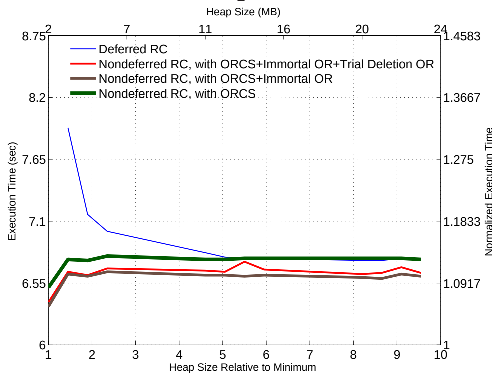
othello



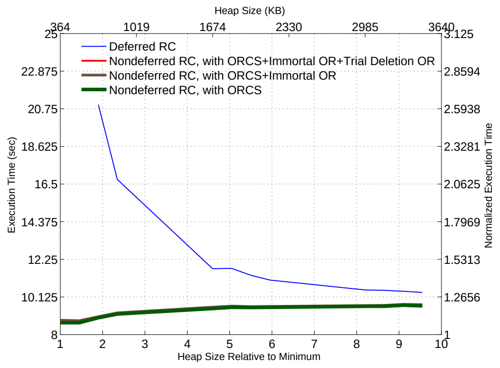
go



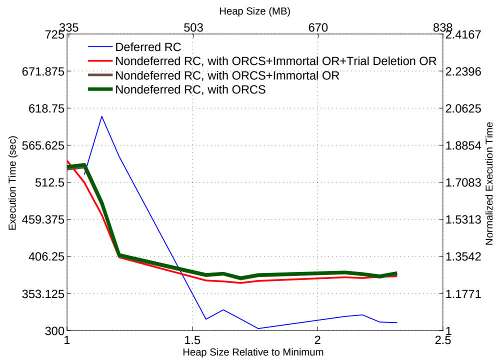
satsol



chess

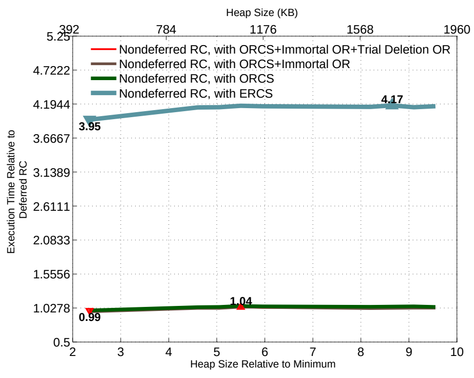


ahcbench

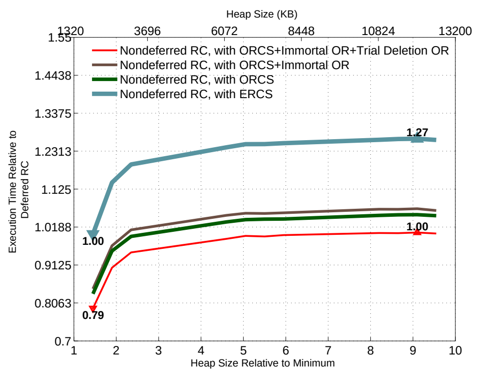


bartok

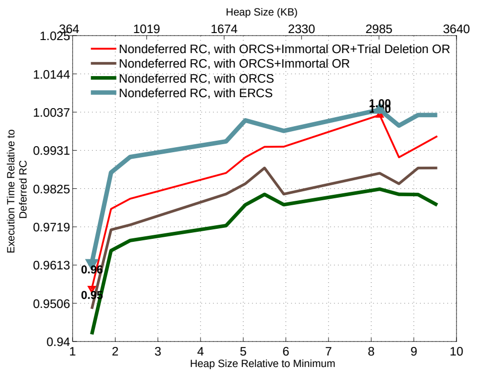
**Figure 8.** Absolute execution times as a function of heap size. Measurements using the nondeferred collector are with ORCS alone, with ORCS plus the immortal OR optimization, and with ORCS plus the immortal and trial deletion OR optimizations.



