The Space Cost of Lazy Reference Counting

Hans-J. Boehm
HP Laboratories
1501 Page Mill Rd.
Palo Alto, CA 94304
Hans.Boehm@hp.com

Abstract

Reference counting memory management is often advocated as a technique for reducing or avoiding the pauses associated with tracing garbage collection. We present some measurements to remind the reader that classic reference count implementations may in fact exhibit longer pauses than tracing collectors.

We then analyze reference counting with lazy deletion, the standard technique for avoiding long pauses by deferring deletions and associated reference count decrements, usually to allocation time. Our principal result is that if each reference count operation is constrained to take constant time, then the overall space requirements can be increased by a factor of $\Omega(R)$ in the worst case, where R is the ratio between the size of the largest and smallest allocated object. This bound is achievable, but probably large enough to render this design point useless for most real-time applications.

We show that this space cost can largely be avoided if allocating an n byte object is allowed to additionally perform $O(n)$ reference counting work.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features — *Dynamic storage management*; D.4.2 [**OperatingSystems**]: Storage Management — *Garbage collection*

General Terms

Languages, Performance.

Keywords

Garbage collection, reference counting, memory allocation, space complexity.

1 Introduction

Reference counting[8, 14] is a commonly used technique for automatically, or semi-automatically deallocating unreferenced memory and other resources. It has been used with great success in some applications, *e.g.* in Unix/Linux file systems. It has also often been suggested and used as a general purpose memory management discipline for programming language objects (cf. [11, 10, 1, 9]).

The fundamental idea behind reference counting memory management is to associate a count of incoming pointers $count(p)$ with each object p in the heap. We maintain the invariant that $count(p)$ is the number of variables or other heap locations that point to p. When $count(p)$ becomes zero, we know that p can no longer be accessed, and hence it is safe to deallocate it.

This approach is completely different from *tracing* garbage collectors, which periodically traverse the entire heap to identify all reachable memory, and then reclaim the rest.[14]

1.1 Some pros and cons of reference counting

Reference counting claims a number of well-known advantages over other automatic garbage collection techniques, including[14]:

- It allows memory to be promptly reused. Hence newly allocated objects may still be resident in the processor cache from the last use of that memory, potentially giving a significant performance benefit.
- The technique works well with a nearly full heap. Hence, if the overhead introduced by the reference count is small, *e.g.* because the average object size is large, it can be a very space-efficient technique.
- It may allow the client code to use destructive updates instead of copies, since it can determine whether an object is uniquely referenced.
- Basic implementations tend to be simpler than other techniques.
- It is often claimed to distribute storage management overhead more evenly throughout the application.

We will concentrate on the last point, both since it is often considered decisive, and since, as we will see, the above statement is a gross oversimplification of the truth.

On the negative side, reference counting potentially makes pointer updates much more expensive, in some cases prohibitively so.

The best-known deficiency of reference counting is that it fails to reclaim "cyclic garbage": If object p points to q and q points to p, with no other pointers to p and q, clearly the program can no longer refer to either object. But $count(p) = count(q) = 1$, and neither will be reclaimed. There are several ways to address this:

- Run a tracing collector occasionally as a backup.
- Require the programmer to avoid cycles. This becomes more feasible in a general purpose context if we introduce "weak" pointers[9, 24] that are not included in reference counts, but are invalidated when an object is deallocated.
- Add a facility to the reference count implementation for reclaiming cycles[3].

Here we concern ourselves with the fundamental performance characteristics of basic reference counting algorithms. Hence we consider only acyclic data structures. Our conclusions still apply for the core algorithm in the presence of any of the extensions to accommodate or prevent cycles.

1.2 Some Terminology

For the purpose of this paper, we will assume we are provided a set of underlying memory allocation and deallocation routines, which we will refer to as *malloc* and *free*.

We will use C programming language syntax for all program fragments.

A reference count implementation can then be viewed as providing implementations for the following operations:

alloc The operation $alloc(size)$ allocates an object of *size* bytes by calling $malloc(size)$. We distinguish it from the underlying *malloc* operation primarily because a reference count implementation may wish to do other work at allocation time. The newly allocated object has an associated reference count of 1.[1]

incr The operation $incr(p)$ increments the reference count associated with p. This is called whenever an additional pointer to p is created. (Since the initial reference count is one, in our formulation, this should not be done the first time a reference to a newly allocated object is saved.)

decr The operation $decr(p)$ decrements the reference count associated with p and arranges for p to be reclaimed (with *free*) if it has no further references. If p is reclaimed, we must further ensure that counts associated with objects referenced by p are suitably adjusted. This is invoked whenever an existing pointer goes out of scope, or is overwritten as the result of a pointer assignment.

These may require adapting the representation of pointers (*e.g.* if a second object is used to store the reference count, as for Boost `shared_ptr`[9]) or the representation of heap objects (*e.g.* if the reference count is explicitly stored in each object).

A reference count implementation is used by a reference-counted program. Such a program allocates heap objects with *alloc*, and issues *incr* and *decr* operations as pointers to heap objects are created and destroyed. A pointer assignment typically requires an *incr* operation on the new value of the pointer, followed by a *decr* operation on the old one.[2]

Typically the *incr* and *decr* calls will not be explicitly present in the source. They will either be added by the compiler (as in [11]), or they may be provided by a reimplementation of the assignment operator etc. in the language itself (as with C++ "smart pointer" implementations such as [9]).

1.3 Towards Constant Time

Traditionally one claimed advantage for reference counting has been that it more evenly distributes storage management overhead. In this paper we explore the consequences of limiting *alloc*, *incr*, and *decr* to (near) constant time, at least in the sense that each will execute at most a (near) constant number of *free* operations.

This is closely related to making a reference-count implementation usable in real-time applications, though there seems to be some disagreement on the precise requirements for real-time memory management (cf. [2, 17]). Certainly a reference count implementation cannot guarantee a minimum processor utilization for a given section of client code unless it is possible to bound the number of *free* calls made on its behalf from the reference count implementation during that section of code.

