A Principled Approach to Nondeferred Reference-Counting Garbage Collection

Pramod G. Joisha
pjoisha@microsoft.com

August 2007

Technical Report
MSR-TR-2007-104

Nondeferred reference-counting (RC) garbage collection is among the oldest memory-management methods. Despite offering unique advantages, little attention has been paid on how to correctly implement it for modern programming languages. This paper revisits this collection method and describes how to implement it for a modern object-oriented language in an optimizing compiler. The main contribution is a general algorithm that realizes one form of nondeferred RC collection for an object-oriented language having features such as exceptions, interior pointers, and object pinning. The algorithm abstracts the pertinent characteristics of instructions using concepts from data-flow analysis, such as def/use information, so that instructions are handled in a uniform manner, instead of in an ad hoc or special-case way. The abstracted information is used to systematically compute what increments and decrements to do, even in the presence of subtle conditions such as exceptional control flow. These techniques enabled us to compile a large suite of programs to use nondeferred RC collection. The paper also discusses the modifications that were necessary in the compiler for supporting the inserted RC operations, and reports measurements from a reference implementation.

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
http://www.research.microsoft.com

1 Introduction

Nondeferred reference counting is one of the original forms of automatic resource management. In the classic version, a count is kept against each managed resource, indicating the number of references to it [15, 11]. The count is increased and decreased as references to the resource are created and destroyed. The resource becomes reclaimable as soon as its count drops to zero.

The approach has notable advantages over other forms of resource management. It is incremental in time and space, having a spatial locality no worse than the mutator [22]. It can be implemented without the need for stack scanning and garbage collection (GC) maps. It can promptly reclaim memory, which facilitates immediate finalization and immediate reuse. These features make it especially attractive in situations where a low memory footprint is desirable, such as virtual machines for embedded devices. However, it has rarely been used in practice because of problems such as cycle reclamation and fragmentation, and primarily because of the high cost of maintaining up-to-date reference counts.

We have been investigating whether nondeferred reference-counting (RC) garbage collection can be made practical, in light of advances in compiler analyses. By "nondeferred", we mean any RC technique that has 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. Thus, while classic (or standard) reference counting is nondeferred, Deutsch and Bobrow's reference counting (called *deferred*) is not [13]. This definition does not imply perfect immediacy of reclamation. A spectrum of nondeferred schemes is possible, each differing in how promptly dead data is reclaimed. The reclamation of dead data is with respect to the interface between the mutator and the collector's run-time system. It is independent of *when* the run-time system returns the reclaimed data back to the allocator.

We recently published compiler optimizations that eliminate a significant number of RC updates in nondeferred RC collection [20, 21]. This paper addresses another obstacle to demonstrating the practicality of the approach. It shows how a compiler can systematically convert an object-oriented program, with modern language features, into one that uses the approach.

The paper describes solutions to the main problems in designing such a conversion algorithm: (1) How should instructions be processed, so that object-oriented instruction sets can be handled in a uniform way? (2) How should RC updates be inserted so that dead data is reclaimed as early as possible? (3) How should modern language features, such as exceptions, interior pointers and object pinning, be supported? The first is solved by identifying instruction characteristics germane to nondeferred RC collection, and abstracting them using concepts from data-flow analysis. The second and third problems are solved using liveness techniques.

The use of liveness in GC is not new [14, 29, 28, 1, 2, 18]. However, unlike this effort, all past work considers it for a tracing collector. Specifically, past efforts perform a live-range analysis to determine the live roots at each GC-safe point. Only data reachable from these live roots is retained. In this work, liveness is used to compute the *death points* of references, so as to reclaim dead data as soon as possible. This is an important distinction, and arises from a basic difference between tracing and reference counting: one looks at live matter and the other at dead matter [5].

There are complications when dealing with reference deaths that do not exist when dealing with live roots. A challenging one is modeling the effects of exceptions. Previous work for tracing collectors did not have to specially treat exceptions, beyond perhaps representing them using factored edges in the control-flow graph (CFG) [10]. And they do not pose a problem to deferred RC collectors because only heap references are counted. For nondeferred RC collection, however, the conversion algorithm must insert RC updates at the appropriate CFG points so as to ensure that when an exception is thrown, data no longer accessible is reclaimed.

The conversion algorithm is intraprocedural, based on only a local analysis. Hence, it is usable in both a static setting—either in whole-program or separate compilation modes—as well as a dynamic setting, such as compiling just-in-time in a virtual machine. It is presently implemented in an optimizing compiler from Microsoft Research called Bartok, and has been successfully used to compile numerous large programs, including Bartok itself, into nondeferred

RC-collected versions. Measurements demonstrating the single-thread performance of some of these converted programs have been reported elsewhere [20, 21]. This paper presents data relating to the algorithm itself. It also discusses a number of compiler issues that were addressed: (1) How should RC updates be represented so that downstream phases preserve their invariants? (2) What effect do phases like method inlining have on the algorithm? (3) How should code in the run-time system be handled?

The rest of the paper is organized as follows. Section 2 presents the algorithm for implementing one form of nondeferred RC collection. We call this version ARCS (Anticipatory Reference Counting for the Stack) because it eagerly applies decrements on local references using liveness. A step in the algorithm involves intercepting the values of live references whose death points are inaccessible. Section 3 describes the computation of these references in the presence of exceptional control flow. Issues relating to the representation of RC updates are elaborated in Section 4. Section 5 shows how the eagerness of reclamation can be traded with code quality. Section 6 explains the special treatment of methods outside the scope of the algorithm, such as some in the run-time system. Section 7 gives data from an implementation of the algorithm. Last, Section 8 discusses related work and Section 9 concludes.

2 A Conversion Algorithm for ARCS Collection

This section describes a compiler phase called *RC update insertion* that automatically converts a program into one that uses ARCS collection. ARCS collection neither requires GC maps nor the scanning of stacks. The phase determines program points at which RC updates should be inserted so as to safely decrement reference counts as early as possible. An "RC update" is an RC increment or decrement on an object targeted by a reference r. It is denoted as $RC_+(r)$, $RC_-(r)$ or $\widehat{RC}_-(r)$, and is a no-op if r is null.

Three main issues had to be solved in designing this phase. The first was figuring a common abstraction for all the instructions so that the insertion algorithm could be expressed in general terms. As Section 2.3 will show, the same algorithm can handle both an instruction like `getfield` in Java [23] and a compare-and-exchange kind of instruction. The second issue was the treatment of reference deaths in the presence of language features such as exceptions. The third was the handling of special pointers into the interior of objects, which the garbage collector is required to honor.

2.1 Preliminaries

The phase operates on the program's intermediate representation (IR) through a series of three stages. In the first, it is converted to a canonical *factored control-flow graph* (FCFG) [10]. A live-range analysis is then performed to determine reference death points in the FCFG. Finally, the IR is transformed by inserting RC increments just after definition points, and RC decrements just after either definition or death points. The decrements are inserted just after death points if we can statically determine last use via liveness information; otherwise, they are inserted just after definition points.

The insertion phase could be run at various places in a pipeline of phases—when the IR is at a high level, a medium level or after it has been lowered into native code. In our implementation, it is applied on a high-level IR. This facilitates optimizations that may not be easily identifiable in other IRs [20]. This of course means that the downstream phases will have to preserve the invariants that the inserted RC updates impose.

2.1.1 Supported Language Features

At the IR level, we assume two kinds of pointers relevant to garbage collection: *references* that resemble the object references in Java [16] and C# [17], and *interior pointers* that resemble the managed pointers in .NET [9]. Interior pointers are similar to conventional pointers in that they

$$
\begin{aligned}
S &::= \boldsymbol{T} := A \\
A &::= \&\boldsymbol{R} \mid \&\boldsymbol{R}[\boldsymbol{I}] \mid \&(W.\boldsymbol{F}) \mid \&.\boldsymbol{F} \mid \boldsymbol{T} \pm \boldsymbol{I} \\
W &::= \boldsymbol{R} \mid \boldsymbol{T}
\end{aligned}
$$

$\boldsymbol{R} \in$ set of local reference variables,
$\boldsymbol{T} \in$ set of local interior-pointer variables,
$\boldsymbol{F} \in$ set of static and instance fields,
$\boldsymbol{I} \in$ set of integer-valued expressions.

