Uniprocessor Garbage Collection Techniques*

Paul R. Wilson

University of Texas
Austin, Texas 78712-1188 USA
(wilson@cs.utexas.edu)

Abstract. We survey basic garbage collection algorithms, and variations such as incremental and generational collection. The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and treadmill collection. *Incremental* techniques can keep garbage collection pause times short, by interleaving small amounts of collection work with program execution. *Generational* schemes improve efficiency and locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects.

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[2] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

This paper is intended to be an introductory survey of garbage collectors for uniprocessors, especially those developed in the last decade. For a more thorough treatment of older techniques, see [Knu69, Coh81].

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be

* This paper will appear in the proceedings of the **1992 International Workshop on Memory Management (St. Malo, France, September 1992) in the Springer-Verlag Lecture Notes in Computer Science series.**

[2] We use the term object loosely, to include any kind of structured data record, such as Pascal records or C structs, as well as full-fledged objects with encapsulation and inheritance, in the sense of object-oriented programming.

deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

Since liveness is a *global* property, this introduces nonlocal bookkeeping into routines that might otherwise be orthogonal, composable, and reusable. This bookkeeping can reduce extensibility, because when new functionality is implemented, the bookkeeping code must be updated.

The unnecessary complications created by explicit storage allocation are especially troublesome because programming mistakes often introduce erroneous behavior that breaks the basic abstractions of the programming language, making errors hard to diagnose.

Failing to reclaim memory at the proper point may lead to slow memory *leaks*, with unreclaimed memory gradually accumulating until the process terminates or swap space is exhausted. Reclaiming memory too soon can lead to very strange behavior, because an object's space may be reused to store a completely different object while an old pointer still exists. The same memory may therefore be interpreted as two different objects simultaneously with updates to one causing unpredictable mutations of the other.

These bugs are particularly dangerous because they often fail to show up repeatably, making debugging very difficult; they may never show up at all until the program is stressed in an unusual way. If the allocator happens not to reuse a particular object's space, a dangling pointer may not cause a problem. Later, in the field, the application may crash when it makes a different set of memory demands, or is linked with a different allocation routine. A slow leak may not be noticeable while a program is being used in normal ways—perhaps for many years—because the program terminates before too much extra space is used. But if the code is incorporated into a long-running server program, the server will eventually exhaust its swap space, and crash.

Explicit allocation and reclamation lead to program errors in more subtle ways as well. It is common for programmers to statically allocate a moderate number of objects, so that it is unnecessary to allocate them on the heap and decide when and where to reclaim them. This leads to fixed limitations on software, making them fail when those limitations are exceeded, possibly years later when memories (and data sets) are much larger. This "brittleness" makes code much less reusable, because the undocumented limits cause it to fail, even if it's being used in a way consistent with its abstractions. (For example, many compilers fail when faced with automatically-generated programs that violate assumptions about "normal" programming practices.)

These problems lead many applications programmers to implement some form of application-specific garbage collection within a large software system, to avoid most of the headaches of explicit storage management. Many large programs have their own data types that implement reference counting, for example. Unfortunately, these collectors are often both incomplete and buggy, because they are coded up for a one-shot application. The garbage collectors themselves are therefore often unreliable, as well as being hard to use because they are not integrated into the programming language. The fact that such kludges exist despite these problems is a testimony to the value of garbage collection, and it suggests that garbage collection should be part of programming language implementations.

In the rest of this paper, we focus on garbage collectors that are built into a language implementation. The usual arrangement is that the allocation routines of the language (or imported from a library) perform special actions to reclaim space, as necessary, when a memory request is not easily satisfied. (That is, calls to the "deallocator" are unnecessary because they are implicit in calls to the allocator.)

Most collectors require some cooperation from the compiler (or interpreter), as well: object formats must be recognizable by the garbage collector, and certain invariants must be preserved by the running code. Depending on the details of the garbage collector, this may require slight changes to the code generator, to emit certain extra information at compile time, and perhaps execute different instruction sequences at run time [Boe91, WH91, DMH92]. (Contrary to widespread misconceptions, there is no conflict between using a compiled language and garbage collection; state-of-the art implementations of garbage-collected languages use sophisticated optimizing compilers.)

1.2 The Two-Phase Abstraction

Garbage collection automatically reclaims the space occupied by data objects that the running program can never access again. Such data objects are referred to as *garbage*. The basic functioning of a garbage collector consists, abstractly speaking, of two parts:

1. Distinguishing the live objects from the garbage in some way, or *garbage detection*, and
2. Reclaiming the garbage objects' storage, so that the running program can use it.

In practice, these two phases may be functionally or temporally interleaved, and the reclamation technique is strongly dependent on the garbage detection technique.

In general, garbage collectors use a "liveness" criterion that is somewhat more conservative than those used by other systems. In an optimizing compiler, a value may be considered dead at the point that it can never be used again by the running program, as determined by control flow and data flow analysis. A garbage collector typically uses a simpler, less dynamic criterion, defined in terms of a *root set* and *reachability* from these roots. At the point when garbage collection occurs[3] all globally visible variables of active procedures are considered live, and so are the local variables of any active procedures. The *root set* therefore consists of the global variables, local variables in the activation stack, and any registers used by active procedures. Heap objects directly reachable from any of these variables could be accessed by the running program, so they must be preserved. In addition, since the program might traverse pointers from those objects to reach other objects, any object reachable from a live object is also live. Thus the set of live objects is simply the set of objects on any directed path of pointers from the roots.

[3] Typically, this happens when allocation of an object has been attempted by the running program, but there is not sufficient free memory to satisfy the request. The allocation routine calls a garbage collection routine to free up space, then allocates the requested object.

Any object that is not reachable from the root set is garbage, i.e., useless, because there is no legal sequence of program actions that would allow the program to reach that object. Garbage objects therefore can't affect the future course of the computation, and their space may be safely reclaimed.

1.3 Object Representations

Throughout this paper, we make the simplifying assumption that heap objects are self-identifying, i.e., that it is easy to determine the type of an object at run time. Implementations of statically-typed garbage collected languages typically have hidden "header" fields on heap objects, i.e., an extra field containing type information, which can be used to decode the format of the object itself. (This is especially useful for finding pointers to other objects.)

Dynamically-typed languages such as Lisp and Smalltalk usually use *tagged* pointers; a slightly shortened representation of the hardware address is used, with a small type-identifying field in place of the missing address bits. This also allows short immutable objects (in particular, small integers) to be represented as unique bit patterns stored directly in the "address" part of the field, rather than actually referred to by an address. This tagged representation supports polymorphic fields which may contain either one of these "immediate" objects or a pointer to an object on the heap. Usually, these short tags are augmented by additional information in heap-allocated objects' headers.

For a purely statically-typed language, no per-object runtime type information is actually necessary, except the types of the root set variables.[4] Once those are known, the types of their referents are known, and their fields can be decoded [App89a, Gol91]. This process continues transitively, allowing types to be determined at every pointer traversal. Despite this, headers are often used for statically-typed languages, because it simplifies implementations at little cost. (Conventional (explicit) heap management systems often use object headers for similar reasons.)

2 Basic Garbage Collection Techniques

Given the basic two-part operation of a garbage collector, many variations are possible. The first part, distinguishing live objects from garbage, may be done in several ways: by *reference counting*, *marking*, or *copying*. [5] Because each scheme has a major influence on the second part (reclamation) and on reuse techniques, we will introduce reclamation methods as we go.

[4] *Conservative* garbage collectors [BW88, Wen90, BDS91, WH91] are usable with little or no cooperation from the compiler—not even the types of named variables—but we will not discuss them here.

[5] Some authors use the term "garbage collection" in a narrower sense, which excludes reference counting and/or copy collection systems; we prefer the more inclusive sense because of its popular usage and because it's less awkward than "automatic storage reclamation."

2.1 Reference Counting

In a reference counting system [Col60], each object has an associated count of the references (pointers) to it. Each time a reference to the object is created, e.g., when a pointer is copied from one place to another by an assignment, the object's count is incremented. When an existing reference to an object is eliminated, the count is

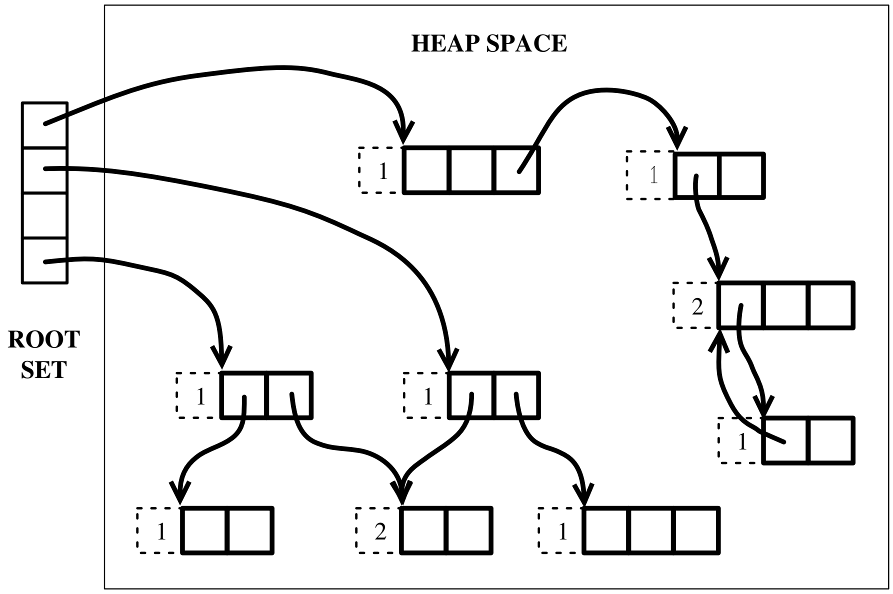

Fig. 1. Reference counting.

decremented. (See Fig. 1.) The memory occupied by an object may be reclaimed when the object's count equals zero, since that indicates that no pointers to the object exist and the running program could not reach it.

