Compiler Optimizations for Nondeferred Reference-Counting Garbage Collection

Pramod G. Joisha
Microsoft Research
pjoisha@microsoft.com

Abstract

Reference counting is a well-known technique for automatic memory management, offering unique advantages over other forms of garbage collection. However, on account of the high costs associated with the maintenance of up-to-date tallies of references from the stack, deferred variants are typically used in modern implementations. This partially sacrifices some of the benefits of nondeferred reference-counting (RC) garbage collection, like the immediate reclamation of garbage and short collector pause times.

This paper presents a series of optimizations that target the stack and substantially enhance the throughput of nondeferred RC collection. A key enabler is a new static analysis and optimization called RC subsumption that significantly reduces the overhead of maintaining the stack contribution to reference counts. We report execution time improvements on a benchmark suite of ten C# programs, and show how RC subsumption, aided with other optimizations, improves the performance of nondeferred RC collection by as much as a factor of 10, making possible running times that are within 32% of that with an advanced traversal-based collector on seven programs, and 19% of that with a deferred RC collector on eight programs. This is in the context of a baseline RC implementation that is typically at least a factor of 6 slower than the tracing collector and a factor of 5 slower than the deferred RC collector.

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

General Terms Algorithms, Languages

Keywords Reference Counting, Static Analyses

1. Introduction

Reference-counting (RC) garbage collection is a technique for reclaiming unreachable data using a count maintained against each logically independent unit of data x. The count indicates whether there are any references to x, and changes as references are added and deleted. A value of zero means that there are no references to x, at which point it is safe to reclaim x.

RC collection, in its original form, tallies all references to an object, be it from the stack, the static data area or the heap [14, 8]. The reference count is kept up to date by special operations, called *RC updates*, that increment and decrement the reference count as references are created and destroyed. This maintenance regimen incurs high processing costs in programs where references are heavily mutated. Consequently, classic RC collection has usually suffered from poor throughput.

Past solutions to the throughput problem have essentially boiled down to counting only a subset of references, postponing the processing of the rest [13, 6]. Since the counts are therefore approximate, these schemes pause the mutator threads once in a while to bring the counts up to date before reclaiming objects with zero reference counts. While the action of deferring RC updates improves a program's overall throughput, it however increases its heap footprint (i.e., the high watermark of the heap) since the reclamation of garbage is also deferred. It also contributes to larger pause times, unlike classic RC collection where collector actions are more uniformly "smeared" throughout the program.

Yet the costs of maintaining accurate tallies in classic RC collection are so high, even for the single-threaded uniprocessor case, that it has never been regarded as a practical garbage collection method, always being supplanted by deferred versions wherever used in modern systems [26, 11, 2, 18].

This paper presents a new static analysis and optimization called *RC subsumption*, that together with other RC-specific compiler optimizations, dramatically enhances not only the throughput of classic RC collection, but that of any nondeferred RC collection scheme as well. (By "nondeferred," we mean the class of RC collection techniques that immediately account for root references as they come into and go out of existence.[1]) On a benchmark suite of ten C# programs using one such nondeferred scheme, the improvement is by as much as a factor of 10, making it possible to achieve execution times that are often within 32% of that with an advanced traversal-based generational collection scheme, and 19% of that with a deferred RC collection scheme.

The other contribution of this paper is that it shows how other optimizations that have either only been suggested (like the omission of RC updates on permanently live data [5]), used in run-time systems for RC collection (like the special treatment of acyclic objects in the trial deletion technique [2]) or are straightforward (like the inlining of RC updates), fare on object-oriented programs.

Moreover, we aren't aware of a stand-alone nondeferred cycle-reclaiming RC garbage collector for a modern object-oriented language. In that sense, this paper reports the first reference measurements on an important memory management design space.

Although our implementation of the collector is single threaded, the optimizations are also applicable to multithreaded code. While the reported measurements demonstrate that nondeferred RC col-

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.

[1] The invariant is that a zero reference count means garbage and a positive reference count means at least one incident reference.

lection can come close to tracing or deferred RC collection methods, the point of the paper isn't to show that it can displace them. Rather, the main contribution is that single-threaded nondeferred RC collection can be made much faster than what has been previously believed. We see the value of this work as shedding new light on nondeferred RC collection and positioning it better for applications such as real-time garbage collection.

The rest of this paper is organized as follows. Sections 2 to 7 describe a series of optimizations for improving the throughput of nondeferred RC collection.[2] Section 8 reports the results of these optimizations on a nongenerational implementation of nondeferred RC collection, and compares the optimized binaries with a generational adaptive tracing collector and a deferred RC collector. Section 9 discusses related work and Section 10 concludes the paper.

2. Reference-Counting Subsumption

Consider the intermediate representation (IR) in Figure 1, in which RC update operations[3] are inserted according to a classic RC collection discipline [17]. We assume that the object creation instruction *newobj*() initializes a newly allocated object's reference count to 1. Since y points to the same object as x in its live range from Line 4 to Line 5, and since this live range is contained in the live range of x from Line 1 to Line 7, the RC updates on Lines 2 and 5 are superfluous.[4] Observe that this redundancy holds true even in a multithreaded, multiprocessor setting. We say that "y is RC subsumed by x" to describe this state of affairs.

```
1   x := newobj()
        ⋮
2   RC+(x)
3   RC−(y)
4   y := x
5   RC−(y)
6   y := null
        ⋮
7   RC−(x)
8   x := null
```

Figure 1. An RC Subsumption Opportunity

It turns out that of the RC updates introduced into real programs by a nondeferred RC collection approach, a significant portion is on local references that are RC subsumed by local references that already have RC updates against them. For instance, the RC updates on formal references are often redundant because formal references are usually RC subsumed by actual references.[5]

Formally, a local reference variable y is defined to be *always* RC subsumed by a local reference variable x if

1. all live ranges of y are contained in live ranges of x;
2. y is never live through a redefinition of either x or y; and
3. the set of objects reachable from y is always a subset of the set of objects reachable from x, i.e., $\mathcal{R}(y) \subseteq \mathcal{R}(x)$.

[2] Some of these optimizations may also reduce the running time of deferred RC collection but that is beyond the scope of this paper.

[3] An RC increment or decrement on an object targeted by a reference r is denoted as $RC_+(r)$ or $RC_-(r)$. These are no-ops when r is null.

[4] The $RC_-(y)$ on Line 3 remains; this is to decrement the reference count of the object that y is about to be swung away from on Line 4.

[5] This may not be true when actual references are overwritten in the callee via pointers and when formal references are redefined in the callee.

Recall that the set of local references live before a statement s is related to those live after s by the equation

$$live_{in}(s) = (live_{out}(s) - defs_{must}(s)) \cup uses_{may}(s), \qquad (1)$$

where $defs_{must}(s)$ and $uses_{may}(s)$ are the sets of local references that must be defined and that may be used in the statement [20].

The reason for Provision 2 is that liveness is a "may" information. In other words, if a more relaxed $uses_{may}$ or a more constrained $defs_{must}$ is used in Equation (1), then a variable could end up being live at a program point even though it may never be used from that point onward prior to a redefinition. This has a subtle consequence on RC subsumption. For example, in Figure 2, x and y will be considered live through Line 7 when no information about the locations pointed to by p is available. However, if the RC updates on Lines 2 and 8 weren't present (on the presumption that y is RC subsumed by x) and the store operation on Line 7 overwrites y, then the reference count of the object targeted by x could prematurely fall to zero after Line 6. This is even though Provision 3 might continue to be true after Line 7. (A similar example can be worked out in which x is overwritten in Line 7.)

```
1   x := newobj()
        ⋮
2   RC+(x)
3   RC−(y)
4   y := x
5   RC+(z)
6   RC−(*p)
7   *p := z
8   RC−(y)
9   y := null
10  RC−(x)
11  x := null
```

Figure 2. A Contingent RC Subsumption Opportunity

Ascertaining the three provisions of RC subsumption at compile time is complicated by two factors. Firstly, live ranges may not be the nice linear stretches shown in Figure 1. They can in general be "webs" [20] spanning multiple definitions and multiple last uses. Secondly, object reachability as stated in Provision 3 is a dynamic, run-time trait and may not always be statically determinable. Nevertheless, we shall demonstrate how Provision 3 can be conservatively decided at compile time and how Provision 1 can be exactly calculated using interference graph construction notions [20].