The principal result presented in this paper is that although constant-time reference-count operations are possible in a bounded amount of heap space, in the presence of variable object sizes the worst-case space bound is impractically large. Thus reference counting with variable-sized objects should generally not be viewed as a real-time technique, unless we are willing to accept more relaxed time bounds on the *alloc* operation.

2 Classic reference counting

The most straightforward and traditional implementations of reference counting use the following approach:

- The *alloc* operation simply calls *malloc* and sets the associated count to one.
- The $incr(p)$ operation is implemented as

  ```
  void incr(counted_obj *p)
  {
    if (p != null)
      count(p)++;
  }
  ```

  In the multi-threaded case the increment operation must be atomic. This is normally accomplished with either an atomic hardware operation or by acquiring a lock.

[1]It is perhaps more natural to allocate objects with a reference count of zero. This formulation is slightly more convenient for us and typically more efficient. If the reference returned by *alloc* is not preserved, an explicit *decr* call is needed.

[2]This order ensures that an assignment such as p = p does not result in a premature deallocation.

- The $decr(p)$ operation is implemented as[3]

```
void decr(counted_obj *p)
{
  if (p != null && --count(p) == 0) {
    for each pointer field f of *p {
      decr(p->f);
    }
    free(p);
  }
}
```

  In the multi-threaded case the count decrement and test must be performed as a single atomic operation.[4]

Implementations along these lines are quite common. For example [9] uses such an implementation.

One disadvantage of this straightforward approach is that it is often expensive to implement in the presence of threads. At a minimum, each *decr* operation requires an atomic fetch-and-add instruction, and *incr* requires an atomic add in order to protect against concurrent updates of the same count field. On modern architectures, these atomic operations usually require a few dozen, or even a few hundred, processor cycles even in the absence of cache misses.[5] In addition, we've added memory references which will require space in the processor cache.[6]

2.1 Pause times for classic reference counting

Here we will concentrate on a second issue: The *decr* operation recursively traverses as much of the data structure as has become unreferenced as a result of the assignment. Hence it, and thus pointer assignments, may take an unbounded amount of time. For example, assume our program consists of the following:

```
q = make_huge_linked_tree();
p = make_little_tree();
q = p;
```

The last assignment will be expanded to something like:

```
incr(p);
decr(q);
q = p;
```

Assuming `make_huge_linked_tree()` builds a large linked acyclic data structure, and returns the only reference to it, the *decr* operation, which is implicitly part of the final assignment, will traverse and invoke *free* on every node in the large data structure.

[3]For brevity, we refer to all pointer fields f within p as p `->` f, even though some of these fields may be array elements, or be contained in nested structures.

[4]If pointer assignments as a whole are to be atomic, or if we need to guarantee type-safety in the presence of data races, we need further synchronization in the assignment operation to ensure that we perform the *decr* operation on the same pointer we are replacing.

[5]Atomic updates can be reduced or eliminated through the use of more sophisticated algorithms which store reference count updates in buffers, and process them in a single thread. Cf. [10, 15, 1]. However, the added complexity seems to make such implementations relatively rare.

[6]Other more sophisticated techniques reduce this need by storing a one bit reference count in the pointer itself[22, 14, 20].

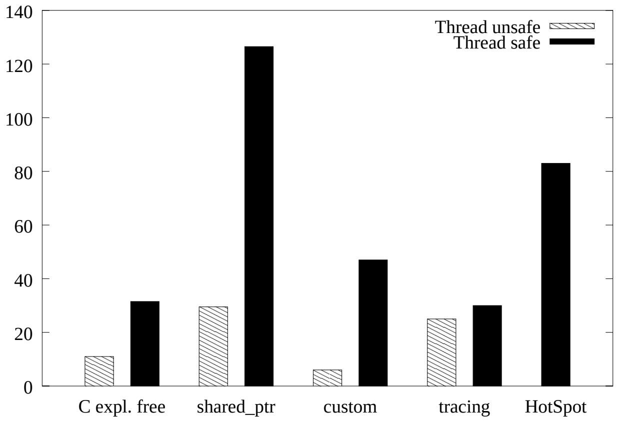

Figure 1. GCBench max "pause" times (msecs).

If we contrast this with the often-maligned pauses of non-incremental tracing garbage collectors[14], the "pauses" introduced by hidden *decr* operations directly affect only a single thread[7] but as we will see, they may certainly be at least comparable in length to a full tracing garbage collection.

A simple tracing collector must examine at least all pointer-containing objects. But a *decr* operation may also *free* nearly all allocated memory. Since, unlike most tracing collectors, it deallocates one object at a time and, in our current version, does not defer any work to allocation, the actual time spent on each object may be appreciably greater. This is especially true if locking is required to support threads.

To illustrate this point, we measured the pause times associated with a run of the GCBench[8] small object allocation benchmark with complete manual explicit deallocation ("C expl. free"), two variants of reference counting ("shared_ptr", "custom"), and two variants of tracing garbage collection ("tracing", "HotSpot").

Here we define the "pause time" associated with explicit deallocation to be the time taken to explicitly walk and deallocate the data structures used in the benchmark.

The first reference-counting variant ("shared_ptr") represents a straightforward application of the `shared_ptr` class in the Boost C++ library[9]. This facility makes it relatively convenient to use reference counting without compiler support. The second variant ("custom") represents a much more aggressively tuned use of other Boost library facilities for reference counting, and it incorporates a custom memory allocator. Both are implementations of "classic" reference counting, in that they do not use the technique we review in the next section.

We included two very different tracing collectors: Our stop-the-world non-moving collector[5] ("tracing"), and Sun's HotSpot Client Java Virtual machine copying generational collector ("HotSpot"). Both collectors are run in their default mode. They could be, but were not, adjusted for significantly reduced pause times on this benchmark.

[7]Of course that thread may happen to hold a crucial lock, and hence block others.

[8]The C, C++ and Java variants we used for this test are available at //www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench/refcnt_tests.

The resulting maximum "pause" times, measured in milliseconds, are given in figure 1. In most cases, we measured both thread-unsafe and thread-safe variants. The former perform enough locking to ensure that multiple copies of the benchmark could correctly have been run concurrently in the same address space. For the thread-unsafe case, this is not true, and no locking or other synchronization is performed.

