A Unified Theory of Garbage Collection

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Tracing and reference counting are uniformly viewed as being fundamentally different approaches to garbage collection that possess very distinct performance properties. We have implemented high-performance collectors of both types, and in the process observed that the more we optimized them, the more similarly they behaved — that they seem to share some deep structure.

We present a formulation of the two algorithms that shows that they are in fact duals of each other. Intuitively, the difference is that tracing operates on live objects, or "matter", while reference counting operates on dead objects, or "anti-matter". For every operation performed by the tracing collector, there is a precisely corresponding anti-operation performed by the reference counting collector.

Using this framework, we show that all high-performance collectors (for example, deferred reference counting and generational collection) are in fact hybrids of tracing and reference counting. We develop a uniform cost-model for the collectors to quantify the trade-offs that result from choosing different hybridizations of tracing and reference counting. This allows the correct scheme to be selected based on system performance requirements and the expected properties of the target application.

General Terms

Algorithms, Languages, Performance

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage collection*

Keywords

Tracing, Mark-and-Sweep, Reference Counting, Graph Algorithms

OOPSLA'04, Oct. 24-28, 2004, Vancouver, British Columbia, Canada.

1. INTRODUCTION

By 1960, the two fundamental approaches to storage reclamation, namely tracing [33] and reference counting [18] had been developed.

Since then there has been a great deal of work on garbage collection, with numerous advances in both paradigms. For tracing, some of the major advances have been iterative copying collection [15], generational collection [41, 1], constant-space tracing [36], barrier optimization techniques [13, 45, 46], soft real-time collection [2, 7, 8, 14, 26, 30, 44], hard real-time collection [5, 16, 23], distributed garbage collection [29], replicating copying collection [34], and multiprocessor concurrent collection [21, 22, 27, 28, 39].

For reference counting, some of the major advances have been incremental freeing [42], deferred reference counting [20], cycle collection [17, 32, 6], compile-time removal of counting operations [9], and multiprocessor concurrent collection [3, 19, 31].

However, all of these advances have been refinements of the two fundamental approaches that were developed at the dawn of the era of high-level languages.

Tracing and reference counting have consistently been viewed as being different approaches to storage reclamation. We have implemented both types of collector: a multiprocessor concurrent reference counting collector with cycle collection [3, 6] and a uniprocessor real-time incremental tracing collector [4, 5]. In this process, we found some striking similarities between the two approaches. In particular, once substantial optimizations had been applied to the naïve algorithms, the difficult issues that arose were remarkably similar. This led us to speculate that the two algorithms in fact share a "deep structure".

In this paper we show that the two fundamental approaches to storage reclamation, namely tracing and reference counting, are algorithmic duals of each other. Intuitively, one can think of tracing as operating upon live objects or "matter", while reference counting operates upon dead objects or "anti-matter". For every operation performed by the tracing collector, there is a corresponding "anti-operation" performed by the reference counting collector.

Approaching the two algorithms in this way sheds new light on the trade-offs involved, the potential optimizations, and the possibility of combining reference counting and tracing in a unified storage reclamation framework.

We begin with a qualitative comparison of tracing and reference counting (Section 2) and then show that the two algorithms are in fact duals of each other (Section 3). We then show that all realistic, high-performance collectors are in fact hybrids that combine tracing and reference counting (Section 4). We then discuss the problem of cycle collection (Section 5) and extend our framework to collectors with arbitrary numbers of separate heaps (Section 6). Using our categorization of collectors, we then present a uniform

	Tracing	Reference Counting
Collection Style	Batch	Incremental
Cost Per Mutation	None	High
Throughput	High	Low
Pause Times	Long	Short
Real Time?	No	Yes
Collects Cycles?	Yes	No

Figure 1: Tracing vs. Reference Counting.

	Tracing	Reference Counting
Starting Point	Roots	Anti-roots
Graph Traversal	Fwd. from roots	Fwd. from anti-roots
Objects Traversed	Live	Dead
Initial RC	Low (0)	High
RC Reconstruction	Addition	Subtraction
Extra Iteration	Sweep Phase	Trial Deletion

Figure 2: Tracing vs. Reference Counting, Revisited.

cost model for the various collectors, which allows their performance characteristics to be compared directly (Section 7). We then discuss some space-time trade-offs that can be made in the implementation of a collector (Section 8). Finally, we present our conclusions.

2. QUALITATIVE COMPARISON

We begin with a qualitative comparison of the differences between tracing and reference counting collectors, and discuss how the algorithms become more similar as optimizations are applied.

2.1 Diametrical Opposites?

The naïve implementations of tracing and reference counting are quite different. The most commonly cited differences are shown in Figure 1.

Reference counting is inherently incremental since it updates reference counts on each pointer write; tracing operates in "batch mode" by scanning the entire heap at once. Tracing incurs no penalty for pointer mutation, while reference counting incurs a high cost — every pointer write (including those to stack frames) results in reference count updates. Throughput for tracing collectors is correspondingly higher.

On the other hand, reference counting collectors incur very short pause times, which makes them naturally suitable for real-time applications.

Finally, tracing collects cyclic garbage while reference counting does not. As a result, when reference counting is applied to heaps that may contain cyclic garbage, cycles must either be collected using a backup tracing collector [43] or with a trial deletion algorithm [6, 17, 32].

2.2 Convergence

Our investigation of the similarities between tracing and reference counting began when we noticed that as we optimized our reference counting and tracing collectors, they began to take on more and more of the each other's characteristics.

The incremental nature of reference counting is generally considered to be its fundamental advantage. However, the cost of updating reference counts every time a new pointer is loaded into a register is typically much too high for high-performance applications. As a result, some form of deferred reference counting [20], in which references from stack frames are accounted for separately, is used in most high-performance implementations of reference counting [3, 19].

However, this means that when an object's reference count drops to zero, it can not be reclaimed immediately, since there might be a reference from the stack that is not accounted for. As a result, collection is deferred until the periodic scanning of the stack references. However, the result is delayed collection, floating garbage, and longer application pauses — the typical characteristics of tracing collectors!

Now consider an implementation of a high-performance tracing collector: two of its fundamental advantages are the lack of per-mutation overhead and the natural collection of cyclic garbage. However, a fundamental disadvantage of tracing is that freeing of dead objects is delayed until the end of a collection cycle, resulting in delayed reclamation of objects and long pause times.

One of the first optimizations that is typically applied to a tracing collector is generational collection [41]. This reduces the average pause time and the delay in reclaiming objects, but it also introduces per-mutation overhead — thus it takes on both some of the positive and the negative aspects of reference counting collection.

A further attempt to precisely limit the pause time is manifest in the train algorithm [25], which breaks memory up into fixed-size *cars* which are grouped together into *trains*. One car is collected at a time, which yields deterministic pause times, except in the presence of *popular objects*. Furthermore, inter-car cycles can cause pathological behavior. But problematic behavior in the presence of cycles is a paradigmatic quality of reference counting!

3. THE ALGORITHMIC DUALS

Our first-hand experience of (and frustration with) the convergence of optimized forms of reference counting and tracing collectors led directly to a deeper study of the algorithms in the hope of finding the fundamental similarities that seem to be appearing in practice.

3.1 Matter vs. Anti-matter

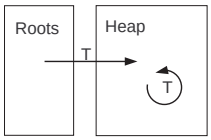
In order to see the connection between the two algorithms it is necessary to view them somewhat differently than usual. First of all, we consider a version of reference counting in which the decrement operations are batched and performed at "collection time" instead of being performed immediately (this can be viewed as a subsequent optimization). Second, we consider a version of tracing in which the tracing process reconstructs the actual reference count of each object instead of simply setting a mark bit (mark bits can be viewed as a subsequent optimization in which the reference count is turned into a one-bit "sticky" reference count).

Viewed in this light, the parallels between the two algorithms are quite striking, and are summarized in Figure 2.

Tracing garbage collection traverses the object graph forward, starting with the roots, to find the live data. Reference counting traverses the object graph forward, starting with the anti-roots (the set of objects whose reference counts were decremented to 0), to find dead data.

Intuitively, one can think of tracing as operating on "matter" and reference counting as operating on "anti-matter". Formally, in the absence of cycles, reference counting computes the graph complement of tracing.

Tracing initializes object reference counts to zero, and in the process of graph traversal increments them until they reach the true reference count. Reference counting "initializes" the reference counts to a value that is in excess of the true count, and in the process



(a) Schematic

```
collect-by-tracing()
    initialize-for-tracing(W)
    scan-by-tracing(W)
    sweep-for-tracing()

scan-by-tracing(W)
    while W ≠ ∅
        remove w from W
        ρ(w) ← ρ(w) + 1
        if ρ(w) = 1
            for each x ∈ [v : (w, v) ∈ E]
                W ← W ⊎ [x]

sweep-for-tracing()
    for each v ∈ V
        if ρ(v) = 0
            V_F ← V_F ∪ {v}
        ρ(v) ← 0

new(x)
    ρ(x) ← 0

initialize-for-tracing(W)
    W ← find-roots()
```

(b) Algorithm

Figure 3: Tracing Garbage Collection

graph traversal decrements them until they reach the true reference count (ignoring the presence of cycles).

Reference counting must perform extra graph iterations in order to complete collection of cyclic data. This is typically viewed as a major drawback of reference counting.

But tracing must also perform an extra iteration, and over the entire object space: the sweep phase, which collects those objects whose reference count is 0 (mark bit clear). While semi-space copying collectors avoid this time cost by copying only live data, this is in fact a simple linear space-time trade-off: a linear traversal is saved at the expense of a linear cost in space.

3.2 Fix-point Formulation

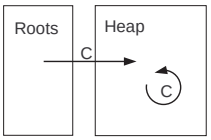
We begin by describing the algorithm for garbage collection abstractly. We will then refine this into the tracing and reference counting duals.

Throughout this paper, we assume that all memory consists of fixed-size objects, and therefore ignore fragmentation [4].

We use the notation $[a, a, b]$ to denote the multiset containing two a's and one b, $[a, b] \uplus [a] = [a, a, b]$ to denote multiset union, and $[a, a, b]^{\odot} = \{a, b\}$ to denote the projection of a multiset onto a set.

We characterize the memory of the system formally as:

- V is the set of vertices in the object graph. V is the universe of all objects, and includes both garbage as well as objects in the "free list". V does not comprise all of memory, since the collector must also maintain meta-data to support collection.



(a) Schematic

```
collect-by-counting(W)

    scan-by-counting(W)
    sweep-for-counting()

scan-by-counting(W)
    while W ≠ ∅
        remove w from W
        ρ(w) ← ρ(w) − 1
        if ρ(w) = 0
            for each x ∈ [v : (w, v) ∈ E]
                W ← W ⊎ [x]

sweep-for-counting()
    for each v ∈ V
        if ρ(v) = 0
            V_F ← V_F ∪ {v}

new(x)
    ρ(x) ← 0

dec(x)
    W ← W ⊎ [x]

inc(x)
    ρ(x) ← ρ(x) + 1

assign(a, p)
    l ← [a]
    [a] ← p
    dec(l)
    inc(p)
```

(b) Algorithm

Figure 4: Reference Counting Garbage Collection. Note the exact correspondence of the scan **and** collect **methods with the Tracing Algorithm in the figure to the left.**

- E is the multiset of edges in the graph (a node can have more than one pointer to the same node).
- R where $R^{\odot} \subseteq V$ is the multiset of roots of the graph (in stack frames and global variables).
- $V_F \subseteq V$ is the "free list", the set of vertices known to be available for allocation.
- $V_L = R^{\star}$ is the set of live vertices in the object graph: the set of vertices reachable from the roots.
- $E_L = \{(x, y) : x \in V_L\}$ is the set of live edges in the object graph.
- $V_D = V - V_L$ is the set of dead vertices. In general $V_F \subseteq V_D$, but throughout this paper we assume that collection is only triggered when $V_F = \emptyset$.