Our solution is in three parts. First, we build a *live-range subsumption graph* $G_L = (V, E_L)$ for a given function f. Nodes in G_L denote local references and directed edges represent live-range containment. That is, $(u, v) \in E_L$, iff the live ranges of u in f are contained in the live ranges of v. Next, an algorithm that determines a subgraph G_U of G_L called the *uncut live-range subsumption graph* is presented. $G_U = (V, E_U)$ has the additional property that if $(u, v) \in E_U$, then u is never live through a redefinition of either itself or v. Finally, we obtain a subgraph $G_R = (V, E_R)$ of G_U such that if $(u, v) \in E_R$, then u is always RC subsumed by v. G_R will be referred to as the *RC subsumption graph*.

2.1 The Live-Range Subsumption Graph

The live range of a program variable u is a collection of du-chains connecting one or more definitions of u with one or more of its last uses [20]. A live range of u could therefore be nonempty, if it includes at least one program point, or empty, if u is never used. When no live range of u contains the program point P, u is said to be *dead* at P. Thus, if u is live and v is dead at some P, then not

1. Initialize V to the set of local references R, and E to $\emptyset$.
2. At every statement s, define

$$live_{wtn}(s) = live_{out}(s) - defs_{must}(s), \quad (2)$$

$$dead_{wtn}(s) = R - live_{wtn}(s), \quad (3)$$

$$dead_{in}(s) = R - live_{in}(s), \quad (4)$$

$$dead_{out}(s) = R - live_{out}(s), \quad (5)$$

and add (u, v) to E if

$$\begin{aligned}&(u \in live_{in}(s) \wedge v \in dead_{in}(s)) \bigvee \\ &(u \in live_{wtn}(s) \wedge v \in dead_{wtn}(s)) \bigvee \\ &(u \in live_{out}(s) \wedge v \in dead_{out}(s)).\end{aligned} \quad (6)$$

3. Find $(V, \overline{E})$, the *complement graph* [10] of (V, E). Then $(V, \overline{E})$ is the live-range subsumption graph (V, E_L).

Figure 3. Computing $G_L = (V, E_L)$

every live range of u can be contained in a live range of v. Figure 3 uses this fact to build the live-range subsumption graph.

In Equation (2), $live_{wtn}(s)$ is the live set *within* s, just before variables in $defs_{must}(s)$ are assigned to, but just after the variables in $uses_{may}(s)$ have been used. $live_{wtn}(s)$ is therefore the smallest live set encountered when traversing through s, from its front to its back. We postulate that at any program point P within or on the boundaries of s, $live(P)$ is either $live_{in}(s)$, $live_{wtn}(s)$ or $live_{out}(s)$. This can be utilized to prove Lemmas 1 and 2 below. The two lemmas taken together imply that $(u, v) \in \overline{E}$ (which is E_L), iff the live ranges of u are contained in the live ranges of v.

LEMMA 1. *If $(u, v) \in \overline{E}$, then for every live range of u, there exists a live range of v that contains it.*

PROOF. Suppose there exists a live range λ of u that isn't contained in any live range of v, and yet $(u, v) \in \overline{E}$. Then λ must contain at least one program point, say P, that isn't contained in any live range of v. Let P be within or on the boundaries of a statement s', and let $live(P)$ be the set of references live at P. Then $live(P)$ must be one of $live_{in}(s')$, $live_{wtn}(s')$ or $live_{out}(s')$. Suppose

$$live(P) = live_{wtn}(s').$$

Since $v \notin live(P)$, we have $v \in dead_{wtn}(s')$. Hence $(u, v) \in E$ after Step 2 of the algorithm. This implies $(u, v) \notin \overline{E}$, a contradiction. The other two cases (when $live(P) = live_{in}(s')$, and when $live(P) = live_{out}(s')$) lead to the same conclusion. □

LEMMA 2. *If for every live range of u there exists a live range of v that contains it, then $(u, v) \in \overline{E}$.*

PROOF. Suppose for every live range of u there exists a live range of v that contains it and yet $(u, v) \notin \overline{E}$. Then $(u, v) \in E$. This means that there exists a statement s' such that at least one of the three conditions in Equation (6) apply. Hence, there is a program point P within or on the boundaries of s' that is contained in a live range of u but not in a live range of v, a contradiction. □

If N is the number of statements in a function, then the worst-case complexity of the algorithm is $O(|V|^2 N)$. This on a par with that of constructing interference graphs [20].

2.2 The Uncut Live-Range Subsumption Graph

Consider $live_{thru}(s)$ in Equation (8), which is the set of references that are live through a statement s. Let $defs_{may}(s)$ be the set of local references that may be defined at s. Then the set $live_{rdef}(s)$ given by Equation (9) consists of references that may be live through their own redefinition. The algorithm in Figure 4 uses this set to arrive at G_U by first initializing E' to E_L and then eliminating all outgoing edges from nodes in $live_{rdef}(s)$, and those among its incoming edges that are from nodes whose references are live through s. The edges that remain, after all s have been accounted for, will therefore satisfy Provisions 1 and 2.

1. Initialize V to the set of local references R, and E' to E_L.
2. At every statement s, define

$$live_{io}(s) = live_{in}(s) \cap live_{out}(s), \quad (7)$$

$$live_{thru}(s) = (R - defs_{must}(s)) \cap live_{io}(s), \quad (8)$$

$$live_{rdef}(s) = defs_{may}(s) \cap live_{thru}(s), \quad (9)$$

and delete $(u, x) \in E'$, if $u \in live_{rdef}(s)$. In addition, delete $(y, u) \in E'$, if $y \in live_{thru}(s)$ and $u \in live_{rdef}(s)$. Then (V, E') is the uncut live-range subsumption graph (V, E_U).

Figure 4. Computing $G_U = (V, E_U)$

A modest estimate of $defs_{may}(s)$, obtainable with an alias analysis [20], is pivotal to the algorithm's efficacy as G_U could otherwise become bereft of edges. A tighter $defs_{may}(s)$ also improves its running time, which in the worst case is $O(|V|^2 N)$ as before.

2.3 The Reference-Counting Subsumption Graph

We use the notation $y \xrightarrow{P} \omega$ to mean that the local reference y targets the object ω at the program point P. Let $\alpha(s)$ and $\beta(s)$ be program points just before and after a statement s. Consider the set

$$\begin{aligned}\mathbb{R}(s, y) = \{x \mid &x \in R \wedge y \in live_{out}(s) \wedge y \xrightarrow{\beta(s)} \omega \wedge \\ &\omega \in \mathcal{R}(x) \text{ on all paths from } \alpha(s) \text{ until} \\ &x \text{ dies or could be redefined}\},\end{aligned} \quad (10)$$

which is the set of local references that "overlook" the object targeted by a live y just after s. These references overlook the object from just before s, at which they are implicitly live by the equation, until their death or possible redefinition.

As an example, suppose s were the IR statement

$$y := x$$

and let $y \in live_{out}(s)$. Then clearly, $x \in \mathbb{R}(s, y)$.

The *overlooking roots' set* $\mathbb{R}(s, y)$ plays a key role in the construction of the RC subsumption graph. The strategy, outlined in Figure 5, starts with a copy E'' of E_U and then eliminates edges that may possibly violate Provision 3. It uses any approximation $\grave{\mathbb{R}}(s, u)$ of $\mathbb{R}(s, u)$; i.e., $\grave{\mathbb{R}}(s, u) \subseteq \mathbb{R}(s, u)$ for all s and $u \in R$.

Using $succ(u)$ to denote the successor nodes of u in (V, E''), Equations (11) and (12) specify the set of edges removed at each statement in each iteration of the algorithm. In essence, an edge (u, v), where $u \neq v$, is deleted from E'' under two circumstances:

- there exists an s that may define u, but at the end of which there is no known reference that overlooks u; or
- there exists an s that may define u, and at the end of which u is overlooked by a $w(\neq v)$ that may not be RC subsumed by v.

Theorem 1 below proves that the edges in E'' after Step 4 of the algorithm indeed belong to the RC subsumption binary relation.