(In a straightforward reference counting system, each object typically has a header field of information describing the object, which includes a subfield for the reference count. Like other header information, the reference count is generally not visible at the language level.)

When the object is reclaimed, its pointer fields are examined, and any objects it holds pointers to also have their reference counts decremented, since references from a garbage object don't count in determining liveness. Reclaiming one object may therefore lead to the transitive decrementing of reference counts and reclaiming many other objects. For example, if the only pointer into some large data structure becomes garbage, all of the reference counts of the objects in that structure typically become zero, and all of the objects are reclaimed.

In terms of the abstract two-phase garbage collection, the adjustment and checking of reference counts implements the first phase, and the reclamation phase occurs when reference counts hit zero. These operations are both interleaved with the execution of the program, because they may occur whenever a pointer is created or destroyed.

One advantage of reference counting is this *incremental* nature of most of its operation—garbage collection work (updating reference counts) is interleaved closely with the running program's own execution. It can easily be made completely incremental and *real time*; that is, performing at most a small and bounded amount of work per unit of program execution.

Clearly, the normal reference count adjustments are intrinsically incremental, never involving more than a few operations for any given operation that the program executes. The transitive reclamation of whole data structures can be deferred, and also done a little at a time, by keeping a list of freed objects whose reference counts have become zero but which haven't yet been processed yet.

This incremental collection can easily satisfy real time requirements, guaranteeing that memory management operations never halt the executing program for more than a very brief period. This can support *real-time* applications in which guaranteed response time is critical; incremental collection ensures that the program is allowed to perform a significant, though perhaps appreciably reduced, amount of work in any significant amount of time. (A target criterion might be that no more than one millisecond out of every two-millisecond period would be spent on storage reclamation operations, leaving the other millisecond for "useful work" to satisfy the program's real-time purpose.)

There are two major problems with reference counting garbage collectors; they are difficult to make *efficient*, and they are not always *effective*.

The Problem with Cycles The effectiveness problem is that reference counting fails to reclaim *circular* structures. If the pointers in a group of objects create a (directed) cycle, the objects' reference counts are never reduced to zero, *even if there is no path to the objects from the root set* [McB63].

Figure 2 illustrates this problem. Consider the isolated pair of objects on the right. Each holds a pointer to the other, and therefore each has a reference count of one. Since no path from a root leads to either, however, the program can never reach them again.

Conceptually speaking, the problem here is that reference counting really only determines a *conservative approximation* of true liveness. If an object is not pointed

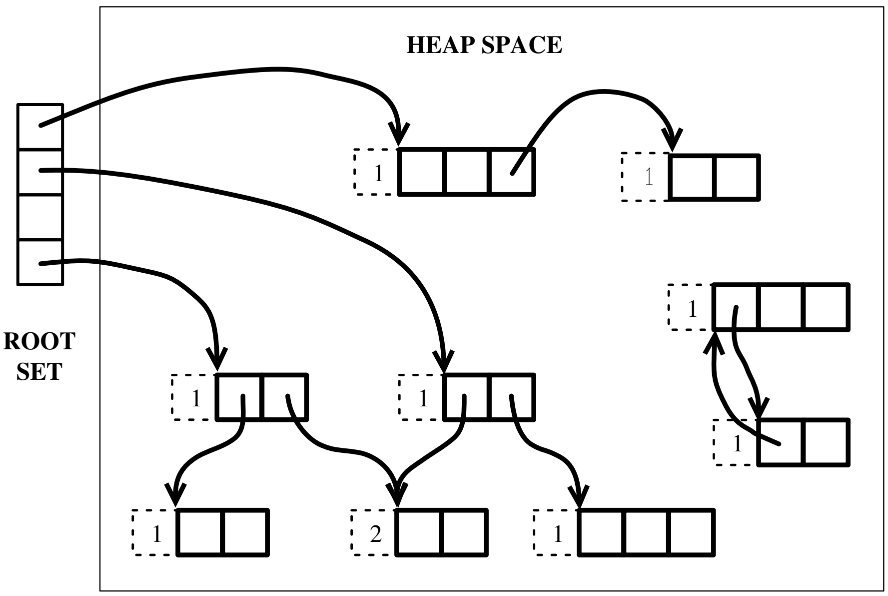

Fig. 2. Reference counting with unreclaimable cycle.

to by any variable or other object, it is clearly garbage, but the converse is often not true.

It may seem that circular structures would be very unusual, but they are not. While most data structures are acyclic, it is not uncommon for normal programs to create some cycles, and a few programs create very many of them. For example, nodes in trees may have "backpointers," to their parents, to facilitate certain operations. More complex cycles are sometimes formed by the use of hybrid data structures which combine advantages of simpler data structures, and the like.

Systems using reference counting garbage collectors therefore usually include some other kind of garbage collector as well, so that if too much uncollectable cyclic garbage accumulates, the other method can be used to reclaim it.

Many programmers who use reference-counting systems (such as Interlisp and early versions of Smalltalk) have modified their programming style to avoid the creation of cyclic garbage, or to break cycles before they become a nuisance. This has a negative impact on program structure, and many programs still have storage "leaks" that accumulate cyclic garbage which must be reclaimed by some other means.[6] These leaks, in turn, can compromise the real-time nature of the algorithm, because the system may have to fall back to the use of a non-real-time collector at a critical moment.

The Efficiency Problem. The efficiency problem with reference counting is that its cost is generally proportional to the amount of work done by the running program, with a fairly large constant of proportionality. One cost is that when a pointer is created or destroyed, its referent's count must be adjusted. If a variable's value is changed from one pointer to another, *two* objects' counts must be adjusted—one object's reference count must be incremented, the other's decremented and then checked to see if it has reached zero.

Short-lived stack variables can incur a great deal of overhead in a simple reference-counting scheme. When an argument is passed, for example, a new pointer appears on the stack, and usually disappears almost immediately because most procedure activations (near the leaves of the call graph) return very shortly after they are called. In these cases, reference counts are incremented, and then decremented back to their original value very soon. It is desirable to optimize away most of these increments and decrements that cancel each other out.

Deferred Reference Counting. Much of this cost can be optimized away by special treatment of local variables [DB76]. Rather than always adjusting reference counts and reclaiming objects whose counts become zero, references from the local variables are not included in this bookkeeping most of the time. Usually, reference counts are only adjusted to reflect pointers from one heap object to another. This means that reference counts may not be accurate, because pointers from the stack may be created or destroyed without being accounted for; that, in turn, means that objects whose count drops to zero may not actually be reclaimable. Garbage collection can only be done when references from the stack are taken into account.

Every now and then, the reference counts are brought up to date by scanning the stack for pointers to heap objects. Then any objects whose reference counts are still zero may be safely reclaimed. The interval between these phases is generally chosen to be short enough that garbage is reclaimed often and quickly, yet still long enough that the cost of periodically updating counts (for stack references) is not high.

This *deferred reference counting* [DB76] avoids adjusting reference counts for most short-lived pointers from the stack, and greatly reduces the overhead of reference counting. When pointers from one heap object to another are created or destroyed, however, the reference counts must still be adjusted. This cost is still roughly proportional to the amount of work done by the running program in most systems, but with a lower constant of proportionality.

---

[6] [Bob80] describes modifications to reference counting to allow it to handle some special cases of cyclic structures, but this restricts the programmer to certain stereotyped patterns.

There is another cost of reference-counting collection that is harder to escape. When objects' counts go to zero and they are reclaimed, some bookkeeping must be done to make them available to the running program. Typically this involves linking the freed objects into one or more "free lists" of reusable objects, out of which the program's allocation requests are satisfied.

It is difficult to make these reclamation operations take less than several instructions per object, and the cost is therefore proportional to the number of objects allocated by the running program.

These costs of reference counting collection have combined with its failure to reclaim circular structures to make it unattractive to most implementors in recent years. As we will explain below, other techniques are usually more efficient and reliable.

(This is not to say that reference counting is a dead technique. It still has advantages in terms of the immediacy with which it reclaims most garbage,[7] and corresponding beneficial effects on locality of reference;[8] a reference counting system may perform with little degradation when almost all of the heap space is occupied by live objects, while other collectors rely on trading more space for higher efficiency. Reference counts themselves may be valuable in some systems. For example, they may support optimizations in functional language implementations by allowing destructive modification of uniquely-referenced objects. Distributed garbage collection is often done with reference-counting between nodes of a distributed system, combined with mark-sweep or copying collection within a node. Future systems may find other uses for reference counting, perhaps in hybrid collectors also involving other techniques, or when augmented by specialized hardware. Nonetheless, reference counting is generally not considered attractive as the primary garbage collection technique on conventional uniprocessor hardware.)

For most high-performance general-purpose systems, reference counting has been abandoned in favor of *tracing* garbage collectors, which actually traverse (trace out) the graph of live objects, distinguishing them from the unreachable (garbage) objects which can then be reclaimed.

2.2 Mark-Sweep Collection

Mark-sweep garbage collectors [McC60] are named for the two phases that implement the abstract garbage collection algorithm we described earlier:

1. *Distinguish the live objects from the garbage.* This is done by tracing—starting at the root set and actually traversing the graph of pointer relationships—usually by either a depth-first or breadth-first traversal. The objects that are reached are *marked* in some way, either by altering bits within the objects, or perhaps by recording them in a bitmap or some other kind of table.

[7] This can be useful for *finalization*, that is, performing "clean-up" actions (like closing files) when objects die [Rov85].

[8] DeTreville [DeT90] argues that the locality characteristics of reference-counting may be superior to those of other collection techniques, based on experience with the Topaz system. However, as [WLM92] shows, generational techniques can recapture some of this locality.

2. *Reclaim the garbage.* Once the live objects have been made distinguishable from the garbage objects, memory is *swept*, that is, exhaustively examined, to find all of the unmarked (garbage) objects and reclaim their space. Traditionally, as with reference counting, these reclaimed objects are linked onto one or more free lists so that they are accessible to the allocation routines.