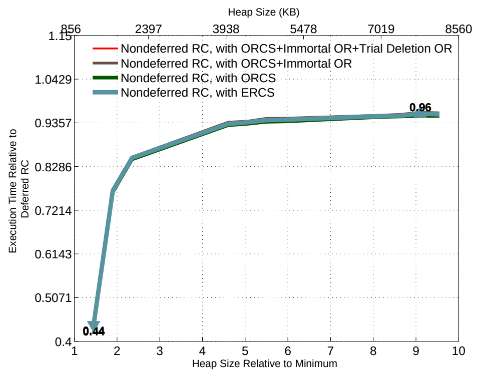
cmp



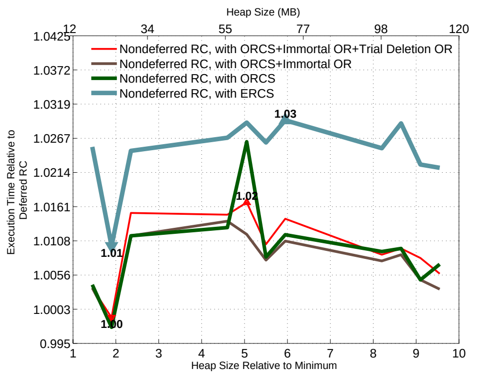
xlisp



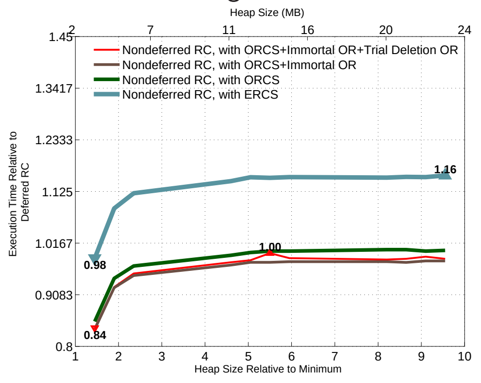
othello



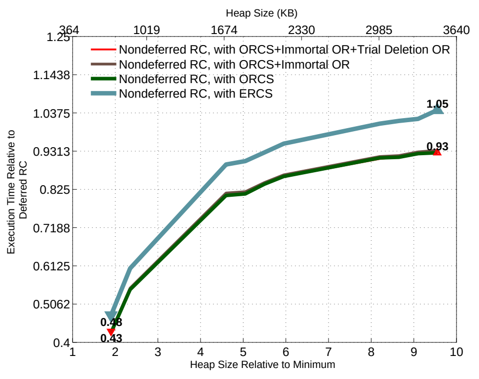
go



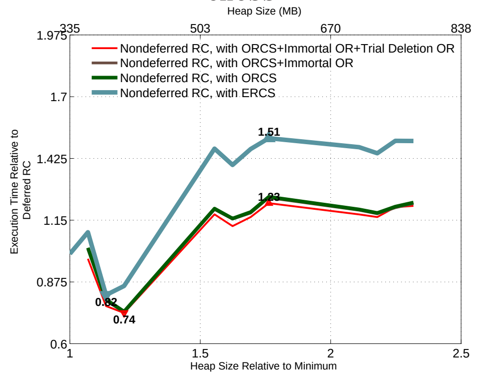
satsol



chess



ahcbench



bartok

**Figure 9.** Execution times relative to the deferred RC collector. The execution times considered are those in Figure 8, and additionally, those with ERCS. The downward and upward triangular markers indicate the minimum and maximum of the first and last curves in each graph.

bartok. The second reason is the high expense of using the trial deletion algorithm in the nondeferred collector. Unlike the deferred version, in which mainly heap references are put on the PLC list, every reference can be potentially buffered as a result of a decrement operation. Since the types of many of the created data structures are not statically inferable as acyclic, the overhead to using trial deletion is much larger in the nondeferred collector.

Table 2 presents profiles of the number of RC updates executed in the benchmarks. These were obtained by first determining the five methods that executed the highest number of RC updates. Averages of the number of RC updates executed in them were then measured. The averages were for those on objects inferred to be acyclic ($RC_A$) and the rest ($RC_O$). The table shows that ORCS can significantly reduce the total number of RC updates executed. Additional improvements from the immortal and trial deletion OR optimizations are less significant. Exceptions in the trial deletion case are the reductions on xlisp and bartok.

For the immortal OR optimization, even though the averages are the same in nearly all cases, this does not mean the optimization had no effect. This is because the averages consider only the top five RC hot methods. In the case of bartok, for instance, the optimization affected the counts in many of the remaining methods.

## 6. Related Work

An early effort at applying compiler optimizations to RC collection was due to Barth [1]. The goal was to determine RC updates superfluous under deferred reference counting. A series of optimizations, such as the removal of RC updates on immortal data, and the cancellation of pairs of increment and decrement operations, were proposed. As this paper shows, overlooking roots provide a framework for the nondeferred counterparts of Barth's proposals.

The problem of finding lifetime-subsumed references has been investigated for functional languages. Schulte observed that RC updates on a function's formal parameters can be optimized out, if its actual parameters are used after its return [20]. Park and Goldberg gave an analysis for establishing *escaping references*, i.e., references returned from creation scopes [18]. No counting is needed on references that do not escape. Their work is different from ours, in that it is more conservative about reachability and it does not consider intraprocedural lifetime information. Their analysis also assumed a language devoid of loops.

Overlooking roots also have applications outside of reference counting—for instance, they could be used to eliminate write barriers that operate on null references. Nandivada and Detlefs proposed two custom analyses, based on abstract interpretation, to do this for object fields and array elements [17]. The case of object fields can be easily handled using the overlooking roots framework. Figure 10 shows the example used by them to illustrate the object field case [17].

```
class Foo { public String s; }
Foo f1 = null; Foo f2 = null;
while (p1) {
  f1 = new Foo();          // F1
  f1.s = "hi1";            // W1
  if (p2) f2 = new Foo(); // F2
  f2.s = "hi2";  }         // W2
```

**Figure 10.** Example used by Nandivada and Detlefs to illustrate write-barrier removal in the object field case [17].

Like their analysis, the application discussed in Section 3.2.2 of this paper will be able to determine that the write barrier at W1 is unnecessary, but not the one at W2. If value-range information on array indices is available, the array element case can also be handled using overlooking roots.