THEOREM 1. *If $(u, v) \in E_R$, then u is RC subsumed by v.*

PROOF. If $u = v$, the claim is obvious. So let u_1, u_2 and so on until u_k be the other nodes from which edges are incident on v in

1. Initialize V to the set of local references R, and E'' to E_U.
2. For every $u \in defs_{may}(s)$ such that $\grave{\mathbb{R}}(s, u) = \emptyset$, define
$$\delta(s, u) = succ(u) - \{u\} \quad (11)$$
and delete $(u, v) \in E''$ if $v \in \delta(s, u)$.
3. For every $u \in defs_{may}(s)$ such that $\grave{\mathbb{R}}(s, u) \neq \emptyset$, define
$$\delta(s, u) = \bigcup_{w \in \grave{\mathbb{R}}(s,u)} \big(succ(u) - (succ(w) \cup \{u, w\})\big) \quad (12)$$
and delete $(u, v) \in E''$ if $v \in \delta(s, u)$.
4. Repeat Steps 2 and 3 on every statement s until a fixed point is reached. Then (V, E'') is the RC subsumption graph (V, E_R).

Figure 5. Computing $G_R = (V, E_R)$

G_R. Now because of Step 2 of the algorithm, we must have

$$\grave{\mathbb{R}}(s, u_i) \neq \emptyset$$

at every s, where $u_i \in defs_{may}(s)$; otherwise (u_i, v), where $1 \leq i \leq k$, would have been deleted. In fact, we must also have

$$\grave{\mathbb{R}}(s, u_i) \subseteq \{u_1, u_2, \ldots, u_k\} \cup \{v\} \quad (13)$$

from Step 3 of the algorithm since otherwise, (u_i, v) would have been deleted. But this implies that after every possible definition of u_i, it would have to be overlooked by at least one other u_j or by v. Because the same argument holds true for any of the u_i, and because $(u_i, v) \in E_U$ in each case, the three provisions of RC subsumption will be satisfied by all the (u_i, v). Hence, the u_i must be RC subsumed by v for all $1 \leq i \leq k$. □

2.3.1 Approximating the Overlooking Roots' Set

The algorithm requires $\grave{\mathbb{R}}(s, u)$ only at those s at which $u \in defs_{may}(s)$. We have already seen how $\grave{\mathbb{R}}(s, u)$ can be determined when s is of the form $u := v$. A few more situations exist in which we can do better than an empty set value for $\grave{\mathbb{R}}(s, u)$.

For example, suppose s is the IR statement

$$u := v.\mathtt{g},$$

where g is a *read-only* field [15]. A field is read-only if it isn't modified after its initialization in any thread. The initialization point is just after object construction for instance fields and just after static construction for static fields. Hence, if the above statement occurs after the initialization point for g, and if $u \in live_{out}(s)$, then a possible value for $\grave{\mathbb{R}}(s, u)$ is $\{v\}$. A corner situation is

$$u := u.\mathtt{g}$$

in which case u would belong to $\grave{\mathbb{R}}(s, u)$ if $u \in live_{out}(s)$.

Another situation is the IR statement

$$u := v[e],$$

where $u \in live_{out}(s)$, v points to a *thread-local object* and $v[e]$ isn't written into before v dies or is possibly redefined in the current thread's code. Then v could be added to $\grave{\mathbb{R}}(s, u)$. Note that it doesn't matter whether the references in v point to objects visible across threads; the thread localness requirement is only on v and is an immediate, nontransitive one. An escape analysis similar to [27] could be utilized to infer this in a number of useful situations.

A fourth situation is the IR statement

$$u := v.\mathtt{f},$$

where $u \in live_{out}(s)$ as before, but where f isn't a read-only field. If v, however, is known to only target thread-local objects and $v.\mathtt{f}$ isn't written into before v dies, then v could be added to $\grave{\mathbb{R}}(s, u)$.

The last situation is specific to a formal reference z of a function f. Now z can be imagined as being initialized by the statement

$$z := \hat{z}$$

on entry to f, where $\hat{z}$ is the actual parameter that corresponds to z. If $\hat{z} \in R$ and $z \in live_{out}(s)$, where s is the imaginary initialization, then $\hat{z}$ could be added to $\grave{\mathbb{R}}(s, z)$. Now $\hat{z}$ can be considered live throughout f.[6] Therefore, if $\hat{z}$ were included in V, then $(z, \hat{z})$ would exist in E_U only if z is never live through a redefinition of either itself or $\hat{z}$. This extension, combined with $\hat{z} \in \grave{\mathbb{R}}(s, z)$, enables the algorithm in Figure 5 to automatically handle the RC subsumption of formal references by actual references. The extension could be viewed as a cheap alternative to a more elaborate interprocedural approach for deriving the same information.

An important point to address here is the validity of the consideration that $\hat{z}$ is live throughout f. At a call site of f, it is questionable as to where the actual argument $\hat{z}$ dies, assuming it isn't used after the call site. Does it die just after the invocation of f, or within the invocation? If within, where exactly? The consideration follows from our view that it dies after the call site. The implication is that an object could be held across the entire duration of a call, even though it might be used only at the beginning of the call.

```
1   function map(F, z)
2   F := F̂
3   z := ẑ
4   x := z.buckets
5   i := 0
6   while (i < x.length),
7       y := x[i]
8       while (y ≠ null),
9           w := y.value
10          y := y.next
11          F(w)
12          end while
13      i := i + 1
14      end while
```

Figure 6. Applying a Function on the Values of a Hash Table

2.3.2 An Example: Traversing a Hash Table

Figure 6 is the IR of a function map that takes as arguments a reference z to a hash table and a reference F to a *delegate* [7]. Delegates are "function objects" that enable method invocation through references, as seen on Line 11. Thus map simply traverses through the linked lists of a hash table, which are organized as an array of buckets, and applies F on the stored values along the way.

Lines 2 and 3 are the imaginary assignments that model the call-by-value parameter passing mechanism. (For clarity, no RC updates have been shown.) $\hat{F}$ and $\hat{z}$ are the actual parameters in the caller that correspond to F and z. The other local references are x, y and w. Figure 7 displays the uncut live-range subsumption graph for map. Observe that G_U reflects the assumption that $\hat{F}$ and $\hat{z}$ are live throughout map. Also, the live range of z doesn't subsume the live ranges of x, y and w; this is because z dies after Line 4.

Next, to compute the RC subsumption graph, we need to arrive at approximations of the overlooking roots' set. These are shown

[6] In the absence of more information, no mutual subsumption among the live ranges of the actual parameters should be considered.

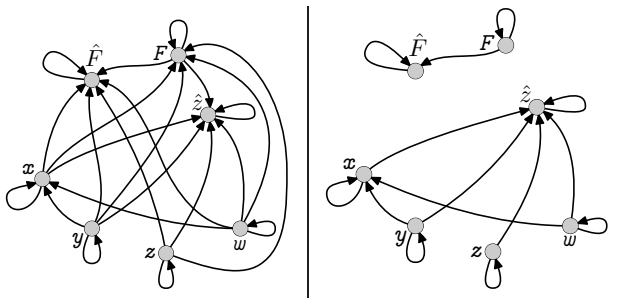

Figure 7. Subsumption Graphs for the map Function

below for the relevant references and statements:

$$\grave{\mathbb{R}}(s_2, \texttt{F}) = \{\hat{F}\}, \qquad \grave{\mathbb{R}}(s_7, \texttt{y}) = \{\texttt{x}\},$$
$$\grave{\mathbb{R}}(s_3, \texttt{z}) = \{\hat{z}\}, \qquad \grave{\mathbb{R}}(s_9, \texttt{w}) = \{\texttt{y}\},$$
$$\grave{\mathbb{R}}(s_4, \texttt{x}) = \{\texttt{z}\}, \qquad \grave{\mathbb{R}}(s_{10}, \texttt{y}) = \{\texttt{y}\}.$$

The determination of the above values for $\grave{\mathbb{R}}(s_2, \texttt{F})$ and $\grave{\mathbb{R}}(s_3, \texttt{z})$ is straightforward. The values for $\grave{\mathbb{R}}(s_4, \texttt{x})$, $\grave{\mathbb{R}}(s_7, \texttt{y})$, $\grave{\mathbb{R}}(s_9, \texttt{w})$ and $\grave{\mathbb{R}}(s_{10}, \texttt{y})$ rely on the immediate targets of `z`, `x` and `y` being thread-local objects.[7] This will be the case if the hash tables passed into map at the various program call sites don't escape threads.

