Incremental Collection of Mature Objects*

Richard L. Hudson[1] and J. Eliot B. Moss[2]

[1] University Computing Services
University of Massachusetts
Amherst, MA 01003, USA
hudson@cs.umass.edu
[2] Object Systems Laboratory
Department of Computer Science
University of Massachusetts
Amherst, MA 01003, USA
moss@cs.umass.edu

Abstract. We present a garbage collection algorithm that extends generational scavenging to collect large older generations (*mature objects*) non-disruptively. The algorithm's approach is to process bounded-size pieces of mature object space at each collection; the subtleties lie in guaranteeing that it eventually collects any and all garbage. The algorithm does not assume any special hardware or operating system support, e.g., for forwarding pointers or protection traps. The algorithm copies objects, so it naturally supports compaction and reclustering.
Keywords: clustering, compaction, copying collection, garbage collection, garbage collector toolkits, generation scavenging, incremental collection, mature objects, non-disruptive collection.

1 Introduction

Generational garbage collection is very effective at reducing total garbage collection time. The majority of the collections are also non-disruptive. However, as good as generational collectors are, they can still be disruptive when the larger, older generations need to be collected. To collect these older objects in a non-disruptive manner, we present an algorithm that has the following properties:

Incremental: The maximum number of bytes moved at each incremental collection is small.
Compaction and Clustering: The algorithm supports compaction and reclustering of objects via copying.
Efficient Implementation: The algorithm can be implemented on stock hardware and does not rely on operating system features such as protected pages.

Bishop [Bishop, 1977] discussed how related objects should be placed in the same area and references into these areas should be handled by a level of indirection so that each area could be collected independently of other areas. Our algorithm borrows heavily from his conceptual work describing areas that hold related objects. Our contribution is to show how this can be used to create a collector with the above characteristics.

* This project is supported by National Science Foundation Grant CCR-8658074 and by Digital Equipment Corporation and Apple Computer.

In Section 2 we discuss the problem of collecting older generations holding mature objects and review some recent work in the area. In Section 3 we give an overview of the collector. Section 4 presents our algorithm, while Section 5 shows an example of how the algorithm works. Section 6 describes how to extend the algorithm to handle a potential problem. Finally, Section 7 discusses some future extensions to support additional techniques that mitigate the disruptive behavior of garbage collection.

2 The Problem and Some History

This paper addresses the problem of collecting older objects incrementally, in the context of a copying, scavenging collector. We insist on copying in order to support compaction and clustering (e.g., hierarchical decomposition [Wilson *et al.*, 1991]). Because copying collectors move objects, we assume the safety property that all pointers (and pointer derived quantities) can be found and updated appropriately (see, e.g., [Diwan *et al.*, 1992]).

Pieces of the problem of doing garbage collection non-disruptively have been worked on for years. In this section, we will review some of these attempts and discuss some of their drawbacks.

Baker [Baker, 1978] discussed the problem of constructing a non-disruptive garbage collector. His solution was a modification of a stop-and-copy algorithm first discussed by Fenichel and Yochelson [Fenichel and Yochelson, 1969]. Baker used a read barrier that trapped all reads of old objects and then copied the objects or updated the pointers to moved objects. White [White, 1980] suggested that collecting unreachable objects was not as much of a problem as improving the locality of reference of live objects, and proposed a scheme that improved locality of reference of running programs but that collected unreachable objects off-line. Both Baker and White assumed special pointer forwarding hardware support for their algorithms.

Lieberman and Hewitt [Lieberman and Hewitt, 1983], Moon [Moon, 1984], and Ungar [Ungar, 1984] all presented algorithms that reduced the running time required by most garbage collections by focusing attention on the youngest and most volatile generations of objects. Lieberman and Hewitt relied on special hardware and a Baker-style algorithm to achieve incremental performance. Moon also relied on the Lisp machine hardware to provide a read barrier. Ungar was concerned only with young objects, and collected older objects "off-line". This work made the time to perform most garbage collections reasonable and the majority of collections non-disruptive. The drawback was the large cost and disruption when large old generations needed to be collected.

Appel, Ellis, and Li [Appel *et al.*, 1988] suggested collecting on stock hardware by using read-protected or no-access pages in older generations: when a page is touched, it is scanned and all pointers to moved objects are updated. For efficiency their algorithm depends on two properties. First, the algorithm requires a fast protection fault reflection mechanism. Providing such a fast mechanism may require modifying the operating system. Second, the algorithm requires high locality of reference in the application being run. Without this property, the scanning resulting from touching several pages shortly after a collection would make collection effectively disruptive.

Boehm [Boehm *et al.*, 1991] showed how collectors could be made "mostly parallel" in the trace phase of a collector. His algorithm also relies on using the page trap hardware and operating system support to do bookkeeping during the mark phase. This choice reduces the amount of mutator cooperation needed.