There are three major problems with traditional mark-sweep garbage collectors. First, it is difficult to handle objects of varying sizes without fragmentation of the available memory. The garbage objects whose space is reclaimed are interspersed with live objects, so allocation of large objects may be difficult; several small garbage objects may not add up to a large contiguous space. This can be mitigated somewhat by keeping separate free lists for objects of varying sizes, and merging adjacent free spaces together, but difficulties remain. (The system must choose whether to allocate more memory as needed to create small data objects, or to divide up large contiguous hunks of free memory and risk permanently fragmenting them. This fragmentation problem is not unique to mark-sweep—it occurs in reference counting as well, and in most explicit heap management schemes.)

The second problem with mark-sweep collection is that the cost of a collection is proportional to the size of the heap, including both live and garbage objects. All live objects must be marked, and all garbage objects must be collected, imposing a fundamental limitation on any possible improvement in efficiency.

The third problem involves locality of reference. Since objects are never moved, the live objects remain in place after a collection, interspersed with free space. Then new objects are allocated in these spaces; the result is that objects of very different ages become interleaved in memory. This has negative implications for locality of reference, and simple mark-sweep collectors are often considered unsuitable for most virtual memory applications. (It is possible for the "working set" of active objects to be scattered across many virtual memory pages, so that those pages are frequently swapped in and out of main memory.) This problem may not be as bad as many have thought, because objects are often created in clusters that are typically active at the same time. Fragmentation and locality problems are is unavoidable in the general case, however, and a potential problem for some programs.

It should be noted that these problems may not be insurmountable, with sufficiently clever implementation techniques. For example, if a bitmap is used for mark bits, 32 bits can be checked at once with a 32-bit integer ALU operation and conditional branch. If live objects tend to survive in clusters in memory, as they apparently often do, this can greatly diminish the constant of proportionality of the sweep phase cost; the theoretical linear dependence on heap size may not be as troublesome as it seems at first glance. As a result, the dominant cost may be the marking phase, which is proportional to the amount of live data that must be traversed, not the total amount of memory allocated. The clever use of bitmaps can also reduce the cost of allocation, by allowing fast allocation from contiguous unmarked areas, rather than using free lists.

The clustered survival of objects may also mitigate the locality problems of re-allocating space amid live objects; if objects tend to survive or die in groups in memory [Hay91], the interspersing of objects used by different program phases may not be a major consideration.

At this point, the technology of mark-sweep collectors (and related hybrids) is rapidly evolving. As will be noted later, this makes them resemble copying collectors in some ways; at this point we do not claim to be able to pick a winner between high-tech mark-sweep and copy collectors.

2.3 Mark-Compact Collection

Mark-compact collectors remedy the fragmentation and allocation problems of mark-sweep collectors. As with mark-sweep, a marking phase traverses and marks the reachable objects. Then objects are *compacted*, moving most of the live objects until all of the live objects are contiguous. This leaves the rest of memory as a single contiguous free space. This is often done by a linear scan through memory, finding live objects and "sliding" them down to be adjacent to the previous object. Eventually, all of the live objects have been slid down to be adjacent to a live neighbor. This leaves one contiguous occupied area at one end of heap memory, and implicitly moving all of the "holes" to the contiguous area at the other end.

This sliding compaction has several interesting properties. The contiguous free area eliminates fragmentation problems so that allocating objects of various sizes is simple. Allocation can be implemented as the incrementing of a pointer into a contiguous area of memory, in much the way that different-sized objects can be allocated on a stack. In addition, the garbage spaces are simply "squeezed out," without disturbing the original ordering of objects in memory. This can ameliorate locality problems, because the allocation ordering is usually more similar to subsequent access orderings than an arbitrary ordering imposed by a garbage collector [CG77, Cla79].

While the locality that results from sliding compaction is advantageous, the collection process itself shares the mark-sweep's unfortunate property that several passes over the data are required. After the initial marking phase, sliding compactors make two or three more passes over the live objects [CN83]. One pass computes the new locations that objects will be moved to; subsequent passes must update pointers to refer to objects' new locations, and actually move the objects. These algorithms may be therefore be significantly slower than mark-sweep if a large percentage of data survives to be compacted.

An alternative approach is to use a *two-pointer algorithm*, which scans inward from both ends of a heap space to find opportunities for compaction. One pointer scans downward from the top of the heap, looking for live objects, and the other scans upward from the bottom, looking for a hole to put it in. (Many variations of this algorithm are possible, to deal with multiple areas holding different-sized objects, and to avoid intermingling objects from widely-dispersed areas.) For a more complete treatment of compacting algorithms, see [Knu69, CN83].

2.4 Copying Garbage Collection

Like mark-compact (but unlike mark-sweep), *copying* garbage collection does not really "collect" garbage. Rather, it moves all of the *live* objects into one area, and the rest of the heap is then known to be available because it contains only garbage.

"Garbage collection" in these systems is thus only implicit, and some researchers avoid applying that term to the process.

Copying collectors, like marking-and-compacting collectors, move the objects that are reached by the traversal to a contiguous area. While compacting collectors use a separate marking phase that traverses the live data, copying collectors integrate the traversal of the data and the copying process, so that most objects need only be traversed once. Objects are moved to the contiguous destination area as they are reached by the traversal. The work needed is proportional to the amount of live data (all of which must be copied).

The term *scavenging* is applied to the copying traversal, because it consists of picking out the worthwhile objects amid the garbage, and taking them away.

A Simple Copying Collector: "Stop-and-Copy" Using Semispaces. A very common kind of copying garbage collector is the *semispace* collector [FY69] using the *Cheney* algorithm for the copying traversal [Che70]. We will use this collector as a reference model for much of this paper.[9]

In this scheme, the space devoted to the heap is subdivided into two contiguous *semispaces*. During normal program execution, only one of these semispaces is in use, as shown in Fig. 3. Memory is allocated linearly upward through this "current" semispace as demanded by the executing program. This is much like allocation from a stack, or in a sliding compacting collector, and is similarly fast; there is no fragmentation problem when allocating objects of various sizes.

When the running program demands an allocation that will not fit in the unused area of the current semispace, the program is stopped and the copying garbage collector is called to reclaim space (hence the term "stop-and-copy"). All of the live data are copied from the current semispace (*fromspace*) to the other semispace (*tospace*). Once the copying is completed, the tospace semispace is made the "current" semispace, and program execution is resumed. Thus the roles of the two spaces are reversed each time the garbage collector is invoked. (See Fig. 4.)

Perhaps the simplest form of copying traversal is the Cheney algorithm [Che70]. The immediately-reachable objects form the initial queue of objects for a breadth-first traversal. A "scan" pointer is advanced through the first object, location by location. Each time a pointer into fromspace is encountered, the referred-to-object is transported to the end of the queue, and the pointer to the object is updated to refer to the new copy. The free pointer is then advanced and the scan continues. This effects the "node expansion" for the breadth-first traversal, reaching (and copying) all of the descendants of that node. (See Fig. 5. Reachable data structures in fromspace are shown at the top of the figure, followed by the first several states of tospace as the collection proceeds—tospace is shown in linear address order to emphasize the linear scanning and copying.)

[9] As a historical note, the first copying collector was Minsky's collector for Lisp 1.5 [Min63]. Rather than copying data from one area of memory to another, a single heap space was used. The live data were copied out to a file, and then read back in, in a contiguous area of the heap space. On modern machines this would be unbearably slow, because file operations—writing and reading every live object—are now many orders of magnitude slower than memory operations.

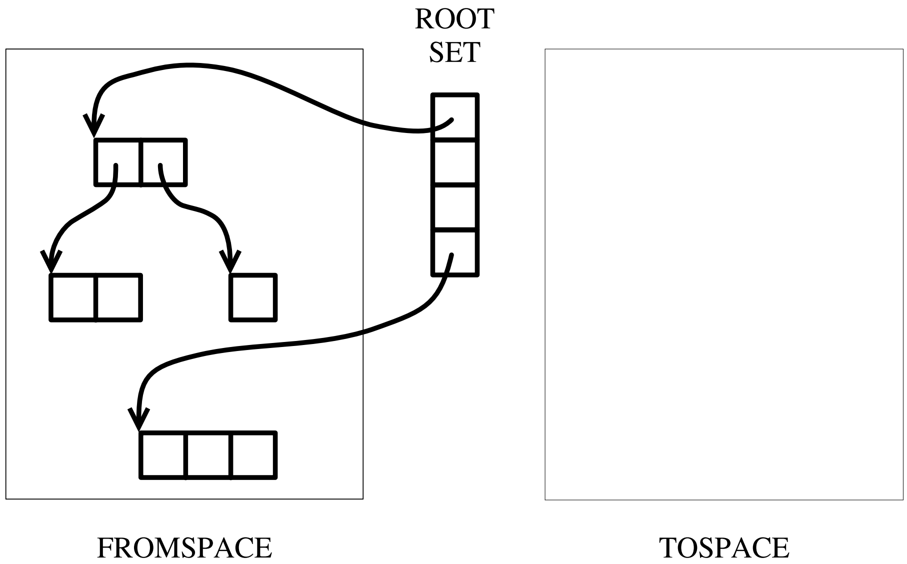

Fig. 3. A simple semispace garbage collector before garbage collection.

Rather than stopping at the end of the first object, the scanning process simply continues through subsequent objects, finding their offspring and copying them as well. A continuous scan from the beginning of the queue has the effect of removing consecutive nodes and finding all of their offspring. The offspring are copied to the end of the queue. Eventually the scan reaches the end of the queue, signifying that all of the objects that have been reached (and copied) have also been scanned for descendants. This means that there are no more reachable objects to be copied, and the scavenging process is finished.