Vechev and Bacon looked at the problem of identifying write barriers inessential to concurrent collectors [21]. At a high level, their goal is comparable to ours—both aim to remove unnecessary write-barrier/RC update operations. But an important difference is that while their work addresses the issue of when to eliminate, ours considers the problem of how to automatically eliminate. They presented conditions that when satisfied allow for the safe removal of barriers. Conditions similar to some of them are known in COM, as programmer guidelines. As an example, COM advises avoiding counting references with nested lifetimes [19].

### 6.1 Comparisons with Ghiya and Hendren's Analysis

Ghiya and Hendren described an analysis for approximately classifying the shapes of heap structures into tree, DAG or cyclic graph [8]. The analysis was built on top of two abstractions called the *direction matrix D* and the *interference matrix I*. Information in $D$ resembles the overlooking roots relation; that in $I$ is a superset of it. However, their analysis differs from ours in three ways:

- *May information.* It computes "may" information in $D$ and $I$, since the objective is to only arrive at heap shape approximations. For our work, must information is needed instead.
- *Field-oblivious approach.* To get at useful may information, their transfer functions for key statement categories (in particular, heap loads and heap stores) had to accommodate the generation of spurious relationships. For example, when faced with a statement like $y$.f := $x$, their analysis retained all of the incoming information for $D$ and $I$. This was because knowledge about the fields or types occurring on a heap path was lacking. Thus, simply tightening their transfer functions will not be enough to produce satisfactory must information.
- *Interprocedural analysis.* Procedures were handled using a context-sensitive interprocedural approach. Abstractions were flowed into a callee at a call site, to compute

| Benchmark | Counts ($\times 10^6$) | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | ERCS | | ORCS | | ORCS+Immortal | | ORCS+Immortal+Trial Deletion | | Deferred | |
| | $RC_A$ | $RC_O$ | $RC_A$ | $RC_O$ | $RC_A$ | $RC_O$ | $RC_A$ | $RC_O$ | $RC_A$ | $RC_O$ |
| `cmp` | 165.472 | 82.479 | 0.022 | – | 0.022 | – | 0.022 | – | 0.002 | – |
| `xlisp` | 616.568 | 607.764 | 392.148 | 468.975 | 392.148 | 468.975 | 510.496 | 350.627 | 225.724 | 299.869 |
| `othello` | 0.284 | – | 0.136 | – | 0.134 | – | 0.134 | – | 0.004 | 0.004 |
| `go` | 4.845 | – | 3.706 | – | 3.706 | – | 3.706 | – | 0.001 | 0.173 |
| `satsol` | 27.630 | 3.879 | 13.966 | 3.879 | 13.966 | 3.879 | 13.966 | 3.879 | 1.540 | 2.677 |
| `chess` | 17.005 | 3.390 | 3.328 | 0.394 | 3.328 | 0.394 | 3.328 | 0.394 | 1.229 | 0.487 |
| `ahcbench` | 48.139 | 43.278 | 23.396 | 21.618 | 23.396 | 21.618 | 23.396 | 21.618 | 6.442 | 6.625 |
| `bartok` | 307.910 | 302.757 | 146.605 | 154.942 | 146.605 | 154.942 | 149.680 | 151.867 | 72.951 | 72.951 |

**Table 2.** Effect of ERCS and the OR optimizations on RC update count distributions. Each entry is an average of the number of RC updates executed in the top five RC "hot" methods. Averages less than 1000 are shown as '–'.

abstractions at the return points of the callee. These were then flowed back to the caller at the call site. The overlooking roots analysis avoids this by using reference mutation summaries. In whole-program compilation mode, it has the option of flowing context-insensitive information back to the call site (see Section 4.2.1).

Other differences are the inclusion of virtual overlookers and MT-safe considerations in the transfer functions.

### 6.2 Grammar-Based Approaches

Jones and Muchnick applied a grammar-based shape analysis to reduce the counting and storage overheads in RC collection [11]. Their analysis can detect objects with reference counts at most one, and objects that can never appear on cycles. Our analysis has goals complementary to theirs.

A shape analysis based on *path expressions* was proposed by Hendren and Nicolau [9]. The aim was to determine if one computation could update a location read or written by another. A path expression is a grammar-based representation of a sequence of links in the object graph. As an example, $\mathtt{L}^2\mathtt{R}^+\mathtt{L}$ is a path that begins with two links of the field `L`, is followed by one or more links of the field `R`, and ends in a link of the field `L`. The analysis computes a *path matrix PM* of these expressions at every program point. An entry $PM(u, v)$ is a set of path expressions describing paths from the object targeted by $u$ to the object targeted by $v$. Paths can be either *definite*, if they are assured to exist, or *possible*, if not. Transfer functions were provided to update the definite and possible paths.

While grammar-based schemes permit more precise path information, their practicality on large programs is not clear.

## 7. Conclusion

This paper showed how a number of previously proposed optimizations for nondeferred RC collection can be unified using the idea of overlooking roots. It demonstrated how overlooking roots permit more powerful versions of many past optimizations, and how new optimizations can be devised using them. An analysis for computing the overlooking root information in a flow-sensitive manner was given. The analysis is intraprocedural; it handles procedures using summaries of the fields and types mutated in them. The article described how tie functions play a key role in producing reasonably good overlooking root information. The execution time improvements possible with optimizations derived from overlooking roots was then shown on a number of benchmarks. It was demonstrated that nondeferred RC collection can reach parity with the deferred version, at even large heap sizes, on nearly all the benchmarks.