- $V_C \subseteq V_D$ is the set of vertices that are cyclic garbage; that is, they are not in V_L but they all have in-edges in the graph.
- $\rho(v)$, where $v \in V$, is the reference count of vertex v, as computed by the collector.

The object graph is the triple $G =< V, E, R >$.

Garbage collection can be expressed as a fix-point computation. A fix-point is computed for the assignment of reference counts $\rho(v)$ to vertices $v \in V$. Reference counts include contributions from the root set R and incoming edges from vertices with non-zero reference counts:

$$\rho(x) = \big|[x : x \in R]\big| + \big|[(w, x) : (w, x) \in E \wedge \rho(w) > 0]\big| \quad (1)$$

Once reference counts have been assigned, vertices with a reference count of 0 are reclaimed:

$$V_F = [v \in V : \rho(v) = 0] \quad (2)$$

In general, there may be many such fix-points for a given object graph. For example, consider the case where $V = \{a, b\}$, $R = \emptyset$ and $E = \{(a, b), (b, a)\}$. Then $\rho(a) = \rho(b) = 0$ and $\rho(a) = \rho(b) = 1$ are the two possible fix-points of the equation. The former is the least fix-point and the latter is the greatest fix-point. Clearly, we wish to collect the garbage identified by the least fix-point.

The fix-point formulation is not in itself an algorithm. We now consider the solutions arrived at by garbage collection algorithms.

3.3 Tracing Garbage Collection

The real algorithms operate on the object graph G defined above. In addition they maintain

- W, the work-list of objects to be processed by the algorithm. When W is empty the algorithm terminates.

The tracing garbage collection algorithm is shown in Figure 3(b). Initially, the reference counts of all vertices are zero, either because they were created that way by new() or because their reference count was reset to zero during the previous sweep-for-tracing(). The initialize-for-tracing() function initializes the work-list to be the root set R.

The heart of algorithm is the function scan-by-tracing(), which reconstructs the reference count of each vertex. It scans forward from each element of the work-list, incrementing the reference counts of vertices that it encounters. When it encounters a vertex w for the first time ($\rho(w) = 1$), it recurses through all of the out-edges of that vertex by adding them to the work-list W.

When the **while** loop terminates, it will have discovered all of the nodes in $R^\star$ — that is, the set of all live nodes V_L — and set their reference counts to be the corresponding number of in-edges in E_L.

Finally, the sweep-for-tracing() function is invoked to return the unused vertices to free storage V_F and reset the reference counts to zero in preparation for the next collection.

The only substantive difference between this algorithm and a standard tracing collector is that we are maintaining a full reference count instead of a boolean flag that tells whether or not a vertex has already been visited. However, this does not change the complexity of the algorithm (although it would affect its running time in practice). As we have already mentioned, the mark "bit" can be viewed as a degenerate reference count that "sticks" at one.

Tracing garbage collection computes the *least fix-point* of the equation in equation 1.

Throughout the paper, we will be using schematic diagrams to show how collectors handle pointers between different regions of memory. In Figure 3(a) we show the structure of the tracing collector, which traces references from the roots to the heap and within the heap. These are shown with arrows labeled with "T". This diagram is trivial, but as we discuss more and more complex collectors, the diagrams provide a simple way to summarize the collector architecture.

3.4 Reference Counting Garbage Collection

The reference counting garbage collection algorithm is shown in Figure 4(b). The horizontal lines that match up with Figure 3(b) are there to emphasize the similarity between the component functions of the algorithm.

This formulation of the reference counting algorithm is somewhat unusual in that decrement operations are buffered, rather than being performed immediately. We are not advocating this as an implementation, but rather as a way of understanding the relationship between the algorithms. Delaying the decrements shifts some of the work in time, but does not affect the complexity of the algorithm.

Therefore, the dec() function adds vertices to the work list W, instead of the vertices being added by the initialize function.

During mutation when a pointer is stored into memory by calling the assign function, which takes the pointer p to be stored and the address a at which to store it. The function loads the old referent l at address a and calls the dec(l) function which adds l to the work-list W. The address is updated, and the reference count of the new referent p is incremented calling inc(p).

When a collection is triggered, all increments have been performed, but the decrements since the last collection have not; they have been recorded in W. As a result, at the commencement of the scan function, the reference counts are over-estimates of the true counts.

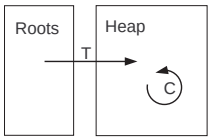
As with tracing collection, the heart of the algorithm is the scanning phase, performed by the function scan-by-counting() at collection time. The algorithm scans forward from each element of the work-list, decrementing the reference counts of vertices that it encounters. When it discovers a garbage vertex w ($\rho(w) = 0$), it recurses through all of the edges of that vertex by adding them to the work-list W. Finally, the sweep-for-counting() function is invoked to return the unused vertices to free storage.

The relationship between the two algorithms becomes obvious when one looks at Figures 3 and 4 side by side. The scan functions are identical except for the use of reference count increments in tracing instead of reference count decrements in reference counting, and the recursion condition which checks whether the reference count is 1 in tracing instead of 0 in reference counting. By changing two characters in the heart of the algorithm, we have changed from tracing to reference counting!

The other interesting difference between the collectors is that the sweep-for-counting() function does not reset reference counts to 0.

The architecture of the simple reference counting collector is shown in Figure 4(a). Arrows labeled with "C" represent reference counted pointers. Reference counting is performed both for references from the roots to the heap and for intra-heap references.

By considering the tracing and reference counting algorithms in this light we see that they share the same fundamental structure. The only difference is that tracing begins with an underestimate of the reference counts and converges toward the true value by incrementing the counts as it encounters vertices in its trace. On the other hand, reference counting starts with an overestimate due to the counted in-edges from objects that are in fact no longer live, and by processing decrement operations it converges toward the true reference count.

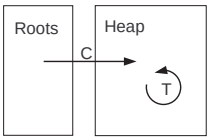


(a) Schematic

```
collect-by-drc(W)
    R ← find-roots()
    trace-roots(R)
    scan-by-counting(W)
    sweep-for-counting()
    untrace-roots(R)

trace-roots(R)
    for r ∈ R
        ρ(r) ← ρ(r) + 1

untrace-roots(R)
    for r ∈ R
        ρ(r) ← ρ(r) − 1

drc-assign(a, p)
    l ← [a]
    [a] ← p
    if ¬ RootPointer(a)
        dec(l)
        inc(p)
```

(b) Algorithm

Figure 5: Deferred Reference Counting



(a) Schematic

```
collect-by-partial-tracing(R)

    scan-by-tracing(R)
    sweep-for-tracing()

pt-inc(x)

    R ← R ⊎ [x]

pt-dec(x)

    R ← R − [x]

pt-assign(a, p)
    l ← [a]
    [a] ← p
    if RootPointer(a)
        pt-dec(l)
        pt-inc(p)
```

(b) Algorithm

Figure 6: Partial Tracing

Viewed in another light, while tracing computes the least fix-point to the equation in equation 1, reference counting computes the greatest fix-point. The set difference between these two solutions comprises the cyclic garbage.

4. TRACING/COUNTING HYBRIDS

Given the similarity in structure that we discovered between tracing and reference counting, we began to re-examine various collector architectures to understand the interplay between these styles of collection. We observed that all realistic garbage collectors are in fact some form of hybrid of tracing and reference counting.

This explains why an optimized "tracing collector" and an optimized "reference counting collector" become more and more similar: because they are in fact taking on characteristics of each other.

As we will see, the only fundamental differences between various collectors are in the division of storage, and in the assignment of responsibility for maintenance of reference counts in the different divisions to either tracing or reference counting. After these decisions have been made, the remaining choices are implementation details which amount to making various space-time trade-offs, which we will discuss in detail in Section 8.

We broadly characterize collectors based on their division of storage: *unified heap* collectors have a single heap in which all data resides; *split heap* collectors divide memory into two regions, such as in a generational collector; and *multiple heap* collectors have more than two memory regions, as for instance distributed garbage collectors or the Train algorithm [25].

Note that in this analysis we consider semi-spaces as a single heap region, since only one semi-space is active at a time (we are not considering concurrent collectors). The use of semi-spaces is one of the time-space trade-offs considered in Section 8.

4.1 Deferred Reference Counting

A Deferred Reference Counting (DRC) collector is a unified heap collector. However, such collectors still have two regions of storage, namely the heap V and the roots R. The various combinations of tracing and reference counting across these two regions yield different algorithms.

Deferred reference counting is a hybrid in which reference counting maintains the counts between heap objects. Objects with reference count 0 are maintained in a *zero count table* (ZCT).

Root references are not counted. Instead, at collection time any elements of the ZCT that are pointed to by roots are removed from the ZCT and the remaining ZCT entries are collected.

Because the root mutation rate is almost always extremely high, deferred reference counting moves the cost of considering the roots from the application to the collector.

But the act of examining the root pointers and removing their referents from the ZCT *is* tracing: it is the act of following a pointer forward and incrementing the reference count of the discovered objects.

The structure of deferred reference counting is shown in Figure 5(a). References from the stack to the heap are traced, while references within the heap are reference counted.

The formulation of the deferred reference counting algorithm is shown in Figure 5(b). The write barrier (assign function) has been modified to filter out any pointers where the source is not a root. Thus at collection time the increments for all intra-heap pointers have been performed, and the decrements have been placed in the work list W.

The collection operation itself finds the root set and increments all of its targets. It then invokes the standard reference counting collection operations (scan-by-counting and sweep-for-counting) which

compute reference counts and then collect objects with reference count 0. Finally, the updates to reference counts from the roots are undone by untrace-roots.

The only difference between this algorithm and the classical DRC algorithm is that we do not explicitly maintain the ZCT; by deferring the decrements to the work list we discover garbage objects when their reference counts drop to zero. But this is merely an implementation choice. Some DRC collectors use this alternative approach [3].

4.2 Partial Tracing

In a unified heap, one could also consider implementing the converse of deferred reference counting, namely reference counting the roots and tracing the heap, which we call a *partial tracing* algorithm. This is shown in Figure 6(a). It is simply deferred reference counting with the role of the edges exchanged; it therefore has a duality with deferred reference counting in a similar way that tracing and reference counting are themselves duals.

The partial tracing algorithm has an assign function that has the complementary filter on pointers: it only considers root pointers. For those pointers, it invokes special increment and decrement operations whose function is to dynamically maintain the root set. In essence, this set can be thought of as a reference count maintained only for roots (that is, different from ρ).

Once the root set has been maintained by the write barrier, collection is simply a matter of invoking the standard tracing algorithm. Instead of finding roots by searching (for example, scanning stacks and global variables), it simply passes the root set it has been maintaining.

The fundamental property of the hybridization is that when tracing starts, reference counting has already caused some vertices to have non-zero reference counts, by virtue of their being in the set R. These "virtual reference counts" are then materialized by being fed as work list inputs to the scan-by-tracing function.

Note that the assign function is always associated with the reference counting, rather than the tracing part of the hybrid collector. For DRC, the heap is reference counted and the barrier records pointers from the heap. For partial tracing, the roots are reference counted and the barrier records pointers from the stack. In general, the presence of a write barrier is an indication of some sort of reference count-like behavior in an algorithm.