The Lisp Machine [Weinreb and Moon, 1981] demonstrated how linked lists could be compacted using cdr-coding. Wilson [Wilson *et al.*, 1991] showed how hierarchical decomposition could also be used to compress data, in addition to improving locality of reference. Wilson's scheme is similar to Moon's "approximately depth-first" algorithm [Moon, 1984] and demonstrates the gains in locality that can be made by reclustering items based on their reachability path characteristics. Unfortunately, mutation of objects requires periodic reclustering. To allow this reclustering and compaction, the objects need to be moved and pointers to the moved objects updated appropriately.

Lang [Lang and Dupont, 1987] showed how the incremental compaction of a large heap can be done using a hybrid mark/sweep and copying collector. The algorithm copies as much of the heap as there is contiguous free space during each collection thus compacting some portion of the heap. The remaining live objects are not copied. Dead objects in areas where live objects were not evacuated are marked and placed on a free list along with the large evacuated area. This process incrementally continues until the entire heap is compacted. This algorithm requires that all live objects be inspected and possible updated during each pass of the collector. Such romping through memory becomes disruptive as the heap grows large enough to affect the cache and virtual memory mechanisms.

Wilson [Wilson and Moher, 1989] tries to make his collector non-disruptive using temporal opportunism, a technique that tries to hide long garbage collection by piggy-backing onto long computations or onto long interactive pauses. Hayes [Hayes, 1991] suggested key object opportunism, which monitors key objects. When a key object become unreachable, one attempts to collect the objects associated with it. By using the key object as an indicator of when a group of objects become unreachable, the collector focuses its attention on a group of objects that are likely to be unreachable. While temporal opportunism uses hints about *when* to do collections, key object opportunism adds hints about *where* to do collections.

Bishop [Bishop, 1977] presented a garbage collection algorithm that divided the heap into multiple areas. Users specified the area in which each object was allocated. These areas were designed to be garbage collected individually. By collecting the areas independently, the collections would not interfere with processes that did not use the area being collected. In order to allow independent collection, each area kept track of pointers both into the area and out of the area. Referencing an object in another area was accomplished using a level of indirection.

Bishop pointed out that related areas could be collected at the same time. He handled multiple area cycles of garbage either by collecting all areas involved in the cycle at the same time, or by using copying to consolidate the cycle of objects into one area. In his thesis, Bishop presented an inductive proof to show that his technique of moving objects guarantees that all unreachable objects are collected.

Bishop did not bound the size of an area or provide ways to collect individual areas incrementally. In addition, his use of levels of indirection to communicate between areas was a source of inefficiency.

Our mature object space algorithm does not require special hardware or special operating system support, and it is not disruptive. It insures that all reachable objects are collected, that they are moved in a manner consistent with compaction and clustering algorithms, and that they are available immediately after each collection. Our algorithm also limits area size, provides ways to collect individual areas incrementally, and eliminates levels of indirection between areas.

3 Overview of the Garbage Collector

To collect young objects, we designed a garbage collection toolkit that supports generational scavenging techniques.[3] Our algorithm for collecting mature objects is an extension of this toolkit. We now offer an overview of the toolkit as a basis for explaining the extensions.

3.1 The Toolkit Concept

The toolkit divides the responsibility for, and support of, garbage collection into two parts: a language-independent part, supplied by the toolkit, and a language (implementation) specific part, nominally supplied by the language implementor. The language-independent part consists mostly of the data structures and code for managing multiple generations and for allocating heap objects. The language implementor must supply the following capabilities: the ability to locate at scavenge-time all *root pointers* (those pointers outside the scavenged generations that refer to objects in the scavenged generations), and the ability to locate all pointers within a heap object, given a pointer to the object. The toolkit includes a library of routines that an implementor can use to locate inter-generational pointers; it is the implementor's responsibility to locate roots lying in the stack(s), registers, and any other areas outside the heap.

3.2 The Structure of the Heap

The toolkit defines the structure of the heap and supplies the necessary allocation routines. The heap consists of a number of *generations*. Generations are numbered 0, 1, 2, ..., in order of increasing age. In any given collection, a selected generation and all younger generations will be scavenged. The total number of generations may vary over time.

Each generation consists of a number of *steps*. Steps segregate objects by age and/or type within a generation, and during scavenging all surviving (reachable) objects in a given step are copied to some other step. This *promotion step* may belong to the same or a different generation, and by adjusting the promotion steps before scavenging, one can introduce new steps, combine existing steps, etc. The number of steps in a generation may vary over time.

A primary function of steps is to eliminate the need for storing or maintaining any age information in individual objects. This reduces storage and time costs, but also gives the collector age information without imposing any requirements on object formats (which are entirely the responsibility of the language implementor).