Figure 1: The syntax of interior-pointer definitions.

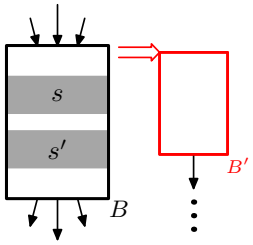

Figure 2: A basic block with an exception header block in an FCFG. Normal arcs are shown as incoming and outgoing arrows. Exception arcs are shown as block arrows.

are dereferenceable. However, they are associated with strong typing information, and can only be used in a few well-defined ways. For this paper, the syntax S of their definitions is given by the grammar in Figure 1.

In the grammar, the "member access" operator ('.') extracts a field, given a reference or an interior pointer to the field's container. The "address of" operator ('&') returns the address of a local reference, array element or field. Thus, $\&(W.\boldsymbol{F})$ is an interior pointer to a field, and $\&.\boldsymbol{F}$ is an interior pointer to a static field.

The grammar restricts interior pointers to reside in local variables. It does not permit the assignment of function-returned interior pointers to local variables. (They can however be passed into functions.) These restrictions are in line with those in MSIL, the low-level common language in .NET [9];[1] they do not indicate a limitation of the insertion approach.

Interior pointer and reference variables can also carry an attribute called pinned. This prevents the garbage collector from reclaiming (or moving) target objects until the interior pointer or reference variables are redefined, or until their lexical scopes end [9].

Last, statements can throw exceptions. Exceptions can be explicit (as with the throw statement), or implicit.

2.1.2 The Factored Control-Flow Graph

The insertion phase transforms an FCFG representation of a function. As in a standard control-flow graph, nodes in an FCFG represent basic blocks. But its arcs can be of two types: *normal arcs*, which denote the normal flow of control from the end of one basic block to the beginning of another, and *exception arcs*, which represent the flow of control from *anywhere* within a basic block to the header block of an exception handler. As an example, Figure 2 shows a basic block B containing two statements s and s', each of which could throw an exception. The exception is processed by a handler having the header block B'. Although the figure shows only one handler,

[1] MSIL stands for Microsoft Intermediate Language.

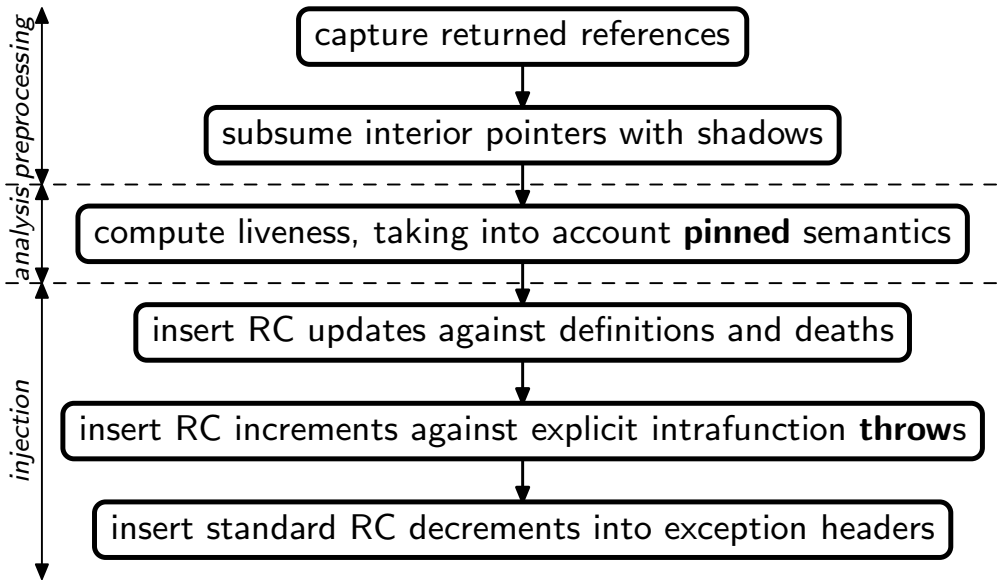

Figure 3: The six steps of the RC update insertion phase, when the phase operates on a high-level IR. The steps are organized into the preprocessing, analysis and injection stages.

a basic block could have multiple exception arcs leaving it, each to the header block of a different handler.

Exception header blocks contain a special statement called an *exception assignment* that catches and assigns the thrown exception to an *exception variable*. The IR represents this statement as

$$c := \mathit{catch}(),$$

where c is the exception variable.

2.2 The Three Stages of the RC Update Insertion Phase

Figure 3 displays the three stages of the insertion phase. The first preprocesses the IR to a normal form so as to simplify subsequent stages. A traditional live-range analysis, modified to model the pinned semantics, is performed in the second stage. The third stage introduces RC updates for local and heap references; their placement is guided by the previously derived liveness information.

2.2.1 Preprocessing Stage

This stage has two steps. The first replaces statements of the form

$$F(x, y, \ldots),$$

where F returns a reference, with statements of the form

$$\dot{t} := F(x, y, \ldots),$$

where $\dot{t}$ is a compiler-generated temporary. This ensures that subsequent stages do not have to account for memory leaking because of uncaptured references returned by functions.

The second step pairs every interior pointer with a compiler-generated local reference called a *shadow*. The objective is to ignore the definitions and deaths of interior pointers in subsequent stages. (Later stages would still have to handle writes of references into the stack and heap through interior pointers.) This is accomplished by introducing definitions and uses of an interior pointer's shadow so as to tightly contain the interior pointer's lifetime.

Shadowing Interior-Pointer Definitions For instance, if $\tilde{p}$ is the shadow of an interior pointer p, then the following is the transformation for a definition of p that points it into an array object:

$$p := \&r[e] \quad \Longrightarrow \quad \begin{array}{l} \tilde{p} := r \\ p := \&r[e] \end{array}$$

From Section 2.1.1, note that r must be a local reference. If p were defined to point into the stack (say, by assigning the address of r to it), then the following would be the code produced:

$$p := \&r \quad \Longrightarrow \quad \begin{array}{l} \tilde{p} := \mathsf{null} \\ p := \&r \end{array}$$

To handle definitions involving offset calculations on interior pointers, the compiler inserts basic blocks with the following code:

$$p := q \pm e \quad \Longrightarrow \quad \begin{array}{l} w := (q \pm e) - \tilde{q} \\ \mathsf{if}\ w \geq 0 \wedge w < sz, \\ \quad \tilde{p} := \tilde{q} \\ \mathsf{else} \\ \quad \tilde{p} := \mathit{findstart}(q \pm e) \\ \mathsf{end} \\ p := q \pm e \end{array}$$

In the transformed code, $\tilde{p}$ and $\tilde{q}$ are the shadows of the interior pointers p and q, e is an integer-valued expression, and sz is the statically determined size of the object pointed to by $\tilde{q}$. The code uses a service called *findstart* provided by the allocator. If p points into an object in the heap, $\mathit{findstart}(p)$ returns a reference to the start of the object. Otherwise, it returns null. Such a service could be built using a run-time technique like crossing maps [3].