Substantial additional measurement details are given in the appendix.

It is clear from the measurements that reference counting "pause times" can easily be on the same order as pause times for tracing collectors. In the thread-safe case, they may well be larger.

The reference count "pause times" are spent deallocating objects and recursively decrementing reference counts, and hence are independent of many optimization issues. Although C++ "smart-pointer"-based implementations do not aggressively try to reduce reference count updates, this has no affect on the work performed during the "pause", and hence these results should apply equally to other languages and implementation styles. They will of course vary much more significantly with the allocation and object retention characteristics of the benchmark.

3 Lazy deletion

The standard method for avoiding the long delays introduced by cascading *decr* operations is to defer the recursive invocations[23, 14, 24, 16] until a more opportune time. When an object's reference count decreases to zero, just the top level object is added to a *to-be-freed* set. The *to-be-freed* objects are then processed later incrementally, *e.g.* during later allocations or possibly in a separate thread.[9]

If we process one element of the *to-be-freed* set during each allocation, the *alloc* and *decr* implementations become:

```
void * alloc(size_t size)
{
  void *result;
  if (<to-be-freed is not empty>) {
    p = <get and remove element of to-be-freed>;
    for each pointer field f of *p {
      decr(p->f);
    }
    free(p);
  }
  result = malloc(size);
  count(result) = 1;
  return result;
}
```

[9]Note that reference counting allows two somewhat distinct kinds of "laziness". A number of papers starting with [11] have advocated deferring *incr* and *decr* operations required directly by the client, *e.g.* to greatly reduce reference counting costs for stack variables, or, in later papers, to reduce synchronization cost. This is different from our kind of laziness, which defers only deletion and otherwise recursive *decr* operations. Both techniques may of course be combined, as is mentioned in [11]. In this paper, we use "lazy reference counting" to mean "reference counting with lazy *free* and recursive *decr* invocations", which is termed "lazy deletion" in [14].

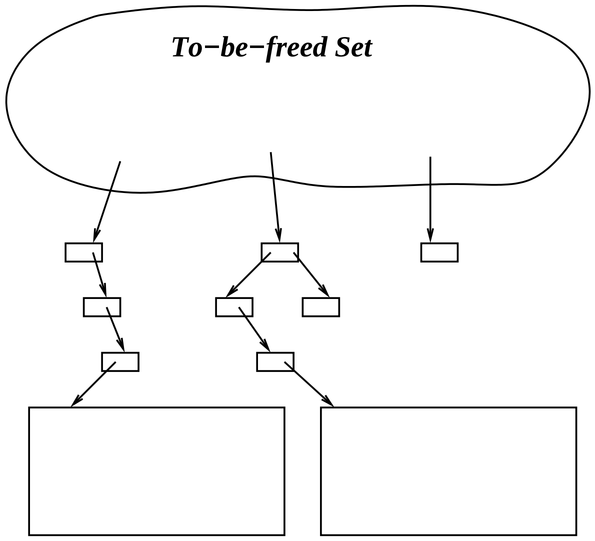

Figure 2. Large hidden to-be-freed objects.

```
void decr(counted_obj *p)
{
  if (p != null && --count(p) == 0)
    <add p to to-be-freed>
}
```

If we assume that the number of pointers in each object is bounded by a constant (i.e. avoid pointer arrays) this approach will ensure that *incr* and *decr* run in constant time, and that *alloc*'s running time is a constant times that of *malloc* and *free*. Techniques for processing pointer arrays incrementally to deal with varying numbers of pointers are discussed briefly below.

If all objects are the same size, as in [23] or the reference counting scheme described in [4], the above can be simplified by letting *alloc* simply return the block *p* which it removed from the *to-be-freed* set without actually deallocating and reallocating it. In this case, we are guaranteed that an allocation request can be satisfied in this way whenever an unreferenced object is available.

Unfortunately, this last property does not hold in the general case. In general, we may try to allocate a large object when there are many suitably-sized objects on the *to-be-freed* set, but none of them have yet been *free*d, *e.g.* if the *to-be-freed* set appears as in figure 2. Thus the heap may need to be grown, or we may need to reintroduce pauses for recursive *decr* calls, even when sufficient objects are unreferenced.

The rest of this paper quantifies the amount of space that may be lost to this effect.

4 Fragmentation

By separating out the underlying memory allocation algorithm from reference counting in our presentation, we have largely dodged the issue of memory fragmentation, i.e. space overhead introduced when enough unallocated space is available in the heap to allocate an object, but that space is not contiguous. This separation is essential in some of the proofs below, but it requires some care in interpreting the results.

We will present results about the amount of *allocated* heap space, which is the amount of memory *malloc*ed, but not *free*d. This usually does not reflect the worst-case space requirements of the application. There may be additional fragmentation overhead.

The reader should recall the following three results about fragmentation overhead:

- Published measurements [13] suggest that typical fragmentation overhead for applications with explicit memory management and well-designed allocators is very low, at least for short-running applications. Here we are primarily concerned with worst-case overhead, which is a different situation.[10]
- Any memory allocator that does not move objects may require heap space a factor of $O(\log(\frac{s_{max}}{s_{min}}))$ larger than allocated space, where s_{min} and s_{max} are the smallest and largest possible object sizes[18, 19].
- It is easy to design an allocator that achieves this bound, to within a constant factor. Consider an allocator that rounds up all requested sizes to the next power of 2, and keeps separate free lists for each object size, never coalescing objects. There are $O(\log(\frac{s_{max}}{s_{min}}))$ size classes, and certainly allocating as much space as the total size of allocated objects to each one is sufficient.

5 A Lower Bound on Space Usage

We consider an object p to be *unreferenced* if one plus the number of *incr* operations performed by the client on the object is equal to the sum of

- The number of *decr* operations performed directly on the object, and
- The number of pointers to this object from other objects that are unreferenced by this definition.

(The latter result in recursive calls to $decr(p)$ by the classic reference count implementation. Thus the sum is just the total number of $decr(p)$ operations in that case.)

We consider an object to be *allocated* if it has been allocated with *malloc*, but not yet deallocated with *free*.