Actually, a slightly more complex process is needed, so that objects that are reached by multiple paths are not copied to tospace multiple times. When an object is transported to tospace, a *forwarding pointer* is installed in the old version of the object. The forwarding pointer signifies that the old object is obsolete and indicates

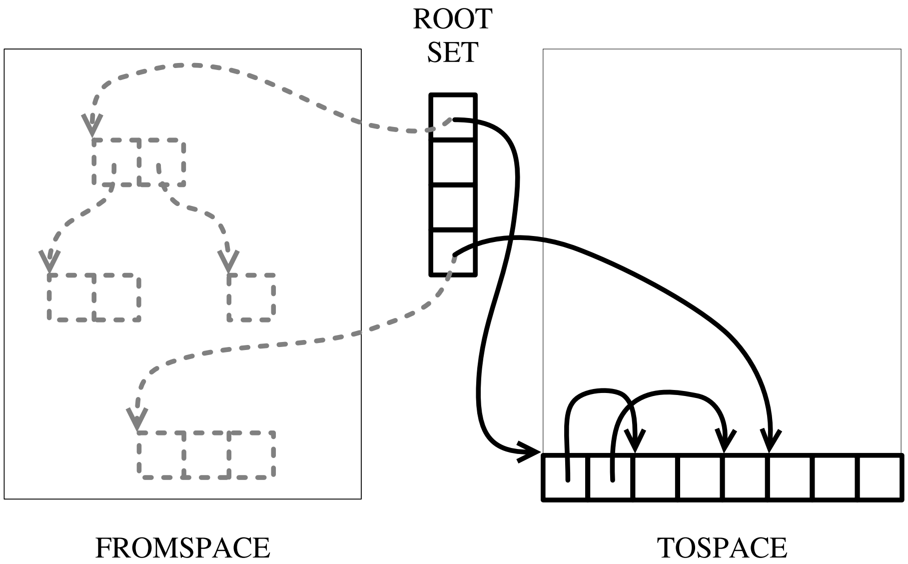

Fig. 4. Semispace collector after garbage collection.

where to find the new copy of the object. When the scanning process finds a pointer into fromspace, the object it refers to is checked for a forwarding pointer. If it has one, it has already been moved to tospace, so the pointer it has been reached by is simply updated to point to its new location. This ensures that each live object is transported exactly once, and that all pointers to the object are updated to refer to the new copy.

Efficiency of Copying Collection. A copying garbage collector can be made arbitrarily efficient if sufficient memory is available [Lar77, App87]. The work done at each collection is proportional to the amount of live data at the time of garbage collection. Assuming that approximately the same amount of data is live at any given time during the program's execution, decreasing the frequency of garbage collections

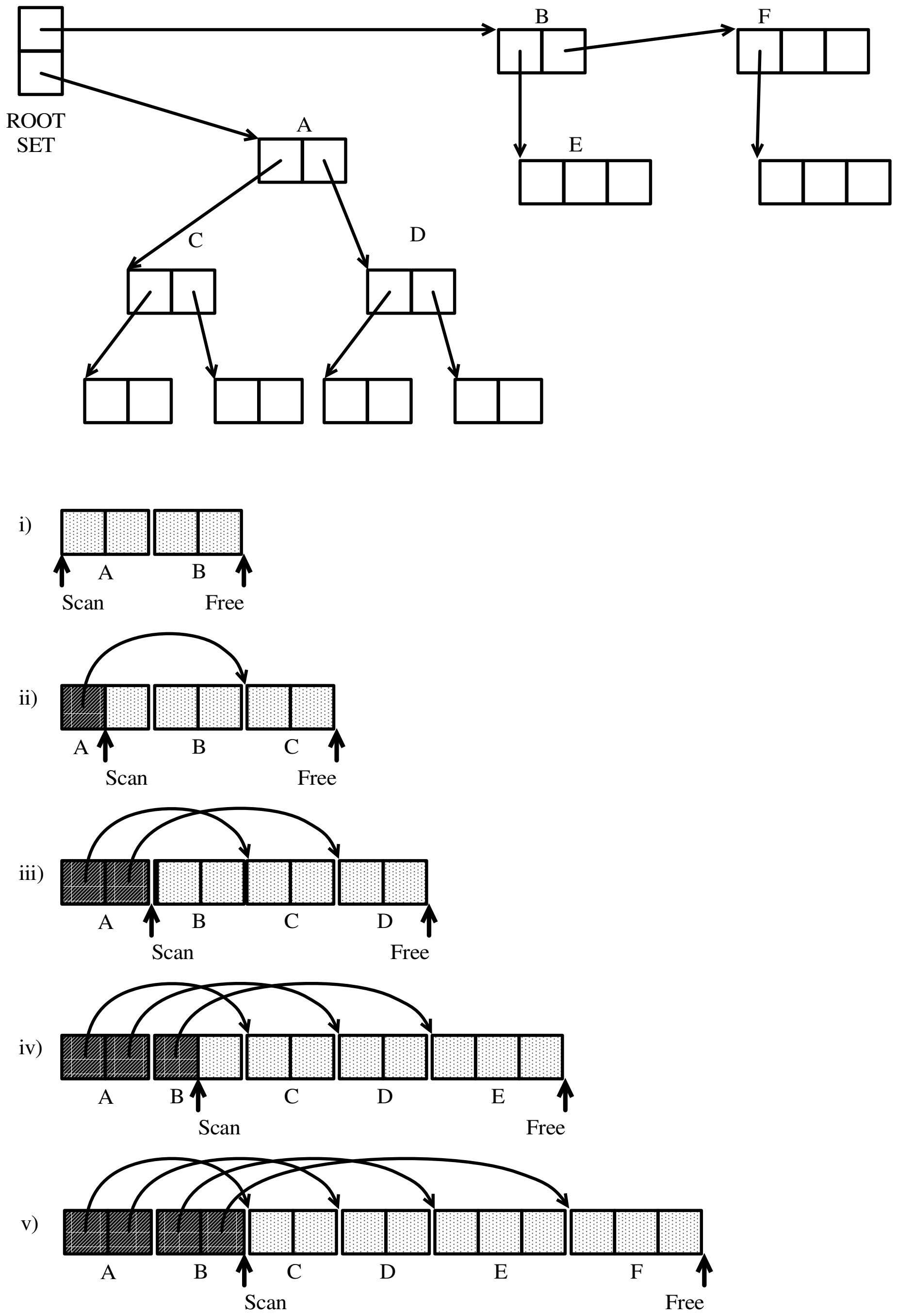

Fig. 5. The Cheney algorithm of breadth-first copying.

will decrease the total amount of garbage collection effort.

A simple way to decrease the frequency of garbage collections is to increase the amount of memory in the heap. If each semispace is bigger, the program will run longer before filling it. Another way of looking at this is that by decreasing the frequency of garbage collections, we are increasing the average age of objects at garbage collection time. Objects that become garbage before a garbage collection needn't be copied, so the chance that an object will *never* have to be copied is increased.

Suppose, for example, that during a program run twenty megabytes of memory are allocated, but only one megabyte is live at any given time. If we have two three-megabyte semispaces, garbage will be collected about ten times. (Since the current semispace is one third full after a collection, that leaves two megabytes that can be allocated before the next collection.) This means that the system will copy about half as much data as it allocates, as shown in the top part of Fig. 6. (Arrows represent copying of live objects between semispaces at garbage collections.)

On the other hand, if the size of the semispaces is doubled, 5 megabytes of free space will be available after each collection. This will force garbage collections a third as often, or about 3 or 4 times during the run. This straightforwardly reduces the cost of garbage collection by more than half, as shown in the bottom part of Fig. 6.

2.5 Non-Copying Implicit Collection

Recently, Baker [Bak92] has proposed a new kind of non-copying collector that with some of the efficiency advantages of a copying scheme. Baker's insight is that in a copying collector, the "spaces" of the collector are really just a particular implementation of sets. Another implementation of sets could do just as well, provided that it has similar performance characteristics. In particular, given a pointer to an object, it must be easy to determine which set it is a member of; in addition, it must be easy to switch the roles of the sets, just as fromspace and tospace roles are exchanged in a copy collector.

Baker's non-copying system adds two pointer fields and a "color" field to each object. These fields are invisible to the application programmer, and serve to link each hunk of storage into a doubly-linked list that serves as a set. The color field indicates which set an object belongs to.

The operation of this collector is simple, and isomorphic to the copy collector's operation. Chunks of free space are initially linked to form a doubly-linked list, and are allocated simply by incrementing a pointer into this list. The allocation pointer serves to divide the list into the part that has been allocated and the remaining "free" part. Allocation is fast because it only requires advancing this pointer to point at the next element of the free list. (Unlike the copying scheme, this does not eliminate fragmentation problems; supporting variable sized objects requires multiple free lists and may result in fragmentation of the available space.)

When the free list is exhausted, the collector traverses the live objects and "moves" them from the allocated set (which we could call the fromset) to another set (the toset). This is implemented by unlinking the object from the doubly-linked fromset list, toggling its mark field, and linking it into the toset's doubly-linked list.