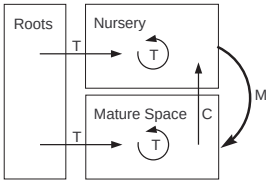
We know of no implementation of partial tracing. For a language run-time system, it would have singularly poor performance properties. It manages to combine most of the worst aspects of both tracing and reference counting: it has extremely high mutation cost, as in pure reference counting, while gaining none of the incrementality of reference counting.

However, in the implementation garbage-collected systems in other environments, where the operations might be performed on disks, networks, or expensive run-time structures, such an algorithm might be worthwhile. For instance, if the cost of writing a root pointer was already fairly high, and finding the roots by scanning was very expensive, then reference counting the roots might be the best solution.

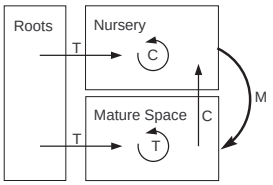
4.3 Generational Garbage Collection

Deferred reference counting illustrates hybridization within a unified heap. We will now consider collectors which split the heap into two regions: a *nursery* and a *mature space*.

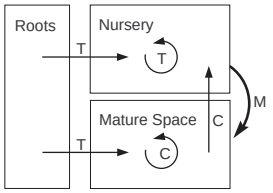
Split-heap collectors actually have three memory regions: the roots, the nursery, and the heap. All split-heap collectors have the property that they are some form of hybridization of tracing and reference counting, as we will now show.



(a) Nursery and Mature Space are Traced (Standard Generational Collection)



(b) Nursery is Reference Counted and Mature Space is Traced



(c) Nursery is Traced and Mature Space is Reference Counted (Ulterior Reference Counting)

Figure 7: Generational Collectors

A generational collector effectively attenuates the allocation rate into the mature space by allocating objects into the nursery, and only moving objects that survive nursery collection into the mature space. Average pause times are substantially reduced and throughput is often increased as well.

4.3.1 Tracing Generational Collection

The most common split-heap collector architecture is a generational collector [41].

In order to collect the nursery independently (without having to trace the entire mature space) a generational collector maintains a *remembered set* of objects in the nursery that are pointed to by objects in the mature space. The remembered set may be implemented with a bitmap, card marking, or a sequential store buffer (SSB).

The remembered set is maintained by a *write barrier* which is executed at every heap pointer update. The write barrier checks whether the pointer crosses from mature space to the nursery, and if so adds it to the remembered set.

By now, the analogy with previous collectors should be somewhat obvious: the write barrier is the assign function, which we have observed is correlated to the reference counting portion of a collector.

The remembered set is in fact a set representation of non-zero reference counts – the complement of the zero count table (ZCT) used in deferred reference counting, except that its range is limited to the nursery. As we have seen, starting with non-zero reference counts is a fundamental feature of reference counting.

```
gen-collect-nursery()
    gen-nursery-initialize()
    gen-nursery-scan()
    nursery-sweep()
```

```
gen-nursery-initialize()
    R ← find-roots()
    R_N ← [r : r ∈ R ∧ r ∈ V_N]
    for each r ∈ R_N
        W ← W ⊎ [r]
```

```
gen-nursery-scan()
    while W ≠ ∅
        remove w from W
        ρ(w) ← ρ(w) + 1
        if ρ(w) = 1
            for each x ∈ [v : (w, v) ∈ E]
                if x ∈ V_N
                    W ← W ⊎ [x]
```

```
gen-assign(a, p)
    l ← [a]
    [a] ← p
    if InMatureSpace(a)
        gen-dec(l)
        gen-inc(p)
```

```
gen-inc(x)
    if x ∈ V_N
        W ← W ⊎ [x]
```

```
gen-dec(x)
    if x ∈ V_N
        ρ(x) ← ρ(x) − 1
```

```
gen-collect-heap()
    gen-collect-nursery()
    collect-by-tracing()
```

```
nursery-sweep()
    for each v ∈ V_N
        if ρ(v) = 0
            V_NF ← V_NF ∪ {v}
        else
            V_N ← V_N \ {v}
            V_H ← V_H ∪ {v}
            ρ(v) ← 0
```

Figure 8: Generational Garbage Collection Algorithm

A generational collection algorithm (for collecting the nursery) using our formalism is shown in Figure 8. The assign function only considers pointers from the mature space into the nursery. Decrements are performed immediately and increments are deferred by placing them into a work list. While this may seem slightly counterintuitive, recall that in our formulations all collectors compute reference counts, rather than just mark bits. If we only needed mark bits, we could omit performing the decrements. The point of recording the increments in a work list is that they form a set of roots from which the tracing of the nursery proceeds (in addition to stack roots).

The gen-nursery-initialize function takes the work list created by the generational write barrier, and adds the roots that point into the nursery. Then it performs the tracing or the nursery with the gen-nursery-scan function. This function is the same as the scan-by-tracing function of the basic tracing algorithm (Figure 3), except that nodes are only added to W if they are in the nursery V_N.

The sweep function moves nodes with non-zero reference counts

```
rcn-collect-nursery()
    rcn-trace-roots()
    rcn-nursery-scan()
    nursery-sweep()
```

```
rcn-trace-roots()
    R ← find-roots()
    R_N ← [r : r ∈ R ∧ r ∈ V_N]
    for each r ∈ R_N
        ρ(r) ← ρ(r) + 1
```

```
rcn-nursery-scan()
    while W ≠ ∅
        remove w from W
        ρ(w) ← ρ(w) − 1
        if ρ(w) = 0
            for each x ∈ [v : (w, v) ∈ E]
                if x ∈ V_N
                    W ← W ⊎ [x]
```

```
rcn-assign(a, p)
    l ← [a]
    [a] ← p

    rcn-dec(l)
    rcn-inc(p)
```

```
rcn-dec(x)
    if x ∈ V_N
        W ← W ⊎ [x]
```

```
rcn-inc(x)
    if x ∈ V_N
        ρ(x) ← ρ(x) + 1
```

```
rcn-collect-heap()
    rcn-collect-nursery()
    collect-by-tracing()
```

Figure 9: Reference Counted Nursery Collection Algorithm

from the nursery into the mature-space, and sets their reference count to zero. This maintains the invariant that between mature space collections, all nodes in the mature space have reference count 0 (tracing invariant).

To collect the whole heap (gen-collect-heap), the nursery is collected and then the standard tracing collector is invoked. The nursery is known to be empty so only mature space objects are considered.

A schematic of the generational collector is shown in Figure 7(a). References from the root set to both the nursery and the mature space are traced. References within both the nursery and the mature space are traced. References from the mature space to the nursery are reference counted. Finally, there is a *macro-edge* from the nursery to the mature space (designated by the arrow labeled with "M").

The macro edge can be thought of as a summary reference count. Since we do not keep track of pointers from the nursery into the mature space, the mature space can not be collected independently. Their might be a reference from the nursery to any or all of the objects in the mature space. Therefore, the only time when it is safe to collect the mature space is when the nursery is empty, because then the reference count from the nursery to the mature space is known to be zero.

4.3.2 Generational with Reference Counted Nursery

We can now start exploring the design space for generational

collectors by considering the different combinations of tracing and reference counting. We first consider the case where we apply the dual approach (reference counting) to the nursery while applying to the same approach (tracing) to the mature space. The result is the algorithm in Figure 9.

This algorithm performs deferred reference counting for the nursery, and maintains reference counts from the mature space into the nursery. The architecture is shown schematically in Figure 7(b).

To be more specific, we apply deferred reference counting to the nursery (we do not reference count updates to root pointers into the nursery). Instead, at the beginning of rcn-collect-nursery, we apply the same operation that we apply at the beginning of DRC, but restricted to the nursery: we trace from the roots into the nursery, incrementing the reference counts of the target objects. Note that this operation is also essentially doing the same thing as the corresponding operation for the generational traced nursery collector, which adds the nursery roots to the work list for tracing.

The rcn-nursery-scan function is simply the reference counting dual of the gen-nursery-scan function: it recursively decrements reference counts instead of incrementing them, except that it does not cross into the mature space.

Finally, the same nursery-sweep function is called as for the generational collector. Unlike the DRC collector, there is no untrace operation that is performed in rcn-nursery-collect. The reason is that nursery-sweep sets the reference count of objects moved into the mature space to zero, undoing the incrementing performed by rcn-trace-roots. Since the mature space will be traced, all reference counts must start at zero and there is no need to accurately undo the effect of tracing the roots and incrementing their reference counts.

The advantage of this collector is that cyclic garbage will eventually be collected because the mature space is traced; the disadvantage is that it reference counts exactly those objects which are likely to have a high mutation rate (the young objects). Therefore, the expensive write barrier operations will be performed for the most frequent operations rather than the least frequent operations.

4.3.3 Generational with Reference Counted Heap

The problems with the previous algorithm suggest taking the opposite approach: tracing the nursery and reference counting the mature space. This has the advantage that mutations in the nursery are not recorded by the write barrier, but the disadvantage that some additional cycle collection mechanism is required for the mature space. The algorithm is shown schematically in Figure 7(c).

This architecture was in fact implemented recently by Blackburn and McKinley under the name "Ulterior Reference Counting" [12]. They used a trial deletion algorithm (see Section 5.3) to collect cycles in the mature space.

As we begin to explore more exotic permutations of tracing and reference counting, the power of the methodology becomes clear: it allows us to easily explore the design space of possible collector architectures, and to clearly classify them relative to each other.

Such an algorithm is shown in Figure 10. Its scan method is the same as for the standard generational collector (Figure 8), with the addition that references from live objects in the nursery to the mature space must increment the reference counts of the mature space objects prior to evacuation (to maintain the mature space invariant that its objects contain their heap reference counts).

The nursery-sweep function is also similar to that of the standard generational collector, except that reference counts of objects being moved from the nursery to the mature space must have the contributions from the roots subtracted. This is because the mature space is being collected by deferred reference counting, so once again we must maintain the DRC heap invariant.

```
urc-collect-nursery()
    urc-nursery-initialize()
    urc-nursery-scan()
    urc-nursery-sweep()

urc-nursery-initialize()
    R ← find-roots()
    R_N ← [r : r ∈ R ∧ r ∈ V_N]
    for each r ∈ R_N
        W ← W ⊎ [r]

urc-nursery-scan()
    while W ≠ ∅
        remove w from W
        ρ(w) ← ρ(w) + 1
        if ρ(w) = 1
            for each x ∈ [v : (w, v) ∈ E]
                if x ∈ V_N
                    W ← W ⊎ [x]
                else
                    ρ(x) ← ρ(x) + 1

urc-nursery-sweep()
    for each v ∈ V_N
        if ρ(v) = 0
            V_NF ← V_NF ∪ {v}
        else
            V_N ← V_N \ {v}
            V_H ← V_H ∪ {v}
    for each r ∈ R_N
        ρ(r) ← ρ(r) − 1

urc-assign(a, p)
    l ← [a]
    [a] ← p
    if inMatureSpace(a)
        urc-dec(l)
        urc-inc(p)

urc-dec(x)
    if x ∈ V_N
        ρ(x) ← ρ(x) − 1
    else
        W_H ← W_H ⊎ [x]

urc-inc(x)
    if x ∈ V_N
        W ← W ⊎ [x]
    else
        ρ(x) ← ρ(x) + 1

urc-collect-heap()
    urc-collect-nursery()
    collect-by-drc(W_H)
```

Figure 10: Traced Nursery and Reference Counted Heap

The assign function performs the same write barrier as the standard generational collector, except that pointers within the mature space are also reference counted: increments are applied eagerly, and decrements are placed in a separate work list for the mature space.