We assume that all allocation requests are for object sizes between s_{min} and s_{max} words.

For simplicity, we make the following *convenience assumptions*:

- s_{min} divides s_{max}.
- s_{min} is large enough for every object to hold two pointers.
- s_{max} is no more than half the maximum referenced memory.

(Any of these assumptions can clearly be dropped at a cost of decreasing the lower bound by a small constant.)

We emphasize that we are uniformly concerned with worst-case overhead here. We suspect that the worst-case lazy reference-counting space overhead is highly atypical in practice, as is the case for fragmentation overhead. But worst-case space behavior must normally be considered for real-time embedded applications, which seems to be an important potential application for lazy reference counting.

We define an object to be *immediately reclaimable* if it is unreferenced, and any other (unreferenced) objects pointing to it have been *free*d.

Standard reference count implementations base deallocation decisions on directly executed reference count operations and on examination of previously *free*d objects. The following definitions capture this notion.

We say that two programs running with the same reference count implementation generate *isomorphic free graphs* at particular points in time if we can establish 1-to-1 correspondences between the free objects[11] and between the immediately reclaimable objects, such that[12]

- The nth objects to become immediately reclaimable correspond, i.e. corresponding objects became reclaimable in the same order.
- Corresponding free objects have the same size and contain the same number of pointer fields.
- At the time of deallocation, corresponding pointer fields in free objects that pointed to free or immediately reclaimable objects pointed to corresponding objects.

We define a reference count implementation to be *lookahead-free* if it obeys the following properties:

- It is sequential, i.e. single-threaded.
- An object is *free*d only if it is immediately reclaimable. Thus no object is ever deallocated while it is referenced from an allocated object. (Among other things, this precludes deallocation of cyclic data structures.)
- The decision about which object will be *free*d next depends only on the order in which reference counts were explicitly reduced to zero and on pointers from free objects. More precisely, if two programs produce isomorphic free graphs with the given reference count implementation, then corresponding objects will be *free*d next.
- Deallocations (with *free*) may be performed in response to *alloc*, *incr* and *decr* calls. The number of deallocations performed for each call depends at most on the size argument to *alloc*, and the sizes of other objects previously *free*d during the same call.

All sequential reference count implementations of which we are aware are lookahead-free.

THEOREM 5.1. *Assume a lookahead-free reference count implementation deallocates (with* free*) at most m objects for any sequence containing at most one* alloc, *two* incr, *and two* decr *calls.*

[10]Our own experience with less space-efficient allocators, longer-running programs, and deallocation delayed by a non-moving tracing collector, suggests that typical fragmentation overhead remains well below a factor of two. This is far better than the theoretical worst case, but significantly worse than [13]. We conjecture that delaying object reclamation is a significant factor in the increase, although it does not affect the worst case.

[11]If the memory for free objects s reused, the old and new objects are treated as distinct objects. Equivalently, we identify objections by their position in the sequence of allocations and not object addresses.

[12]This definition is a bit arbitrary, particularly in that we do not insist that corresponding immediately reclaimable objects have the same size. This again affects only the constants in our result.

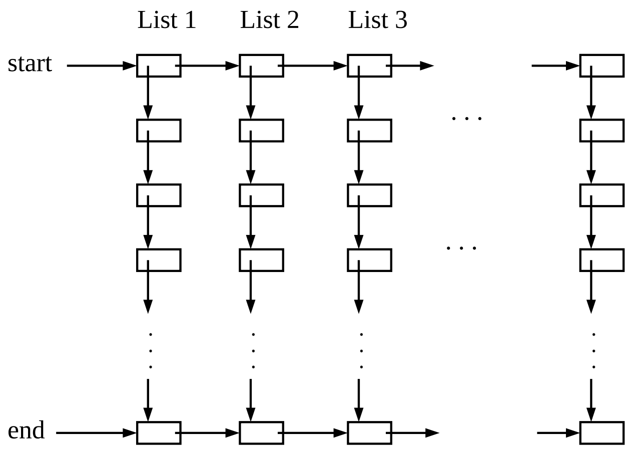

Figure 3. Initial $P_{experiment}$ data structure.

Then for every N there exists a reference-counted program such that

- *No cyclic data structures are ever constructed.*
- *No more than N bytes are referenced at any point.*
- *At some point at least $\frac{Ns_{max}}{2ms_{min}}$ bytes will be allocated.*

Proof First consider a program $P_{experiment}$ which we use only to learn about the order of deallocations performed by the reference count implementation. $P_{experiment}$ allocates $\frac{N}{2ms_{min}}$ singly linked lists each consisting of N objects[13] of size s_{min}. We use the second word in the first and last nodes of each list to link together the first nodes of all lists, and similarly for the last node. This gets us the data structure in figure 3. Note that the only horizontal links are at the top and bottom of the figure.

$P_{experiment}$ then continues by repeatedly performing the following operations for $\frac{N}{2ms_{min}}$ iterations, i.e. once for each list:

1. Allocate a large object of size s_{max} bytes.
2. Replace the *null* pointer at the end of the first remaining list with a pointer to the newly allocated object. This requires no *incr* or *decr* operations, since by convention *alloc* returns an object with a reference count of one.
3. Replace the two references to the start and end of the first remaining list by references to the next list, using two *incr* and two *decr* operations.[14]

After the first iteration of this loop, we get the data structure in figure 4.