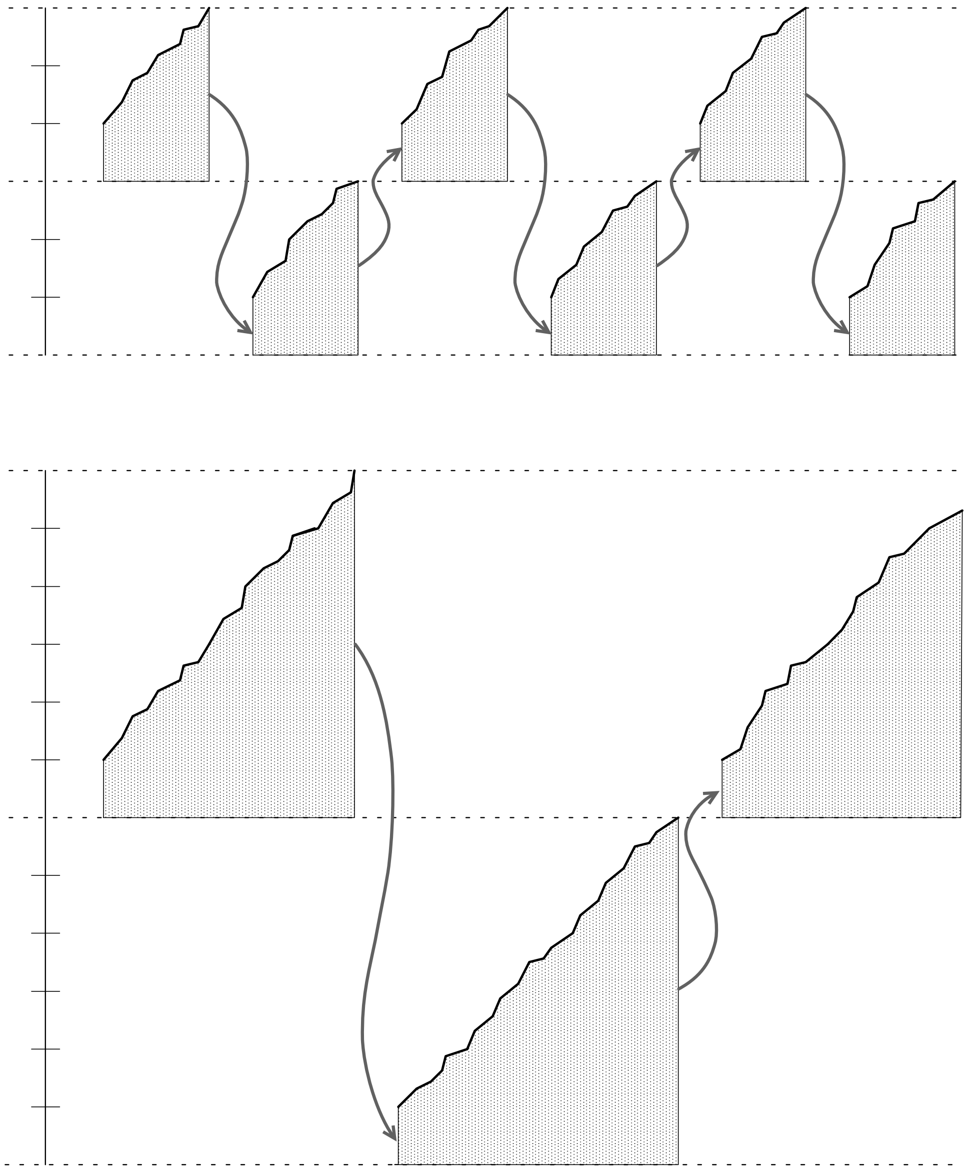

Fig. 6. Memory usage in a semispace GC, with 3 MB (top) and 6 MB (bottom) semispaces

Just as in a copy collector, space reclamation is implicit. When all of the reachable objects have been traversed and moved from the fromset to the toset, the fromset is known to contain only garbage. It is therefore a list of free space, which can immediately be put to use as a free list. (As we will explain in section 3.3, Baker's scheme is actually somewhat more complex, because his collector is incremental.) The cost of the collection is proportional to the number of live objects, and the garbage ones are all reclaimed in small constant time.

This scheme has both advantages and disadvantages compared to a copy collector. On the minus side, the per-object constants are probably a little bit higher, and fragmentation problems are still possible. On the plus side, the tracing cost for large objects is not as high. As with a mark-sweep collector, the whole object needn't be copied; if it can't contain pointers, it needn't be scanned either. Perhaps more importantly for many applications, this scheme does not require the actual language-level pointers between objects to be changed, and this imposes fewer constraints on compilers. As we'll explain later, this is particularly important for parallel and real-time incremental collectors.

2.6 Choosing Among Basic Techniques

Treatments of garbage collection algorithms in textbooks often stress asymptotic complexity, but all basic algorithms have roughly similar costs, especially when we view garbage collection as part of the overall free storage management scheme. Allocation and garbage collection are two sides of the basic memory reuse coin, and any algorithm incurs costs at allocation time, if only to initialize the fields of new objects.

Any of the efficient collection schemes therefore has three basic cost components, which are (1) the initial work required at each collection, such as root set scanning, (2) the work done at per unit of allocation (proportional to the amount of allocation, or the number of objects allocated) and (3) the work done during garbage detection (e.g., tracing).

The latter two costs are usually similar, in that the amount of live data is usually some significant percentage of the amount of garbage. Thus algorithms whose cost is proportional to the amount of allocation (e.g., mark-sweep) may be competitive with those whose cost is proportional to the amount of live data traced (e.g., copying).

For example, suppose that 10 percent of all allocated data survive a collection, and 90 percent never need to be traced. In deciding which algorithm is more efficient, the asymptotic complexity is less important than the associated constants. If the cost of sweeping an object is ten times less than the cost of copying it, the mark-sweep collector costs about the same as as copy collector. (If a mark-sweep collector's sweeping cost is billed to the allocator, and it's small relative to the cost of initializing the objects, then it becomes obvious that the sweep phase is just not terribly expensive.) While current copying collectors appear to be more efficient than current mark-sweep collectors, the difference is not large for state-of-the art implementations.

Further, real high-performance systems often use hybrid techniques to adjust tradeoffs for different categories of objects. Many high-performance copy collectors use a separate *large object area* [CWB86, UJ88], to avoid copying large objects from space to space. The large objects are kept "off to the side" and usually managed in-place by some variety of marking traversal and free list technique.

A major point in favor of in-place collectors (such as mark-sweep and treadmill schemes) is the ability to make them *conservative* with respect to data values that may be pointers or may not. This allows them to be used for languages like C, or off-the-shelf optimizing compilers [BW88, Bar88, BDS91], which can make it difficult or impossible to unambiguously identify all pointers at run time. A non-moving collector can be conservative because anything that looks like a pointer object can be left where it is, and the (possible) pointer to it doesn't need to be changed. In contrast, a copying collector must know whether a value is a pointer—and whether to move the object and update the pointer. For example, if presumed pointers were updated, and some were actually integers, the program would break because the integers would be mysteriously changed by the garbage collector.

2.7 Problems with a Simple Garbage Collector

It is widely known that the asymptotic complexity of copying garbage collection is excellent—the copying cost approaches zero as memory becomes very large. Treadmill collection shares this property, but other collectors can be similarly efficient if the constants associated with memory reclamation and reallocation are small enough. In that case, garbage detection is the major cost.

Unfortunately, it is difficult in practice to achieve high efficiency in a simple garbage collector, because large amounts of memory are too expensive. If virtual memory is used, the poor locality of the allocation and reclamation cycle will generally cause excessive paging. (Every location in the heap is used before any location's space is reclaimed and reused.) Simply paging out the recently-allocated data is expensive for a high-speed processor [Ung84], and the paging caused by the copying collection itself may be tremendous, since all live data must be touched in the process.)

It therefore doesn't generally pay to make the heap area larger than the available main memory. (For a mathematical treatment of this tradeoff, see [Lar77].) Even as main memory becomes steadily cheaper, locality within cache memory becomes increasingly important, so the problem is simply shifted to a different level of the memory hierarchy [WLM92].

In general, we can't achieve the potential efficiency of simple garbage collection; increasing the size of memory to postpone or avoid collections can only be taken so far before increased paging time negates any advantage.

It is important to realize that this problem is not unique to copying collectors. *All* garbage collection strategies involve similar space-time tradeoffs—garbage collections are postponed so that garbage detection work is done less often, and that means that space is not reclaimed as quickly. On average, that increases the amount of memory wasted due to unreclaimed garbage.[10]

[10] Deferred reference counting, like tracing collection, also trades space for time—in giving up continual incremental reclamation to avoid spending CPU cycles in adjusting reference counts, one gives up space for objects that become garbage and are not immediately reclaimed. At the time scale on which memory is reused, the resulting locality character-

While copying collectors were originally designed to improve locality, in their simple versions this improvement is not large, and their locality can in fact be *worse* than that of non-compacting collectors. These systems may improve the locality of reference to long-lived data objects, which have been compacted into a contiguous area. However, this effect is swamped by the pattern of references due to allocation. Large amounts of memory are touched *between* collections, and this alone makes them unsuitable for a virtual memory environment.

The major locality problem is not with the locality of compacted data, or with the locality of the garbage collection process itself. The problem is an *indirect* result of the use of garbage collection—by the time space is reclaimed *and reused*, it's likely to have been paged out, simply because too many other pages have been allocated in between. Compaction is helpful, but the help is generally *too little, too late*. With a simple semispace copy collector, locality is likely to be worse than that of a mark-sweep collector, simply because the copy collector uses more total memory—only half the memory can be used between collections. Fragmentation of live data is not as detrimental as the regular reuse of two spaces.[11]

The only way to have good locality is to ensure that memory is large enough to hold the regularly-reused area. (Another approach would be to rely on optimizations such as prefetching, but this is not feasible at the level of virtual memory—disks simply can't keep up with the rate of allocation because of the enormous speed differential between RAM and disk.) *Generational* collectors address this problem by reusing a smaller amount of memory more often; they will be discussed in Sect. 4. (For historical reasons, is widely believed that only copying collectors can be made generational, but this is not the case. Generational mark-sweep collectors are somewhat harder to construct, but they do exist and are quite practical [DWH+90].

Finally, the temporal distribution of a simple tracing collector's work is also troublesome in an interactive programming environment; it can be very disruptive to a user's work to suddenly have the system become unresponsive and spend several seconds garbage collecting, as is common in such systems. For large heaps, the pauses may be on the order of seconds, or even minutes if a large amount of data is dispersed through virtual memory. Generational collectors alleviate this problem, because most garbage collections only operate on a subset of memory. Eventually they must garbage collect larger areas, however, and the pauses may be considerably longer. For real time applications, this may not be acceptable.