Figure 7 shows the RC subsumption graph obtained using these overlooking roots' sets. Since `F`, `z`, `x`, `y` and `w` are all RC subsumed by a reference other than themselves, the RC updates against them can be eliminated, permitting an RC update-free map.

3. Specializing RC Updates on Acyclic Objects

Our nondeferred RC collector doesn't depend on any backup collector for completeness. It reclaims cycles using a local scan technique called *trial deletion* [19], as extended in [3, 2].

The basic idea behind trial deletion is that when a reference disappears from an object o whose reference count is greater than 1, o could potentially be lost as a leaked cycle. Consequently, the RC_- operation conservatively puts o onto a "potentially leaked cycles" (PLC) list for later processing, possibly when a memory pressure situation arises. If a reference to o is subsequently created, the associated RC_+ operation pulls o out of the PLC list as o then can't lie on a lost cycle. Hence, statically knowing types whose instances never lie on cycles allows the compiler to generate specialized RC updates that avoid the cycle bookkeeping steps needed in a more general RC update operation under trial deletion.

Although the aforementioned compile-time optimization hasn't been implemented in the past, knowledge about acyclic objects has been used to optimize the run-time actions of the general RC update operation [2]. That is, the cycle bookkeeping steps could be performed conditionally depending on whether the object's exact type has only acyclic instantiations. The steps are skipped if the test returns true. This is a collector-side optimization and also exists in our implementation. Rather than accessing an object's exact type on every RC update, it could be accessed once at object creation time through the vtable data structure and then used to set a bit in the object's RC field indicating the acyclic nature of the type [2].

We say that a type t is *acyclic* if an instance of t doesn't transitively point to itself through a chain of references. The state of being acyclic could be spoken of at different levels of granularity and in senses involving a timeline. Thus, t could be acyclic in certain areas of the object graph, at certain program points, or always in all runs of a given program. For example, the `System.String` class in .NET is an acyclic type in *any* program. Given a type hierarchy, information on the fields in the types and a mode of compilation (whole program or separate), Figure 8 presents a procedure that determines the set of types that are acyclic in all programs.

[7] Whether the `value` field targets a thread-local object is immaterial.

1. Initialize V_T to the set of reference types, and E_T to $\emptyset$.
2. If $s \in V_T$ has an instance field of static reference type t, or if s is an array type whose element has the static reference type t, then add (s, t') to E_T for every t' that is a subtype of t.
3. If $s \in V_T$ is an array type whose element is a value type that directly or indirectly contains an instance field of reference type t, then add (s, t') to E_T for every t' that is a subtype of t.
4. Add (s', t) to E_T if (s, t) was added due to some field in Step 2 or Step 3, and s' is a strict subtype of s.
5. If the analysis is being done under a separate mode of compilation, then add (s, s) to E_T for all extendable types s.
6. Delete t and all its edges if t is an immortal type.
7. Decompose $G_T = (V_T, E_T)$ into SCCs.
8. Define
$$rcyclic(G_T) = \{t | t \text{ lies on a nontrivial SCC of } G_T\}. \quad (14)$$
Then $V_T - rcyclic(G_T)$ is a set of acyclic run-time types.
9. Define
$$scyclic(G_T) = \{t | \exists t' \in rcyclic(G_T) \text{ such that } t' \text{ is a subtype of } t \in V_T\}. \quad (15)$$
Then $V_T - scyclic(G_T)$ is a set of acyclic static types.

Figure 8. Acyclic Type Analysis

3.1 The Type Connectivity Graph

A *type connectivity graph* $G_T = (V_T, E_T)$ has nodes representing types and directed edges summarizing the connections between type instances. If (u, v) exists in E_T, then an instance of exact or *run-time* type u could have an edge to an instance of run-time type v in some object graph. It isn't necessary to represent all types in G_T; only those whose instances can reside on the heap (like classes and *array types* [7] in .NET) need to be considered.[8] We call them *reference types* in accordance with .NET terminology.

The construction of G_T begins with the empty set for E_T. A directed edge (s, t') is added to E_T if s has a field with the *static* reference type t or if s is an array type whose elements have the static reference type t, and if t' is a subtype of t (t' could be t itself). The static type is the textually declared type. Hence in .NET, a field with the declared type `System.Object` would have the static type `System.Object`; its run-time type could be any class or array type since all of them derive from `System.Object`. A similar explanation is the reason for Step 3.

Edges also need to be added because of the subtyping relations that may exist on a field's containing type. In other words, if $(s, t) \in E_T$ because of some field `f` in s and if s' is a *strict* subtype of s (i.e., s' is a subtype of s such that $s' \neq s$), then the edge (s', t) would have to be added to E_T since `f` would also exist in s'.

3.2 Covariant or Invariant Subtyping of Arrays

Note that Step 4 bases its action on edges added in the previous steps due to fields. Specifically, if $(s, t) \in E_T$ where s is an array

[8] As an example, .NET value types can be excluded because their instances reside only on the stack. On the other hand, boxed value types and array value types should be included because their instances exist on the heap.

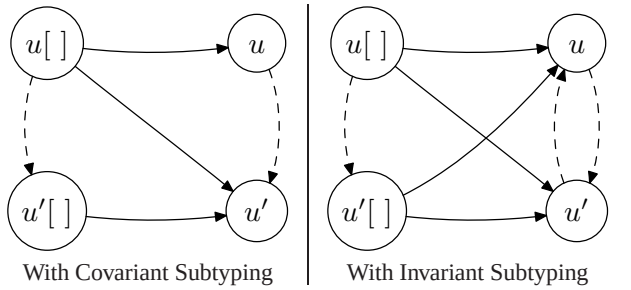

Figure 9. Array Subtyping Effects on a Type Connectivity Graph

type whose element is the reference type t, then it *doesn't* add edges of the form (s', t) where s' is a strict subtype of s. This is because we assume arrays to have a covariant subtyping [24] as in Java and C#; if arrays had an invariant subtyping instead, then Step 4 would have to be expanded to include all edges added in Step 2.

The aforementioned array subtyping effect on type connectivity graphs is illustrated in Figure 9. The nodes u, u', $u[\,]$ and $u'[\,]$ represent four types. The dashed arcs represent the subtyping relations among them. The solid arcs depict the possible connections between their corresponding instances in the object graph. Under covariant subtyping, the array type $u'[\,]$ is a subtype of the array type $u[\,]$ if u' is a subtype of u. Under invariant subtyping, $u'[\,]$ is a subtype of $u[\,]$ if u' and u are subtypes of each other.

3.3 Separate Compilation

If the analysis is done under separate compilation, then nodes in G_T representing extendable types have self loops added to them; in .NET, extendable types would be nonsealed classes and interfaces.

3.4 Omitting Immortal Object Types

If all instances of a reference type t are known to be immortal, then cycles involving instances of t can be ignored. Thus G_T can be made simpler by deleting t and all its edges.

3.5 Decomposition into Strongly Connected Components

Once G_T is built, it is decomposed into *strongly connected components* (SCCs) [10]. We call an SCC *nontrivial* if it contains two nodes u and v such that $(u, v) \in E_T$ (u and v could be the same). Consider $rcyclic(G_T)$ defined by Equation (14) as being the set of types that belong to a nontrivial SCC of G_T. Then since instances of these types *may* form cycles in some object graph, elements in $V_T - rcyclic(G_T)$ must be acyclic run-time types by definition.

Now consider the set $scyclic(G_T)$ defined by Equation (15) as consisting of those types w such that a subtype of w belongs to $rcyclic(G_T)$. Then a reference of type w could point to a cyclic data structure in some object graph. Therefore, $V_T - scyclic(G_T)$ is the set of all acyclic static reference types in V_T.

From Equations (14) and (15), it is clear that the set of acyclic run-time types is a superset of the set of acyclic static types. The former are used in the collector-side optimization of RC updates and the latter in the compile-time specialization of RC updates.