For each of the $\frac{N}{2ms_{min}}$ iterations of the loop one *alloc*, two *incr*, and two *decr* operations are performed. Thus at most $\frac{N}{2s_{min}}$ *free* calls may be made during the loop. (None can be *free*d earlier, since all objects are referenced before this point.) Since objects may not be *free*d before the objects referring to them, none of the large objects

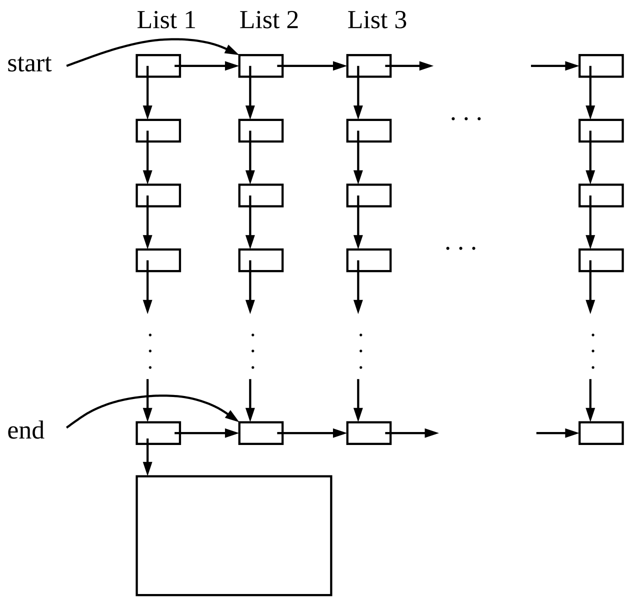

Figure 4. $P_{experiment}$ data structure after iteration 1.

can be deallocated. Let k_i be the number of small objects *free*d from the i^{th} list. We have $\sum_i k_i \leq \frac{N}{2s_{min}}$.

We then construct P_{proof} to be identical to $P_{experiment}$, with one important difference: We arrange that the i^{th} linked list has length exactly k_i.[15] If any of the k_i are less than 2, we also need to adjust P_{proof} so that we still maintain the horizontally linked *start* and *end* lists correctly.[16]

Since none of the large objects $P_{experiment}$ are ever *free*d, inductively $P_{experiment}$ and P_{proof} generate isomorphic free graphs after each main loop iteration.[17] Specifically, P_{proof} must still *free* exactly k_i elements from the i^{th} list, and P_{proof} still fails to free any of the large objects.

Since at any given point, at most one of the large objects is referenced, the total amount of referenced memory never exceeds that in all of the lists allocated at the beginning, plus the size of one large object. for P_{proof}, the former includes at most $\frac{N}{2s_{min}}s_{min}$ or $\frac{N}{2}$ bytes. The latter consists of s_{max} bytes, which we assumed to be at most $\frac{N}{2}$. Thus at most N bytes are referenced at any point.

[13]The length N here is clearly sufficient but largely arbitrary.

[14]We can strengthen the result slightly by observing that the reference count updates for the pointers to the end of the lists are not strictly necessary, and could be omitted. Hence we really only need one *incr* and one *decr* operation per iteration, and we could strengthen the definition of m correspondingly.

[15]For all realistic reference counting implementations, it will be easy to compute k_i without the use of additional heap space and P_{proof} can be constructed out of loops, as we implied for $P_{experiment}$. If k_i is not easily computable, it suffices for our stated theorem to fully unroll the outer loop for constructing the lists, and include the k_i as constants in a program constructed for a specific N. Or we could possibly arrange to read the k_i from a file.

[16]We arrange for the *start* list to contain the first nodes of all nonempty lists, and for the *end* list to contains the last nodes of all lists of length at least two. Thus for small k_i only the start list or neither list may need to be adjusted in the i^{th} iteration. We also omit appending the newly allocated large block to empty lists. The large blocks corresponding to an empty list are immediately dropped.

[17]Note that an immediately reclaimable small object in $P_{programmable}$ may correspond to an immediately reclaimable large object in P_{proof}, but neither will be *free*d and hence, by our assumptions, the difference will not affect the deallocation sequence.

Since none of the large objects are *free*d, at the end we have one large allocated block for for each of the $\frac{N}{2ms_{min}}$ loop iterations, for a total of $\frac{Ns_{max}}{2ms_{min}}$ bytes. •

OBSERVATION 5.1. *Theorem 5.1 implies that if* alloc*,* incr *and* decr *each perform a constant number of* frees*, and with no constraints on the size of allocated objects, the amount of allocated memory can be quadratic in the amount of referenced memory. This is significantly worse than the logarithmic worst-case overhead of fragmentation with a good memory allocator.*

Conversely, the above theorem also tells us that in order to reduce the space overhead to a constant factor, we need $m = \Omega(\frac{s_{max}}{s_{min}})$. Up to this value we have a real tradeoff between latency and space; beyond this value the result becomes vacuous.

In particular, for the fixed size, "real-time" case, i.e. with $s_{max} = s_{min}$ and $m = 1$, this is a vacuous result. (The factor of 2 imprecision comes from the fact that we reserved $N/2$ bytes in our referenced memory quota for the single "large" object.) This is of course expected, since we know that lazy reference counting works well for fixed object size.

Perhaps more surprisingly, the theorem suggests that there is nothing special about the fixed size case; allowing a small amount of variation in allocation sizes, results in a (relatively) small amount of space overhead. The next section confirms this.

6 An Upper Bound on Space Usage

LEMMA 6.1. *Lazy reference counting preserves the property that all allocated unreferenced objects are in the to-be-freed set or reachable from a to-be-freed object via unreferenced objects.*

Proof Whenever an object's reference count reaches zero, we put it in the *to-be-freed* list. Hence any unreferenced object not in the set must be pointed to by an unreferenced object, which must either itself be in the *to-be-freed* set, or must be reachable from another object in the set. •

THEOREM 6.2. *Consider a reference count implementation in which each* alloc *call calls* free *on exactly one unreferenced object whenever there are unreferenced objects, as in the lazy reference count implementation we presented above. Let s_{min} and s_{max} be the smallest and largest requested object size as before. Assume that at most N bytes in the heap are referenced at any one time. Then at most $\frac{s_{max}}{s_{min}}N$ bytes will be allocated at any point.*

Proof The number of allocated (*malloc*ed but not *free*d) objects is equal to the largest number of objects referenced at a previous point in the computation. This is easily shown by induction:

Initially it is true since no objects are referenced, *malloc*ed, or *free*d.

Neither *incr* nor *decr* affects either the number of allocated objects, or the largest number ever referenced. Hence it suffices to show that each *alloc* operation preserves the property.

When a new object is *alloc*ed we consider two cases:

- No allocated objects are unreferenced. Hence the *to-be-freed* list is empty. By induction hypothesis, the number of allocated objects is equal to the maximum number ever referenced. The *alloc* call increases both quantities by one.
- There are unreferenced allocated objects. By induction hypothesis more objects were referenced at some point in the past. Thus the maximum number of referenced objects is not affected by the *alloc* invocation.