If the expression $q \pm e$ typically arises from intra-object data accesses, then the predicate $w \geq 0 \wedge w < sz$ will be true in most cases. Since the code's else branch would then be the infrequent path, the cumulative cost of using *findstart* will not be appreciable. [2]

Shadowing Interior-Pointer Uses The shadowing step also introduces uses of shadows, so as to tightly subsume the lifetimes of the corresponding interior pointers. In the case of interior pointers used in a non-call instruction, this is done by creating fake uses of their shadows after the instruction:

$$\ldots := \ldots\ p\ \ldots \quad \Longrightarrow \quad \begin{array}{l} \ldots := \ldots\ p\ \ldots \\ \mathit{fakeuse}(\tilde{p}) \end{array}$$

In the case of interior pointers passed into a function, the function's signature is rewritten to include the shadow:

$$r := F(p, \ldots) \quad \Longrightarrow \quad r := F(p, \tilde{p}, \ldots)$$

The *fakeuse* operator is later lowered into a no-op.

2.2.2 Live-Range Analysis Stage

Let $\mathit{defs}_{must}(s)$ and $\mathit{uses}_{may}(s)$ be the sets of local references that must be defined and that may be used in a statement s. Then the following relates local references live before and after s [24]:

$$\mathit{live}_{in}(s) = (\mathit{live}_{out}(s) - \mathit{defs}_{must}(s)) \cup \mathit{uses}_{may}(s). \tag{1}$$

We say a local reference *dies across s* if it belongs to the set

$$\mathit{dieacross}(s) = (\mathit{live}_{in}(s) \cup \mathit{defs}_{must}(s)) - \mathit{live}_{out}(s). \tag{2}$$

[2] Under an optimization for the accessing of two arrays at the same index, q and $q \pm e$ can fall on two different objects [14]. In MSIL, this would require the pinned attribute on both p and q, which is equivalent to using the `fixed` construct on both the arrays at the C# source level [17].

These are references that can be decremented just after s, without turning uses further down the control-flow path into dangling references. Supplementary decrements may be needed, however, to prevent memory leaks. Specifically, additional decrements are needed if (1) references are both defined in s and live on entry to it, or (2) references other than those in $defs_{must}(s)$ are defined in s (say, through interior pointers). Section 2.2.3 shows a decrement sequence that covers all these cases.

Modeling the pinned Semantics A decrement cannot be inserted after the last use of a pinned reference r because the object that it targets must be held until its redefinition, or until the end of its lexical scope. Simply treating r as live throughout a function is insufficient, because a decrement is needed just before each of its redefinitions. Our solution is to extend r's live ranges so that they span the definition points of r and reach the end of the function's body. This can be done by introducing a fake use of r at two places: (1) each statement s whose $defs_{must}$ set contains r:

$$uses'_{may}(s) = uses_{may}(s) \cup \{r\} \qquad \text{if } r \in defs_{must}(s), \tag{3}$$

and (2) ends of basic blocks that return control from the function.

Liveness of Thrown References Besides returned references, references that are expressly thrown (using the throw statement) from basic blocks lacking a handler for the exception are considered live on exit from the function. If a basic block could implicitly throw an exception and no handler already exists for it, a default one is created that simply re-throws the exception via a throw.

2.2.3 Injection Stage

RC updates are inserted by this stage in three steps using liveness information. The first step injects increments for reference definitions, and decrements for references deaths or redefinitions. (The statements processed are those that exist before this stage.) The second step injects increments against explicitly thrown references having exception handlers in the function. (This can be established by comparing the static type of the thrown reference with the static type of the exception variable.) The increment is injected just before the throw. The third step introduces decrements for local references that die in a basic block into that block's exception header (if it has one). It is important that the last two steps occur after the first; doing so guarantees that in an adjoining pair of RC updates on the same reference, the increment always precedes the decrement.

Injection Step 1: Inserting RC Updates against Definitions and Deaths This step is best explained by describing its net effect on statements. Depending on whether the statement is a call or non-call instruction, the net effect is one of the code patterns shown in Figures 4 and 5. In these figures, and all other code in this paper, an RC_+ indicates an increment, and an RC_- and $\widehat{RC}_-$ a decrement. The difference between RC_- and $\widehat{RC}_-$ is that the former is inserted on the basis of reachability changes in the object graph and the latter on the basis of liveness information. We therefore refer to RC_- as a *standard decrement*, and to $\widehat{RC}_-$ as an *eager decrement*. (We use the term "standard" because decrements done in standard versions of reference counting are of this type [22].)

Non-Call Instructions Let $ldefs(s)$ be the set of *l-value* expressions [24] of all references (stack and heap) that *may* be defined in a statement s. Let $\mathscr{L}(Q)$ be the set of l-values for variables in a set Q. Then, when s is not a call, the code after injection corresponds to one of the two templates in Figure 4. The two templates are necessary because the compiler's knowledge of what *will* be defined may not be the same as what *could* be defined. When its information on

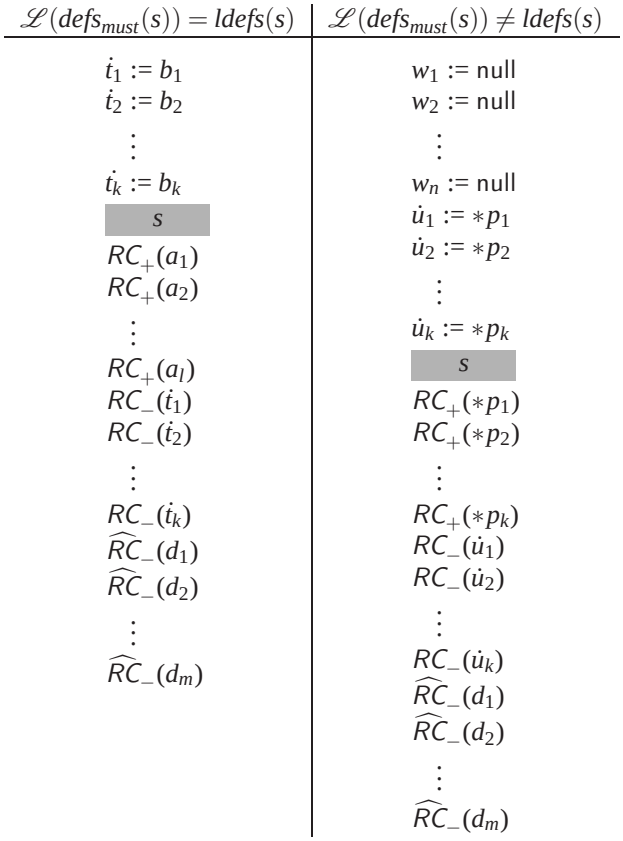

Figure 4: Templates of code produced by the insertion phase, after processing a non-call instruction s. Code matches either template, depending on whether $\mathscr{L}(defs_{must}(s))$ equals $ldefs(s)$.

the two match, more efficient code can be generated in accordance with the template on the left. Otherwise, the compiler will have to generate code according to the template on the right.

In the two templates, as well as elsewhere in this paper, dot accents are used to represent temporaries. The other variables belong to the following sets of references:

$$\begin{aligned} &a_i \in defs_{must}(s), \\ &b_i \in defs_{must}(s) \cap live_{in}(s), \\ &d_i \in dieacross(s), \\ &w_i \in defs_{must}(s) - uses_{may}(s), \\ &p_i \in ldefs(s). \end{aligned} \tag{4}$$

The compiler produces code according to the template on the left, if it can determine that $\mathscr{L}(defs_{must}(s))$ and $ldefs(s)$ are equal.[3] The first set of RC updates it generates are increments against a_i, which are references that must be defined in s. The next set of RC updates are decrements for *segueing live references*. These are live references that can transition to a redefined state by going through an inaccessible death point. Their values before s are therefore captured in $\dot{t}_i$, for doing decrements after s. Segueing live references are described further in Section 3. The last set of RC updates are decrements against references that die across s.

Code corresponding to the second column of Figure 4 is produced when $\mathscr{L}(defs_{must}(s))$ and $ldefs(s)$ cannot be determined to be equal. The assignments to $\dot{u}_i$ capture values of references

[3] It may appear that $ldefs(s)$ could be written as $\mathscr{L}(defs_{may}(s))$, where $defs_{may}(s)$ is the "may" analogue of $defs_{must}(s)$. This, however, is not true because $defs_{must}(s)$ traditionally only contains local variables.