While the meaning of steps is somewhat arbitrary, we impose a convention that the lowest numbered step in a generation has the youngest objects in that generation, etc. Further, we number the steps 0, 1, 2, ..., such that every step in the system has a unique number. For example generation 0 might have steps 0 and 1, generation 1 might have steps 2 through 4, and so on. A simple promotion policy is to promote survivors of step k to step $k+1$. In that case, the number of steps in a generation determines the number of scavenges (of that generation) necessary to promote objects to the next generation.

Each step consists of a number of *blocks*. A block is 2^n bytes, aligned on a 2^n byte boundary for some value of n chosen when the system is built. A typical block size might be 64K bytes. The number of blocks in a step may vary over time. While the blocks of a step

[3] For a more detailed discussion of the toolkit see [Hudson *et al.*, 1991].

are usually not contiguous, a *nursery* may be set up to consist of a number of contiguous blocks, so that one might more readily use a page trap (rather than an explicit limit check) to detect nursery overflow and trigger a scavenge.

Blocks have four primary advantages. First, they allow sizes of steps and generations to change easily since the storage of a step need not be contiguous. Second, they allow speedy determination of the generation, step, and promotion step of an object: the address of the object is simply shifted right by n bits and indexes a block table containing the needed information. Third, blocks match naturally with page trapping or card marking schemes (both of which the toolkit supports). Fourth, they reduce the storage needed under some circumstances when compared with copying collectors that use semi-spaces. If b bytes are present in a generation before a scavenge and the survivors consume a bytes, then a semi-space scheme uses $2 \times b$ bytes whereas our scheme uses $b + a$ bytes (modulo rounding resulting from the block size). The degree of advantage depends on the survival rate a/b, but may be significant in some applications.

Blocks do introduce a problem, however. They cannot handle objects larger than the block size. To handle such objects we provide a *large object space* (LOS), as suggested in [Ungar and Jackson, 1988]. In fact, it is probably a good idea to put into LOS any object that consumes a significant fraction of a block; we used the heuristic threshold of 1/8 of a block. Further, as also discussed in [Ungar and Jackson, 1988], any object that contains few pointers and that exceeds some threshold in size should be stored in LOS to avoid the overhead of copying. LOS uses free list allocation based on splay trees [Sleator and Tarjan, 1983, Sleator and Tarjan, 1985, Jones, 1986] and, once allocated, an LOS object is never moved. However, LOS objects still belong to a step, which is indicated by threading the objects onto a doubly linked list rooted in the step data structure. When a LOS object is promoted, we simply unchain it from one list and chain it into another. When scavenging is complete, any LOS objects remaining on a scavenged step's LOS list are freed.

While the generation, step, and block, of a non-LOS object can be determined using the simple shift and index technique, LOS may combine objects from different steps and generations in the same block. Therefore, we store a back reference from a LOS object's header to its containing step. It is relatively easy to determine the step given a pointer to the base of an LOS object, but determining the step given a pointer into the middle of the object requires locating the object header, which is supported but involves additional work.

3.3 Phases of a Scavenge

A scavenge consists of two phases. First, the root set for the scavenge is determined based on the remembered sets, as well as the stack, register, and global variable contents. All objects directly reachable from the roots are copied into new space, and the roots updated to point to the copied object. All objects reachable from the new space objects are then copied over using a non-recursive Cheney scan [Cheney, 1970].[4] As each object is copied, a forwarding pointer is left in the old copy, so that other references to the object can be updated as they are encountered. Since the toolkit makes no assumptions about object format, language implementors can define the details of the forwarding pointer format. The toolkit does

[4] The toolkit might be adapted to support mark-sweep or other approaches to collection, but currently it provides only copying collection. Also, it would not be hard to incorporate suggestions such as hierarchical clustering [Wilson *et al.*, 1991].

determine automatically where to allocate the new copy of the object, given the object's size (which must be determined by language-specific code).

Before a scavenge begins, the toolkit, following a dynamically modifiable plan supplied by the language implementor, determines the generations to be scavenged and creates new steps accordingly. It also sets up all the promotion step references. After a scavenge, all the old steps of the scavenged generations are deleted and their blocks become available for allocation.

These scavenge techniques work well for small heaps. In large heaps, however, scavenging older and older generations along with all younger generations becomes disruptive. In order to avoid this disruption we limit the number of generations in the heap. Any object that lives through several scavenges is moved into *mature object space* (MOS) where it is collected using our non-disruptive algorithm.

4 The Mature Object Space Algorithm

We now describe mature object space, its structure and its collection algorithm. The components used to implement this algorithm are the same as those used to implement the garbage collection toolkit for young objects. In particular, the blocks, the remembered sets, and the scanning and copying mechanisms are the same for both mature object space and generational space.

First, we will describe how mature space is divided into areas. Second, we will discuss the remembered set mechanisms used to track pointers between mature space areas. Third, we will present the rules that determine where mature objects are placed. Fourth, we will show how collection of an area results in objects being moved so that any unreachable object is eventually isolated and collected.