None of the optimizations look at heap references. Extending this work to them would be a next step. Although the optimizations incorporate MT-safe considerations, the collector implementations are for single-thread programs. It remains to be seen how effective they would be in multithread programs. As is, they should be useful in single-thread settings, such as embedded systems and microcontrollers.

## References

[1] Jeffrey M. Barth. Shifting Garbage Collection Overhead to Compile Time. *Communications of the ACM*, 20(7):513–518, July 1977.

[2] Don Box and Chris Sells. *Essential .NET: The Common Language Runtime*. Addison-Wesley Publishing Company, Inc., Redwood City, CA 94065, USA, 2003.

[3] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 345–359. Association for Computing Machinery, June 1990.

[4] George E. Collins. A Method for Overlapping and Erasure

of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[5] L. Peter Deutsch and Daniel G. Bobrow. An Efficient, Incremental Automatic Garbage Collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[6] H. Gelernter, J. R. Hansen, and C. L. Gerberich. A FORTRAN-Compiled List-Processing Language. *Journal of the ACM*, 7(2):87–101, April 1960.

[7] Sanjay Ghemawat, Keith H. Randall, and Daniel J. Scales. Field Analysis: Getting Useful and Low-Cost Interprocedural Information. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 334–344. Association for Computing Machinery, June 2000.

[8] Rakesh Ghiya and Laurie J. Hendren. Is it a Tree, a DAG, or a Cyclic Graph? A Shape Analysis for Heap-Directed Pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15. Association for Computing Machinery, January 1996.

[9] Laurie J. Hendren and Alexandru Nicolau. Parallelizing Programs with Recursive Data Structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[10] Pramod G. Joisha. Compiler Optimizations for Nondeferred Reference-Counting Garbage Collection. In *Proceedings of the 5th International Symposium on Memory Management*, pages 150–161. Association for Computing Machinery, June 2006.

[11] Neil D. Jones and Steven S. Muchnick. Flow Analysis and Optimization of LISP-like Structures. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice-Hall, Inc., Englewood Cliffs, NJ 07458, USA, 1981.

[12] John B. Kam and Jeffrey D. Ullman. Monotone Data Flow Analysis Frameworks. *Acta Informatica*, 7:305–317, January 1977.

[13] Yossi Levanoni and Erez Petrank. An On-the-fly Reference Counting Garbage Collector for Java. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 367–380, October 2001.

[14] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification, Second Edition*. The Java Series. Addison-Wesley Publishing Company, Inc., 1999.

[15] Rafael D. Lins. Cyclic Reference Counting with Lazy Mark-Scan. *Information Processing Letters*, 44(4):215–220, December 1992.

[16] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., San Francisco, CA 94104, USA, 1997.

[17] V. Krishna Nandivada and David Detlefs. Compile-Time Concurrent Marking Write Barrier Removal. In *Proceedings of the 5th International Symposium on Code Generation and Optimization*, pages 37–48. IEEE Computer Society Press, March 2005.

[18] Young Gil Park and Benjamin Goldberg. Reference Escape Analysis: Optimizing Reference Counting Based on the Lifetime of References. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 178–189. Association for Computing Machinery, June 1991.

[19] Dale E. Rogerson. *Inside COM*. Microsoft Press, Redmond, WA 98052, USA, 1997.

[20] Wolfram Schulte. Deriving Residual Reference Count Garbage Collectors. In *Proceedings of the 6th International Symposium on Programming Language Implementation and Logic Programming*, pages 102–116, September 1994.

[21] Martin T. Vechev and David F. Bacon. Write Barrier Elision for Concurrent Garbage Collectors. In *Proceedings of the 4th International Symposium on Memory Management*, pages 13–24. Association for Computing Machinery, October 2004.

[22] Joseph Weizenbaum. Symmetric List Processor. *Communications of the ACM*, 6(9):524–536, September 1963.

## A. Transfer Functions for Other Statements

### A.1 Simple Assignments

$\underline{y := x}$ This statement kills pairs in which the overlooker is $y$, and the overlookee is not $x$ or something overlooked by $x$. It also kills pairs in which the overlookee is $y$, and the overlooker is not $x$ or something that overlooks $x$. Thus,

$$\begin{aligned} kill(s) = \{(y, v) \mid v \neq x \wedge (x, v) \notin olook_{in}(s)\} \\ \bigcup \{(u, y) \mid u \neq x \wedge (u, x) \notin olook_{in}(s)\}. \end{aligned} \tag{10}$$

The statement generates two kinds of overlooking pairs: (1) those in which the overlooker is $y$, and the overlookee is $x$ and whatever is overlooked by $x$; and (2) those in which the overlookee is $y$, and the overlooker is $x$ and whatever overlooks $x$. This gives

$$\begin{aligned} & gen(s) = \\ & \quad \{(y, v) \mid v \neq y \wedge (v = x \vee (x, v) \in olook^*_{in}(s))\} \\ & \bigcup \{(u, y) \mid u \neq y \wedge (u = x \vee (u, x) \in olook^*_{in}(s))\}. \end{aligned} \tag{11}$$

Pairs of the form $(u, u)$ should not exist in the *gen* sets because of the irreflexivity of the *olook* sets. This is the reason for the predicates $v \neq y$ and $u \neq y$ in Equation (11).