A full collection first collects the nursery, after which the nursery is empty and the mature space obeys the DRC invariant: the reference count of every object is the number of references from objects in the heap. Then the DRC algorithm is invoked collect garbage in the mature space.

5. CYCLE COLLECTION

One of the primary disadvantages of reference counting collectors is that they do not find cyclic garbage. Therefore, an additional mechanism is required.

So far our examination of reference counting has ignored cycles. We now consider various ways of collecting cyclic garbage.

There are two fundamental methods: a backup tracing collector, or a cycle collector. We first present these algorithms in the context of a single-heap collector.

5.1 Backup Tracing Collection

The first and most commonly used is a reference counting collector that occasionally performs a tracing collection in order to free cycles [43]. Reference counting is performed for both references from the roots to the heap and intra-heap references, but occasionally tracing is used over the whole heap, in which case it will collect garbage cycles missed by reference counting.

5.1.1 Reference Counting with Sticky Counts

A reference counting system with sticky counts is an extension of reference counting with tracing backup. A value $2^\xi - 1$ is chosen at which the reference count "sticks", and ceases to be further incremented or decremented. This is typically done to reduce the space that must be allocated in the object header to a few bits.

When tracing is performed, it recomputes all of the reference counts. Live objects whose count was stuck but now have less than $2^\xi - 1$ references will have the correct count, and dead objects (including those that were stuck and those that were part of garbage cycles) will have count 0.

5.2 Fix-point Formulation

In equation 1 we presented garbage collection as an abstract fix-point computation. Tracing computes the least fix-point V_L while reference counting computes the greatest fix-point. Thus the set difference between these two solutions comprises the cyclic garbage V_C.

There may exist other fix-point solutions between the least (tracing) and greatest (reference counting) fix-points. If there are n non-trivial strongly connected components in V_C then there will be between $n+1$ and 2^n solutions to the fix-point equation, depending upon the topology of the graph.

The general method for finding the cyclic garbage is to find a subset of nodes, $S \subseteq V$, such that

$$(S \cap R = \emptyset) \ \wedge \ \{(x,y) \in E : y \in S \wedge x \in V - S\} = \emptyset \quad (3)$$

In other words, S is a set which contains no roots and only internal references. Therefore, there is a fix-point solution in which for $s \in S, \rho(s) = 0$ and S is garbage.

The difficult question for cycle collection procedures is how to choose the set S.

```
collect-by-counting-with-cc()
    scan-by-counting()
    collect-cycles()
    sweep-for-counting()

collect-cycles()
    S ← ∅
    trial-deletion()
    trial-restoration()
    P ← ∅

trial-deletion()
    for each r ∈ P
        if ρ(r) = 0
            P ← P \ {r}
        else
            try-deleting(r)

try-deleting(v)
    if v ∉ S
        S ← S ∪ {v}
        for each w ∈ [x : (v, x) ∈ E]
            ρ(w) ← ρ(w) − 1
            try-deleting(w)

trial-restoration()
    for each r ∈ P
        try-restoring(r)

try-restoring(v)
    S ← S \ {v}
    if v ∈ S
        if ρ(v) > 0
            restore(v)
        else
            for each w ∈ [x : (v, x) ∈ E]
                try-restoring(w)

restore(v)
    for each w ∈ [x : (v, x) ∈ E]
        ρ(w) ← ρ(w) + 1
        if w ∈ S
            restore(w)

assign-cc(a, p)
    l ← [a]
    [a] ← p
    dec(l)
    inc(p)

inc(x)
    if x ≠ null
        ρ(x) ← ρ(x) + 1
        P ← P \ {x}

dec(x)
    if x ≠ null
        W ← W ⊎ [x]
        P ← P ∪ {x}
```

Figure 11: Algorithm for Reference Counting with Cycle Collection by Trial Deletion.

5.3 Reference Counting with Trial Deletion

We now consider reference counting with cycle collection by trial deletion rather than with a backup tracing collector.

In the trial deletion we start with a node x, which we suspect to be part of a garbage cycle, and consider the set $S = x^\star$ the set of all nodes reachable from x. If x is part of a cycle, then all nodes of the cycle are reachable from x. Therefore, the cycle is a subset of $x^\star$. We decrement the internal reference count to get the external reference count of each node. If we find any node has an external reference count greater than zero, we remove it from the set and restore all the reference counts for its children, since this node is now external to the set. We do this until no more nodes with non-zero external reference count are left in the set. If the set is not null, we have found a set of garbage nodes.

There remains the issue of how to choose the nodes x from which we start the trial deletion procedure. For example, the algorithm of Bacon and Rajan [6] produces a complete set of candidates (if trial deletion is performed on all candidates then all cyclic garbage will be found) while employing a number of heuristics to reduce the number of candidates.

The algorithm is shown in Figure 11. During mutation, the algorithm maintains a set P of "purple" vertices, those that are potentially roots of cyclic garbage. A vertex becomes purple when its reference count is decremented; it ceases to be purple when its reference count is incremented. At collection time, purple vertices with non-zero reference counts are considered as the potential roots of cyclic garbage. Trial deletion is then performed, with the set S representing those vertices that have been marked "gray", or visited by the algorithm. Finally, if trial deletion gives rise to a region of vertices with reference count 0, those vertices will be reclaimed by sweep(). Otherwise, the reference counts are restored.

Other algorithms in this family include those of Christopher [17] and of Martínez et al. [32].

6. MULTI-HEAP COLLECTORS

So far we have discussed whole-heap and generational garbage collection systems. We now extend our analysis of garbage collection to multiple heaps, which may be treated asymmetrically (as in generational systems) or symmetrically. Multi-heap garbage collection was pioneered by Bishop [10] for the purpose of allowing efficient collection of very large heaps that greatly exceeded the size of physical memory.

Multi-heap systems generally partition the heap in order to collect some regions independently of others. The benefits are reduced pause times and increased collector efficiency by preferentially collecting regions with few live objects.

Together, the duality of tracing and reference counting, its extension to multiple heaps, and the approaches to cycle collection provide an intellectual framework in which we can understand the inter-relation of fundamental design decisions and their algorithmic properties.

6.1 Ubiquitous use of Reference Counting

As is well known, reference counting is fundamentally incremental while tracing is not. However, we claim that *any* algorithm that collects some subset of objects independently is fundamentally making use of reference counting.

Reference counting allows one to collect an object x without consulting other objects. Similarly, incremental collection algorithms allow one to collect a set of objects X without consulting some other set of objects Y. (Here, we are referring to the class of collectors that incrementally collects the heap by scavenges a well-defined portion of the heap.) Thus the only issue is the *granularity* at which reference counting is performed. We use the generic term *macro-node* to refer to a collection of objects that typically serves as a unit of collection.

We have already seen that remembered sets are a reference count-like abstraction. Thus generational collectors keep reference counts for heap to nursery references. However, there is also a nursery-to-heap reference count, although it is implicit. By considering the nursery and the mature space as two macro-nodes, there is a macro-node edge from the nursery to the mature space. In other words, the mature space macro-node has at least a reference count of at least one even if there are edges from the root set. This implies that the mature space may not be collected independently from the nursery. Instead, we must first collect the nursery and move all live objects into the heap (at which point, the nursery macro-node disappears and the mature space macro-node's count drops to zero). Alternatively, we can perform a single, unified collection of the heap and the nursery by temporarily considering them as a single macro-node. Coalescing the macro-nodes eliminates the macro-node edge and is sensible since their internal edge type are both of the tracing flavor. A macro-node edge summarizes the fact that there may be edges between the constituent objects of the two macro-nodes.

Fundamentally, there are two ways of collecting cycles: (1) move all of the objects into a single heap and trace it, or (2) perform a trial deletion algorithm in which cyclic garbage is (logically) moved into a single region which is then entirely discarded. Moving participants of a cycle into a single heap does not imply that there must only be one such heap. For instance, it may be possible to statically or dynamically partition objects to avoid such references.

6.2 Object Relocation

Multi-heap collectors group objects into macro-nodes (variously called windows, cars, trains, regions, increments, and belts). In order to perform collection, these algorithms copy objects from one macro-node to another.

However, though reference counts can be used to detect that an object is live (or dead), direct pointers preclude object relocation without traversing the entire heap. Indirect pointers allow relocation without the space cost of remembered sets but pay the run-time cost of indirection and the space cost of both the indirection object and the fragmentation it may create. Read-barrier techniques typically treat every object as its own indirection object, except when it has actually been moved. This provides flexibility at the expense of read-barrier execution, and still suffers from fragmentation induced by indirection objects.

Remembered sets is the other approach to allowing relocation. They avoid the run-time cost of a read barrier while allowing incremental relocation and compaction, but the object relocation and the update of the pointers from its remembered set must be performed atomically, which limits the level of incrementality. In particular, when there are many references to a single object, remembered sets suffer from the *popular object problem*. The space cost of the remembered set is high because there are many incoming pointers, and the incrementality is poor because an arbitrary number of pointers must be updated in a single atomic step.

6.3 The Train Algorithm

Hudson and Moss designed an algorithm for garbage collection which later came to be know as the Train algorithm [25]. Seligmann and Garup found and fixed a small flaw in the algorithm and implemented it [37].

The primary purpose of the Train algorithm is to reduce the pause time associated with the collection of the Mature Object Space. It is assumed the Train is a generational system: there is a nursery

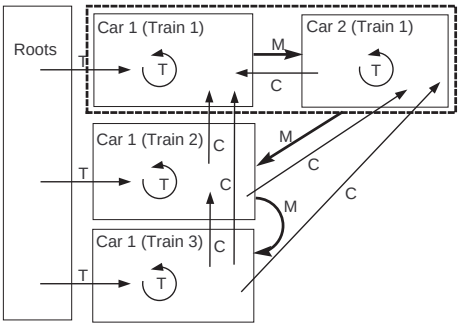


Figure 12: Schematic of the Train Algorithm

which keeps the recently created objects and promotes the objects found during its collection.

The train algorithm divides the Mature Object Space (MOS) into *cars* of fixed size. During the collection of the MOS one car is collected at a time – making the train algorithm incremental at the level of a car. Thus the pause time requirements determine the size of the cars. The cars are organized into *trains* of variable size. Both the cars in a train and the trains themselves are ordered. When an object is moved from the nursery to the MOS, it is generally moved to the end of first train, or to the second train if the first train is in the midst of being collected.

Collection is always done on the first train. If there are no external pointers to the first train (from roots or other trains), then all the objects in the train are garbage and the whole train is collected. If not, the first car of the first train is examined. If there are no incoming pointers (in the remembered set), then the car can be collected. If there are objects with pointers only from later cars in the first train, then they are moved to the last car which points to them, or, if there is not enough space in the last car, into any later car. If necessary, a new car is created for the object.

If there are pointers from later trains, then the object is moved to a later train. The best choice would be to move it to the last train that has pointers to this object. If there is a pointer to the object from outside the MOS (that is, from the nursery), then the object is moved to a later train. After this is done with all the objects in the current car with pointers from outside, there would be no pointers from outside the car, and so any remaining objects are garbage and the whole car can be collected. After this is done with all the cars in the first train, all the external pointers to the train will be gone, and the whole train can be collected.

For this to work, the algorithm maintains remembered sets of pointers to objects in each car from later cars in the same train and from objects in later trains, and from outside the MOS. The pointers from earlier trains and earlier cars need not be remembered since they will be collected before this car is collected.