4.1 The Structure of Mature Object Space

Mature object space is divided into *areas*, just as young object space is divided into generations. The structure of an area is similar to a generation in that all pointers into an area can be found at scavenge time (i.e., each area has a remembered set). An area consists of one or more blocks. These blocks are the same as the blocks used in young object space and share the same bookkeeping functions, including quick determination of the area in which the block resides, and during a collection, determination of the area to which an object should be copied. In addition, blocks support determination of whether a pointer should be recorded in a remembered set. Unlike generations in the heap, age information is no longer interesting, so areas do not have steps.

Unlike Bishop's areas, our areas are sized so that each individual area can be collected quickly. The collector works on one area at a time. The problem with a straightforward implementation of Bishop's algorithm with limited area size is that a multiple area circular structure might not be collected because local information is not sufficient to determine if an object is globally unreachable. Hence, just as in Bishop's approach, we must migrate a multi-area cycle of garbage into a single area in order to reclaim it. However, the limit on area size makes this impossible if the linked structure is larger than can fit into a single area. Since Bishop did not restrict the size of areas, he did not have this problem. Hence, the key contribution of our algorithm is insuring that we reclaim large structures of garbage while still imposing the limit on area size, so that collections will be non-disruptive.

To further describe the structure of MOS, we first introduce some terminology. Pointers to mature objects from outside mature object space are *root pointers*. Root pointers reside in young object space, large object space, on the stack, in registers, and in static areas. Objects immediately reachable from roots are *leaders*. Objects that are not immediately reachable from roots, but still reachable from objects in mature object space, are *followers*.

We will use a train metaphor to describe the algorithm. An area can be thought of as a railroad *car*. The cars are used to bound the amount of work that is done during each invocation of the collector. A group of cars holding a linked structure of objects can be thought of as a *train*. Trains are used to group large related objects so that they can be managed as a unit.

4.2 Roots and Remembered Sets

Each area has an associated remembered set, which allows us to find all pointers from outside the area that refer to objects in the area. However, since we will scavenge an area only when all young spaces are also scavenged, and since all scavenges process all roots (stack(s), registers, static areas), remembered sets for areas need only track references from other MOS areas. The remembered set for a *train* is simply the union of the remembered sets of its cars, less any intra-train references.

A more subtle remembered set property comes from the fact that the algorithm processes areas in round-robin order. To understand this, suppose we assign each area a sequence number, and when an area is scavenged, it is assigned the next highest number. Then a remembered set need only record references from higher numbered to lower numbered areas. When an area is collected, its number will be the lowest, and hence we will be able to find all the references from other areas into the collected area.

We gain two advantages from handling the remembered sets this way. First, we reduce the total volume of remembered set information. If pointers are evenly distributed in terms of the direction they point, the remembered sets would be half as big, but it is not clear that the algorithm leads to such distributions, so the magnitude of this benefit is unclear. Second, and perhaps more importantly, we do not have to update *other* area's remembered sets when an area is scavenged. This is because none of the scavenged area's information could possibly be recorded in the other area's remembered sets, since such entries would record pointers from lower-numbered areas to higher-numbered ones, which our directionality rule specifically does not record.

The toolkit leaves the structure of the remembered sets up to the language implementor. The toolkit does, however, provide several alternative implementations including remembering slots, objects, cards, or pages. See [Hosking *et al.*, 1992] for performance studies comparing the available techniques.

4.3 Collecting an Area in Mature Object Space

As previously mentioned, we process areas in round robin order, collecting one area (or car) upon each scavenge of MOS (which implies that all young generations are also scavenged at the same time).

There is a check that is always done before collecting a car: if there are no root pointers to the train whose car is about to be collected, then we check the train's remembered set. If the remembered set is empty, then the entire train can be reclaimed with no further effort. We

can readily enhance the remembered set bookkeeping to make the check efficient (though possibly inaccurate for one round robin cycle of scavenging): record with each car the number of extra-train references to objects in that car, and also keep a sum across all the cars (easily updated as cars are collected (removed) or added).

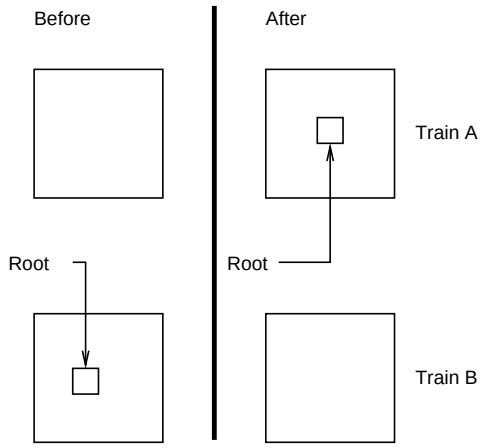

Fig. 1. Leaders in Train B are moved into another train.