As discussed in Section 4.1.1, the tie function is updated at only those pairs that are in $gen^*(s)$. From Equations (11) and (5), these are of the form $(y, v)$ or $(u, y)$. For the $(y, v)$ pairs, $\mathscr{T}$ is updated to include the ties for $(x, v)$. For the $(u, y)$ pairs, it is updated to include the ties for $(u, x)$. This leads to the update equation

$$\mathscr{T}(u, v) \overset{\cup}{\leftarrow} \begin{cases} \mathscr{T}(x, v) & \text{if } u = y, \\ \mathscr{T}(u, x) & \text{if } v = y, \end{cases} \tag{12}$$

applied at all $(u, v) \in gen^*(s)$. In the above, $\mathscr{T}(u, v) \overset{\cup}{\leftarrow} X$ is a concise representation of $\mathscr{T}(u, v) \leftarrow \mathscr{T}(u, v) \cup X$.

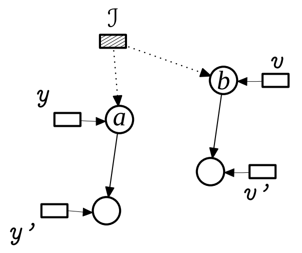


**Figure 11.** The single immortal root $\mathcal{I}$ directly overlooks $y$ and $v$ at the same time. But neither $y$ nor $v$ directly overlook $\mathcal{I}$ because there is no path from *a* to *b*, or from *b* to *a*.

$\underline{y := c}$ Since *c* is a constant reference, its target can be viewed as the target of an immortal root. The analysis has a range of options on how many immortal roots to model. At one extreme, a unique immortal root is associated with every different *c*. The statement can then be treated the same way as $y := x$, by substituting $x$ with the immortal root corresponding to *c*. At the other extreme, a single immortal root, say $\mathcal{I}$, simultaneously targets all immortal objects, as displayed in Figure 11. The latter offers simplicity over precision, and was our implementation choice. All pairs in which the overlookee is $y$ and the overlooker is not $\mathcal{I}$, or in which the overlooker is $y$, would then have to be killed:

$$kill(s) = \{(u, v) \mid (u \neq \mathcal{I} \wedge v = y) \vee u = y\}. \tag{13}$$

The *gen* set calculation for this statement should produce the pair $(\mathcal{I}, y)$. With a single immortal root, this will be the only pair generated; the pair $(y, \mathcal{I})$ is *not* generated because $\mathcal{I}$ may target more than one object, of which some may not be reachable from $y$ (refer Figure 11). Thus

$$gen(s) = \{(\mathcal{I}, y)\}. \tag{14}$$

$\mathscr{T}$ is not updated, because $\mathcal{I}$ has no overlookers.

## A.2 Allocations

$\underline{y := allocobj(T)}$ Section 3.2.2 explained that the object returned by $allocobj(T)$ can be thought to be the target of a pristine root. Like in the immortal case, numerous options are available on how many pristine roots to consider. There could be one per allocated type, or one per field per allocated type, or even one per allocation site. For the sake of simplicity, our implementation assumes one pristine root, say $\mathcal{P}$, for all allocated objects. With a single pristine root, the same issues that pertained to the calculations in Equations (13) and (14) apply to the *kill* and *gen* set calculations here:

$$kill(s) = \{(u, v) \mid (u \neq \mathcal{P} \wedge v = y) \vee u = y\}, \tag{15}$$

$$gen(s) = \{(\mathcal{P}, y)\}. \tag{16}$$

$\mathscr{T}$ is also not updated here because $\mathcal{P}$ has no overlookers.

## A.3 Heap Loads

$\underline{y := x.\mathtt{f}}$ If $x$ points to a thread-local object, or if `f` is a thread-safe field (i.e., only accessed by a particular thread) or a read-only field, we say that the statement is *multithread (MT) safe*. For such statements, all pairs in which $y$ is the overlooker, and all pairs in which the overlookee is $y$ and the overlooker is not $x$ or something that overlooks $x$, must be killed. For other statements, all pairs in which $y$ is either the overlooker or overlookee are killed:

$$kill(s) = \begin{cases} \{(u, v) \mid u = y \vee \big(v = y \\ \quad \wedge (u, x) \notin olook_{in}(s) & \text{if MT-safe } s, \\ \quad \wedge u \neq x\big)\} \\ \{(u, v) \mid u = y \vee v = y\} & \text{otherwise.} \end{cases} \tag{17}$$

There are a couple of cases in the *gen* set analysis for this statement. The easiest are the ones where *s* is not known to be MT safe. Then depending on whether `f` is an *immortal field*, the *gen* set is either $\emptyset$ or has the single pair $(\mathcal{I}, y)$. Fields are immortal if they *always* target immortal objects (i.e., even when simultaneously mutated by multiple threads). An example is the vtable field that all objects possess in many object-oriented language implementations.