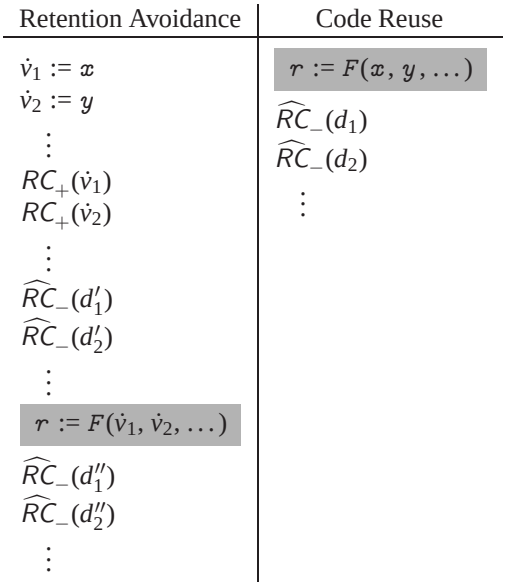

Figure 5: Templates of alternative code sequences for the call instruction $r := F(x, y, \ldots)$. The choice of template depends on decisions regarding issues such as storage retention, instruction cache performance and implementation complexity.

potentially to be overwritten in s, by applying the dereference operator ('$*$') on l-value expressions in $ldefs(s)$. Decrements are applied on them, after increments are applied on references that may be defined in s.[4] Decrements against the w_i would exist earlier, because they die before their redefinition in s. They are hence assigned null to preclude double decrements due to the $RC_-(\dot{u}_i)$. If an alias analysis can prove that a p_i will point to a w_q, then the statements $w_q :=$ null, $\dot{u}_i := *p_i$ and $RC_-(\dot{u}_i)$ can be omitted.

The decrements against the $\dot{u}_i$ are standard decrements, since they come from references being overwritten. Those against the d_i are eager decrements because they arise from references dying.

It should be mentioned that to insert the eager decrements, new basic blocks may have to be created. This will be the case when s is the last statement in a basic block with two or more outgoing arcs, and references die along one arc but not another.

Call Instructions The treatment is different when s is a call instruction, and corresponds to one of the two templates in Figure 5. The templates have different goals: the one on the left avoids unnecessary storage retention across a function call, while the one on the right requires fewer RC updates to be inserted. In particular, if the Retention Avoidance (RA) template is used and an actual reference parameter is used last in s, then the storage targeted by it will be freed before the call returns, assuming no other references alias it. The Code Reuse (CR) template replaces all caller-side increments with a single set of increments on the callee side.

The RA template realizes its goal by splitting the decrements into two groups, one that occurs before the call and one after it. The first group consists of decrements on d'_i, where

$$d'_i \in (dieacross(s) \cap uses_{may}(s)) - addrtaken(s). \tag{5}$$

The set $addrtaken(s)$ consists of references whose addresses are passed into F; an example of such a reference is the local variable z, if $\&z$ is passed into F. These references are considered

[4] If a reference does not actually get assigned to a location pointed to by p_j $(1 \leq j \leq k)$, then the $RC_+(*p_j)$ and $RC_-(\dot{u}_j)$ cancel out.

live for the entire duration of the call. The second group consists of decrements against the remaining references that die across s:

$$d_i'' \in \mathit{dieacross}(s) - (\mathit{uses}_{\mathit{may}}(s) - \mathit{addrtaken}(s)). \tag{6}$$

Increments against the $\dot{v}_i$ account for the actual-to-formal copying of reference parameters at the time of function invocation. Because there will be decrements inside F against their formal counterparts (unless these formal counterparts are thrown or returned), no decrements should be applied on the $\dot{v}_i$ after F returns.

In the RA template, the original reference arguments are replaced by temporaries in the transformed call instruction. This is because the d_i' will be these arguments, and the $\widehat{RC}_-$ operations occurring before the call kill these d_i'. (The reason for this is explained in the discussion for the third step in this stage.)

In the CR template, all of the inserted decrements come after s, against the references that die across it (Equation (4) defines the d_i). No other RC updates or assignments are inserted on the caller side. For the transformed sequence to work, increments must be inserted on the callee side, against the formal reference parameters.

In both the templates, interior pointers passed into the call need no special consideration because they are indirectly taken care of through their shadows (see Section 2.2.1). No increment is applied against the returned reference because an increment would have already occurred when it is defined in F. (As stated in Section 2.2.2, returned references are considered live on exit from a function. Therefore, they will not have a decrement after their last definition.)

Apart from function calls, exception assignments (refer Section 2.1.2) and allocation statements are classified as call instructions by this stage. An example of an allocation statement is

$$r := \mathit{allocobj}(T),$$

where r is a reference to a newly allocated, unconstructed object of type T. (Only the vtable and RC fields are set up in such an object; all other fields contain zeros.) An $RC_+(r)$ is not inserted here, because objects begin life with a reference count of 1.

There is a subtle issue when generating code in accordance with the RA template. There may be statements in the IR whose designation as a call or non-call instruction is unknown when the insertion phase runs. An example is the $\mathit{monitorenter}(x)$ statement, which enters a monitor for an object. A downstream phase could either lower it into a compiler intrinsic or a call. If the downstream phase handles it opposite to the way the insertion phase treats it, then the generated RC update sequence could produce incorrect counts. A solution is to always treat such statements as non-call instructions in the insertion phase, and to produce an RC-instrumented version of the called function in which increments exist against the formal references. The lowered call would then have to be to this instrumented version. This is where the CR template offers the advantage of a simpler implementation because no a priori knowledge is needed on how an instruction gets subsequently lowered. Ambiguous statements can always be considered non-call instructions and no special handling is required in the downstream phases.

Another difficulty to using the RA template is that important subsumption-based optimization opportunities become somewhat harder to detect [20, 21]. This is because the increment, decrement pair for a function's reference argument straddles the function's boundary. For these two reasons, our implementation presently uses the CR template. Of course, with the CR template, there is a small chance that garbage will be held longer than even a standard tracing collector. But we have not seen this occur in any of our tests.

Injection Step 2: Inserting Increments against throw Exceptions that are explicitly thrown from basic blocks without exception handlers are treated the same way as returned references. That is, no decrement against the thrown reference exists after its last definition, because it is considered live on exit from the function. This is why exception assignments are regarded as call

instructions, because this avoids an increment against the exception variable when the exception is caught further up the call stack.

On the other hand, when explicitly thrown references are caught in the same function, the absence of an increment against the exception variable at the exception assignment point must be countered by an increment at the point of the `throw` statement, or earlier. Hence the second step in this stage.

Injection Step 3: Inserting Decrements into Exception Headers If the statement s in Figures 4 or 5 was to throw an exception, none of the ensuing RC updates will get executed. The increments among them should not happen anyway, because an exception-throwing s does not side-effect a program's variables. However, among the decrements, those against references that die across s should be performed. Therefore, since any statement in a basic block B could throw an exception, the third step inserts decrements against references in the set

$$D' = \left(live_{in}(B) \cup \left(\bigcup_{s \in B} defs_{must}(s)\right)\right) - live_{in}(B') \tag{7}$$

into B's exception header B'. (The sets $live_{in}(B)$ and $live_{in}(B')$ in Equation (7) are references that are live on entry to B and B'.)

But when an exception is thrown at execution time, decrements on a subset of D' would have already occurred in B, due to references that die in B. This means that the decrements inserted into B' could operate on dangling references. To forestall this, the $\widehat{RC}_{\_}$ operation is given the following semantics: It sets its operand reference to `null` after doing the decrement. We call this the *decrement-and-assign-null* (DAN) semantics. Suppose that the operand is x; then the solution does not conflict with later uses of x because the $\widehat{RC}_{\_}$ operation is introduced just after the death point of x.