When a car is collected we refer to it as a *from* car. Each reachable object in a *from* car has an associated *to* car which is determined by how the object is reached. First, we copy any objects in the car referred to by roots into either a new train or some *other* train (Figure 1). Which train we choose is a policy decision that does not affect the correctness of the algorithm. Next, we scan the copied objects and copy over, in typical copy collector style, all other objects in the *from* car reachable from objects in the other train.

At the same time we move objects being promoted from young generations into trains holding references to them, or if they are referred to by roots, into any train. Since the young generations are bounded in size, the volume of promoted objects is also bounded, so we can bound the disruption caused by promotions[5].

At this point the *from* car may still contain reachable follower objects, but they must be reachable only from other cars. Using the *from* car's remembered set, we locate all references from outside the train to objects still in the *from* car, and move them to the train containing the reference. See Figure 2.

The only remaining reachable objects in the *from* car are reachable from other cars in the same train. These objects are moved into the last car of the train as illustrated in Figure 3. This leaves only unreachable objects in the car. The space for these objects is then recycled.

This is similar to Bishop's approach. If the train to which we want to move an object is full, we add a car to that train and copy the object there. In any case, an object that is reachable from outside the train being collected is moved to some other train (thus collapsing garbage into fewer trains and eventually a single train), or (if unreachable) is reclaimed.

[5] Setting the size of young generations is a policy decision. The size can be limited by collecting young generations more often or by promoting more objects into mature space during each collection.

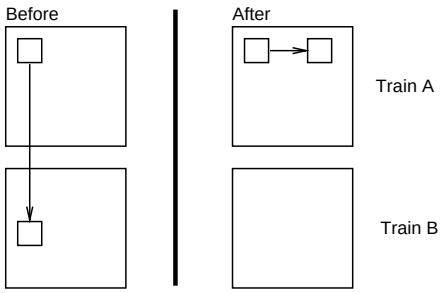

Fig. 2. Followers in Train B reachable from another train are moved there.

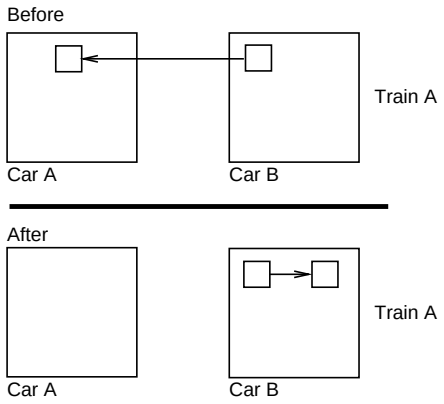

Fig. 3. Other followers are moved to the last car.

Of course, we scan moved objects, and evacuate any remaining reachable objects from the car. We collect cars in the order they were linked onto the train. Since we check to see if an object is referred to by another train *before* we check references inside the train, objects referred to directly from other trains will always moved out of this train. This is important since a train might contain a multi-area cycle of objects that "belongs in" another train, i.e., is not reachable from the leaders of this train.

The objects that are referenced during any one invocation of the algorithm are just those objects that are involved with the car where we are currently focused, either as a member of the car or by pointing into the car. This locality is known ahead of time so the algorithm will be able to provide the operating system hints about what locations it will need during the next cycle of the collector.

Since the algorithm periodically copies all reachable objects in mature space, it reclusters live objects at no additional cost. During the copying, we can apply sophisticated compaction and clustering techniques such as those described by Wilson [Wilson *et al.*, 1991]. In addition, this algorithm avoids the fragmentation that can occur with mark and sweep collectors.

4.4 Why the Collector Works

Having presented the collection algorithm, we now argue that it will eventually collect all unreachable objects, even large cycles of garbage. Suppose we have some garbage that threads through a number of trains. As we process the lowest numbered train, car by car, one of three things will happen to each object: it will be detected as garbage and reclaimed; it will be moved to another train; or it will be moved to another car of the same train (but not a car of the *new* train). The last case will not repeat indefinitely, since eventually we reach a situation where we have reclaimed or moved to other trains all objects reachable from roots or other trains, and the remembered set of the current train will be empty. By induction, then, in one round robin pass of the trains, the garbage structure will be compacted into a single train. Again, we see that as we process, car by car, eventually the train's remembered set will be empty and the garbage then reclaimed.

Each train pass may require objects to be copied several times to other cars in the train, but each pass through the cars in a train will reduce the number of objects since any object referenced from outside the train will be moved. By induction each pass through the cars on a train will either reduce the size of the train or reclaim the entire train.

One way to conceptualize the algorithm is as pulling different threads or chains of objects apart, until garbage is isolated and then reclaimed. Of course, smaller garbage structures are reclaimed sooner and with less copying, but the point is that the algorithm is guaranteed to reclaim garbage in a train or evacuate it into another train within $O(n^2)$ car collections, where n is the number of cars in the train. Since pieces of garbage structures can not be copied back into a train from which they were evacuated, the algorithm takes at most one pass through the trains to collect a garbage structure[6] while retaining the desired non-disruptive, incremental behavior.