6.3.1 The Train as Hybrid

We now consider the train algorithm from our perspective as shown in Figure 12. Obviously it has a tracing component, since tracing is performed within each car. However, there is also reference counting component since there are remembered sets to handle inter-car references.

The train algorithm also has the interesting feature that some reference counts are encoded *positionally:* pointers to subsequent cars and cars from subsequent trains are not recorded in remembered sets. Each car can be considered as a supernode aggregating objects and a train as a supernode of cars. The macro-node counts are induced by the links from earlier to later cars of the same train and from earlier to later trains.

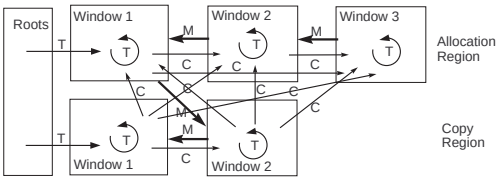


Figure 13: Schematic of the Older-First Algorithm

As in the generational case, the macro-node edges are directly reflected in the train algorithm. For example, the first train as a whole can be collected because there are no root or remembered set references after processing all the cars and because the macro-node count of the first train is zero.

6.3.2 Cycle Collection in the Train Algorithm

The hybridization in the train algorithm is especially apparent because it suffers from one of the fundamental problems of reference counting, namely cycle collection. The train algorithm works well with intra-car cycles since cars are collected with tracing. But inter-car and inter-train cycles can cause significant problems. But this is unsurprising as these are exactly the cycles at the macro-node level, which is reference counted rather than traced.

In the multi-heap collectors, such as the train algorithm, we use the same procedure. For each node in a train, we keep a list of pointers to it from nodes in the later trains (remembered sets). We collect the first train in the sequence of trains. If there are no pointers in the list, and no pointers from the root, then the set of nodes in this train satisfy the condition in equation 3 and therefore can be collected. Thus we can view the set of nodes in a train as S in our discussion in Section 5.2. The list of pointers from later trains can be viewed as a measure of external reference counts to S.

If however, a node x in train 1 is being pointed to by a node y in train n, then we move x to train n. This is equivalent to trial deletion from the point of view of train n, since the pointer from y to x is no longer part of any remembered set. At the same time, from the point of view of train 1, it corresponds to the restoration step of the trial deletion algorithm, since we have an external pointer to x, and therefore we removed it from the set corresponding to train 1.

After we have removed all the externally referenced nodes from train 1, the remaining nodes are garbage and are collected. In actual practice we collect a car at a time, rather than the whole train. But that is done only to limit the pause time, and does not essentially change the logic of the algorithm.

If x and y are part of a garbage set S, and train n is the highest numbered train containing nodes that can reach x, then the set S will be collected when train n is collected.

Thus the train algorithm, and trial deletion algorithm are both ways of finding the set S that satisfies the condition in equation 3.

6.4 The Older-First Algorithm

In [40], Stefanovič et al. advocated a collector which scavenges older objects before the youngest ones so that the young objects are more likely to die. In the extreme case, their oldest-only collector will always collect some subset of the oldest objects. However, it will copy old, live objects many times. Instead, they propose the older-first collector (Figure 13) which sweeeps through the heap, collecting groups of objects from the oldest to the youngest group.

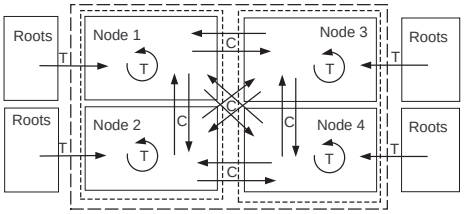


Figure 14: Schematic of Lang's Distributed GC Algorithm

Because of the sweep, recently allocated objects will be collected at least once before the oldest objects are reconsidered. Because older objects are collected first, the remembered sets in these collectors record references from older regions to younger objects, unlike a traditional generational collector.

Unlike the train collector, there are no macro-nodes other than the windows. Because cyclic garbage can permanently span multiple macro-nodes and be permanently uncollected, this algorithm is incomplete. In contrast, all cyclic garbage in a train algorithm eventually is promoted to a train that is otherwise dead at which point the entire train is collected en masse.

6.5 Distributed Garbage Collection

In [29], Lang et al. describe a garbage collector suitable for a distributed system as illustrated in Figure 14. Each node in the system performs tracing which conservatively detects garbage. However, cyclic garbage that spans more than one node will never be collected. They solve this with the concept of groups of processors, which correspond to our notion of macro-nodes. By using larger groups, all garbage will be eventually collected. In practice, a node may choose to not participate in a particular group collection. Reducing the group dynamically does not compromise correctness as long as the notion of exterior references is appropriately adjusted.

6.6 The Log-Structure Filesystem

It is instructive to consider how the implementation of a garbage collector changes as the trade-offs between operations change. Log-structured filesystems [35] implement a Unix filesystem by writing updated filesystem data and metadata sequentially to the disk. The Unix filesystem is a reference-counted directed acyclic graph (hard links can create acyclic cross-edges).

As files are overwritten, data earlier in the log becomes dead. To reclaim this data, "cleaning" (garbage collection) is performed. Cleaning is performed on fixed-size increments called segments: some number of segments are compacted at a time.

However, the performance of the system is critically dependent on the quality of the cleaning algorithm. In order to achieve good performance, a second level of reference counting is performed, treating the segments as macro-nodes (this is called the "segment usage table"). In addition, the age of each segment is recorded. The collector then tries to clean old segments with low reference counts – effectively dynamically selecting a nursery.

Such an approach is practical for filesystems because the overheads are large enough that the tradeoffs change. In log-structured filesystems, all pointers (inode numbers) are followed indirectly in order to allow the inodes themselves to move as their changed versions are appended to the log.

Therefore, there is no need to "fix" the pointers from other segments, and consequently there is no need for remembered sets. In addition, the "write barrier" is executed for disk block operations, so the additional overhead of maintaining a second reference count is trivial.

Thus we see that "obviously bad" garbage collection algorithms may work very well when garbage collection is applied to other domains than language run-time systems.

6.7 Other Multiple Heap Collectors

There are other potential bases for splitting the heap.

Shuf et al. allocate objects of a prolific type (high allocation rate) separately from those of a non-prolific type [38]. Optimizations are used to reduce the overhead of the remembered set between the two regions. It is like the distributed algorithm in that there are no macro-node edges.

The connectivity-based algorithm uses static analysis to obviate the write barriers so that no remembered sets are required [24]. Instead the macro-nodes and the macro-node edges are computed online to form a macro-node DAG. Whenever a particular macro-node needs to be collected, all its predecessors (transitively) must be collected at the same time.

The Beltway collector by Blackburn et al. generalizes various copying collectors including the tracing collector, generational tracing collector, and older-first collectors [11]. Like the train algorithm, there are two levels: belts (like trains) and increments (like cars). However, the macro-node edges between increments of the same belt follow the order found in the older-first collector. Because of this ordering, there is a potential problem with cyclic garbage which is solved by stipulating that the oldest belt be composed of a single (potentially large) increment. Additionally, aside from the degenerate oldest belt, the belts are not macro-nodes because a belt is never discarded as a whole.

7. COST ANALYSIS

For each collector, we analyze the cost in a common framework. This allows precise comparison of the strengths and weaknesses of each algorithm with respect to a given combination of storage availability and application characteristics. We analyze the cost per collection κ, the frequency of collection ϕ, and the total cost imposed by the garbage collector τ, including both collection and collector code inserted into the mutator (such as write barriers).

In general, we make steady state assumptions about the mutator: in particular, that the allocation rate and fraction of the heap that is garbage are constants. Naturally these are unrealistic assumptions, but they allow us to quantify the collectors with a reasonable number of variables, and provide a solid foundation for comparing the characteristics of the different collectors.

7.1 Cost Factors

We analyze the cost of each collector, including constant factors. However, we will not specify the coefficients for each parameter, since this would make the formulae unwieldy. The reader should be aware that these are real costs with implicit coefficients, rather than "big-Oh" notation.

The cost of a collector X is characterized with the following five quantities:

- $\kappa(X)$ is the time required for a single garbage collection in seconds;
- $\sigma(X)$ is the space overhead in units of m (defined below) words for the collector meta-data;
- $\phi(X)$ is the frequency of collection in hertz;
- $\mu(X)$ is the mutation overhead as a fraction of the application running time; and

- $\tau(X)$ is the total time overhead for collection. If a program in isolation takes 1 second to execute, and $\tau(X) = 0.5$, then the same program running with collector X takes 1.5 seconds to execute. the program

Quantities that are not parameterized by the collector name, such as μ_E or V_L, are invariant across all collector types.

7.2 Collector-Independent Parameters

To analyze the performance of the collectors, we define the following additional terms:

- m is the (fixed) size of each object, in words.
- $\mathcal{M}$ is the size of memory, in m-word objects. $\mathcal{M}' = m\mathcal{M}$ is the size of memory in words.
- ω is the object size in bits and $\omega' = \omega/m$ is the word size in bits. Because a word must be large enough to hold the address of any word, we also have $\omega' \geq \lceil \log_2 \mathcal{M}' \rceil$.
- a is the allocation rate, in objects/second.
- ϵ is the pointer density, that is the average number of non-null pointers per object;
- μ_R is the root mutation rate in writes/second.
- μ_E is the edge mutation rate in writes/second.
- $\rho(v)$ is the reference count of v.

For sets and multisets, we use script letters to denote cardinality. For example, $\mathcal{V}_L = |V_L|$ is the number of live vertices.

Note that while the set of live vertices V_L is always collector-independent, the total number of vertices V and the number of dead vertices V_D are both collector-dependent. For collector X,

$$\mathcal{V}(X) = \mathcal{M} - \sigma(X) \tag{4}$$

and the free memory available after collection is

$$\mathcal{V}_D(X) = \mathcal{V}(X) - \mathcal{V}_L \tag{5}$$

also, since we are assuming a pointer density of ϵ it follows that $\mathcal{E}_L = \epsilon\mathcal{V}_L$ and $\mathcal{E}_D(X) = \epsilon\mathcal{V}_D(X)$.

7.3 Total Time

The total time cost of collection is generally

$$\tau(X) = \phi(X)\kappa(X) \ + \ \mu(X). \tag{6}$$

For each collector we will define the cost of the component terms, and then present an equation for the total cost that is expressed in terms of collector-independent parameters. This allows direct comparison of the different algorithms.

The actual cost of collection operations is dependent on the particular implementation and the hardware/software platform upon which it runs.

Therefore, we convert the abstract component operations into time via an n-ary linear function with implicit coefficients denoted by $\mathcal{L}(...)$. For example, in a mark-sweep collector, there is a cost for each root, mark cost for each live vertex, a mark-check cost for each live edge, and a sweep-cost for all vertices. So the time cost is $\mathcal{L}(\mathcal{R}, \mathcal{V}_L, \mathcal{E}_L, \mathcal{V})$.

7.4 Unified Heap Collectors

We begin with a cost analysis of the simple tracing and reference counting algorithms, and then examine the various unified-heap hybrid algorithms.

```
void mark(Object obj) {
    do {
        obj.mark();
        Object cdr ← null;

        for (Object p: obj.referents())
            if (p ≠ null && ¬ p.isMarked())
                if (cdr = null)
                    cdr ← p;
                else
                    mark(p);

        obj ← cdr;

    } while (obj ≠ null);
}
```

Figure 15: Optimization of traversal stack space during the mark phase.

7.5 T: Tracing

We begin by discussing the space and time costs for the simple tracing garbage collection algorithm; these formulae form the basis of subsequent trace-based collector analyses.