An outcome of this solution is that the `null` assignments against the w_i in Figure 4 become unnecessary.

Because the standard $RC_{\_}$ operation is based on reachability (i.e., references being overwritten), it does not need to have the DAN semantics. It is enough for it to just do a decrement. Since an exception header cannot itself throw an exception, this means that the decrements inserted into B' can be of the standard kind.

2.3 Examples

To demonstrate how the insertion phase transforms specific statements, we consider its effect on Java's `getfield` instruction, and an atomic compare-and-exchange instruction called *cmpxchg*.

A possible IR for `getfield` is `o.f`, where `o` is a local reference and `f` a field. The phase considers it a non-call instruction. The compiler computes $defs_{must}(s)$ and $ldefs(s)$ to be $\{o\}$ and $\{\&o\}$; it thus determines that $\mathscr{L}(defs_{must}(s))$ equals $ldefs(s)$. Hence, the transformed code matches the left template in Figure 4:

$$\texttt{o := o.f} \quad \Longrightarrow \quad \begin{array}{l} \dot{t}_1 := \texttt{o} \\ \texttt{o := o.f} \\ RC_{+}(o) \\ RC_{\_}(\dot{t}_1) \\ \widehat{RC}_{\_}(d_1) \\ \widehat{RC}_{\_}(d_2) \\ \quad \vdots \\ \widehat{RC}_{\_}(d_m) \end{array}$$

The d_i in the above are references that are live on entry to s, but not after. These will form a nonempty set, if the statement could throw an exception and if $live_{in}$ at the beginning of the exceptional control-flow path is a proper superset of $live_{out}(s)$.

The *cmpxchg* instruction is similar to the `CompareExchange` method in .NET.[5] It takes an interior pointer p to a reference, a pair of references x and y, and compares y with the reference at p for equality. If equal, the reference at p is replaced by x and the original reference at p is returned. If unequal, only the reference at p is returned. The insertion phase regards the statement as a non-call instruction. The compiler determines $defs_{must}(s)$ and $ldefs(s)$ as being $\{r\}$ and $\{p, \&r\}$. Now, depending on whether an alias analysis can prove that p always equals $\&r$, the transformed code could match either of the two patterns in Figure 4. The code below is when $\mathscr{L}(defs_{must}(s))$ and $ldefs(s)$ cannot be ascertained as being equal:

$$
\begin{array}{lcl}
r := cmpxchg(p, x, y) & \Longrightarrow & \dot{u}_1 := *p \\
& & \dot{u}_2 := *(\&r) \\
& & r := cmpxchg(p, x, y) \\
& & RC_+(*p) \\
& & RC_+(*(\&r)) \\
& & RC_-(\dot{u}_1) \\
& & RC_-(\dot{u}_2) \\
& & \widehat{RC}_-(d_1) \\
& & \widehat{RC}_-(d_2) \\
& & \quad \vdots \\
& & \widehat{RC}_-(d_m)
\end{array}
$$

Two trivial optimizations are possible on the above code. First, $*(\&r)$ is replaceable by r. Second, $RC_+(*(\&r))$ and $RC_-(\dot{u}_1)$ cancel out because r equals $\dot{u}_1$ after the *cmpxchg* statement.

3 The Capturing of Segueing Live References

We say r is a "segueing live reference" at a program point P if it is live at P, if there is a subsequent point Q at which it is redefined, and if all points between P and Q are inaccessible in the IR. For instance, the b_i in Figure 4 are references overwritten in s, but that are live when control reaches s. They are all segueing live references because they *can* go from being live to being redefined by passing through a death point inside the statement. [6]

Segueing live references are determined using the predicate

$$b_i \in defs_{must}(s) \cap live_{in}(s), \tag{8}$$

and not by using what would appear as the more obvious choice:

$$b_i \in defs_{must}(s) \cap uses_{may}(s). \tag{9}$$

This is because when an exceptional control-flow path emanates from s, $live_{in}(s)$ can be a proper superset of $uses_{may}(s)$.

As an example, consider Figure 2, and suppose $ldefs(s)$ is such that $\mathscr{L}(defs_{must}(s))$ and $ldefs(s)$ are equal. Then RC updates for s will be injected in accordance with the first column in Figure 4. Let x be a local reference such that

$$
\begin{aligned}
& x \in defs_{must}(s), \\
& x \notin uses_{may}(s), \\
& x \in live_{in}(B').
\end{aligned}
$$

Then from Figure 4, no decrement would be applied after s on the old value of x, if segueing live references were calculated using Equation (9). This will lead to a memory leak if the old value is the last reference to an object *and* control normally flows through s.

[5] This is a member of the `System.Threading.Interlocked` class [9].

[6] We say "can" because control can also flow along an exceptional path.

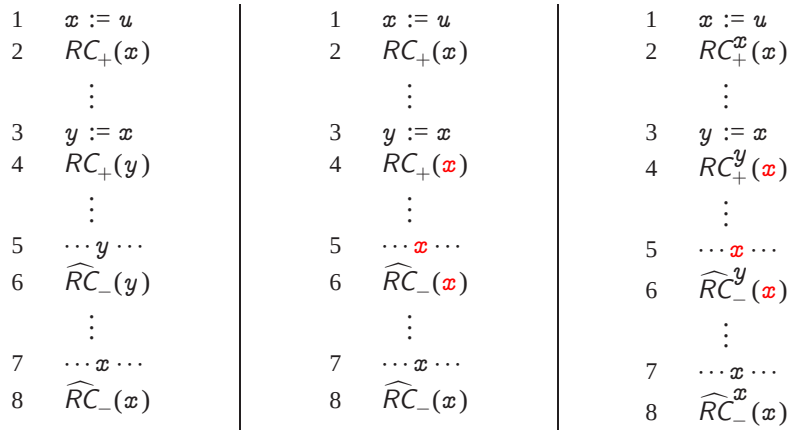

Figure 6: An illustration of the problem of representing the $\widehat{RC}_-$ operation. In the middle is the result after the copy propagator processes code on the left, wrongly replacing y on Line 6 by x. On the right is the result if a two-operand representation were used.

4 Representing Eager Decrements in the IR

The DAN (decrement-and-assign-null) semantics of the $\widehat{RC}_-$ operation has consequences on the way it is modeled in the IR. Its decrement action utilizes a reference *value*, while its kill action affects a reference *lifetime*. Although value and lifetime may relate to the same variable, they need to be separately represented in the IR, so that downstream phases correctly treat the RC updates.

To motivate the issue, consider the left code fragment in Figure 6, which shows an IR after the insertion phase. The $\widehat{RC}_-(y)$ and $\widehat{RC}_-(x)$ on Lines 6 and 8 are due to the last uses of y and x on Lines 5 and 7. Assume a copy propagator is run on this IR, and that it propagates x into the occurrences of y on Lines 4, 5 and 6. The result is the middle fragment in Figure 6. The transformation, however, is wrong because the reaching definition of x on Line 7 is changed to null. The problem arises from propagating a value into an operation that has both use and kill roles associated with it.

Our solution is to use two operands for the $\widehat{RC}_-$ operation, one for each role. We represent it as $\widehat{RC}_-^a(b)$, where the superscript a is the killed reference, and where the parenthetical argument b is the used reference. To a compiler phase that only manipulates values, such as copy propagation, the superscript will be opaque, but not the parenthetical argument. If this two-operand representation were used, the copy propagator can correctly transform the left fragment into the right fragment in Figure 6.

The superscript is essentially the label of a live range. Since the RC_+ operation is inserted at definition points, it is useful to also give the operation a superscript that denotes the live range of the corresponding definition. This is shown in the right fragment in Figure 6. These superscripts are valuable for RC optimizations that are based on lifetime information [20, 21].