5 An Example

The next several figures illustrate a simple example of how the algorithm works. For simplicity we will assume the maximum number of objects that can fit in a car is 3. This means that any given invocation of the collector will move at most three objects.

In Figure 4 we show three data structures. One structure, consisting of objects R, S, and T, is reachable from a root. The structure consisting of object A and B is circular garbage spanning two trains. The other structure, consisting of objects C, D, E and F, forms a large circular structure of garbage that can not fit into one car. We will show how the structure C-D-E-F is isolated and freed and how A-B is consolidated and freed.

We start by applying the rules to train B. Is any object in the train reachable from outside of train B? Both object A and object R are reachable so we focus our attention on car 1. Leader R is evacuated to another train. The choice of which train is a policy decision. Here we chose train A instead of creating a new train. Next follower B is reachable from train A so it is evacuated to train A. Object C is only reachable from train B so C is moved into the last car in train B. The space used for car 1 is now recycled. Figure 5 shows the state of the trains after the first invocation of the algorithm.

[6] It might require two passes through the objects if the train remembered set information is managed as previously discussed.

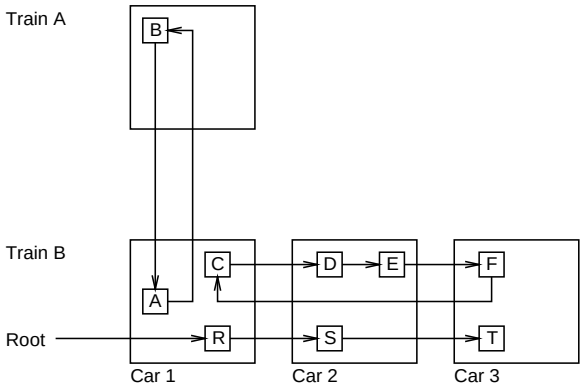

Fig. 4. The starting configuration

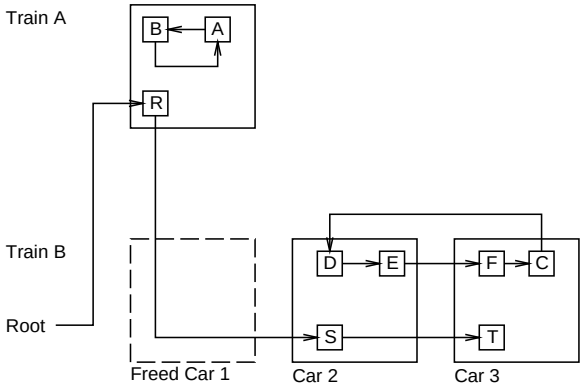

Fig. 5. Evacuate leader R, group A-B and copy C.

The second invocation of the collector focuses on car 2 in Figure 5. No objects are referenced by roots, so we look for objects in car 2 referenced from outside train B. Object S is referenced from train A, so we move S into train A. Since all cars in train A are full, we need to add another car to make room for S. Finally, we look for objects referenced from other cars in the train. Object D is moved into the last car of the train. Car 3 is full so we create a new car to make room for object D. The scanning of object D finds object E in car 2. E is evacuated into car 4. This gives us the state found in Figure 6. Notice how the live R-S-T structure is being extracted from the dead C-D-E-F structure.

On the next invocation of the algorithm we note that train B is still referenced so we focus on car 3. Again no objects are referenced by roots. Follower T is referenced by train A so it is moved into train A. Object T is scanned but contains no references into car 3. Next we consider references from within the train B. Object F is so referenced so it is moved into car 4. The scan of object F finds a reference to C. Since car 4 is full a new car is attached to the end of the train and C is moved into it. At this point (shown in Figure 7) structure R-S-T has been separated from structure C-D-E-F like pulling spaghetti out onto a fork.

The next invocation of the algorithm notes that train B has no references into it, so the

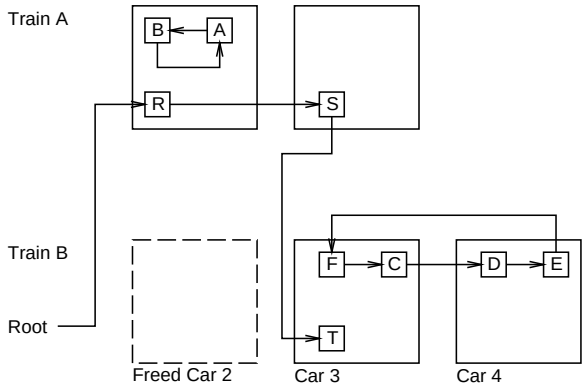

Fig. 6. Evacuate follower S and copy D and E.