4. Eliding RC Updates on Immortal Objects

Certain data structures, such as vtables, string literals and garbage collection (GC) tables, live permanently in the static data area or the bootstrap memory and therefore needn't be subjected to the RC regime. This section outlines the design of an analysis that identifies references to such objects. We say that a local reference is an *immortal target variable* if it points to an immortal object. It is easy to extend this definition to cover value-type variables that embed references to immortal objects; we therefore consider only reference variables for the rest of the section's discussion.

4.1 Static Optimization

The analysis finds the immortal target variables' sets $\mathcal{I}_{in}(s)$ and $\mathcal{I}_{out}(s)$ before and after every statement s. The desired information is obtained by combining a simple type and field inspection with an iterative forward and backward data-flow analysis.

4.1.1 Forward Data Flow

For instance, suppose s is the IR instruction

$$y := checkcast\langle v\rangle(x)$$

that checks whether the reference x (of type u) points to an object of type v and if so, assigns it to the reference y whose type is a supertype of v.[9] Hence, the data-flow equation for s is

$$\mathcal{I}_{out}(s) = \mathcal{I}_{out}(s) \bigcup \begin{cases} \mathcal{I}_{in}(s) \cup \{y\} & \text{if } u \text{ is immortal,} \\ \mathcal{I}_{in}(s) \cup \{y\} & \text{if } v \text{ is immortal,} \\ \mathcal{I}_{in}(s) \cup \{y\} & \text{if } x \in \mathcal{I}_{in}(s), \\ \mathcal{I}_{in}(s) - \{y\} & \text{otherwise.} \end{cases} \quad (16)$$

In the above, u (or v) being immortal means that instances of u (or v) are always immortal. The equation assumes that references declared with an immortal type point only to immortal objects.

Another example is the `ldfld` instruction in MSIL (Microsoft Intermediate Language) [7], which in IR form can be written as

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

where `f` is the loaded field. Now if y is a reference and `f` a field that is ensured to always point to immortal data—not because of its type but by virtue of its containing class—then y will also point to immortal data.[10] The relevant data-flow equation here is

$$\mathcal{I}_{out}(s) = \mathcal{I}_{out}(s) \bigcup \begin{cases} \mathcal{I}_{in}(s) \cup \{y\} & \text{if } \mathtt{f} \text{ is immortal,} \\ \mathcal{I}_{in}(s) - \{y\} & \text{otherwise,} \end{cases} \quad (17)$$

where a field is immortal if it always targets immortal objects.

Note that Equation (17) doesn't have a case that tests whether the type w of the reference y is immortal. Such a test would be redundant. This is because if `f` isn't an immortal field, then its type w' can't be immortal. Since w must be a supertype of w' by the semantics of the `ldfld` instruction, then w also can't be immortal.

For the purposes of the analysis, a null reference is considered to target a special immortal object. Thus, for the IR instruction

$$y := \mathsf{null},$$

the data-flow equation is simply

$$\mathcal{I}_{out}(s) = \mathcal{I}_{out}(s) \cup \mathcal{I}_{in}(s) \cup \{y\}. \quad (18)$$

4.1.2 Backward Data Flow

Consider the `stfld` MSIL instruction, whose IR is

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

and suppose x is a reference and `f` an immortal field. Then x should point to an immortal object just before the statement. This

[9] The instruction throws an exception if the check fails.

[10] For instance, classes implementing a garbage collector may have fields pointing to permanently live data structures that are maintained in the bootstrap memory. In this situation, it may not be possible to identify the immortal nature of the fields' targets by a mere inspection of their types.

gives the following data-flow equation in the backward direction:

$$\mathcal{I}_{in}(s) = \mathcal{I}_{in}(s) \bigcup \begin{cases} \mathcal{I}_{out}(s) \cup \{x\} & \text{if } \mathtt{f} \text{ is immortal,} \\ \mathcal{I}_{out}(s) & \text{otherwise.} \end{cases} \quad (19)$$

Similar equations can be worked out for different cases of other IR instructions. The only requirement is that they form transfer functions that are monotonic on the underlying finite-height lattice. For example, it is easy to see that the transfer functions represented by Equations (16) to (18) are monotonic in the following way:

$$T_{\mathtt{o}}(\mathcal{I}_{in}(s)) \subseteq T_{\mathtt{o}}(\mathcal{I}'_{in}(s)) \qquad \text{if } \mathcal{I}_{in}(s) \subseteq \mathcal{I}'_{in}(s), \quad (20)$$

where $T_{\mathtt{o}}$ is the transfer function for the instruction with opcode $\mathtt{o}$.

4.1.3 Confluence Points

The immortal target variables' set at the entry of a basic block is

$$\mathcal{I}_{in}(B) = \mathcal{I}_{in}(B) \cup \big(\bigcap_{B' \in preds(B)} \mathcal{I}_{out}(B') \big), \quad (21)$$

where $preds(B)$ are the predecessors of the basic block B, $\mathcal{I}_{in}(B)$ corresponds to the $\mathcal{I}_{in}$ set for the first statement in B, and $\mathcal{I}_{out}(B')$ corresponds to the $\mathcal{I}_{out}$ set for the last statement in B'.

Equation (21) represents a forward transfer function between entire basic blocks. While a similar transfer function can be set up in the backward direction, our design presently doesn't propagate information backwards between basic blocks. Setting up such backward functions requires care because they apply in only limited scenarios—for example, immortal information from the beginning of a basic block B can't be simply propagated to the end of its predecessor basic blocks when B has more than one predecessor.

By starting with empty sets for $\mathcal{I}_{in}(s)$ (which are the $\mathcal{I}_{out}$ sets for previous statements) and then repeatedly applying the forward and backward transfer functions, we can arrive at a fixed point for the immortal target variables at all statements in the program's IR. RC updates on such variables can then be removed.

4.2 Run-Time Optimization

Obviously, the previous analysis won't detect all references to immortal objects. And yet, it would be wasteful to perform the general RC update operation on references that end up targeting immortal objects since the operations are moderately heavyweight (see Section 7). Consequently, our prototype reserves a bit in the RC field, set at object creation time, to flag objects whose exact type is immortal. This bit is tested on entry into an RC update operation and a quick exit is done if the test succeeds.

5. Specializing Non-Null Operand RC Updates

An RC update operation must first check whether its operand reference is non-null before attempting to increment or decrement the target object's reference count. If the operand reference is known to be non-null however, then a specialized version of the RC update can be generated that doesn't perform the null check. Observe that this optimization is complementary to the elimination of RC updates on null references described in Section 4.

Opportunities for this optimization arise from the null check semantics of many object-oriented instructions. For instance, given

```
1  y := x.f
2  RC+(x)
3  RC−(z)
4  z := x
```

where x is a reference and the instruction on Line 1 has the `ldfld` semantics of MSIL for reference operands [7], then x would have to be non-null at the beginning of Line 2. This is because Line 1 would have otherwise thrown an exception.

```
1   y := newobj()
2   x := newobj()
        ⋮
3   RC+(x)
4   RC−(y)
5   y := x
        ⋮
6   RC−(y)
7   y := null
        ⋮
8   ... := ... x ...
        ⋮
9   RC−(x)
10  x := null
```

Figure 10. An Aggressive but Erroneous Coalescing Opportunity

Another opportunity is in references returned by object creation instructions. As an example, given the code sequence

```
1  y := newobj()
2  RC−(y)
```

the RC_- operation can omit the null check on the reference y.

These opportunities can be formalized by forward data-flow equations as in Section 4. They can then be used to obtain the sets of non-null references at all statements by a fixed-point calculation.

6. Coalescing RC Updates

This optimization replaces a pair of RC increment and decrement operations on the same reference with a no-op. The current implementation is conservative in two ways: (1) an $RC_+(x)$ is canceled with a following $RC_-(x)$ only within a contiguous sequence of RC updates; and (2) coalescing opportunities are restricted to within basic blocks. While the reason for the latter was to keep the analysis simple, the reason for the former is that coalescing across longer stretches of code could lead to premature reclamation.