The compiler eventually lowers $\widehat{RC}_-^a(b)$ into a pair of statements. The first statement calls the `release` method on b to do the decrement, and the second assigns null to a. Why not just use such a statement pair in the high-level IR instead of the nonstandard $\widehat{RC}_-^a(b)$ representation? The reason is that a general code motion phase, which does not specially recognize the `release`(b) statement, may move a use of b from before the statement to after. This can cause the creation of a dangling reference, as shown in Figure 7.

The root of the issue is that the parenthetical argument may be an object's last reference. So transformational phases that extend live ranges can frustrate the `release`-null representation by introducing a use between the statement pair. It is for this reason that the lowering should happen

1	$\cdots b \cdots$	1	$\cdots b \cdots$	$1'$	`release(`b`)`
2	$\widehat{RC}^{a}_{-}(b)$	2.1	`release(`b`)`	$2'$	$\cdots b \cdots$
		2.2	a := null	$3'$	a := null

Figure 7: An example of the problem with representing $\widehat{RC}^{a}_{-}(b)$ as a `release`-null statement pair. A code motion phase could move the use of b on Line 1 to after Line 2.1, producing the fragment on the right. This could turn b on Line $2'$ into a dangling reference.

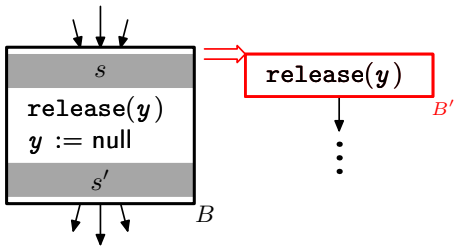

Figure 8: An obstructed dead-code elimination opportunity. In B and B' are the lowered representations of $\widehat{RC}^{y}_{-}(y)$ and $RC_{-}(y)$. If s does not throw an exception, but s' does, the null assignment in B precludes a repeat decrement in B' on the same object.

after all such phases have executed.

The lowering phase also converts $RC^{a}_{+}(b)$ into a call to the `addref` method on b, which does the increment. The superscript is ignored here, since it has no role in the operation's semantics.

The IR after the RC updates have been lowered can be safely operated upon by other phases, if those phases satisfy three conditions: (1) they do not introduce new objects, in the form of statements such as z = *allocobj*(T) (these will lead to leaks); (2) they do not extend existing reference lifetimes; and (3) new references introduced by them have lifetimes subsumed by existing reference lifetimes. These conditions are sufficient, but not necessary.

5 Trading Code Quality with Eagerness

Because of exceptional control-flow paths, not all of the null assignments that come from lowering the $\widehat{RC}_{-}$ operations can be removed. For instance, consider a basic block B, in which s and s' are two adjacent statements. Suppose both could throw an exception that is serviced by a common handler. Assume that y dies across s, is not redefined after that in B, and is not live on entry to the handler's header block B'. Then the insertion phase will insert an $\widehat{RC}^{y}_{-}(y)$ after s, as well as an $RC_{-}(y)$ into B'. Figure 8 shows the result after the $\widehat{RC}_{-}$ and RC_{-} operations are lowered.

The key point in Figure 8 is that the null assignment cannot be removed. This is because the exception thrown by B might be due to s', and not s. Thus, there is a reaching use of this assignment in B'. Obviously, the assignment could have been removed if there was another definition of y after it but before s'.

The obstruction comes from a combination of two factors: (1) the null assignment reaches a use in B'; and (2) it is not *postdominated* by that use [24]. This lack of postdominance is a consequence of the FCFG representation—it cannot be changed by moving the null assignment to another possible point in B. But the first factor can be broken, by moving the null assignment to after s'. Hence, by relaxing the placement of $\widehat{RC}^{y}_{-}(y)$ to a point further away from the death point of y, the dead-code elimination obstruction can be lifted. Figure 9 displays the lowered form of one such placement. The null assignment can now be removed, thereby improving the code path through B, at the expense of holding onto garbage longer.

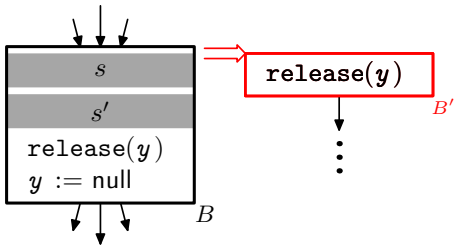

Figure 9: By relaxing the placement of $\widehat{RC}_{\_}^{y}(y)$ to the end of B, an opportunity for dead-code elimination will be created.

5.1 Effect of Placement Relaxation on Register Allocation

The previous discussion showed how code quality can be improved by relaxing the placement of the $\widehat{RC}_{\_}$ operation, potentially delaying the identification of dead data. Other benefits may follow from the relaxation—for example, opportunities might be created to *coalesce* [20] and *batch* [6] RC updates.

In the extreme, the $\widehat{RC}_{\_}$ operations can be placed just before references are redefined, and just before they go out of scope. That exactly corresponds to a standard RC collection scheme! Placing decrements much after the death points of references also extends lifetimes in the downstream phases. This would mean more mutually interfering live ranges by the time a graph-coloring register allocator gets to work on the IR, and therefore, more spill code.

This work, in fact, began as an attempt to realize standard RC collection in a high-level IR. The effort soon ran aground on account of inordinately large register allocation phase times, and poorly performing binaries due to excessive register pressure. And attempting to realize RC collection on a low-level IR, after register allocation, would have made optimizations much harder [21].

6 Methods Outside the Insertion Phase's Scope

In our implementation, run-time services for the RC collector are packaged as part of the larger Bartok run-time system. The run-time system is written in C#, and is compiled into the executable code produced. Thus, the insertion phase also transforms code in the run-time system to use nondeferred RC collection.

However, not all methods should have RC updates automatically inserted into them. For instance, those invoked before the run-time system's data structures are initialized should not execute RC updates. A way to flag a method outside the insertion phase's scope is by affixing annotations, called *attributes*, to it. This section discusses four such attributes, which encapsulate different reasons for suppressing automatic insertion. Rules governing their affixation are explained. These rules can be mechanized, so as to automatically deduce the methods that might need them.

6.1 Attributes for Suppressing Automatic Insertion

`[PreInitRefCounts]` This attribute is attached to methods invoked before run-time initialization. They constitute a small set, belonging to a trusted computing base. Their typical tasks are to allocate bootstrap memory for data structures in the allocator and collector, and to run the allocator's and collector's static initializers.

`[RecursiveRefCounts]` Because the compiler eventually lowers RC updates into calls to the `addref` and `release` methods, any method transitively reachable from `addref` and `release`, including `addref` and `release`, should not be in the scope of the insertion phase. This is

because an inserted RC update could cause an endless recursion at execution time. Suppressing insertion for this reason is indicated by the `[RecursiveRefCounts]` attribute.

`[ManualRefCounts]` Certain methods may need to directly manipulate an object's reference count. As an example, object allocation routines initialize an object's reference count to 1. If RC updates were automatically inserted into them, the programmer-inserted reference-count manipulations could interact with the RC updates to produce undefined results. Such methods are therefore marked with the `[ManualRefCounts]` attribute.

`[ZombieRefCounts]` The insertion phase should also ignore methods that have references to *zombie* objects. These are objects that have become garbage (i.e., reference counts have dropped to zero), but that are yet to be returned to the allocator. Between registration by the collector and return to the allocator, zombies can be operated upon in a variety of ways—for instance, decrements could be applied on their descendents, and they could be subjected to assertion checks. If RC updates were automatically inserted into zombie-referring methods, they could cause the resurrection of a zombie at run time. This may lead to erroneous behavior, either in the form of a memory leak, or a double registration by the collector (if the resurrected zombie falls back to the zombie state). The `[ZombieRefCounts]` attribute is attached to prevent this.

6.1.1 Attribute Usage Issues

In our implementation, the four attributes have only been used on some methods in the collector's run-time system. If the `addref` and `release` methods were exposed to user code, a programmer could also use `[ManualRefCounts]` to hand-optimize the RC efficiency of a method, by manually inserting calls to `addref` and `release`.