  From the preceding lemma, the *to-be-freed* list is nonempty. Hence *alloc* both *free*s and *malloc*s exactly one object. Hence the number of allocated objects is also unaffected.

Let K be the maximum number of objects ever referenced. The size of any K objects is at least Ks_{min}. Thus the total size of referenced objects N must have been at least Ks_{min} at some point, and thus $K \leq \frac{N}{s_{min}}$.

The number of allocated objects never decreases with this particular algorithm. The final (and largest) number of allocated objects is also K. The total size of those K objects is at most $Ks_{max} \leq \frac{s_{max}}{s_{min}}N$. •

Note that the upper and lower bounds differ only by a factor of 2 for the standard lazy reference count algorithm, at least with our convenience assumptions. (The reason for the factor of 2 is described at the end of section 5.) In that case the m in the lower bound result is 1, since only *alloc* calls *free*, and it calls it at most once.

In spite of the unpleasant space bound, the above algorithm doesn't quite give us constant time reference counting operations, since *alloc* may have to traverse an unbounded number of pointers in a large object, though it only invokes *free* once. This can be reduced to constant time plus a constant number of *free* operations per reference count operation with a more complex algorithm.

There appear to be at least two ways to accomplish this:

- We arrange to be able to find all n non-*null* pointers inside an object in $O(n)$ time, *e.g.* by building a doubly-linked list inside the object when necessary. We then scan large pointer arrays incrementally. To compensate for the fact that the average *alloc* will thus do less than one *free*, we arrange for *incr* to also process deferred decrements. Since allocation and initialization of objects with many pointers require many *incr* calls, this ensures that *free* calls will keep pace with allocations, although the bound on the number of allocated objects will be higher.
- David Wise[18] has pointed out that reference count decrements during a deferred deallocation can be deferred even further, until the pointers are overwritten in the newly allocated object. This also avoids the flurry of decrement operations during allocation.

It is possible to obtain much more reasonable space bounds if we relax the assumption on the number of *free*s performed by each *alloc* call:

THEOREM 6.3. *Consider a reference count implementation in which each $alloc(n)$ call calls* free *on unreferenced objects of total size at least n, whenever there are sufficient allocated but unreferenced objects available. Assume that at most N bytes in the heap are referenced at any one time. Then at most N bytes will be allocated at any point.*

[18]See http://lists.tunes.org/archives/gclist/2000-September/001835.html. They previously built a hardware implementation of this approach for fixed size heap cells.[24]

Proof We prove the hypothesis directly by induction. Clearly it is true at program start with no allocated objects.

When a new object is allocated with $alloc(n)$ we again consider two cases:

- Fewer than *n* bytes are allocated but not referenced. Immediately after the *alloc* call all allocated unreferenced objects will have been *free*d, and all allocated bytes will be referenced. Thus clearly the number of allocated bytes will be less than N.
- At least *n* bytes are allocated but not referenced. In that case, the *alloc* operation will free at least as many bytes as are newly allocated, and the number of allocated bytes cannot increase.

Again neither *incr* nor *decr* operations affect the validity of the induction hypothesis. •

Again, this nicely matches the lower bound result; to be able to *free n* bytes when allocating *n* bytes, in the worst case we will need to make $\frac{s_{max}}{s_{min}}$ *free* calls in response to a single *alloc* call. Thus we are exactly in the $m = \frac{s_{max}}{s_{min}}$ case, which we know is within a constant factor of optimal for constant space overhead.

This does not imply that lazy reference counting costs no space at all in this case; when we try to allocate an object with fewer than *N* referenced bytes, the *to-be-freed* list may be nonempty. These allocated but not referenced objects may cause or compound fragmentation in the underlying allocator. But this is already accounted for by the potential $O(\frac{s_{max}}{s_{min}})$ fragmentation space overhead for the underlying allocator.

Also note that in spite of the preceding result, if the program has at most *n* objects of a given size *s* referenced at one point, it may still be the case that more than *n* objects of size *s* will need to be allocated; the theorem is purely a statement about the total size of all allocated objects. Thus the objects on the *to-be-freed* list may still impact the amount of space the underlying allocator needs to reserve for a given object size, assuming it segregates objects by size.

The preceding theorem does tell us that we can get worst-case space bounds for lazy reference counting very similar to those for manual memory management by letting *alloc* take time in proportion to the size of the allocated object. This is basically the same requirement as for real-time tracing collection[4, 2].[19]

7 Implications for Concurrency

In the above, we considered only sequential or single-threaded reference count operations, and we assumed deterministic execution. But especially the lower bound result also has some implications for nondeterministic or concurrent algorithms.

OBSERVATION 7.1. *The lower bound result assumed deterministic execution to compute the k_i values. If we are interested in the worst case among several possible nondeterministic executions, it suffices to choose a possible k_i sequence, and otherwise the same proof applies.*

[19]For time-based collection as recommended in [2], this is implicit in that a bound on the allocation rate is assumed.

Nondeterministic or random behavior can't help the worst case.

OBSERVATION 7.2. *The above still applies to concurrent implementations, such as [11], if the total number of frees performed by any thread during a loop iteration of P_{proof} is bounded by m. With a bounded number of processors, the final loop in P_{proof} either cannot run in near constant time per iteration (even if the large objects are uninitialized), or we again risk a large increase in heap size.*

8 Related work and history

There has been much work on tracing collectors that simultaneously satisfy space and pause-time constraints (cf. [4, 2]).

The earliest work on lazy reference counting was by Weizenbaum[23]. It was limited to fixed size objects, but that restriction has usually received minimal attention.

The fact that large objects may be hidden from reallocation by smaller ones is mentioned briefly by [14], but no attempt is made to quantify the cost.

We are not aware of much discussion of the space cost of lazy reference counting with variable-sized objects, in spite of the fact that the technique was frequently suggested. The issue came up in discussion on the *gclist* mailing list[20], where we reminded readers of the issue, and asked for a bound on the space cost. No answer was posted.