For instance, consider the IR shown in Figure 10 in which the inserted RC update operations realize a classic style of RC collection. The RC_+ and RC_- operations on Lines 3 and 4 are for the assignment on Line 5, and the RC_- operations on Lines 6 and 9 are for the assignments on Lines 7 and 10 respectively. Now, if the $RC_+(x)$ on Line 3 were coalesced with the $RC_-(x)$ on Line 9, the x on Line 8 would become a dangling reference.[11]

7. Inlining RC Updates

Besides the increment and decrement actions, the general RC updates perform other tasks like testing whether their operand reference is non-null, quick exiting on immortal objects (see Section 4.2), determining whether an object should be put into a PLC list or pulled out from it (this is done by examining a bit in the RC field as described in Section 3), establishing whether the reference count drops to zero and if so reclaiming the object, and checking whether the reclaimed object has associated data structures that also need to be reclaimed.[12] If the general RC update were therefore inlined into a mutator's code, it could cause considerable code bloat.

[11] As an aside, notice that neither x nor y is RC subsumed by the other.

[12] In .NET, *sync blocks* are created against objects on which a lock is held; these need to be recycled when the object is reclaimed [7].

Benchmark	Description	Execution Time (seconds)
`wc`	line and word count Unix tool run on two files	3.81
`go`	Game of Life, playing on a 40×19 board	16.04
`ijpeg`	conversions of an in-memory PPM bitmap image	0.38
`grep`	regular expression search on a 31,195-line file	24.63
`satsol`	Boolean formula satisfiability solver	15.39
`mandel`	graphical rendering of a Mandelbrot set fractal	1.89
`comp95`	file compression utility	29.10
`sort`	merge sort of a 1,048,576-integer array	4.10
`cmp`	file comparison tool run on two 1006KB files	3.07
`xlisp`	Xlisp interpreter executing `au`, `boyer`, `browse`, etc., as part of a workload of 21 Lisp programs	57.85

Table 1. C# Benchmarks

On account of the specializations in Sections 3 and 5, three other kinds of RC update operations are possible: those on static acyclic reference types, those on non-null references, and those on references that are both non-null *and* have a static acyclic type. The only extra tasks performed in the last of these is quick exiting on immortal objects (this happens in both the increment and decrement operations) and checking whether data structures held against a reclaimed object also need to be reclaimed (this happens only in the decrement operation). Because of their relatively lightweight nature, only they are inlined in our current implementation.

8. Results

Experiments were performed on a nongenerational nondeferred RC collection implementation for .NET, with and without the six optimizations described in this paper. The prototype was built under Bartok, an optimizing compiler and run-time system from Microsoft Research that translates MSIL into x86 binaries. The design is single threaded and handles various aspects of .NET like exceptions, object pinning, interior pointers and sync blocks [7]. As stated in Section 3, garbage cycles are reclaimed using trial deletion. To keep pause times bounded, the collected garbage is recycled using a delayed deallocation policy [32].

8.1 Platform Specifications

Table 1 lists the benchmarks considered, which are all single-threaded C# programs. The `go`, `ijpeg`, `comp95` and `xlisp` benchmarks are ports of SPEC CINT95 programs. The benchmarks are translated into MSIL by a front end and then into x86 code by Bartok. Their execution times were normalized against their execution times when Bartok's default collector was used, which are displayed in the last column of the table. This is a two-generation adaptive tracing collector that uses semispace collection in a 16MB nursery and semispace or sliding collection in the second generation depending on the available memory. The reported execution time numbers are an average of three runs of a total of four runs from which the first was discarded. (The runs were on a lightly loaded machine disconnected from the network.)

Comparisons were also made against a nongenerational deferred RC collection scheme. This collector shares much of the tuned run-time code base of the nondeferred RC collector, including the allocator and the code that realizes the trial deletion technique. A number of parameters are common to both. For example, both trigger a garbage collection when the delayed deallocation (DD) list exceeds 2^{15} objects. It means recycling a constant number of objects from the DD list in the nondeferred case, and additionally processing the zero-count table [13] in the deferred case. Both cases could also trigger the processing of the PLC list, which happens when the PLC list size exceeds 8MB. Optimizations on the compiler side are currently ongoing for the deferred RC collector.[13]

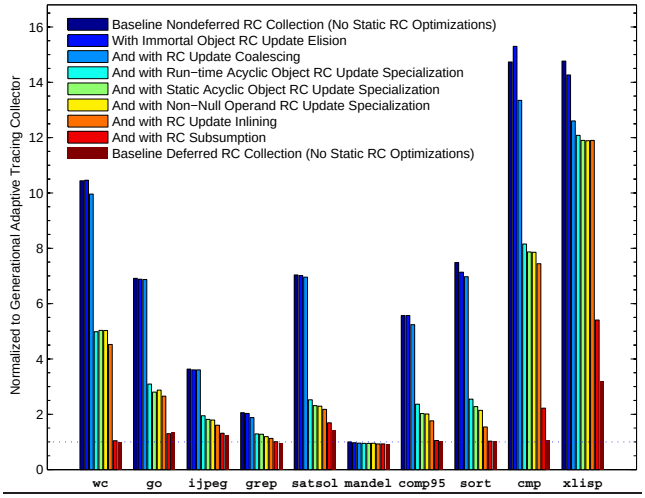

Figure 11. Impact of RC Optimizations on Execution Times

When compiling for any of the collectors, all other optimizations offered by Bartok, like common subexpression elimination, dead-code elimination and copy propagation, were turned on.

The platform was an HP XW8000 workstation with an Intel Xeon 2.8GHz CPU, running Windows XP Version 2002 (Service Pack 2) in hyperthreaded mode, and having 2GB of main memory, 512KB of secondary cache and 8KB of primary cache. Version 7.10 of `csc`, the .NET C# compiler, was used as the front end.

8.2 Relative Execution Times

Figure 11 displays the effects of the six optimizations on the nondeferred RC-collected execution times. The times are relative to execution times when the generational adaptive tracing collector is used. The first column for each benchmark is the time with none of the six optimizations active; this is the time for the baseline implementation. The eighth column is when all the optimizations are turned on. The second to the eighth columns are the progression in times as optimizations are successively switched on one after another. With all activated, overall times come within 32% of the tracing collector in seven out of the ten programs. This should be

[13] Some of the optimizations would have to be reworked for the deferred RC case. For example, RC subsumption as defined in Section 2 doesn't consider the subsumption of heap references. However, it could be adapted for reducing the stack scanning pressure in deferred RC collection. Also, RC update coalescing currently only combines RC updates on local references.

Benchmark	Counts ($\times 10^6$)									
	Before	Excluding RC Subsumption				Including RC Subsumption				$RC^{deferred}$
	RC_G	RC_G	RC_N	RC_A	RC_{NA}	RC_G	RC_N	RC_A	RC_{NA}	
`wc`	1242.2	414.1	0	207.0	621.1	0	0	0.8	0.4	4.8×10^{-3}
`go`	306.3	0	0	153.2	153.2	0	0	3.7	3.7	0.9
`ijpeg`	6.7	0	0	3.4	3.3	0	0	2.4	2.4	8.0×10^{-2}
`grep`	620.9	0	0	310.4	310.4	0	0	39.4	10.7	1.7×10^{-3}
`satsol`	808.8	36.1	25.8	347.6	358.0	10.3	0	200.8	211.2	13.9
`mandel`	9.7	2.8	0	0	0.7	4.1	0	0	0.7	1.4
`comp95`	2090.7	0	0	1045.4	1045.4	0	0	1.4×10^{-3}	4.0×10^{-4}	5.2×10^{-4}
`sort`	894.8	0	0	434.9	434.9	0	0	5.0×10^{-4}	4.0×10^{-6}	5.2×10^{-4}
`cmp`	1235.8	411.9	0	206.0	617.9	206.0	206.0	1.9×10^{-3}	2.0×10^{-4}	2.7×10^{-3}
`xlisp`	> 4295.0	> 4295.0	1797.6	0	0	1075.8	664.3	0	0	1039.0

Table 2. Distribution of Counts Across Different Kinds of RC Updates in RC "Hot" Methods

viewed in the context of baseline times that are at least thrice that of the tracing collector in eight programs. Hence, net improvements range from 7.7% in `mandel`, at least a factor of 2 in `ijpeg`, `grep` and `xlisp`, to at least a factor of 4 in the rest.

Each column from the second to the third, and from the sixth to the eighth, corresponds to one optimization. The fourth and fifth columns correspond to the acyclic object RC update specialization. In particular, the fourth column is when only the collector side of the optimization is in effect and the fifth column is when both the and collector side and static versions are in effect (see Section 3).