7.5.1 Space Cost

There are two fundamental sources of space overhead in tracing collection: the mark bits and the stack for the recursive traversal of the object graph. While pointer reversal techniques [36] exist that allow the elimination of the latter overhead, this is a significant space-time trade-off since it means more writes and more traversal. In this paper, to avoid an explosion of alternate cost models, we will assume that pointer reversal is not used, while noting that it is an option.

The space cost for the traversal stack is proportional to the depth of the object graph. This is typically viewed as problematic given the existence of long linked structures. However, we argue that with a few simple optimizations which have the flavor of tail-recursion elimination, the traversal depth can be greatly reduced at very little cost.

Consider the code in Figure 15. The mark(obj) function has been modified slightly so that its parameter is known to be non-null and unmarked. The loop over the referents of obj stores the first non-null pointer to an unmarked object that it encounters in the variable cdr. If more non-null unmarked referents are discovered, the mark(p) call recurses to process them, consuming another stack frame.

However, if there is at most one non-null unmarked referent, there is no recursion. The function either returns if there is no non-null unmarked referent, or if there is exactly one it handles the cdr pointer via tail-recursion elimination.

As a result, both singly- and doubly-linked lists will have a recursion depth of 1: singly-linked lists because there is only one referent, and doubly-linked lists because one of the two referents is guaranteed to have been previously marked.

We therefore define

- $\mathcal{D}'$ as the *traversal depth* of the object graph at collection time, that is the depth of the recursion stack required for traversing the live object graph after the above optimizations have been applied.

The space cost for the mark phase will then be proportional to $\mathcal{D}'$. In a system that works with interior pointers, the space is sim-

ply $\mathcal{D}'$ pointers. In a system without interior pointers, a "cursor" into the object being traversed must be separately maintained. The space cost $\mathcal{D}$, measured in objects, of the traversal stack is therefore

$$\mathcal{D} = \frac{\mathcal{D}'}{m} \tag{7}$$

or, in a system without interior pointers,

$$\mathcal{D} = \frac{\mathcal{D}'}{m} \cdot \left(1 + \frac{\lceil \log_2 m \rceil}{\omega}\right) \tag{8}$$

For programs whose data structures are tree-like, the traversal depth will be logarithmic rather than linear in the number of objects. For more general graph structures, the depth is less predictable. Since $\mathcal{D}$ is in general not predictable, and it is undesirable to reserve $\mathcal{V}(T)$ words for the traversal stack, most collectors in fact use a fixed-size pre-allocated traversal stack. When the stack overflows, an alternative method is used. While pointer reversal is asymptotically better, in practice it is often preferable to use a simple $O(n^2)$ algorithm in which the heap is simply rescanned when the recursion stack overflows.

7.5.2 Generic Space Formula

We now present a generic formula for space overhead that can be specialized to a number of collectors, including tracing.

We generically assume that b bits are required for the "reference count". If $b = 1$, it is a mark bit; if $b = \omega'$, it is a full-word reference count; if $1 < b < \omega'$, it represents a trade-off between these extremes.

For some collector X that uses b bits per object and has a traversal stack of depth $\mathcal{D}$, the space required is

$$\begin{aligned}
\sigma(X) &= \mathcal{D} + \frac{b\mathcal{V}(X)}{\omega} \\
\mathcal{V}(X) &= \mathcal{M} - \sigma(X) \\
&= \mathcal{M} - \mathcal{D} - \frac{b\mathcal{V}(X)}{\omega} \\
&= \frac{\mathcal{M} - \mathcal{D}}{1 + \frac{b}{\omega}} \qquad (9) \\
\sigma(X) &= \frac{\mathcal{D} + \frac{b}{\omega}\mathcal{M}}{1 + \frac{b}{\omega}} \qquad (10)
\end{aligned}$$

7.5.3 Space Cost of the Tracing Collector

For a tracing collector that is optimized to use a single mark bit per object, $b = 1$ and the formula simplifies to

$$\begin{aligned}
\mathcal{V}(T) &= \frac{\mathcal{M} - \mathcal{D}}{1 + \frac{1}{\omega}} \qquad (11) \\
\sigma(T) &= \frac{\mathcal{D} + \frac{\mathcal{M}}{\omega}}{1 + \frac{1}{\omega}} \qquad (12)
\end{aligned}$$

On a machine with a word size $\omega' = 32$ and a fixed object size $m = 2$ (such as Lisp cons cells), $\omega = 64 \gg 1$, so we can approximate the space required as one bit for every possible object (as if there were no metadata at all), plus the space for the recursion stack. Note that we implicitly also assume that $\mathcal{M} \gg \mathcal{D}$.

$$\begin{aligned}
\mathcal{V}(T) &\simeq \mathcal{M} - \mathcal{D} \qquad (13) \\
\sigma(T) &\simeq \frac{\mathcal{M}}{\omega} + \mathcal{D} \qquad (14)
\end{aligned}$$

7.5.4 Time Cost

The cost per collection is proportional to the size of the root set, the number of vertices examined, and the number of edges traversed for the trace (mark) phase, and the total number of vertices for the sweep phase. So the total cost of a tracing collection is

$$\kappa(T) = \mathcal{L}_T(\mathcal{R}, \mathcal{V}_L, \mathcal{E}_L, \mathcal{V}(T)) \tag{15}$$

A collection has to be performed after the reclaimed cells are consumed, so the frequency of collection (in collections/second) is

$$\begin{aligned}
\phi(T) &= \frac{a}{V_D(T)} \\
&\simeq \frac{a}{\mathcal{M} - \mathcal{D} - \mathcal{V}_L} \qquad (16)
\end{aligned}$$

using the approximation from Equation 13.

Therefore the total cost of collection is

$$\begin{aligned}
\tau(T) &= \phi(T) \cdot \kappa(T) \\
&\simeq \frac{a}{\mathcal{M} - \mathcal{D} - \mathcal{V}_L} \mathcal{L}_T(\mathcal{R}, \mathcal{V}_L, \mathcal{E}_L, \mathcal{V}(T)) \qquad (17)
\end{aligned}$$

7.6 C: Reference Counting

We now develop space and time cost functions for reference counting and its derivative collectors.

7.6.1 Space Cost

While reference counting does not perform tracing, it does perform recursive deletion when an object's reference count drops to zero. There is no difference in principle between the two, so it would seem that the amount of space consumed by recursive traversal is the same.

However, by using Weizenbaum's method for non-recursive freeing [42], the space for the traversal stack can be eliminated, at the expense of delaying freeing until the next allocation.

There is in fact a much simpler variant of "pointer threading" that can be applied which allows efficient and immediate traversal of a stack of dead objects without requiring any extra memory or cache read/write operations. To our knowledge this technique has not been published previously.

Recursive deletion implies that the object from which we are recursing has aleady been identified as dead; therefore, we do not need to preserve the data values it contains; we only need to preserve the pointers which have not yet been traversed. The ability to use the space in dead objects is the key to our technique.

When recursively traversing the object graph to decrement reference counts and free those with $\rho = 0$, there are two fundamental pieces of information in every stack frame: one points to the object being collected; the other identifies which field was most recently processed. When we recurse for the first time, there is obviously at least one pointer in the object, and since we are traversing it, we no longer need it. Therefore, we can store a pointer to the parent object in that freed pointer cell. And since we already know the reference count of the object (zero), we can over-write the reference count with a field that indicates which pointer we are currently traversing.

A reference count is necessarily large enough to count the total number of pointers in an object, since they might all point to some other (reference counted) object. Therefore, we can guarantee that any object that might cause recursion will have sufficient space to store the necessary "stack frame" of the recursive deletion.

Furthermore, this technique does not require the reading or writing of any additional cache lines – the first cache line of each object has to be examined anyway to perform the storage reclamation.

However, unlike with tracing, the reference count field can not be optimized down to a single bit: it occupies a full word (this assumption can be relaxed with the use of sticky counts).

$$\mathcal{V}(C) = \frac{\mathcal{M}}{1+\frac{1}{m}} = \frac{m\mathcal{M}}{m+1} \quad (18)$$

$$\sigma(C) = \frac{\mathcal{M}}{m+1} \quad (19)$$

That is, the space overhead is one word per object, which increases the effective object size by 1 word, which is precisely what we obtain by substituting $b = \omega'$ and $\mathcal{D} = 0$ into equations 9 and 10.

7.6.2 Time Cost

The cost per collection is simply

$$\kappa(C) = \mathcal{L}_{C_1}(\mathcal{V}_D(C), \mathcal{E}_D(C)) \quad (20)$$

in other words, the cost of identifying the dead vertices.

The frequency of collection is the same as with tracing, namely

$$\phi(C) = \frac{a}{\mathcal{V}_D(C)} \quad (21)$$

7.6.3 Mutation Cost

However, reference counting skips the initialization step performed by tracing and instead inserts a write barrier into the code. Therefore, the total cost must include the mutation rate, which for naïve reference counting is proportional to μ_R and μ_E, so

$$\mu(C) = \mathcal{L}_{C_2}(\mu_R, \mu_E) \quad (22)$$

So the total cost of reference counting in the absence of cycles is

$$\begin{aligned} \tau(C) &= \phi(C)\cdot\kappa(C) + \mu(C) \\ &= \frac{a}{\frac{m\mathcal{M}}{m+1} - \mathcal{V}_L}\mathcal{L}_{C_1}(\mathcal{V}_D(C), \mathcal{E}_D(C)) + \\ &\quad \mathcal{L}_{C_2}(\mu_R, \mu_E) \end{aligned} \quad (23)$$

7.7 CT: Reference Counting, Tracing Backup

We now consider reference counting collectors which periodically use a tracing collector to collect cyclic garbage, as described in Section 5.1.

7.7.1 Space Cost

In the CT algorithm, each object has a reference count, and space is required for recursive traversal by the tracing portion of the collector. Therefore, the space cost of CT is

$$\mathcal{V}(CT) = \frac{\mathcal{M}-\mathcal{D}}{1+\frac{1}{m}} \quad (24)$$

$$\sigma(CT) = \frac{\mathcal{M}+\mathcal{D}}{1+\frac{1}{m}} \quad (25)$$

7.7.2 Time Cost

In order to evaluate the performance of cycle collection, we define

- $c \le a$, the rate of generation of cyclic garbage.

In order to compute the total cost of the collector we must compute the costs due to both reference counting and tracing. We will examine the cost from the point of view of the tracing collections, since those happen on a regular (and less frequent) basis. This leads to a simpler analysis; if one tries to evaluate the cost from the point of view of the reference counting portion of the collector, the model is complicated because each successive collection frees less memory, causing the frequency of collection to increase over time, until a tracing collection is performed.

From the point of view of the tracing collector, one can think of the reference counting collector as a "nursery" collector that reduces the allocation rate into the "mature space" of the tracing collector. In particular, while the application allocates data at rate a, the system only creates data for the tracing collector at rate c, since all garbage except cyclic garbage is collected by the reference counting collector. Therefore, the rate of tracing collection is

$$\phi(CT) = \frac{c}{\mathcal{V}_D(CT)} \quad (26)$$

which is the same as the equation for $\phi(T)$ except that c has replaced a. For the purposes of modelling the two collectors together, we will assume that the reference counting collector has the same frequency as the tracing collector.