A method is *auto RC-suppressed* if any of the four attributes is attached to it. It is otherwise said to be *auto RC-enabled*. More than one of these attributes can be affixed to an auto RC-suppressed method. For example, `release` will have both `[RecursiveRefCounts]` and `[ZombieRefCounts]`, since there could be a reference within it to a zombie.

6.2 Issues with Inlining Methods Before Automatic Insertion

If the insertion phase processes an auto RC-enabled method *after* an auto RC-suppressed method is inlined into it, the result could be an erroneous sequence of RC updates. We demonstrate the problem by considering auto RC-enabled code that contains the statement z := *allocobj*(T). This statement assigns a newly allocated, unconstructed object of type T to z. The compiler can choose to lower the statement into an invocation of a run-time method that does the job of *allocobj*. Such a method is shown in the first column of Figure 10. It allocates an appropriately aligned, zero-initialized memory for the object, and sets up its reference count and vtable fields. Note that the reference count of the vtable object is not adjusted; this is because vtable structures are assumed to live forever.

Now suppose z := *allocobj*(T) is lowered to a call to `allocobj`, and the call is inlined, before the insertion phase processes the auto RC-enabled code. Then the code obtained at the end of the processing is shown on the right in Figure 10. Line 2 involves a call to the run-time allocation routine `allocmem`. Hence, because it is a call instruction, no increment is inserted against the z defined on Line 2. The z on Line 8 is a last-use occurrence in the original and transformed codes. But because of the last use of y on Line 6, the $\widehat{RC}_{-}^{y}(y)$ on Line 7 turns it into a dangling reference.

The problem in this case arises from the way `allocobj` was written. If the definition of y on Line $3'$ were removed, and if x were used instead on Line $5'$, then the inserted RC updates will not conflict with the reference-count initialization on Line 5.

Our solution is to simply avoid such problems by not inlining auto RC-suppressed methods into auto RC-enabled methods prior to the insertion phase. Among the three other scenarios possible when inlining happens before the insertion phase, there is one more problematic case,

```
     [ManualRefCounts]
     function allocobj(T)
1'   sz := sizeof(T)
2'   x := allocmem(sz)
3'   y := x
4'   x.RC := 1
5'   y.vtable := T.vtable
6'   return y
```

```
1    sz := sizeof(T)
2    z := allocmem(sz)
3    y := z
4    RC+^y(y)
5    z.RC := 1
6    y.vtable := T.vtable
7    RC_^y(y)
       ⋮
8    ··· z ···
9    RC_^z(z)
```

Figure 10: The fragment on the right is the result of applying the insertion phase on auto RC-enabled code. The code contained the statement $z := allocobj(T)$. The method shown on the left was inlined at this statement, just before running the insertion phase.

		callee	
		RC-suppressed	*RC-enabled*
caller	*RC-suppressed*	✓, ✓	✗, ✓
	RC-enabled	✗, ✓	✓, ✓

Table 1: A ✓ or ✗ mark shows the validity of an inlining scenario. Each caller-callee combination has a pair of marks; these correspond to inlining the callee before and after the insertion phase.

which is also solved by not inlining. This is when the caller is auto RC-suppressed and the callee is auto RC-enabled.

There can be four other scenarios, which come from inlining the callee *after* the insertion phase. These post-insertion phase scenarios are not problematic, as long as the caller and callee have correct RC update sequences. Hence, there are a total of eight inlining scenarios; these are summarized in Table 1. In the scenarios that involve an auto RC-suppressed callee or caller, the programmer must take care to ensure that the interactions do not lead to memory leaks, dangling references, or reference-count corruptions.

7 Measurements

This section gives experimental data evaluating an implementation of the RC insertion phase in Bartok. The data collected was for the C# applications displayed in Table 2. These are single-thread programs that were first converted into MSIL using version 7.10 of the .NET C# compiler. The MSIL files were then compiled by Bartok into stand-alone x86 code. The platform for the experiments was an HP XW8000 workstation with an Intel Xeon 2.8GHz processor, running Windows XP Version 2002 (Service Pack 2) in hyperthread mode. The capacities of its RAM, primary cache and secondary cache were 2GB, 8KB and 512KB respectively. (Measurements demonstrating the run-time performance of the generated x86 binaries on the same platform are in a recent paper [21].)

Table 2 displays the number of methods, statements and basic blocks processed for each program. The counts are inclusive of .NET's Base Class Library (BCL) and Bartok's C# run-time system; these get compiled into the outputted native code.[7] The method, statement and basic block counts consider all managed-code methods, including the auto RC-suppressed ones. Among the auto RC-suppressed methods are a number of increment and decrement methods

[7] Not all of the run-time system or BCL gets compiled into an outputted binary; there is a phase called "tree shaking" that only pulls in the referenced portions of the class hierarchy, casting out the rest.

Benchmark	Description	Methods	Statements	Basic Blocks		Statement Types	
				Total	Max.	Call	Non-Call
`cmp`	File comparison tool, run on two 1006KB files.	1728	55242	14700	258	5458	30118
`xlisp`	Xlisp interpreter executing `au`, `boyer`, `browse`, etc., as part of a workload of 21 Lisp programs. SPEC CINT95 port.	2167	71362	19188	693	7465	39722
`othello`	Othello (aka Reversi) strategic board game, on an 8×8 grid.	1472	44112	11345	258	3839	23784
`go`	Game of Life, on a 40×19 board. SPEC CINT95 port.	2108	127520	24429	258	7986	91322
`satsol`	Boolean formula satisfiability solver. Available from `www.research.microsoft.com/research/downloads`.	1849	60529	15991	258	5997	33564
`chess`	Chess-playing program. SPEC CINT2000 port.	1994	90004	21088	439	8221	54241
`ahcbench`	The Adaptive Huffman Compression algorithm applied on files. Available from `www.research.microsoft.com/research/downloads`.	1702	53235	13997	258	5067	29151
`bartok`	MSIL to x86 ahead-of-time optimizing compiler, compiling itself to use generational copying collection.	6322	457238	123000	495	71541	263001

Table 2: Details on C# programs transformed by the conversion algorithm to use ARCS collection.

Benchmark	Time (secs)	Statements		Basic Blocks			$\mathscr{L}(defs_{must}(s)) = ldefs(s)$?		
		Total	% Increase	Max.	Total	% Increase	*true*	*false*	% *true*
`cmp`	1.063	82626	49.57	284	16296	10.86	27809	2309	92.33
`xlisp`	1.485	112871	58.17	866	21326	11.14	35787	3935	90.09
`othello`	0.797	65399	48.26	284	12479	10.00	21670	2114	91.11
`go`	2.281	175081	37.30	284	27236	11.49	88143	3179	96.52
`satsol`	1.186	90734	49.90	284	17757	11.04	31211	2353	92.99
`chess`	1.750	130488	44.98	493	23636	12.08	51725	2516	95.36
`ahcbench`	0.969	79298	48.96	284	15476	10.57	26877	2274	92.20
`bartok`	11.718	833103	82.20	622	153123	24.49	241426	21575	91.80

Table 3: Times for the conversion algorithm (run as part of compilation), as well as characteristics of the transformed code.

Benchmark	ARCS			DRC	
	Base	Opts.	Opts.+Inlining	Base	Inlining
cmp	972	824	1056	852	1052
xlisp	1236	1028	1320	1068	1280
othello	840	736	928	752	940
go	1600	1296	1548	1340	1568
satsol	1028	864	1100	892	1092
chess	1308	1076	1352	1108	1332
ahcbench	944	808	1032	836	1024
bartok	6324	4240	5440	4176	4912

Table 4: File sizes, in kilobytes, of x86 ARCS and DRC versions.

synthesized by the compiler for `struct` types. These are used for RC_+, RC_- and $\widehat{RC}_-$ operations on `struct` variables.