Ritzau's later thesis[16] dismisses lazy reference counting for real-time applications, insisting instead on a fixed object size as in [21]. He states, referring to [23]: "However this technique can not be used directly in systems where objects have varying sizes. If differently size objects are used, the worst case of allocation becomes to free all objects on the heap (just to find an object of the right size)."

We have effectively refined this claim in two ways:

- We consider the possibility of simply using more space instead of "freeing all objects on the heap". We quantify the cost of this approach. Since *malloc*/*free* users routinely sacrifice a worst-case factor on the order of $\log(\frac{s_{max}}{s_{min}})$ in space usage from fragmentation overhead, we believe that something like our lower bound result is needed to support his argument.
- Theorem 6.3 points out an alternative design point: Rather than breaking up an $alloc(n)$ call into $O(n)$ small allocations, we could let it make $O(n)$ free calls, and use an allocator that guarantees bounded fragmentation.[21] This would probably require more space but improve performance of data structure accesses.

9 Acknowledgements

The tuned Boost-reference-counting version of GCBench used for the measurements was contributed by Peter Dimov. He also pointed out some problems (since repaired) in the untuned C++ benchmark.

[20]See http://lists.tunes.org/archives/gclist/2000-September/001836.html

[21]Given a hard a priori bound of *N* referenced bytes, it would suffice to only allow allocation of power-of-two object sizes, and allocate separate heap regions of size *N* for each of the $\log(\frac{s_{max}}{s_{min}})+1$ possible size classes, as we suggested earlier. Note that if the program is known to have only *n* referenced objects of size *s*, it is not necessarily safe to reduce the heap size associated with *s* to *sn*.

David Wise enlightened me about some of the relevant history. Both he and the anonymous reviewers suggested many improvements to the paper.

10 References

[1] D. F. Bacon, C. R. Attanasio, H. B. Lee, V. T. Rajan, and S. Smith. Java without the coffee breaks: A nonintrusive multiprocessor garbage collector. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation*, pages 92–103. ACM, 2001.

[2] D. F. Bacon, P. Cheng, and V. T. Rajan. A realtime garbage collector with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, pages 285–298, January 2003.

[3] D. F. Bacon and V. T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the Fifteenth European Conference on Object-Oriented Programming, LNCS 2072*, pages 207–235. Springer, 2001.

[4] H. Baker. List processing in real time on a serial computer. *Communications of the ACM*, pages 280–294, April 1978.

[5] H.-J. Boehm. A garbage collector for C and C++. http://www.hpl.hp.com/personal/Hans_Boehm/gc/.

[6] H.-J. Boehm. Fast multiprocessor memory allocation and garbage collection. Technical Report HPL-2000-165, HP Laboratories, December 2000.

[7] H.-J. Boehm. Reducing garbage collector cache misses. In *Proceedings of the 2000 International Symposium on Memory Management*, pages 59–64, 2000.

[8] G. E. Collins. A method for overlapping and erasure of lists. *CACM*, 13(12):655–657, December 1960.

[9] G. Colvin, B. Dawes, P. Dimov, and D. Adler. Boost smart pointer library. http://www.boost.org/libs/smart_ptr/.

[10] J. DeTreville. Experience with concurrent garbage collectors for modula-2+. Technical Report 64, Digital Systems Research Center, August 1990.

[11] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[12] S. Dieckman and U. Hölzle. A study of the allocation behavior of the SPECjvm98 Java benchmarks. Technical Report TRCS98-33, University of California at Santa Barbara, December 1998.

[13] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: solved? In *Proceedings of the International Symposium on Memory Management 1998*, pages 26–36, October 1998.

[14] R. Jones and R. Lins. *Garbage Collection*. John Wiley and Sons, 1996.

[15] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for Java. In *Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 367–380, 2001.

[16] T. Ritzau. *Memory Efficient Hard Real-Time Garbage Collection*. PhD thesis, Department of Computer and Information-Science, Linköping University, April 2003.

[17] S. G. Robertz and R. Henriksson. Time triggered garbage collection. In *LCTES '03*, pages 93–102. ACM, 2003.

[18] J. M. Robson. An estimate of the store size necessary for dynamic storage allocation. *Journal of the ACM*, 18(3):416–423, 1971.

[19] J. M. Robson. Bounds for some functions concerning dynamic storage allocation. *Journal of the ACM*, 21(3):491–499, 1974.

[20] D. J. Roth and D. S. Wise. One bit counts between unique and sticky. In *Proceedings of the International Symposium on Memory Management 1998*, pages 49–56, October 1998.

[21] F. Siebert and A. Walter. Deterministic execution of Java's primitive bytecode operations. In *Java Virtual Machine Research and Technology Symposium*, April 2001.

[22] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Conference on Lisp and Functional Programming*, pages 159–166, 1984.

[23] J. Weizenbaum. Symmetric list processor. *Communications of the ACM*, 6(9):524–544, 1963.

[24] D. S. Wise, B. Heck, C. Hess, W. Hunt, and E. Ost. Research demonstration of a hardware reference-counting heap. *LISP Symb. Computat.*, 10(2):159–181, July 1997.

A Appendix: Measurement details

All measurements were obtained on a machine with two 2 GHz Pentium 4 Xeon processors running RedHat 7.2. The second processor was basically unused for any of the experiments, since even the thread-safe code was single-threaded for nearly the complete program execution. C and C++ executables were compiled with gcc 3.2 and statically linked.[22]

The GCBench benchmark primarily allocates and drops complete binary trees of various heights up to 16, while keeping some other permanent data structures live during program execution. The maximum pause times were obtained allocating and dropping height 16 trees, i.e. trees containing $2^{17} - 1$ vertices. (It also includes some initialization code that allocates a single tree occupying the entire heap to ensure a reasonable heap size. The deallocation time for that was not considered as a pause for explicit deallocation/reference counting.)

GCBench has often been correctly criticized as being an unrealistic garbage collection benchmark. Among other issues, it is clearly too small and too regular to be fully representative of real programs. But precisely that feature makes it relatively easy to translate between languages, and to use it as a "sanity test" for comparing different garbage collectors and memory management approaches.