The cost of the tracing collection itself is unchanged, so

$$\kappa(CT_T) = \kappa(T) = \mathcal{L}_T(\mathcal{R}, \mathcal{E}_L, \mathcal{V}_L, \mathcal{V}(CT)) \quad (27)$$

To analyze the cost of reference counting, we will now account for the cost as though it were run in a continuous, incremental fashion. There are $\frac{\mathcal{V}_D}{c}$ seconds between tracing collections, during which $\frac{\mathcal{V}_D}{c}a$ cells are allocated. Of those, $\mathcal{V}_D$ are not freed by reference counting (because they are cyclic garbage). So the number of cells freed by reference counting between tracing collections is

$$a\frac{\mathcal{V}_D}{c} - \mathcal{V}_D = \left(\frac{a}{c} - 1\right)\mathcal{V}_D$$

We assume that the pointer density is uniform, so that for each vertex v we traverse $\mathcal{E}_D/\mathcal{V}_D$ pointers, or $\mathcal{E}_D$ pointers in all. Therefore, the cost of reference counting for each "tracing period" is

$$\kappa(CT_C) = \left(\frac{a}{c} - 1\right)\cdot\mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D) \quad (28)$$

Combining these results we get the total cost of reference counting with a tracing backup collector as

$$\begin{aligned} \tau(CT) &= \mathcal{L}_{C_2}(\mu_R, \mu_E) + \phi(CT)\cdot(\kappa(CT_T) + \kappa(CT_C)) \\ &= \mathcal{L}_{C_2}(\mu_R, \mu_E) + \frac{a}{\mathcal{V}_D}\mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D) + \\ &\quad \frac{c}{\mathcal{V}_D}(\mathcal{L}_T(\mathcal{R}, \mathcal{V}_L, \mathcal{E}_L, \mathcal{V}(CT)) - \mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D)) \end{aligned} \quad (29)$$

Of course, when there is no cyclic garbage $c = 0$ and the last term of the equation drops out, so that $\tau(CT) = \tau(C)$.

7.8 CD: Deferred Reference Counting

Deferred reference counting (Section 4.1) moves the cost of considering the roots from the application to the collector. Therefore, the cost of collection becomes

$$\kappa(CD) = \mathcal{L}_{CD}(\mathcal{R}) + \mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D) \quad (30)$$

The frequency of collection is the same as with tracing and reference counting, namely

$$\phi(CD) = \phi(C) = \frac{a}{\mathcal{V}_D} \quad (31)$$

so the total cost of collection is

$$\begin{aligned} \tau(CD) &= \phi(CD)\cdot\kappa(CD) + \mathcal{L}_{C_2}(0, \mu_E) \\ &= \frac{a}{\mathcal{V}_D}(\mathcal{L}_{CD}(\mathcal{R}) + \mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D)) + \\ &\quad \mathcal{L}_{C_2}(0, \mu_E) \end{aligned} \quad (32)$$

7.8.1 Space Cost

The space cost of deferred reference counting is the cost of keeping the reference costs plus the space consumed by the zero count table (ZCT). In general the number of entries in the ZCT can equal the total number of live and dead objects in the system. Therefore, robust implementations use a bit in each object to indicate whether it is the ZCT. So the total cost is the space for the reference counts plus the space for the ZCT "presence bits":

$$\begin{aligned}\sigma(CD) &= \frac{\mathcal{V}}{\omega} + \frac{\mathcal{V}\lceil \log_2 \mathcal{V} \rceil}{\omega} \\ &= \frac{\mathcal{V}}{\omega} \cdot (1 + \lceil \log_2 \mathcal{V} \rceil) \end{aligned} \tag{33}$$

7.9 CC: Reference Counting, Trial Deletion

We now consider reference counting with cycle collection by trial deletion (Section 5.3) rather than with a backup tracing collector.

The cost of this algorithm is the cost of the basic reference counting algorithm plus the additional work performed by the trial deletion of cycles. However, trial deletion will also be applied to some portion of the live subgraph, and this cost must also be factored in. We introduce the parameter

- λ, the fraction of live vertices V_L that are examined by the trial deletion algorithm ($0 \leq \lambda \leq 1$).

To compute the cost of collection, we use the same methodology as with the CT algorithm: the trial deletion phase CC_D is viewed as "dominant" and collection is described in terms of the period of that collection.

The trial deletion phase is invoked when all free memory is consumed by cyclic garbage:

$$\phi(CC) = \phi(CT) = \frac{c}{\mathcal{V}_D} \tag{34}$$

The cost of the trial deletion algorithm is two traversals over all of the cyclic garbage (which is all of V_D in this case) plus the traversals over the live subgraph:

$$\kappa(CC_D) = \mathcal{L}_{CC_D}(\mathcal{V}_D + \lambda\mathcal{V}_L, \mathcal{E}_D + \lambda\mathcal{E}_L) \tag{35}$$

The time cost of the reference counting collection for non-cyclic garbage is the same as for the CT algorithm:

$$\kappa(CC_C) = \kappa(CT_C) = \left(\frac{a}{c} - 1\right) \cdot \mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D) \tag{36}$$

Thus the total cost of the CC collector is

$$\begin{aligned}\tau(CC) &= \phi(CC) \cdot (\kappa(CC_D) + \kappa(CC_C)) + \mathcal{L}_{C_2}(\mu_R, \mu_E) \\ &= \mathcal{L}_{C_2}(\mu_R, \mu_E) + \frac{a}{\mathcal{V}_D}\mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D) + \\ &\quad \frac{c}{\mathcal{V}_D}(\mathcal{L}_{CC_D}(\mathcal{V}_D, \lambda\mathcal{V}_L, \mathcal{E}_D, \lambda\mathcal{E}_L) - \mathcal{L}_{C_1}(\mathcal{V}_D, \mathcal{E}_D)) \end{aligned} \tag{37}$$

7.9.1 Space Cost

The space cost of reference counting with cycle collection is the cost of reference counts plus the space consumed by the purple set P plus the per-object color information, represented by the set Y in the abstract algorithm. Let π be the fraction of nodes that are purple, then the space cost is

$$\sigma(CC) = \mathcal{V}(CC)\left(\frac{\lceil \log_2 \mathcal{V}(CC) \rceil}{\omega} + \frac{1}{\omega} + \frac{\pi}{m}\right) \tag{38}$$

7.10 Split-Heap Collectors

We characterize split-heap collectors with the following quantities:

- $N \subset V$ is the nursery.
- $H = \overline{N}$ is the mature space heap.
- V_{NL} and E_{NL} are the set of live vertices and the set of live edges in the nursery.
- V_{ND} and E_{ND} are the set of dead vertices and the set of dead edges in the nursery.
- V_{HL}, E_{HL}, V_{HD}, and E_{HD} are the corresponding sets in the mature space heap.

7.11 GT: Tracing Generational Collection

We now consider standard tracing generational collectors as described in Section 4.3.1. We will assume a simple generational collector in which the size of the nursery is fixed and objects are evacuated into the mature space after surviving a single nursery collection. We will use the term "heap" in this section as a synonym for "mature space".

To collect the nursery we trace forward within the nursery from the roots R and the roots R' from the mature space into the nursery. Afterwards, the nursery is cleared. The cost is

$$\kappa(GT_N) = \mathcal{L}_T(\mathcal{R} + \mathcal{R}', \mathcal{V}_{NL}, \mathcal{E}_{NL}, \mathcal{N}) \tag{39}$$

The frequency of nursery collection is simply

$$\phi(GT_N) = \frac{a}{\mathcal{N}} \tag{40}$$

The size of the mature root set can be as high as V_H and is further discussed below.

Collecting the mature heap is just like performing a simple tracing collection. The cost is

$$\kappa(GT_H) = \mathcal{L}_T(\mathcal{R}, \mathcal{V}_{HL}, \mathcal{E}_{HL}, \mathcal{H}) \tag{41}$$

The allocation rate into the heap is attenuated by the use of the nursery. The resulting heap allocation rate is $a\frac{\mathcal{V}_{NL}}{\mathcal{N}}$. Therefore the frequency of heap collection is

$$\phi(GT_H) = \frac{a\frac{\mathcal{V}_{NL}}{\mathcal{N}}}{\mathcal{V}_{HD}} \tag{42}$$

The total cost of generational collection is the sum of the costs of the nursery collections, the heap collections, and the write barriers:

$$\begin{aligned}\tau(GT) &= \phi(GT_N) \cdot \kappa(GT_N) + \\ &\quad \phi(GT_H) \cdot \kappa(GT_H) + \mathcal{L}_{GT}(\mu_E) \\ &= \frac{a}{\mathcal{N}} \cdot \mathcal{L}_T(\mathcal{R} + \mathcal{R}', \mathcal{V}_{NL}, \mathcal{E}_{NL}, \mathcal{N}) + \\ &\quad \frac{a\frac{\mathcal{V}_{NL}}{\mathcal{N}}}{\mathcal{V}_{HD}} \cdot \mathcal{L}_T(\mathcal{R}, \mathcal{V}_{HL}, \mathcal{E}_{HL}, \mathcal{H}) + \mathcal{L}_{GT}(\mu_E) \end{aligned} \tag{43}$$

7.11.1 Space Cost

Like its non-generational counterpart, there is a space cost for the mark bits and the traversal depth. In addition, if the remembered set is maintained as a sequential store buffer (i.e. a list of modified objects), it can take V_H space if every object in the mature space is modified between consecutive collection. The space overhead is equal to that of the tracing collector plus the remembered set

$$\sigma(GT) = D + V/\omega$$

7.12 GCH: Generational with Counted Heap

We now consider a generational collector with a traced nursery and a reference counted heap (Section 4.3.3). For the nursery, the cost and frequency is identical to the GT algorithm. The frequency of heap collection is also the same. The cost of heap collection is the cost of deferred reference counting applied to the heap.

$$\kappa(GCH_N) = \kappa(GT_N) \quad (44)$$

$$\phi(GCH_N) = \phi(GT_N) = \frac{a}{\mathcal{N}} \quad (45)$$

$$\kappa(GCH_H) = \mathcal{L}_{CD}(\mathcal{R}) + \mathcal{L}_{C_1}(\mathcal{V}_{HD}(GCH), \mathcal{E}_{HD}(GCH)) \quad (46)$$

$$\phi(GCH_H) = \phi(GT_H) = \frac{a\frac{\mathcal{V}_{NL}}{\mathcal{N}}}{\mathcal{V}_{HD}(GCH)} \quad (47)$$

$$\tau(GCH) = \phi(GCH_N) \cdot \kappa(GCH_N) + \phi(GCH_H) \cdot \kappa(GCH_H) + \mathcal{L}_{GT}(\mu_{E_H}) \quad (48)$$

8. SPACE-TIME TRADEOFFS

So far we have always considered a single, canonical implementation of each collector. Of course, there are many implementation decisions that can be made that will cause the performance characteristics to vary. However, almost all of these are some form of space-time trade-off that can be applied to individual memory regions once the basic collector design has been selected.

8.1 Copying Collectors

Copying collectors trade space for time by allocating an extra semi-space and copying the live data into the new semi-space. Thus they save the factor of $\mathcal{V}$ iteration over the dead data required by mark-and-sweep style collectors (the sweep-for-tracing() function of Figure 3), but at an equivalent cost of $\mathcal{V}/2$ in space.

8.2 Remembered Set

Although the remembered set in a generational collector could contain all objects in the mature space, this is very unlikely. Pre-allocating space for this situation is excessive over-provisioning. Instead, one can dynamically borrow space for the remembered set from the mature space and nursery. It is preferable to borrow from the mature space. If the mutation rate is low enough that the borrowed space is not exhausted, the nursery collection frequency is unaffected. Alternatively, borrowing space from the nursery will definitely increase the frequency of minor collection to

$$\phi(GT_N) = a/(N - \Omega) \quad (49)$$

where Ω is the amount of space allowed for the remembered set in the nursery. This technique trades space for time by making the usually true assumption that mutation rate is not that high.