All of the reported counts are in the high-level IR, which generally bears a one-to-one correspondence with MSIL. The column labeled "Max." in Table 2 gives the maximum number of basic blocks in a method before the insertion phase executes. This is 258 in the majority of cases, which is the number of basic blocks in the `FormatCustomized` method belonging to the `DataTimeFormat` class in the BCL. The last two columns indicate the number of call and non-call instructions, as classified by the injection stage and as seen at its beginning (refer Figure 3). On average, non-call instructions occur six times more often than call instructions. Their sum is less than the number shown in the "Statements" column because they only take into account the auto RC-enabled methods.

The times, in seconds, taken by the insertion phase to transform the high-level program IRs is shown in Table 3. These times were measured when running Bartok in the CLR (Common Language Run-time), which is .NET's virtual machine for executing MSIL [9]. The table also shows the total number of statements and basic blocks immediately after the conversion. In general, the high-level IR statements increase by about 48%; in the case of Bartok compiling itself, the increase is a little over 80%. The increase in the number of basic blocks is less pronounced, typically being about 11%, and reaching up to 24% for Bartok. The new maximum number of basic blocks is mostly 284, which is the number of basic blocks in the transformed `FormatCustomized` BCL method.

The last three columns in Table 3 indicate the usage frequencies of the two templates in Figure 4, both as absolute counts as well as the percentage usage of the left template. For any benchmark, the sum of the absolute counts equals the number underneath the "Non-Call" column in Table 2. We thus see that the more efficient template is used over 90% of the time in all cases.

Finally, Table 4 shows the sizes in kilobytes of the generated x86 binaries. For comparison, a conversion phase to realize deferred RC collection was also implemented in Bartok. The last two columns in Table 4 show the sizes of the binaries produced with this phase turned on. The "Base." columns give the sizes of the ARCS and DRC (deferred RC) baseline versions. The DRC binaries contain code components not present in the ARCS binaries, such as GC maps and a stack-scanning module. Despite this, the sizes of the baseline ARCS versions are consistently larger than the baseline DRC versions. The "Opt." column shows the sizes of the ARCS versions after the coalescing and immortal object RC update elision optimizations described in [20], and the three overlooking-root-based optimizations discussed in [21]. These optimizations statically detect and eliminate redundant RC updates on stack references. The "Opts.+Inlining" and "Inlining" columns display the sizes of the ARCS and DRC binaries after the lightweight RC updates in them are inlined [20]. These two columns show that the sizes of the final binaries are comparable in many cases.

8 Related Work

In spite of RC collection's long history, there has been no documented work on a principled approach to realizing it with a compiler. Compile-time RC collection research has been mainly in the area of optimizations for deferred RC collection [6, 19, 12, 25], and reference counting for achieving deterministic finalization [27].

A study by Diwan et al. demonstrated that large improvements in an application's heap usage is possible when accurate liveness information is available [18]. It showed that increasing degrees of liveness knowledge can enable increasing reductions in a program's heap footprint. The study gathered liveness data by performing a run-time analysis of a program's trace.

In the ARCS collection scheme, reclamation can be initiated at any program instruction. In most previous efforts, it can be initiated at only a few places in a method, namely the GC-safe points. An exception is the work by Stichnoth et al., which considers every program point as being a GC-safe point. It addresses the resulting problem of coping with a large number of GC maps [28].

Our work supports interior pointers. Past research has either ignored them, or handled them using approaches different from ours, e.g., the Diwan et al. derivation table technique [14].

A topic related to this work is unsafe compiler optimizations that move references to earlier points in the code, causing premature reclamation of their referents [7].

In the work by Sells and Tavares, RC operations were performed on only references that live on the evaluation stack of the CLR [27]. (The evaluation stack is different from the call stack.) Their goal was not to provide a complete GC—the CLR's backup collector was used for that—but to run finalizers as early as possible.

The C++ Standards Committee has proposed *smart pointer* classes for inclusion in a future C++ standard [4]. Smart pointers are a language aid for easing the management of dynamically allocated resources. They resemble the standard C++ pointers, except for under-the-hood bookkeeping in the form of RC operations. This allows for the automatic deletion of a resource, once all smart pointers to it are overwritten or go out of scope. Smart pointer implementations are available as part of the Boost C++ Libraries [8].

While our work uses information on reference deaths, there has also been work that uses information on object deaths. The two are related, in that when all the references to an object die, the object also dies. Various static analyses have been suggested to estimate object lifetimes. An example is the data-flow analysis by Ruggieri and Murtagh [26]. It computes a mapping of references to object-creating expressions. The goal is to partition the heap into subheaps, one for each procedure, so that objects contained in a procedure's lifetime are allocated in its subheap. Their approach could delay reclaiming an object, to well beyond its disconnection from the rest of the object graph.

9 Summary

This paper presented an algorithm for transforming an object-oriented program into one that uses a nondeferred form of RC collection. Rather than special-casing the treatment of instructions, the algorithm handles them in a uniform way, by using abstractions based on data-flow analysis notions. Modern object-oriented language features, such as object pinning, interior pointers and exceptions, are accounted for in the transformation. The algorithm inserts two kinds of decrements to reclaim dead data, one based on reachability and the other on liveness. Some of the complications it copes with are the consideration of live references whose death points are inaccessible, and exceptional control-flow paths.

The algorithm has been implemented in an optimizing compiler. It has been used to successfully compile numerous large programs into nondeferred RC-collected versions. The paper also discussed the compiler issues that were addressed for this. These include representing the eager decrements so that subsequent phases can correctly operate on the IR, constraints on running phases like method inlining before or after RC updates are inserted, and the treatment of code in

the run-time system.

References

[1] Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In A. Michael Berman, editor, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 269–279, June 1998.

[2] Bowen Alpern, C. R. Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn-Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-Time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 207–216, June 1988.

[4] Matt Austern. Draft Technical Report on C++ Library Extensions. Technical Report ISO/IEC DTR 19768, The C++ Standards Committee, June 2005.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. A Unified Theory of Garbage Collection. In *Proceedings of the 2004 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 50–68, October 2004.

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

[7] Hans-Juergen Boehm. Simple GC-Safe Compilation. In Paul R. Wilson and Barry Hayes, editors, *Addendum to OOPSLA'91 Proceedings*, October 1991.

[8] Boost C++ Libraries. At `http://www.boost.org`.

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

[10] Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and Precise Modeling of Exceptions for the Analysis of Java Programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 21–31, September 1999.

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

[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, 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] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler Support for Garbage Collection in a Statically Typed Language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–282, June 1992.

[15] 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.

[16] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Second Edition*. The Java Series. Addison-Wesley Publishing Company, Inc., 2000.

[17] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Publishing Company, Inc., Redwood City, CA 94065, USA, 2003.

[18] Martin Hirzel, Amer Diwan, and Johannes Henkel. On the Usefulness of Type and Liveness Accuracy for Garbage Collection and Leak Detection. *ACM Transactions on Programming Languages and Systems*, 24(6):593–624, November 2002.

[19] 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, April 1986.

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

[21] Pramod G. Joisha. Overlooking Roots: A Framework for Making Nondeferred Reference-Counting Garbage Collection Fast. To appear in *Proceedings of the 6th International Symposium on Memory Management*. ACM Press, October 2007.

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

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

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

[25] 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, June 1991.

[26] Cristina Ruggieri and Thomas P. Murtagh. Lifetime Analysis of Dynamically Allocated Objects. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 285–293, January 1988.

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

[28] James M. Stichnoth, Guei-Yuan Lueh, and Michał Cierniak. Support for Garbage Collection at Every Instruction in a Java Compiler. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 118–127, May 1996.

[29] David R. Tarditi, Greg Morrisett, Perry Cheng, Christopher Stone, Robert Harper, and Peter Lee. TIL: A Type-Directed Optimizing Compiler for ML. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.