3 Incremental Tracing Collectors

For truly real-time applications, fine-grained incremental garbage collection appears to be necessary. Garbage collection cannot be carried out as one atomic action

istics must share basic performance tradeoff characteristics with generational collectors of the copying or mark-sweep varieties, which will be discussed later.

[11] Slightly more complicated copying schemes appear to avoid this problem [Ung84, WM89], but [WLM92] demonstrates that *cyclic* memory reuse patterns can fare poorly in hierarchical memories because of recency-based (e.g., LRU) replacement policies. This suggests that freed memory should be reused in a LIFO fashion (i.e., in the opposite order of its previous allocation), if the entire reuse pattern can't be kept in memory.

while the program is halted, so small units of garbage collection must be interleaved with small units of program execution. As we said earlier, it is relatively easy to make reference counting collectors incremental. Reference counting's problems with efficiency and effectiveness discourage its use, however, and it is therefore desirable to make tracing (copying or marking) collectors incremental.

In most of the following discussion, the difference between copying and mark-sweep collectors is not particularly important. The incremental tracing for garbage detection is more interesting than the incremental reclamation of detected garbage.

The difficulty with incremental tracing is that while the collector is tracing out the graph of reachable data structures, the graph may change—the running program may *mutate* the graph while the collector "isn't looking." For this reason, discussions of incremental collectors typically refer to the running program as the *mutator* [DLM+78]. (From the garbage collector's point of view, the actual application is merely a coroutine or concurrent process with an unfortunate tendency to modify data structures that the collector is attempting to traverse.) An incremental scheme must have some way of keeping track of the changes to the graph of reachable objects, perhaps re-computing parts of its traversal in the face of those changes.

An important characteristic of incremental techniques is their degree of conservatism with respect to changes made by the mutator during garbage collection. If the mutator changes the graph of reachable objects, freed objects may or may not be reclaimed by the garbage collector. Some *floating garbage* may go unreclaimed because the collector has already categorized the object as live before the mutator frees it. This garbage *is* guaranteed to be collected at the next cycle, however, because it will be garbage at the *beginning* of the next collection.

3.1 Tricolor Marking

The abstraction of *tricolor marking* is helpful in understanding incremental garbage collection. Garbage collection algorithms can be described as a process of traversing the graph of reachable objects and coloring them. The objects subject to garbage collection are conceptually colored white, and by the end of collection, those that will be retained must be colored black. When there are no reachable nodes left to blacken, the traversal of live data structures is finished.

In a simple mark-sweep collector, this coloring is directly implemented by setting mark bits—objects whose bit is set are black. In a copy collector, this is the process of moving objects from fromspace to tospace—unreached objects in fromspace are considered white, and objects moved to tospace are considered black. The abstraction of coloring is orthogonal to the distinction between marking and copying collectors, and is important for understanding the basic differences between incremental collectors.

In incremental collectors, the intermediate states of the coloring traversal are important, because of ongoing mutator activity—the mutator can't be allowed to change things "behind the collector's back" in such a way that the collector will fail to find all reachable objects.

To understand and prevent such interactions between the mutator and the collector, it is useful to introduce a third color, grey, to signify that an object has been reached by the traversal, but that *its descendants may not have been*. That is, as the traversal proceeds outward from the roots, objects are initially colored grey. When they are scanned and pointers to their offspring are traversed, they are blackened and the offspring are colored grey.

In a copying collector, the grey objects are the objects in the unscanned area of tospace—the ones between the scan and free pointers. Objects that have been passed by the scan pointer are black. In a mark-sweep collector, the grey objects correspond to the stack or queue of objects used to control the marking traversal, and the black objects are the ones that have been removed from the queue. In both cases, objects that have not been reached yet are white.

Intuitively, the traversal proceeds in a wavefront of grey objects, which separates the white (unreached) objects from the black objects that have been passed by the wave—that is, there are no pointers directly from black objects to white ones. This abstracts away from the particulars of the traversal algorithm—it may be depth-first, breadth-first, or just about any kind of exhaustive traversal. It is only important that a well-defined grey fringe be identifiable, and that the mutator preserve the invariant that no black object hold a pointer directly to a white object.

The importance of this invariant is that the collector must be able to assume that it is "finished with" black objects, and can continue to traverse grey objects and move the wavefront forward. If the mutator creates a pointer from a black object to a white one, it must somehow coordinate with the collector, to ensure that the collector's bookkeeping is brought up to date.

Figure 7 demonstrates this need for coordination. Suppose the object A has been completely scanned (and therefore blackened); its descendants have been reached and greyed. Now suppose that the mutator swaps the pointer from A to C with the pointer from B to D. The only pointer to D is now in a field of A, which the collector has already scanned. If the traversal continues without any coordination, C will be reached again (from B), and D will never be reached at all.

Incremental approaches There are two basic approaches to coordinating the collector with the mutator. One is to use a *read barrier*, which detects when the mutator attempts to access a pointer to a white object, and immediately colors the object grey; since the mutator can't read pointers to white objects, it can't install them in black objects. The other approach is more direct, and involves a *write barrier*—when the program attempts to write a pointer into an object, the write is trapped or recorded.

Write barrier approaches, in turn, fall into two different categories, depending on which aspect of the problem they address. To foil the garbage collector's marking traversal, it is necessary for the mutator to 1) write a pointer to a white object into a black object *and* 2) destroy the original pointer before the collector sees it.

If the first condition (writing the pointer into a black object) does not hold, no special action is needed—if there are other pointers to the white object from grey objects, it will be retained, and if not, it is garbage and needn't be retained anyway. If the second condition (obliterating the original path to the object) does not hold, the object will be reached via the original pointer and retained. The two write-barrier approaches focus on these two aspects of the problem.

Snapshot-at-beginning collectors ensure that the second condition cannot happen—rather than allowing pointers to be simply overwritten, they are first saved so that

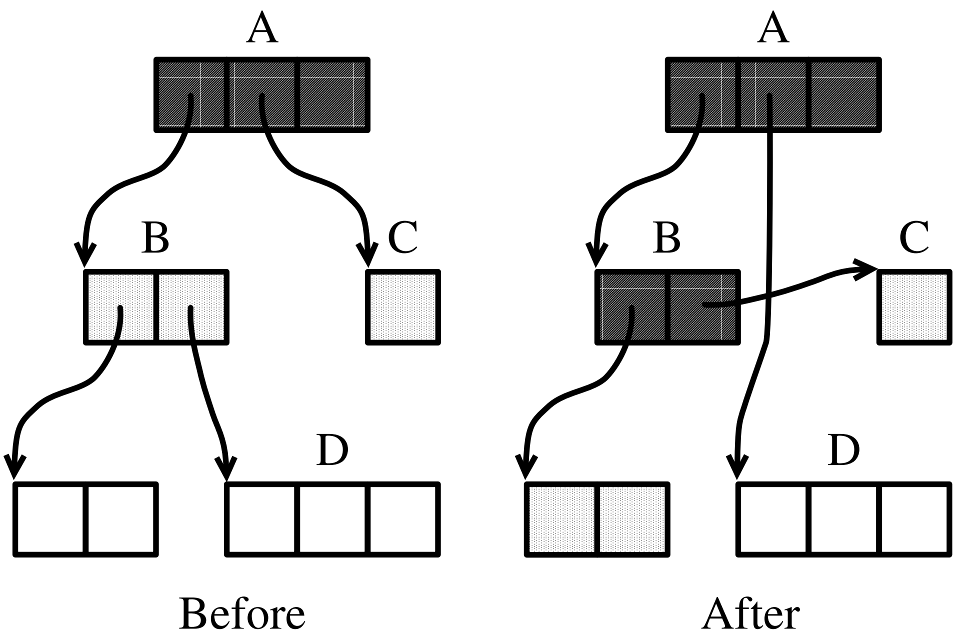

Fig. 7. A violation of the coloring invariant.

the collector can find them. Thus no paths to white objects can be broken without providing another path to the object for the garbage collector.

Incremental update collectors are still more direct in dealing with these troublesome pointers. Rather than saving copies of all pointers that are overwritten (because they *might* have already been copied into black objects) they actually record pointers stored into black objects, and catch the troublesome pointers at their destination, rather than their source. That is, if a pointer to a white object is copied into a black object, that new copy of the pointer will be found. Conceptually, the black object (or part of it) is reverted to grey when the mutator "undoes" the collector's traversal. (Alternatively, the pointed-to object may be greyed immediately.) This ensures that the traversal is updated in the face of mutator changes.

3.2 Baker's Incremental Copying.

The best-known real-time garbage collector is Baker's incremental copying scheme [Bak78]. It is an adaptation of the simple copy collection scheme described in Sect. 2.5, and uses a *read barrier* for coordination with the mutator. For the most part, the copying of data proceeds in the Cheney (breadth-first) fashion, by advancing the scan pointer through the unscanned area of tospace and moving any referred-to objects from fromspace. This *background scavenging* is interleaved with mutator operation, however.

An important feature of Baker's scheme is its treatment of objects allocated by the mutator during incremental collection. These objects are allocated in tospace and are treated as though they had already been scanned—i.e., they are assumed to be live. In terms of tricolor marking, new objects are *black* when allocated, and none of them can be reclaimed; they are never reclaimed until the next garbage collection cycle.[12]

In order to ensure that the scavenger finds all of the live data and copies it to tospace before the free area in newspace is exhausted, the rate of copy collection work is tied to the rate of allocation. Each time an object is allocated, an increment of scanning and copying is done.

In terms of tricolor marking, the scanned area of tospace contains black objects, and the copied but unscanned objects (between the scan and free pointer) are grey. As-yet unreached objects in fromspace are white. The scanning of objects (and copying of their offspring) moves the wavefront forward.

In addition to the background scavenging, other objects may be copied to tospace as needed to ensure that the basic invariant is not violated—pointers into fromspace must not be stored into objects that have already been scanned, undoing the collector's work.