Notice that two optimizations—acyclic object RC update specialization and RC subsumption—speed up running times significantly. The former improves running times from about 1.01 in `mandel` to about 3.06 in `sort`. Improvements due to the latter range from about 1.00 in `mandel` to about 4.33 in `wc`. If a performance increase of at least 50% were considered, then the former achieves it in seven programs whereas the latter in six programs. However, bear in mind that the acyclic object RC update specialization optimization is a consequence of using trial deletion for cycle reclamation. If a tracing backup collector were used instead, then the baseline execution times would be closer to the fourth columns and RC subsumption would still continue to wield a dramatic effect.

It should be mentioned here that the execution time of 5.40 for `xlisp` (fully optimized nondeferred RC collection) is also an outlier on a performance suite comprising over 45 programs, on which the minimum, arithmetic mean, geometric mean and median were 0.21, 1.39, 1.20 and 1.01 respectively.

Observe that the immortal object RC update elision optimization produces a slight slowdown in `cmp`, from 45.23 seconds to 46.97 seconds. We suspect this to be due to data cache effects because the execution of an RC increment could cause an object to be loaded into cache lines much before its contents are accessed. Also observe that RC update inlining has a slightly negative effect on `xlisp`. This is possibly because of the increased code size, which influences the performance of the instruction cache.

The rightmost column for each benchmark indicates the running time under the deferred RC collection scheme. Without any RC compiler optimizations, the execution time of nondeferred RC collection is at least twice as much as deferred RC collection in nine programs. With all the optimizations switched on, this turns around to being within 19% of deferred RC collection in eight programs.

8.3 RC Update Profiles

An infrastructure that permits RC instrumentation code to be compiled into a binary has also been implemented. The instrumented code outputs the number of RC updates executed during a program run. The output gives the break up across different kinds of RC updates, and is sorted from the "hottest" to the coldest method from the point of view of the total number of RC updates performed. Table 2 shows the distributions of these counts, expressed in millions, for instrumented versions of the benchmarks. On each line, the RC_G (general), RC_N (non-null operand but not of acyclic static type), RC_A (operand of acyclic static type but not detected to be non-null) and RC_{NA} (non-null operand of acyclic static type) update counts are in the hottest method of a benchmark under nondeferred RC collection. The $RC^{deferred}$ column shows the number of RC updates in the hottest method of a benchmark under deferred RC collection. The third to the sixth columns show the distribution with all but the RC subsumption optimization active. The seventh to the tenth columns are with all optimizations switched on.

The numbers were obtained using 32-bit counters in the instrumented code. This sufficed for all cases except `xlisp`, where the RC_G count overflowed when RC subsumption wasn't active.

Notice that the sum of the RC updates in the third to the sixth columns is less than the second column in `satsol`, `mandel` and `sort`. This is due to the immortal object RC update elision and the RC update coalescing optimizations. In the other cases except `xlisp`, the sum equals the second column. (This is up to a round-off error, which shows up on `go`, `grep` and `comp95`.) The fact that these sums are drastically more than the corresponding sums of the seventh to the tenth columns in all but `mandel` confirms the significant benefit of the RC subsumption optimization.

8.4 Effects of Optimization Phase Ordering

The counter-intuitive behavior on `mandel` in Table 2 is an interesting one because it reveals how the ordering of the *static* optimizations can affect execution times. For the experiments in this paper, the ordering chosen was: (1) immortal object RC update elision, (2) RC subsumption, (3) RC update coalescing, (4) static acyclic object RC update specialization, (4) non-null operand RC update specialization, and (5) RC update inlining. There is no particular reason why this ordering was chosen; it only reflects the sequence in which they were implemented. For the most part, the individual optimizations don't interact with one another. For example, the specializations don't affect RC subsumption, and vice versa. However, coalescing could interact with RC subsumption in a way so that per-

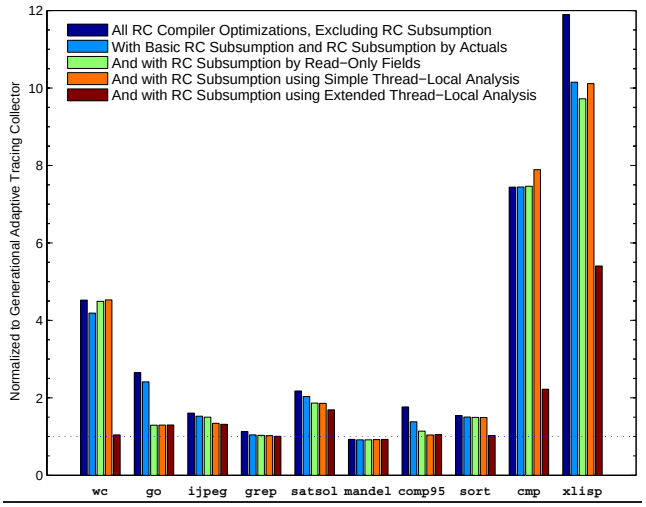

Figure 12. Effect of Overlooking Roots' Set on RC Subsumption

formance is actually *better* without RC subsumption. Essentially, when RC subsumption happens before coalescing, it may prevent the coalescing of a pair of RC updates that occur in a hot loop. This is the reason why the sum of the third to the sixth columns is less than the sum from the seventh to the tenth columns in `mandel`.

8.5 Influence of the Overlooking Roots' Set Computation

Figure 12 demonstrates how better approximations of the overlooking roots' set can appreciably improve the effectiveness of RC subsumption, all other optimizations being the same. The leftmost and rightmost bars for each benchmark are the execution times when RC subsumption is fully off and on; these therefore correspond to the seventh and eighth columns in Figure 11.

The second bar is the result of an RC subsumption in which $\grave{\mathbb{R}}(s, u)$ is only computed for statements of the forms $u := v$ and $z := \hat{z}$. This produces a performance improvement of about between 1% to over 20% relative to without any RC subsumption.

The third bar is after including statements of the form $u := v.\mathtt{g}$, where `g` is a read-only field. This contributes up to an additional 87% performance improvement, with the biggest happening on `go`.

The fourth bar is after including statements of the form $u := v.\mathtt{f}$, where $v.\mathtt{f}$ isn't written into before v dies, and v is inferred to point to a thread-local object using a simple thread-escape analysis. The analysis treats every static reference field as pointing to globally visible data and similarly marks any local reference variable that may alias such a field. The extension contributes up to another 12% improvement over the performance for the third bar.

The last bar is after considering the same statements as in the fourth bar, except with a more powerful thread-escape analysis that takes thread creation information into account. This contributes up to a factor of 4.33 in additional improvement. Of course, our benchmarks were single threaded to begin with, so we expect the improvements to be somewhere between the fourth and fifth bars in multithreaded applications. However, it should also be mentioned that we haven't yet fully capitalized on all opportunities. For instance, we are very conservative about writes and assume that $v.\mathtt{f}$ could be written into before v dies, if v is live at the end of a basic block and is written into somewhere else in the program.

9. Related Work

There has been a considerable amount of past research in RC collection. Much of it has centered on important collector-side techniques for handling concurrency, reclaiming garbage cycles and improving throughput [11, 3, 2, 18, 6, 23]. The relatively fewer efforts in the area of compile-time optimizations for RC collection in imperative-style procedural languages have focused on optimizing RC-related operations in deferred RC collection [5].

An analysis was outlined in [5] for cancelling adjacent RC increment and decrement operations. When an object allocation is followed by a reference creation (as is often the case), the analysis merges the RC update required for the reference creation event with the code sequence for the allocation event. Additionally, it takes care of directly returning an allocated object to the free list if all references to that object are subsequently lost. The RC update coalescing optimization of Section 6 corresponds to the cancelling part of this analysis. Three other optimizations were suggested in [5]: (1) removing RC updates on null references; (2) removing RC updates on data structures that are known to persist (like storage management tables); and (3) batching adjacent RC updates (called "transactions" in the paper) into a larger one. The first two correspond to the immortal object RC update elision optimization of Section 4. While [5] didn't report any performance data, Section 8 of this paper shows that at least the first two along with the cancelling optimization don't significantly affect the throughputs of nondeferred RC-collected object-oriented programs.