In our experience, it is not unrealistic for programs to drop a large fraction of referenced memory with a single pointer assignment, the benchmark characteristic that is most relevant here. This is confirmed by live object measurements such as those in [12] for several of the SPECjvm benchmarks.

Each tree node allocated by GCBench contains two pointers and

[22]Note that atomic memory update instructions, and hence locks, are relatively more expensive on a Pentium 4 than for many other processors. The difference between thread-unsafe and thread-safe variants is somewhat higher than on other architectures, but the analogous graphs for an Itanium machine look qualitatively similar.

two integers, which total 16 bytes on this hardware.[23] This object size is probably more representative of Scheme than C or C++ programs, a property that favors tracing collection. The benchmark does essentially nothing with its data structures other than allocating and deallocating them, which favors reference counting for total execution time, though it has minimal impact on pause time measurements.

The different memory management approaches were measured as follows:

C expl. free The C version of the benchmark, using explicit deallocation with `free`. Deallocation requires an explicit tree traversal. We used the standard RedHat 8.0 `malloc`/`free` implementation. Pause times were measured as the largest amount of time taken to deallocate one of the height 16 trees. This is intended to be representative of standard C coding practice.

In the thread-safe case, a second thread is forked at the beginning of program execution. It exits immediately, but memory allocation primitives continue to acquire and release locks as a result.

shared_ptr Straightforward use of the Boost `shared_ptr` reference-counted "smart pointer" class[9]. The implementation can be used for pointers to arbitrary classes. Hence reference counts have to be allocated in separate objects, which require separate allocation calls. The thread-safe version uses the default pthread locks to protect against concurrent reference-count updates.

custom Much more heavily tuned use of Boost reference counting using Boost `intrusive_ptrs` embedded pointers. Uses a custom spin-lock implementation to protect reference count updates and memory allocation. The lock implementation executes only one atomic instruction per lock/release cycle, and is presumably only slight slower than using a fetch-and-add instruction for reference count updates. Although some effort is invested in reducing the cost of each synchronization operation, no attempt is made to reduce their number, unlike [15] or [1].

Memory is allocated using Boost `quick_allocator`, which keeps separate regions for different sizes and alignment requirements. It never coalesces free objects. This may possibly result in significantly worse space utilization than the standard allocator on real applications, but it works very well for this benchmark, since basically all objects have the same size. The fact that the allocator can be inlined also doesn't hurt.

This implementation is still based on pointer operations redefined by the client code, and does not involve any understanding of reference counting by the compiler. No reference-count-specific compiler optimization is performed, and stacks are not traced as in [11]. We would expect such optimizations to have a noticeable impact on overall execution time, but not "pause" time.

tracing This uses version 6.2 of our tracing garbage collector[5] with the C version of the benchmark, configured without explicit deallocation. The thread-safe version uses thread-local allocation buffers[6], which accounts for the small overhead for thread-support. (These are also used in all common Java implementations.) For this experiment, the collector was not configured for either parallel or incremental collection, either of which would have significantly reduced pause times.[24] The intent was to compare to a straightforward stop-the-world tracing collector.

Pause time measurements were obtained from measurements made by the collector itself. These are probably slightly lower than what is actually observed by the client.

HotSpot We used the HotSpot 1.4.1 client VM without performance tuning arguments. Pause times were obtained from -verbosegc output. It is probably possible to reduce pause times to the minor collection pause time of 17-19 msecs by tuning heap sizes, but including the full collection is probably more representative of typical use.

The total execution time (again in milliseconds) and space (in Megabytes) for each benchmark run are given in figures 5 and 6 respectively.

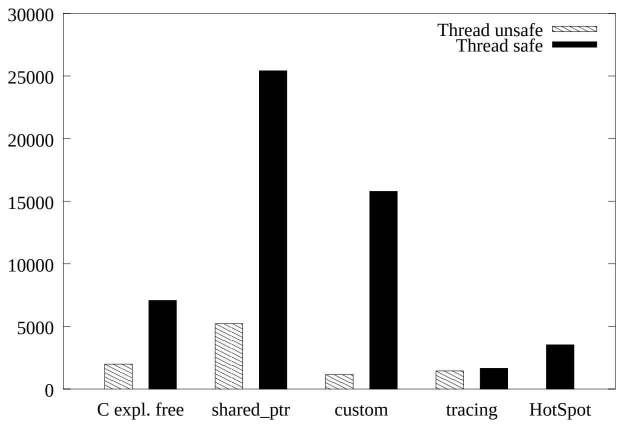

Figure 5. GCBench total execution times.

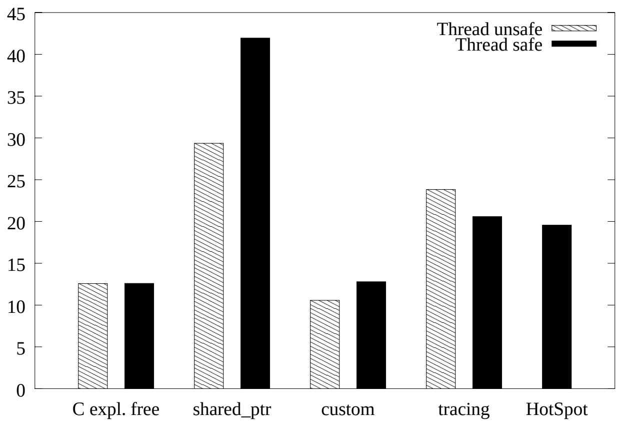

Figure 6. GCBench Heap space use.

[23] Standard allocators will typically allocate 24 bytes, since they need at least a small header and 8 byte alignment. Our tracing collector allocates 24 bytes so that it can correctly handle C pointers just past the end of an object. Java implementations are likely to allocate at least 20 bytes to accommodate a method table pointer. Thus height 16 trees actually contain about 3 MB.

[24] We also did not configure the collector for explicit memory prefetching as in [7], both since this currently results in non-portable X86 code, and because it has minimal impact in this case. We expect the latter is due to the simplicity of the benchmark, coupled with the aggressive hardware prefetch engine in a Pentium 4.