Baker's approach is to couple the collector's copying traversal with the mutator's traversal of data structures. The mutator is never allowed to see pointers into fromspace, i.e., pointers to white objects. Whenever the mutator reads a (potential) pointer from the heap, it immediately checks to see if it is a pointer into fromspace; if so, the referent is copied to tospace, i.e., its color is changed from white to grey. In effect, this advances the wavefront of greying just ahead of the actual references by the mutator, keeping the mutator inside the wavefront.[13]

It should be noted that Baker's collector itself changes the graph of reachable objects, in the process of copying. The read barrier does not just inform the collector of changes by the mutator, to ensure that objects aren't lost; it also shields the *mutator* from viewing temporary inconsistencies created by the collector. If this were not done, the mutator might encounter two different pointers to versions of the same object, one of them obsolete.

12 Baker suggests copying old live objects into one end of tospace, and allocating new objects in the other end. The two occupied areas of tospace thus grow toward each other.

13 Nilsen's variant of Baker's algorithm updates the pointers without actually copying the objects—the copying is lazy, and space in tospace is simply reserved for the object before the pointer is updated [Nil88]. This makes it easier to provide smaller bounds on the time taken by list operations, and to gear collector work to the amount of allocation—including guaranteeing shorter pauses when smaller objects are allocated.

This shielding of the mutator from white objects has come to be called a *read barrier*, because it prevents pointers to white objects from being read by the program at all.

The read barrier may be implemented in software, by preceding each read (of a potential pointer from the heap) with a check and a conditional call to the copying-and-updating routine. (Compiled code thus contains extra instructions to implement the read barrier.) Alternatively, it may be implemented with specialized hardware checks and/or microcoded routines.

The read barrier is quite expensive on stock hardware, because in the general case, any load of a pointer must check to see if the pointer points to a fromspace (white) object; if so, it must execute code to move the object to tospace and update the pointer. The cost of these checks is high on conventional hardware, because they occur very frequently. Lisp Machines have special purpose hardware to detect pointers into fromspace and trap to a handler[Gre84, Moo84, Joh91], but on conventional machines the checking overhead is in the tens of percent for a high performance system.

Brooks has proposed a variation on Baker's scheme, where objects are *always* referred to via an indirection field embedded in the object itself [Bro84]. If an object is valid, its indirection field points to itself. If it's an obsolete version in tospace, its indirection pointer points to the new version. Unconditionally indirecting is cheaper than checking for indirections, but would still incur overheads in the tens of percent for a high-performance system. (A variant of this approach has been used by North and Reppy in a concurrent garbage collector [NR87].) Zorn takes a different approach to reducing the read barrier overhead, using knowledge of important special cases and special compiler techniques. Still, the time overheads are on the order of twenty percent [Zor89].

3.3 The Treadmill

Recently, Baker has proposed a non-copying version of his scheme, which uses doubly-linked lists (and per-object color fields) to implement the sets of objects of each color, rather than separate memory areas. By avoiding the actual moving of objects and updating of pointers, the scheme puts fewer restrictions on other aspects of language implementation.[14]

This non-copying scheme preserves the essential efficiency advantage of copy collection, by reclaiming space implicitly. (As described in Sect. 2.5, unreached objects on the allocated list can be reclaimed by appending the remainder of that list to the free list.) The real-time version of this scheme links the various lists into a cyclic structure, as shown in Fig. 8. This cyclic structure is divided into four sections.

The *new* list is where allocation of new objects occurs during garbage collection—it is contiguous with the free list, and allocation occurs by advancing the pointer that separates them. At the beginning of garbage collection, the new segment is empty.

The *from* list holds objects that were allocated before garbage collection began, and which are subject to garbage collection. As the collector and mutator traverse

[14] In particular, it is possible to deal with compilers that do not unambiguously identify pointer variables in the stack, making it impossible to use simple copy collection.

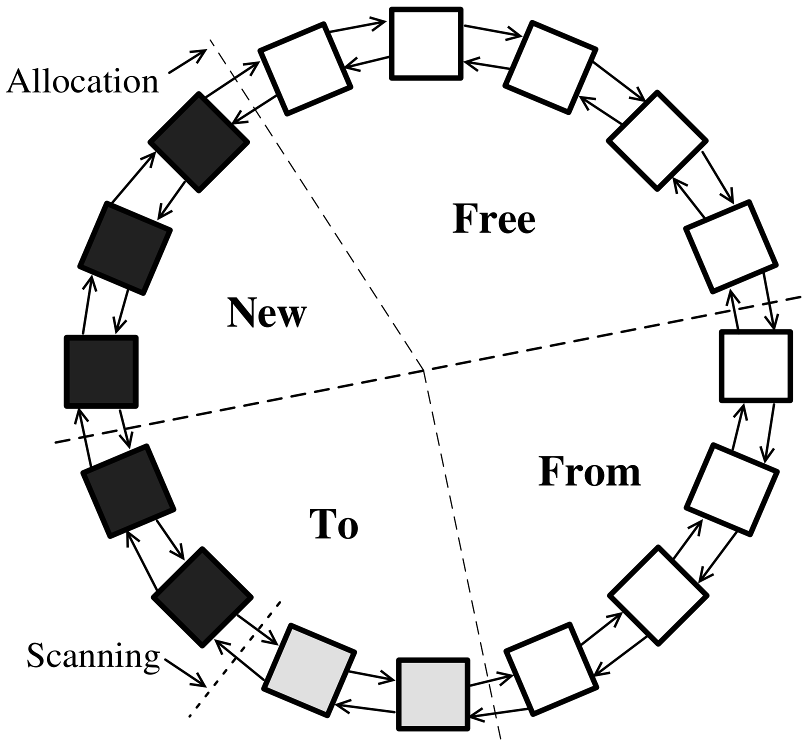

Fig. 8. Treadmill collector during collection.

data structures, objects are moved from the from list to the to list. The to list is initially empty, but grows as objects are “unsnapped” (unlinked) from the from list (and snapped into the to list) during collection.

The new list contains new objects, which are allocated black. The to-list contains both black objects (which have been completely scanned) and grey ones (which have been reached but not scanned). Note the isomorphism with the copying algorithm—even an analogue of the Cheney algorithm can be used. It is only necessary to have a scan pointer into the from list and advance it through the grey objects.

Eventually, all of the reachable objects in the from list have been moved to the to list, and scanned for offspring. When no more offspring are reachable, all of the objects in the to-list are black, and the remaining objects in the from list are known to be garbage. At this point, the garbage collection is complete. The from list is now available, and can simply be merged with the free list. The to list and the new list both hold objects that were preserved, and they can be merged to form the new to-list at the next collection.[15]

The state is very similar to the beginning of the previous cycle, except that the segments have "moved" partway around the cycle—hence the name "treadmill."

Baker describes this algorithm as being isomorphic to his original incremental copying algorithm, presumably including the close coupling between the mutator and the collector, i.e., the read barrier.

Conservatism in Baker's scheme. Baker's garbage collector uses a somewhat conservative approximation of true liveness in two ways.[16] The most obvious one is that objects allocated during collection are assumed to be live, even if they die before the collection is finished. The second is that pre-existing objects may become garbage after having been reached by the collector's traversal, and they will not be reclaimed—once an object has been greyed, it will be considered live until the next garbage collection cycle. On the other hand, if objects become garbage during collection, and all paths to those objects are destroyed *before* being traversed, then they *will* be reclaimed. That is, the mutator may overwrite a pointer from a grey object, destroying the only path to one or more white objects and ensuring that the collector will not find them. Thus Baker's incremental scheme incrementally updates the reachability graph of pre-existing objects, only when grey objects have pointers overwritten. Overwriting pointers from black objects has no effect, however, because their referents are already grey. The degree of conservatism (and floating garbage) thus depends on the details of the collector's traversal and of the program's actions.

3.4 Snapshot-at-Beginning write-barrier algorithms

If a non-copying collector is used, the use of a read barrier is an unnecessary expense; there is no need to protect the mutator from seeing an invalid version of a pointer. *Write barrier* techniques are cheaper, because heap writes are several times less common than heap reads. *Snapshot-at-beginning* algorithms use a write barrier to ensure that *no* objects ever become inaccessible to the garbage collector while collection is in progress. Conceptually, at the beginning of garbage collection, a *copy-on-write* virtual copy of the graph of reachable data structures is made. That is, the graph of reachable objects is fixed at the moment garbage collection starts, even though the actual traversal proceeds incrementally.

Perhaps the simplest and best-known snapshot collection algorithm is Yuasa's [Yua90]. If a location is written to, the overwritten value is first saved and pushed on a marking stack for later examination. This guarantees that no objects will become unreachable to the garbage collector traversal—all objects live at the beginning of

[15] This discussion is a bit oversimplified; Baker uses four colors, and whole lists can have their colors changed instantaneously by changing the sense of the bit patterns, rather than the patterns themselves.

[16] This kind of conservatism is not to be confused with the conservative treatment of pointers that cannot be unambiguously identified. (For a more complete and formal discussion of various kinds of conservatism in garbage collection, see [DWH+90].)

garbage collection will be reached, even if the pointers to them are overwritten. In the example shown in Fig. 7, the pointer from B to D is pushed onto the stack when it is overwritten with the pointer to C.

Yuasa's scheme has a large advantage over Baker's on stock hardware, because only heap pointer writes must be treated specially to preserve the garbage collector invariants. Normal pointer dereferencing and comparison does not incur any extra overhead.

On the other hand, Yuasa's scheme is more conservative than Baker's. Not only are all objects allocated during collection retained, but *no* objects can be freed during collection—all of the overwritten pointers are preserved and traversed. These objects are reclaimed at the next garbage collection cycle.