If *s* is MT safe, then $gen(s)$ will at least have pairs in which the overlookee is $y$, and the overlooker is $x$ and whatever overlooks $x$. These cases yield the following equation:

$$gen(s) = \begin{cases} \{(u, y) \mid u \neq y \\ \quad \wedge \big((u, x) \in olook^*_{in}(s) & \text{if MT-safe } s, \\ \quad \vee u = x\big)\} \cup \psi \\ \{(\mathcal{I}, y)\} & \text{else if immortal } \mathtt{f}, \\ \emptyset & \text{otherwise.} \end{cases} \tag{18}$$

In Equation (18), the second case occurs when `f` is an immortal field and *s* is not MT safe. In the first case, $\psi$ is nonempty only when `f` is immortal, or when $\mathcal{P}$ directly overlooks $x$. If $\mathcal{P}$ directly overlooks $x$, then $y$ can be considered to be overlooked by $\mathcal{I}$, since that instance of `f` will then be in a pristine state. We therefore obtain the following equation for $\psi$:

$$\psi = \begin{cases} \{(\mathcal{I}, y)\} & \text{if immortal } \mathtt{f}, \\ \emptyset & \text{else if } (\mathcal{P}, x) \notin olook^*_{in}(s), \\ \emptyset & \text{else if } \mathscr{T}(\mathcal{P}, x) \neq \emptyset, \\ \{(\mathcal{I}, y)\} & \text{otherwise.} \end{cases} \tag{19}$$

According to Equation (18), there may be two types of pairs in $gen^*(s)$. The first is $(\mathcal{I}, y)$. The tie function will have to be updated here only if *s* is MT safe and $(\mathcal{I}, x) \in olook^*_{in}(s)$. The second type is $(u, y)$, where $u \neq \mathcal{I}$. For these pairs, either $(u, x) \in olook^*_{in}(s)$ or $u = x$; in both of these cases, `f` may tie $u$ to $y$. If $u \neq x$, then whatever ties $u$ to $x$ may also tie $u$ to $y$. This leads to the following update of the tie function, performed at all $(u, v) \in gen^*(s)$:

$$\mathscr{T}(u, v) \overset{\cup}{\longleftarrow} \begin{cases} \emptyset & \text{if not MT-safe } s, \\ \{\mathtt{f}\} & \text{else if } u = x, \\ \mathscr{T}(u, x) \cup \{\mathtt{f}\} & \text{otherwise.} \end{cases} \tag{20}$$

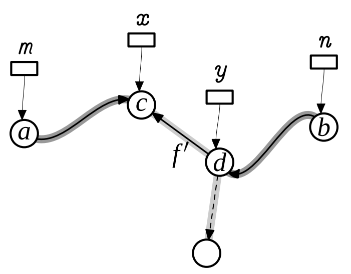


**Figure 12.** Object graph before and after `y.f := x`. The dashed arc represents the state of the specific instance $f'$ of `f` before the statement.

The treatment of $y := x[e]$ is the same, except that instead of a tie field, the discussion involves a tie array type.

### A.4 Heap Stores

<u>`y.f := x`</u> We say that the statement is MT safe if $y$ points to a thread-local object, or if `f` is a read-only or thread-safe field.[8] Irrespective of whether it is MT safe, it will kill all overlooking pairs involving the pristine root, if the number of pristine roots is one, and if $x$ is not directly overlooked by the immortal root. This is because the update could then destroy the pristine state of any newly allocated object.

Figure 12 graphically shows the effect on the heap of the MT-safe version of this statement. The lightly-shaded dashed and solid arcs represent the field's state before and after the statement. The heavily-shaded arcs display paths in the heap by which $m$ and $n$ respectively overlook $x$ and $y$.

At first glance, it would appear that all pairs in $olook_{in}(s)$ that are tied by `f` would have to be killed. But the statement has two important properties, from the standpoint of the overlooking roots relation, which permit better kill information. Lemmas 1 and 2 state and prove these properties.

LEMMA 1. *Let s be the statement* `y.f := x`*, which is given to be MT safe. If* $m$ *overlooks* $x$ *just before s, and if* $m$ *is not the pristine root, then* $m$ *will also overlook* $x$ *just after s.*

PROOF. The claim is obvious if $m$ overlooks $x$ by a path $p$ that is free of the specific `f`, say $f'$, updated by this statement; so suppose $p$ does include $f'$. Then from Figure 12, $p$ can be written as $p_1 f' p_2$, where $p_1$ is a path free of $f'$ from the object $a$ to the object $d$. When $s$ is MT safe, there will be the path $p_1 f'$ just after $s$, from $a$ to the object $c$. □

LEMMA 2. *Let s be the statement* `y.f := x`*,* not *necessarily MT safe. If* $n$ *overlooks* $y$ *just before s, and if* $n$ *is not the pristine root, then* $n$ *will also overlook* $y$ *just after s.*

PROOF. The claim is obvious if $n$ directly overlooks $y$ just before $s$, so suppose it does not directly overlook $y$ just before $s$. Then from Figure 12, $n$ overlooks $y$ by a nontrivial path that is free of the specific `f`, say $f'$, updated by $s$. Even in the presence of multiple threads, at least one such path is assured to exist during the execution of $s$, because if not, $n$ is not guaranteed to overlook $y$ just before $s$. □