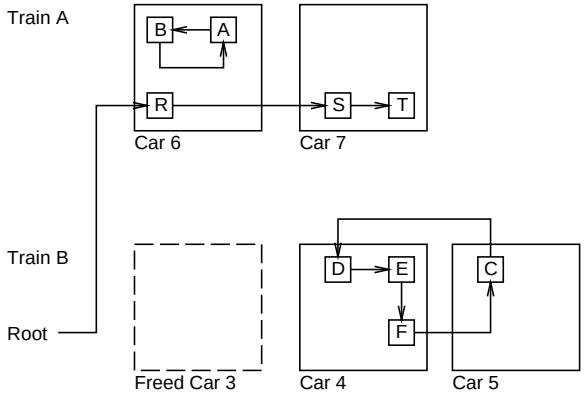

Fig. 7. Large cyclic dead structures are isolated into one train.

entire train is recycled immediately. Note that we isolated the structure C-D-E-F into one train where it can be reclaimed even though it is larger than any area we incrementally considered. This leaves us with only train A.

In Figure 8, we note that train A has a reference from outside the train so we focus on car 6. Since object R is reachable from a root we move it to another train. In this case we create a new train C and move R into it. Structure A-B, which used to form a circular list that spanned multiple trains, is now isolated and is recycled.

Figure 9 does not show recycled train B but does show the new train C that holds object R. We now consider car 7. Object S is moved into the train C and object S is scanned for references into car 7. Object T is found and moved into train C. Car 7 can now be recycled.

In Figure 10, what remains is a train with three neatly clustered live objects. These were the only three reachable objects present at the beginning of the example. The algorithm successfully grouped all unreachable objects into unreachable trains where they could be freed without disruption.

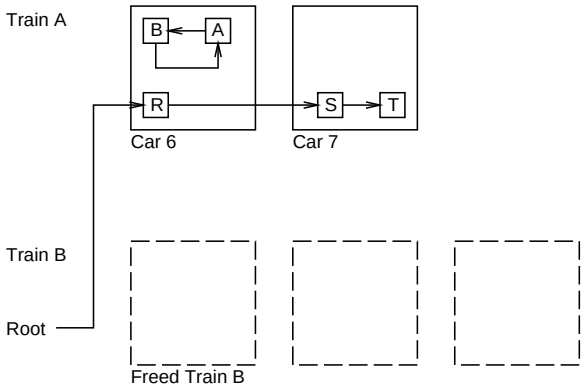

Fig. 8. Trains with no references can be freed.

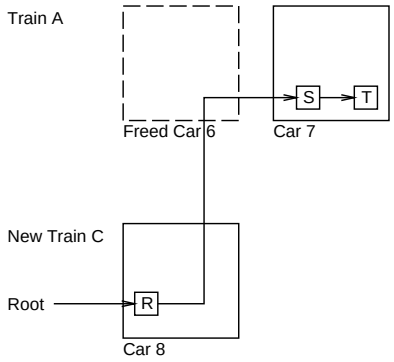

Fig. 9. Evacuate R so cycle A-B can now be freed.

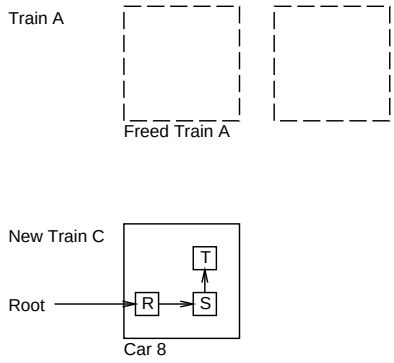

Fig. 10. Live structure R-S-T is clustered into one train.

6 Popular objects

There is one possible way in which our algorithm as presented might be disruptive. Call an object *popular* if there are many references to it. To copy a popular object, we must process a large remembered set and update many pointers.[7] In fact, we cannot bound the number of references to an object, so we cannot bound the work involved in moving an object.

Our solution is not to move such objects, analogously to the treatment of large objects. We can detect popular objects (or popular cars, anyway) by considering the size of their remembered set. If the remembered set size exceeds some threshold, we simply retain the whole area, logically (but not physically) copying it and having it start a new train (if it is an engine) or join the newest train (if it is a boxcar). With some cleverness we might be be able to clear out some objects, but it may not be worthwhile. The remembered set is discarded and will be rebuilt over time as we cycle through all the other areas. We need only take care that the threshold that determines popular versus non-popular areas is high enough that we can still collect highly linked cyclic garbage. Thus, the threshold should be no smaller than the number of pointers that fit in one area. We have yet to work out the details and correctness argument.

7 Future Work

We can add Wilson's temporal opportunism to our algorithm with no problem. Hayes's key object opportunism is more problematic since we assumed round-robin processing of the areas. To process areas in arbitrary order, we would have to remember all inter-area references (instead of just those pointing in one "direction") and we would have to deal with updating remembered sets. We might avoid updating remembered sets by including a "time-stamp" with remembered set entries, which would allow us to detect and ignore stale entries, rather than having to remove them immediately. The costs and benefits are unclear.