Alternatively, one can consider a bitmap or card-marking approach where the overhead is greatly diminished by either w or k, the size of the card. In the former case, there is no loss of precision but the write barrier must perform bit updates. With card marking, the write barrier code is shorter and faster though the loss in precision (one card corresponds many objects) will result in a longer scanning time during garbage collection.

For slot-based bitmap, the space overhead is reduced to

$$\sigma(GT) = D + V/\omega + V_H/\omega \quad (50)$$

For card marking, the space overhead is even less:

$$\sigma(GT) = D + V/\omega + V_H/k \quad (51)$$

where k is the card size.

8.3 Traversal

For tracing collectors, the recursive traversal in the reconstruct function requires stack space D. This can be eliminated entirely at the cost of an extra pass over the live objects V_L by using pointer reversal [36]. However, this is usually a poor trade-off because the stack depth in practice is much smaller than the number of live objects ($\mathcal{D} \ll \mathcal{V}_L$).

A better strategy is to use a fixed-size recursion stack and then use some technique to handle stack overflow. One can either fall back on pointer reversal when the stack overflows, or one can discard the stack and begin scanning the heap from left to right to find unmarked objects. While this approach is simpler to implement and is a more natural implementation in a concurrent system, it does have the distinct disadvantage that its complexity is n^2.

9. CONCLUSIONS

We have shown that tracing and reference counting garbage collection, which were previously thought to be very different, in fact share the exact same structure and can be viewed as duals of each other.

This in turn allowed us to demonstrate that all high-performance garbage collectors are in fact hybrids of tracing and reference counting techniques. This explains why highly optimized tracing and reference counting collectors have surprisingly similar performance characteristics.

In the process we discovered some interesting things: a write barrier is fundamentally a feature of reference counting; and the existence of cycles is what makes garbage collection inherently non-incremental: cycle collection is "trace-like".

We have also provided cost measures for the various types of collectors in terms of collector-independent and -dependent quantities. These cost measures do not "cheat" by for example ignoring the space cost of collector metadata. They therefore allow direct "head-to-head" comparisons.

Design of collectors can be made more methodical. When constructing a collector, there are three decisions to be made:

Partition. Divide memory (heap, stack, and global variables) into a set of partitions, within which different strategies may be applied;

Traversal. For each partition, decide whether the object graph will be traversed by tracing or reference counting; and

Trade-offs. For each partition, choose space-time trade-offs such as semi-space vs. sliding compaction, pointer reversal vs. stack traversal, etc.

Our unified model of garbage collection allows one to systematically understand and explore the design space for garbage collectors, and paves the way for a much more principled approach to selecting a garbage collector to match the characteristics of the associated application.

In the future, this methodology may help enable the dynamic construction of garbage collection algorithms that are tuned to particular application characteristics.

Acknowledgements

Alex Aiken suggested the fix-point formulation of the garbage collection algorithm. We gratefully thank Ras Bodik, Hans Boehm, Richard Fateman, Matthew Fluet, Richard Jones, Greg Morrisett, Chet Murthy, and George Necula, and the anonymous referees for their helpful comments.

This work began with a “Five Minute Madness” presentation by the first author at the 2004 Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE) Workshop in Venice, Italy. We thank the organizers of and participants in the workshop for providing the stimulating atmosphere which gave rise to this work.

10. REFERENCES

[1] APPEL, A. W. Simple generational garbage collection and fast allocation. *Software – Practice and Experience 19*, 2 (1989), 171–183.

[2] APPEL, A. W., ELLIS, J. R., AND LI, K. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988). *SIGPLAN Notices, 23*, 7 (July), 11–20.

[3] BACON, D. F., ATTANASIO, C. R., LEE, H. B., RAJAN, V. T., AND SMITH, S. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices, 36*, 5 (May), 92–103.

[4] BACON, D. F., CHENG, P., AND RAJAN, V. T. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems* (San Diego, California, June 2003). *SIGPLAN Notices, 38*, 7, 81–92.

[5] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices, 38*, 1, 285–298.

[6] BACON, D. F., AND RAJAN, V. T. Concurrent cycle collection in reference counted systems. In *European Conference on Object-Oriented Programming* (Budapest, Hungary, June 2001), J. L. Knudsen, Ed., vol. 2072 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 207–235.

[7] BAKER, H. G. List processing in real-time on a serial computer. *Commun. ACM 21*, 4 (Apr. 1978), 280–294.

[8] BAKER, H. G. The Treadmill, real-time garbage collection without motion sickness. *SIGPLAN Notices 27*, 3 (Mar. 1992), 66–70.

[9] BARTH, J. M. Shifting garbage collection overhead to compile time. *Commun. ACM 20*, 7 (July 1977), 513–518.

[10] BISHOP, P. B. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Laboratory for Computer Science, Massachussets Institute of Technology, May 1977. MIT/LCS/TR-178.

[11] BLACKBURN, S. M., JONES, R., MCKINLEY, K. S., AND MOSS, J. E. B. Beltway: getting around garbage collection gridlock. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Berlin, Germany, June 2002). *SIGPLAN Notices, 37*, 5, 153–164.

[12] BLACKBURN, S. M., AND MCKINLEY, K. S. Ulterior reference counting: Fast garbage collection without a long wait. In *Proceedings of the Conference on Object-oriented Programing, Systems, Languages, and Applications* (Anaheim, California, Oct. 2003). *SIGPLAN Notices, 38*, 11, 344–358.

[13] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.

[14] CHEADLE, A. M., FIELD, A. J., MARLOW, S., PEYTON JONES, S. L., AND WHILE, R. L. Non-stop Haskell. In *Proc. of the Fifth International Conference on Functional Programming* (Montreal, Quebec, Sept. 2000). *SIGPLAN Notices, 35*, 9, 257–267.

[15] CHENEY, C. J. A nonrecursive list compacting algorithm. *Commun. ACM 13*, 11 (1970), 677–678.

[16] CHENG, P., AND BLELLOCH, G. A parallel, real-time garbage collector. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (Snowbird, Utah, June 2001). *SIGPLAN Notices, 36*, 5 (May), 125–136.

[17] CHRISTOPHER, T. W. Reference count garbage collection. *Software – Practice and Experience 14*, 6 (June 1984), 503–507.

[18] COLLINS, G. E. A method for overlapping and erasure of lists. *Commun. ACM 3*, 12 (Dec. 1960), 655–657.

[19] DETREVILLE, J. Experience with concurrent garbage collectors for Modula-2+. Tech. Rep. 64, DEC Systems Research Center, Aug. 1990.

[20] DEUTSCH, L. P., AND BOBROW, D. G. An efficient incremental automatic garbage collector. *Commun. ACM 19*, 7 (July 1976), 522–526.

[21] DIJKSTRA, E. W., LAMPORT, L., MARTIN, A. J., SCHOLTEN, C. S., AND STEFFENS, E. F. M. On-the-fly garbage collection: An exercise in cooperation. In *Hierarchies and Interfaces*, F. L. Bauer et al., Eds., vol. 46 of *Lecture Notes in Computer Science*. Springer-Verlag, 1976, pp. 43–56.

[22] DOLIGEZ, D., AND LEROY, X. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *Conf. Record of the Twentieth ACM Symposium on Principles of Programming Languages* (Jan. 1993), pp. 113–123.

[23] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, July 1998.

[24] HIRZEL, M., DIWAN, A., AND HERTZ, M. Connectivity-based garbage collection. In *Proceedings of the Conference on Object-oriented Programing, Systems, Languages, and Applications* (Anaheim, California, Oct. 2003). *SIGPLAN Notices, 38*, 11, 359–373.

[25] HUDSON, R. L., AND MOSS, J. E. B. Incremental collection of mature objects. In *Proc. of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*, pp. 388–403.

[26] JOHNSTONE, M. S. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, University of Texas at Austin, Dec. 1997.

[27] KUNG, H. T., AND SONG, S. W. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposium on Foundations of Computer Science* (1977), pp. 120–131.

[28] LAMPORT, L. Garbage collection with multiple processes: an exercise in parallelism. In *Proc. of the 1976 International Conference on Parallel Processing* (1976), pp. 50–54.

[29] LANG, B., QUENNIAC, C., AND PIQUER, J. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages* (Jan. 1992), SIGPLAN Notices, pp. 39–50.

[30] LAROSE, M., AND FEELEY, M. A compacting incremental collector and its performance in a production quality compiler. In *Proc. of the First International Symposium on Memory Management* (Vancouver, B.C., Oct. 1998). *SIGPLAN Notices, 34*, 3 (Mar., 1999), 1–9.

[31] LEVANONI, Y., AND PETRANK, E. 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* (Tampa Bay, Florida, Oct. 2001), pp. 367–380.

[32] MARTÍNEZ, A. D., WACHENCHAUZER, R., AND LINS, R. D. Cyclic reference counting with local mark-scan. *Inf. Process. Lett. 34*, 1 (1990), 31–35.

[33] MCCARTHY, J. Recursive functions of symbolic expressions and their computation by machine. *Commun. ACM 3*, 4 (1960), 184–195.

[34] NETTLES, S., AND O'TOOLE, J. Real-time garbage collection. In *Proc. of the SIGPLAN Conference on Programming Language Design and Implementation* (June 1993). *SIGPLAN Notices, 28*, 6, 217–226.

[35] ROSENBLUM, M., AND OUSTERHOUT, J. K. The design and implementation of a log-structured file system. In *Proc. of the Thirteenth*

ACM symposium on Operating Systems Principles (Pacific Grove, California, Oct. 1991). *SIGOPS Operating Systems Review, 25*, 5, 1–15.

[36] SCHORR, H., AND WAITE, W. M. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM 10*, 8 (1967), 501–506.

[37] SELIGMANN, J., AND GRARUP, S. Incremental mature garbage collection using the Train algorithm. In *Ninth European Conference on Object-Oriented Programming* (Åarhus, Denmark, 1995), W. G. Olthoff, Ed., vol. 952 of *Lecture Notes in Computer Science*, pp. 235–252.

[38] SHUF, Y., GUPTA, M., BORDAWEKAR, R., AND SINGH, J. P. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, Jan. 2002). *SIGPLAN Notices, 37*, 1, 295–306.

[39] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM 18*, 9 (Sept. 1975), 495–508.

[40] STEFANOVIČ, D., MCKINLEY, K. S., AND MOSS, J. E. B. Age-based garbage collection. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Denver, Colorado, Oct. 1999). *SIGPLAN Notices, 34*, 10, 370–381.

[41] UNGAR, D. M. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, Pennsylvania, 1984), P. Henderson, Ed. *SIGPLAN Notices, 19*, 5, 157–167.

[42] WEIZENBAUM, J. Symmetric list processor. *Commun. ACM 6*, 9 (Sept. 1963), 524–536.

[43] WEIZENBAUM, J. Recovery of reentrant list structures in SLIP. *Commun. ACM 12*, 7 (July 1969), 370–372.

[44] YUASA, T. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software 11*, 3 (Mar. 1990), 181–198.

[45] ZEE, K., AND RINARD, M. Write barrier removal by static analysis. In *Proc. of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Seattle, Washington, Oct. 2002), ACM Press. *SIGPLAN Notices, 37*, 11 (Nov.), 191–210.

[46] ZORN, B. Barrier methods for garbage collection. Tech. Rep. CU-CS-494-90, University of Colorado at Boulder, 1990.