Lemma 2 does not impose MT-safe requirements on $s$. This is because if $n$ overlooks $y$ just before $s$, then during the execution of $s$, there will be a path by which $n$ overlooks $y$ and that is free of the specific instance of `f` updated by $s$. Hence, from Lemmas 1 and 2, and the discussion on killing pairs that involve the pristine root, we get

$$kill(s) = \kappa \cup \{(\mathcal{P}, v) \mid (\mathcal{I}, x) \notin olook_{in}(s) \vee \mathscr{T}(\mathcal{I}, x) \neq \emptyset\}, \tag{21}$$

where

$$\kappa = \begin{cases} \{(u,v) \mid v \neq x \wedge v \neq y \wedge u \neq \mathcal{P} \wedge \mathtt{f} \in \mathscr{T}(u,v)\} & \text{if MT-safe } s, \\ \{(u,v) \mid v \neq y \wedge u \neq \mathcal{P} \wedge \mathtt{f} \in \mathscr{T}(u,v)\} & \text{otherwise.} \end{cases} \tag{22}$$

In Equation (21), the predicate $(\mathcal{I}, x) \notin olook_{in}(s) \vee \mathscr{T}(\mathcal{I}, x) \neq \emptyset$ is true if $\mathcal{I}$ might not directly overlook $x$. The equation then includes all pairs in which $\mathcal{P}$ is the overlooker.

If $s$ is MT safe, then all of the overlookers of $y$, including $y$, will end up overlooking $x$ as well as whatever is overlooked by $x$. If $s$ is not MT safe, but if `f` is known to be an immortal field, then $x$ will be directly overlooked by $\mathcal{I}$. This is subtle, because even if another thread mutates `f` as $s$ is executed, its target, by definition, remains immortal. Neither $y$ nor $x$, however, end up overlooking $\mathcal{I}$ because $\mathcal{I}$ could target multiple objects. This gives

$$gen(s) = \begin{cases} \{(u,v) \mid (u = y \vee (u,y) \in olook^*_{in}(s)) \wedge ((x,v) \in olook^*_{in}(s) \vee v = x)\} \cup \xi & \text{if MT-safe } s, \\ \{(\mathcal{I}, x)\} & \text{else if immortal } \mathtt{f}, \\ \emptyset & \text{otherwise.} \end{cases} \tag{23}$$

$\xi$ in Equation (23) is similar to $\psi$ in Equation (18). It is usually $\emptyset$, except when `f` is immortal:

$$\xi = \begin{cases} \{(\mathcal{I}, x)\} & \text{if immortal } \mathtt{f}, \\ \emptyset & \text{otherwise.} \end{cases} \tag{24}$$

From Equation (23), a pair in $gen^*(s)$ can be of three forms. If it is $(y,x)$, then only `f` needs to be added to $\mathscr{T}(y,x)$. If it is of the form $(y,v)$, where $v \neq x$, then $\{\mathtt{f}\}$ and $\mathscr{T}(x,v)$ would have to be added to $\mathscr{T}(y,v)$. Pairs in $gen^*(s)$ that do not match $(y,v)$ will be of the form $(u,v)$, where either $(u,y) \in olook^*_{in}(s)$ or $u = \mathcal{I}$. In both cases, a safe update of $\mathscr{T}(u,v)$ is to add $\mathscr{T}(u,y)$, $\{\mathtt{f}\}$ and $\mathscr{T}(x,v)$

[8] It is not contradictory for `f` to be read-only despite this being an update of `f`; this will be the case if the update is in an object construction sequence. Then, all accesses to `f` in the sequence will still be thread safe because the object being constructed will only be accessible to the initializing thread.

to it. By observing that both $\mathscr{T}(x,x)$ and $\mathscr{T}(y,y)$ equal $\emptyset$, all of these cases can be combined into

$$\mathscr{T}(u,v) \overset{\cup}{\leftarrow}$$
$$\begin{cases} \emptyset & \text{if not MT-safe } s, \\ \mathscr{T}(u,y) \cup \{\mathtt{f}\} \cup \mathscr{T}(x,v) & \text{otherwise,} \end{cases} \tag{25}$$

where $(u,v) \in gen^*(s)$.

The statement $y[e] := x$ is handled the same way.

### A.5 The Meet Operator

The meet operation is set intersection, except that overlooking pairs containing the $\top$ root are specially dealt with. $\top$ can only occur in pairs of the form $(\top, u)$. This is because when there are no upward-exposed uses of a concrete root $x$, $x$ will not overlook any other root until defined. Its only overlooker until its definition will be $\top$, after which it will no longer be overlooked by $\top$. Thus, if $olook_1$ and $olook_2$ are two *olook* sets that reach a confluence point, their meet $olook_1 \sqcap olook_2$ at that point will be

$$\begin{aligned} olook_1 \sqcap olook_2 = \\ \{(u,v) \mid \big((\top,v) \in olook_1 \wedge (u,v) \in olook_2\big) \\ \vee \big((u,v) \in olook_1 \wedge (\top,v) \in olook_2\big) \\ \vee \big((u,v) \in olook_1 \wedge (u,v) \in olook_2\big)\}. \end{aligned} \tag{26}$$

Let $\check{R}$ be the set of all roots. Because $\sqcap$ is an idempotent, commutative and associative operator on the set of ordered pairs $\check{R} \times \check{R}$, the pair $(\check{R} \times \check{R}, \sqcap)$ defines a semilattice. The semilattice, the $\sqcap$ operator, and the transfer functions in Section 4.2, form a monotone data-flow analysis framework [12].