We envision a distributed version of the mature space algorithm. Though it falls outside the scope of this paper, we intend to develop a version where each node in a distributed system holds multiple complete trains. The algorithm does not change. If node *A* holds a structure *S* without a leader, then *S* will be migrated to some train in node *B* that holds a reference to *S*. If no node *B* is willing to accept the structure then either the structure will be discarded or node *B* and node *A* would have to agree on some sort of "rent" so node *A* could afford to retain *S*. Such a rental agreement would be equivalent to introducing a root in node *A* referencing *S*.

The MOS approach also seems promising for collect large persistent heaps for persistent and database languages. Some details would need to be worked out to insure that the algorithms makes as few secondary storage accesses as possible. It will probably pay to be opportunistic and do whatever processing one can on parts of the heap that are brought into main memory by normal application activity, as well as to exploit temporal opportunism to make more progress during periods of light load, etc.

[7] Large objects could also be a problem, but we can put them in large object space just as we do for the young generations.

8 Conclusions

We have described what we believe is the first efficient non-disruptive copy collection algorithm for mature objects. The algorithm is incremental, supports fast allocation, and supports compaction and clustering via copying. We believe this algorithm goes a long way towards making garbage collection palatable for a variety of languages and long running applications.

9 Acknowledgements

We appreciate Tony Hosking's work on implementing the toolkit discussed here. Amer Diwan and David Moon provided extensive comments on drafts of the paper. Other colleagues also read and critiqued the paper. Finally, we thank Barry Hayes for challenging us to implement key opportunism; it was thinking about that problem that led to our invention of the algorithm described here.

References

[Appel *et al.*, 1988] Andrew W. Appel, John R. Ellis, and Kai Li. Realtime concurrent collection on stock multiprocessors. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, June 1988), *ACM SIGPLAN Not. 23*, 7 (July 1988), pp. 11–20.

[Baker, 1978] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM 21*, 4 (April 1978), 280–294.

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Boehm *et al.*, 1991] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In [OOPSLA, 1991], pp. 157–164.

[Cheney, 1970] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM 13*, 11 (November 1970), 677–678.

[Diwan *et al.*, 1992] Amer Diwan, J. Eliot B. Moss, and Richard L. Hudson. Compiler support for garbage collection in a statically typed language. In *Conference on Programming Language Design and Implementation* (San Francisco, California, June 1992), SIGPLAN, ACM Press, pp. 273–282.

[Fenichel and Yochelson, 1969] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM 12*, 11 (November 1969), 611–612.

[Hayes, 1991] Barry Hayes. Using key object opportunism to collect old objects. In [OOPSLA, 1991], pp. 33–46.

[Hosking *et al.*, 1992] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Vancouver, Canada, October 1992). To appear.

[Hudson *et al.*, 1991] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, September 1991. Submitted for publication.

[Jones, 1986] Douglas W. Jones. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM 29*, 4 (April 1986), 300–311.

[Lang and Dupont, 1987] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. *SIGPLAN '87 – Symposium on Interpreters and Interpretive Techniques* (1987), 253–263.

[Lieberman and Hewitt, 1983] Henry Lieberman and Carl Hewitt. A real-time garbage collection based on the lifetimes of objects. *Communications of the ACM 26*, 6 (June 1983), 419–429.

[Moon, 1984] David Moon. Garbage collection in a large Lisp system. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Austin, TX, August 1984), pp. 235–246.

[OOPSLA, 1991] *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Phoenix, Arizona, October 1991), *ACM SIGPLAN Not. 26*, 11 (November 1991).

[Sleator and Tarjan, 1983] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. In *Proceedings of the ACM SIGACT Symposium on Theory* (Boston, Massachusetts, April 1983), pp. 235–245.

[Sleator and Tarjan, 1985] Daniel Dominic Sleator and Robert Endre Tarjan. Self-adjusting binary search trees. *Journal of the ACM 32*, 3 (July 1985).

[Ungar, 1984] David Ungar. 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, April 1984), *ACM SIGPLAN Not. 19*, 5 (May 1984), pp. 157–167.

[Ungar and Jackson, 1988] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Diego, California, September 1988), *ACM SIGPLAN Not. 23*, 11 (November 1988), pp. 1–17.

[Weinreb and Moon, 1981] Daniel Weinreb and David Moon. *Lisp Machine Manual*, third ed. Massachusetts Institute of Technology, 1981.

[White, 1980] Jon L. White. Address/memory management for a gigantic Lisp environment or, GC considered harmful. In *Proceedings of the ACM Symposium on Lisp and Functional Programming* (Stanford, California, August 1980), ACM, pp. 119–127.

[Wilson *et al.*, 1991] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective "static-graph" reorganization to improve locality in garbage-collected systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Canada, June 1991), *ACM SIGPLAN Not. 26*, 6 (June 1991), pp. 177–191.

[Wilson and Moher, 1989] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, Louisiana, October 1989), *ACM SIGPLAN Not. 24*, 10 (October 1989), pp. 23–35.