The use of reference counting for deterministic finalization is discussed in [30]. The work is in the context of .NET's Common Language Runtime (CLR). RC operations are performed only on references that live in the evaluation stack of the CLR. (The evaluation stack is a special data structure in the CLR's programming model.) The reference counting isn't necessarily deferred. The paper mentions the inlining of RC updates as an optimization. As future work, it also mentions that being more intelligent about which objects are reference counted could improve performance.

There have been a number of past efforts at optimizing RC updates in functional languages [16, 22]. These however exploit an operational semantics in which references can be updated at most once. For instance, the abstract interpretation approach in [16] relies on every occurrence of a reference in a function being consumed at most once in an activation. It isn't clear how applicable or effective these techniques would be in a programming paradigm that supports loops. Moreover, since approaches like those in [16] only compute reference counts approximately, they would have to be coupled with a backup storage reclamation method.

An abstract interpretation scheme for obtaining lifetime information on dynamically allocated data is presented in [12]. It can therefore be applied to optimizing RC updates. However, the analysis is for functional languages—only forward jumps are permitted and the effect of looping is achieved by function application.

In [22], an abstract interpretation scheme called *reference escape analysis* with similar aims as RC subsumption is presented. The objective is to determine when a reference "escapes", i.e., when its *value* is returned out from the function scope in which it is created. RC updates against non-escaping references can then be eschewed if they point to objects also targeted by escaping references. The analysis differs from RC subsumption in three main ways: (1) references can only be updated by direct assignment and not indirectly through devices like pointers; (2) it treats lifetimes at the granularity of procedures and doesn't consider finer grain intraprocedural lifetime information; and (3) the analysis was worked out for a toy language without loops.

The omission of RC updates on the formal parameters of a function when the actual parameters at the call site are used after the call was identified in [28]. RC optimizations aimed at an implementation of the OPAL functional language are described in [29]. These include specializing RC updates when the operand is known to be a reference, and eliminating RC updates when they are known to be non-references. [29] also discusses the possible reuse of storage

when the reference count of an object is about to fall to zero and a similarly sized object is allocated at a following point.

A run-time approach that reduces the number of RC updates in deferred RC collection was presented in [18]. The idea is based on the observation that RC updates needn't be applied on the intermediate values of a reference between two garbage collections.

An alternate approach to RC update elision is to adhere to a programming model that expressly creates and destroys references and that uses language constructs with explicit lifetime information [4].

The elimination of extraneous write barriers in concurrent garbage collection presents issues analogous to the identification of unnecessary RC updates. [31] discusses *covering conditions* for recognizing redundant write barriers. Although the conditions relate to the provisions of RC subsumption, [31] doesn't provide a scheme for automatically deducing them. Their potential was however demonstrated by examining dynamic program traces and showing that a large percentage of write barriers satisfy them.

Static analyses identifying writes to heap locations that contain null references before the write were described in [21]. The analyses were based on abstract interpretation, and considered object fields and array elements for the heap locations. The motivation was that write barriers for such operations could be removed.

A data structure similar to the live-range subsumption graph called the containment graph was described in [9]. Nodes in the containment graph denote live ranges (unlike the live-range subsumption graph where they represent local variables). A directed edge is inserted from a node j to a node i if i is live at a definition or use of j [9]. Thus the containment relation in [9] is different from the live-range subsumption relation; a simple example showing the difference is when the live ranges of two distinct variables x and y only partially overlap. The purpose of the containment graph was to reduce spill code costs through the splitting of live ranges.

Optimizations to eliminate RC update operations are also beneficial in areas like programming for Microsoft's Common Object Model (COM) [25]. Opportunities similar to RC subsumption have been known for a long time in COM. However, these opportunities have to be exploited by the programmer, and the model only provides guidelines that assist in their realization.

10. Summary

This paper presented a set of six optimizations that dramatically reduce the execution time of a nondeferred RC collection scheme. The reduction is chiefly realized by a new analysis and optimization called RC subsumption that finds and removes a particular kind of redundancy in RC updates. The reported measurements show that throughput increases due to RC subsumption alone can be significant, often by at least 50%. When combined with the other optimizations, this makes it possible to achieve nondeferred RC collection times that are on a par with both a deferred RC collector and an advanced tracing collector on a number of programs.

In fact, we believe that there is more scope for improvement, given that techniques like the division of the heap into generations would be complimentary to RC collection.

While the work demonstrates that nondeferred RC collection can be made much more efficient than what has been believed in the past, there is still the question of how the optimizations would fare on multithreaded code. Issues for further investigation include extending RC subsumption to heap references, and whether there is a commonality of structure to the optimizations presented in the paper. At this point, it at least appears that RC subsumption has potential beyond RC collection—for instance, to reduce the stack scanning pressure in a concurrent collector.

Acknowledgments

I would like to thank David R. Tarditi, Mark Plesko, Bjarne Steensgaard, James R. Larus, Benjamin G. Zorn and John DeTreville for helpful discussions on this work. I would also like to thank David R. Tarditi, James R. Larus, Benjamin G. Zorn, John DeTreville and Francesco Logozzo for feedback on drafts of this paper.

The first version of the deferred RC collector was implemented by Ting Yang from the University of Massachusetts, Amherst.

References

[1] Association for Computing Machinery. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2000.

[2] David F. Bacon, Clement R. Attanasio, Han B. Lee, V. T. Rajan, and Stephen Smith. Java without the Coffee Breaks: A Nonintrusive Multiprocessor Garbage Collector. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 92–103. Association for Computing Machinery, June 2001.

[3] David F. Bacon and V. T. Rajan. Concurrent Cycle Collection in Reference Counted Systems. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 207–235. Springer-Verlag, June 2001.

[4] Henry G. Baker. Minimizing Reference Count Updating with Deferred and Anchored Pointers for Functional Data Structures. *ACM SIGPLAN Notices*, 29(9):38–43, September 1994.

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

[6] Stephen M. Blackburn and Kathryn S. McKinley. Ulterior Reference Counting: Fast Garbage Collection without a Long Wait. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 344–358, October 2003.

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

[8] George E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[9] Keith D. Cooper and L. Taylor Simpson. Live Range Splitting in a Graph Coloring Register Allocator. In *Proceedings of the 7th International Conference on Compiler Construction*, volume 1383 of *Lecture Notes in Computer Science*, pages 174–187. Springer-Verlag, March 1998.

[10] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Electrical Engineering and Computer Science Series. The MIT Press, Cambridge, MA 02142, USA, 1995.

[11] John DeTreville. Experience with Concurrent Garbage Collectors for Modula-2+. Technical Report SRC-RR-64, Digital Systems Research Center, November 1990.

[12] Alain Deutsch. On Determining Lifetime and Aliasing of Dynamically Allocated Data in Higher-Order Functional Specifications. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 157–168. Association for Computing Machinery, January 1990.

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

[14] 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.

[15] 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* [1], pages 334–344.

[16] Paul Hudak. A Semantic Model of Reference Counting and its Abstraction (Detailed Summary). In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 351–363. Association for Computing Machinery, April 1986.

[17] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Inc., New York City, NY 10158, USA, 1996.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] Harel Paz, Erez Petrank, David F. Bacon, Elliot K. Kolodner, and V. T. Rajan. An Efficient On-the-Fly Cycle Collection. In *Proceedings of the 14th International Conference on Compiler Construction*, volume 3443 of *Lecture Notes in Computer Science*, pages 156–171. Springer-Verlag, April 2005.

[24] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, Cambridge, MA 02142, USA, 2002.

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

[26] Paul Rovner. On Adding Garbage Collection and Runtime Types to a Strongly-Typed, Statically-Checked, Concurrent Language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, July 1985.

[27] Erik Ruf. Effective Synchronization Removal for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* [1], pages 208–218.

[28] 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.

[29] Wolfram Schulte and Wolfgang Grieskamp. Generating Efficient Portable Code for a Strict Applicative Language. In *Proceedings of the PHOENIX Seminar and Workshop on Declarative Programming*, pages 239–252, November 1991.

[30] Chris Sells and Christopher Tavares. Adding Reference Counting to the Shared Source Common Language Infrastructure. At `http://www.sellsbrothers.com/writing`.

[31] 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.

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