## B. Overlooking RC Subsumption Analysis

We define the *live cover* of a root $r$ at a point $P$ as the set of live roots at least one of which overlooks $r$ at $P$. Let $liver(P,r)$ denote this set. If $liver(P,r)$ is nonempty at all $P$ in a live range $l$ of $r$, and if Provision B2 is also satisfied, then $l$ is an ORCS live range by the definition in Section 3.1.

Live cover sets have some unusual properties. For one, every subset of a live cover is not assured to be a live cover. As an example, if $liver(P,r)$ is $\{x_1, x_2\}$, then $\{x_1\}$ may not be a live cover of $r$ at $P$. But every superset (comprising live roots) of a nonempty live cover is a live cover. We call the former the *subset property*, and the latter the *superset property*. The empty set is a special case, and is considered to be a trivial (and an uninteresting) live cover.

Because of the subset property, computing $liver(P,r)$ becomes tricky. Of course, a guaranteed live cover at $P$ is

$$\widetilde{liver}(P,r) = live(P) \cap xproj(olook(P), r), \tag{27}$$

where *xproj* is the x-projection operator described in Section 4.2.1. But $\widetilde{liver}(P,r)$ could be $\emptyset$, as in at confluence points. Therefore, the goal of this section is to derive better information. "Better" means ascertaining a nonempty $liver(P,r)$ when $\widetilde{liver}(P,r)$ is $\emptyset$.

### B.1 Live Cover Transfer Functions

If $liver_{in}(s,r)$ and $liver_{out}(s,r)$ are the live covers of $r$, just before and just after a statement $s$, then as usual

$$\begin{aligned} liver_{out}(s,r) = \\ (liver_{in}(s,r) - KILL(s,r)) \cup GEN(s,r). \end{aligned} \tag{28}$$

When figuring out the $KILL(s,r)$ and $GEN(s,r)$ sets, there are a few cases to consider. Let $s_{out}$ be the program point just after $s$. If $\widetilde{liver}(s_{out},r)$ is nonempty, then by the superset property, a valid $liver_{out}(s,r)$ is $liver_{in}(s,r) \cup \widetilde{liver}(s_{out},r)$. Otherwise, if $s$ does not kill (with respect to the overlooking roots relation) any of the roots in $liver_{in}(s,r)$, and if none of these roots die as control flows through $s$, then $liver_{out}(s,r)$ can be set to $liver_{in}(s,r)$. Thus,

$$GEN(s,r) = \widetilde{liver}(s_{out},r), \tag{29}$$

$$KILL(s,r) = \begin{cases} \emptyset & \text{if } \widetilde{liver}(s_{out},r) \neq \emptyset, \\ \check{R} & \text{else if } xproj(kill(s),r) \cap liver_{in}(s,r) \neq \emptyset, \\ \check{R} & \text{else if } diethru(s) \cap liver_{in}(s,r) \neq \emptyset, \\ \emptyset & \text{otherwise.} \end{cases} \tag{30}$$

The expression $kill(s)$ in the above is the overlooking roots' kill set, from Section 4. The set $diethru(s)$ are the roots that die as control flows through $s$. It is

$$\begin{aligned} diethru(s) = \\ (live_{in}(s) - live_{out}(s)) \cup (live_{in}(s) \cap defs_{must}(s)), \end{aligned} \tag{31}$$

where $live_{in}(s)$ are the roots that are live on entry to $s$, and where $defs_{must}(s)$ are the roots that must be defined in $s$.

### B.2 Live Cover Meet Operator

Let the program point $Q$ lie at the confluence of $P_1$ and $P_2$, and let $liver_1$ and $liver_2$ be the live covers of $r$ at $P_1$ and $P_2$ respectively. If nothing in both $liver_1$ and $liver_2$ dies as control flows from $P_1$ and $P_2$ to $Q$, then $liver_1 \cup liver_2$ is a valid $liver(Q,r)$. So suppose something dies in either set. Now, if there is an overlooker $x$ of $r$ at $P_1$ that is live at $Q$, and if there is an overlooker $y$ of $r$ at $P_2$ that is also live at $Q$, then either $x$ or $y$ will be a live overlooker of $r$ at $Q$. A possible meet operation is therefore

$$liver_1 \sqcap liver_2 =$$
$$\begin{cases} liver_1 \cup liver_2 & \text{if } liver_1 \cup liver_2 \subseteq live(Q), \\ \widetilde{liver}(Q,r) & \text{else if } \widetilde{liver}_1 \cap live(Q) = \emptyset, \\ \widetilde{liver}(Q,r) & \text{else if } \widetilde{liver}_2 \cap live(Q) = \emptyset, \\ \widetilde{liver}_1 \cup \widetilde{liver}_2 & \text{otherwise,} \end{cases} \tag{32}$$

where $\widetilde{liver}_1 = \widetilde{liver}(P_1,r)$ and $\widetilde{liver}_2 = \widetilde{liver}(P_2,r)$.