3.5 Incremental Update Write-Barrier Algorithms

While both are write-barrier algorithms, snapshot-at-beginning and *incremental update* algorithms are quite different. Unfortunately, incremental update algorithms have generally been cast in terms of parallel systems, rather than as incremental schemes for serial processing; perhaps due to this, they have been largely overlooked by implementors targeting uniprocessors.

Perhaps the best known of these algorithms is due to Dijkstra *et al.* [DLM+78]. (This is similar to the scheme developed independently by Steele [Ste75], but simpler because it does not deal with compactification.) Rather than retaining everything that's in a snapshot of the graph at the *beginning* of garbage collection, it heuristically (and somewhat conservatively) attempts to retain the objects that are live at the *end* of garbage collection. Objects that die during garbage collection—and before being reached by the marking traversal—are not traversed and marked.

To avoid the problem of pointers escaping into reachable objects that have already been scanned, such copied pointers are caught at their *destination*, rather than their source. Rather than noticing when a pointer escapes *from* a location that hasn't been traversed, it notices when the pointer escapes *into* an object that *has* already been traversed. If a pointer is overwritten without being copied elsewhere, so much the better—the object is garbage, so it might as well not get marked.

If the pointer is installed into an object already determined to be live, that pointer must be taken into account—it has now been incorporated into the graph of reachable data structures. Such pointer stores are recorded by the write barrier—the collector is notified which black objects may hold pointers to white objects, in effect reverting those objects to grey. Those formerly-black objects will be scanned again before the garbage collection is complete, to find any live objects that would otherwise escape. (This process may iterate, because more black objects may be reverted while the collector is in the process of traversing them. The traversal is guaranteed to complete, however, and the collector eventually catches up with the mutator.)

Objects that become garbage during garbage collection may be reclaimed at the end of that garbage collection, not the next one. This is similar to Baker's read-barrier algorithm in its treatment of pre-existing objects—they are not preserved if they become garbage before being reached by the collector.

It is less conservative than Baker's and Yuasa's algorithms in its treatment of objects allocated by the mutator during collocation, however. Baker's and Yuasa's schemes assume such newly-created objects are live, because pointers to them may get installed into objects that have already been reached by the collector's traversal. In terms of tricolor marking, objects are allocated "black", rather than white—they are conservatively assumed to be part of the graph of reachable objects. (In Baker's algorithm, there is no write barrier to detect whether they have been incorporated into the graph or not.)

In the Dijkstra *et al.* scheme, objects are assumed *not* to be reachable when they're allocated. In terms of tricolor marking, objects are allocated *white*, rather than black. At some point, the stack must be traversed and the objects that are reachable *at that time* are marked and therefore preserved.

We believe that this has a potentially significant advantage over Baker's or Yuasa's schemes. Most objects are short-lived, so if the collector doesn't reach those objects early in its traversal, they're likely never to be reached, and instead to be reclaimed very promptly. Compared to Baker's or Yuasa's scheme, there's an extra computational cost—by assuming that *all* objects allocated during collection are reachable, those schemes avoid the cost of traversing and marking those that actually *are* reachable. On the other hand, there's a space benefit with the incremental update scheme—the majority of those objects can be reclaimed at the end of a collection, which is likely to make it worth traversing the others. (In Steele's algorithm, some objects are allocated white and some are not, depending on the colors of their referents [Ste75]. This heuristic attempts to allocate short-lived objects white to reclaim their space quickly, while treating other objects conservatively to avoid traversing them. The cost of this technique is not quantified, and its benefits are unknown.)

3.6 Choosing Among Incremental Techniques

In choosing an incremental collection design, it is instructive to keep in mind the abstraction of tricolor marking, as distinct from mechanisms such as mark-sweep or copy collection. For example, Brooks' collector [Bro84] is actually a write barrier algorithm, even though Brooks describes it as an optimization of Baker's scheme.[17] Similarly, Dawson's[Daw82] copy collection scheme is cast as a variant of Baker's, but it is actually an incremental update scheme, similar to Dijkstra *et al.*'s; objects are allocated in fromspace, i.e., white.

The choice of a read- or write-barrier scheme is likely to be made on the basis of the available hardware. Without specialized hardware support, a write barrier appears to be easier to implement efficiently, because heap pointer writes are much less common than pointer traversals.

Appel, Ellis and Li [AEL88] use virtual memory (pagewise) access protection facilities as a coarse approximation of Baker's read barrier[AEL88, AL91, Wil91].

[17] The use of uniform indirections may be viewed as *avoiding* the need for a Baker-style read barrier—the indirections isolate the collector from changes made by the mutator, allowing them to be decoupled. The actual coordination, in terms of tricolor marking, is through a write barrier.

Rather than checking each load to see if a pointer to fromspace is being loaded, the mutator is simply not allowed to see any page that might contain such a pointer. Pointers in the scanned area of tospace are guaranteed to contain only pointers into tospace. Any pointers from fromspace to tospace must be from the unscanned area, so the collector simply access-protects the unscanned area, i.e., the grey objects. When the mutator accesses a protected page, a trap handler immediately scans the whole page, fixing up all the pointers (i.e., blackening all of the objects in the page); referents in fromspace are relocated to tospace (i.e., greyed) and access-protected.

Unfortunately this scheme fails to provide meaningful real-time guarantees in the general case. (It does support concurrent collection, however, and greatly reduces the cost of the read barrier.) In the worst case, each pointer traversal may cause the scanning of a page of tospace until the whole garbage collection is complete.[18]

Of write barrier schemes, incremental update appears to be more effective than snapshot approaches—because most short-lived objects are reclaimed quickly—but with an extra cost in traversing newly-allocated live objects. This cost might be reduced by carefully choosing the ordering of root traversal, traversing the most stable structures first to avoid having the collector's work undone by mutator changes.

Careful attention should be paid to write barrier implementation. Boehm, Demers and Shenker's [BDS91, Boe91] incremental update algorithm uses virtual memory dirty bits as a coarse pagewise write barrier. All black objects in a page must be re-scanned if the page is dirtied again before the end of a collection. (As with Appel, Ellis and Li's copy collector, this coarseness sacrifices real-time guarantees, while supporting parallelism. It also allows the use of off-the-shelf compilers that don't emit write barrier instructions along with heap writes.)

In a system with compiler support for garbage collection, a list of stored-into locations can be kept, or dirty bits can maintained (in software) for small areas of memory, to reduce scanning costs and bound the time spent updating the marking traversal. This has been done for other reasons in generational garbage collectors, as we will discuss in Sect. 4.

4 Generational Garbage Collection

Given a realistic amount of memory, efficiency of simple copying garbage collection is limited by the fact that the system must copy all live data at a collection. In most programs in a variety of languages, *most objects live a very short time, while a small percentage of them live much longer* [LH83, Ung84, Sha88, Zor90, DeT90, Hay91]. While figures vary from language to language and program to program, usually between 80 and 98 percent of all newly-allocated objects die within a few million instructions, or before another megabyte has been allocated; the majority of objects die even more quickly, within tens of kilobytes of allocation.

(Heap allocation is often used as a measure of program execution, rather than wall clock time, for two reasons. One is that it's independent of machine and im-

[18] Ralph Johnson has improved on this scheme by incorporating lazier copying of objects to fromspace [Joh92]. This decreases the maximum latency, but in the (very unlikely) worst case a page may still be scanned at each pointer traversal until a whole garbage collection has been done "the hard way".

plementation speed—it varies appropriately with the speed at which the program executes, which wall clock time does not; this avoids the need to continually cite hardware speeds.[19] It is also appropriate to speak in terms of amounts allocated for garbage collection studies because the time between garbage collections is largely determined by the amount of memory available.[20] Future improvements in compiler technology may reduce rates of heap allocation by putting more "heap" objects on the stack; this is not yet much of a problem for experimental studies, because most current state-of-the-art compilers don't do much of this kind of lifetime analysis.)

Even if garbage collections are fairly close together, separated by only a few kilobytes of allocation, most objects die before a collection and never need to be copied. Of the ones that do survive to be copied once, however, *a large fraction survive through many collections.* These objects are copied at every scavenge, over and over, and the garbage collector spends most of its time copying the same old objects repeatedly. This is the major source of inefficiency in simple garbage collectors.

Generational collection [LH83] avoids much of this repeated copying by segregating objects into multiple areas by age, and scavenging areas containing older objects less often than the younger ones. Once objects have survived a small number of scavenges, they are moved to a less frequently scavenged area. Areas containing younger objects are scavenged quite frequently, because most objects there will generally die quickly, freeing up space; copying the few that survive doesn't cost much. These survivors are *advanced* to older status after a few scavenges, to keep copying costs down.

(For historical reasons and simplicity of explanation, we will focus on generational copying collectors. The choice of copying or marking collection is essentially orthogonal to the issue of generational collection, however [DWH+90].)

4.1 Multiple Subheaps with Varying Scavenge Frequencies

Consider a generational garbage collector based on the semispace organization: memory is divided into areas that will hold objects of different approximate ages, or *generations*; each generation's memory is further divided into semispaces. In Fig. 9 we show a simple generational scheme with just two age groups, a New generation and an Old generation. Objects are allocated in the New generation, until its current semispace is full. Then the New generation (only) is scavenged, copying its live data into the other semispace, as shown in Fig. 10.

If an object survives long enough to be considered old, it can be copied out of the new generation and into the old, rather than back into the other semispace. This removes it from consideration by single-generation scavenges, so that it is no longer copied at every scavenge. Since relatively few objects live this long, old memory will

[19] One must be careful, however, not to interpret it as the ideal abstract measure. For example, rates of heap allocation are somewhat higher in Lisp and Smalltalk, because more control information and/or intermediate data of computations may be passed as pointers to heap objects, rather than as structures on the stack.

[20] Allocation-relative measures are still not the absolute bottom-line measure of garbage collector efficiency, though, because decreasing work per unit of allocation is not nearly as important if programs don't allocate much; conversely, smaller percentage changes in garbage collection work mean more for programs whose memory demands are higher.

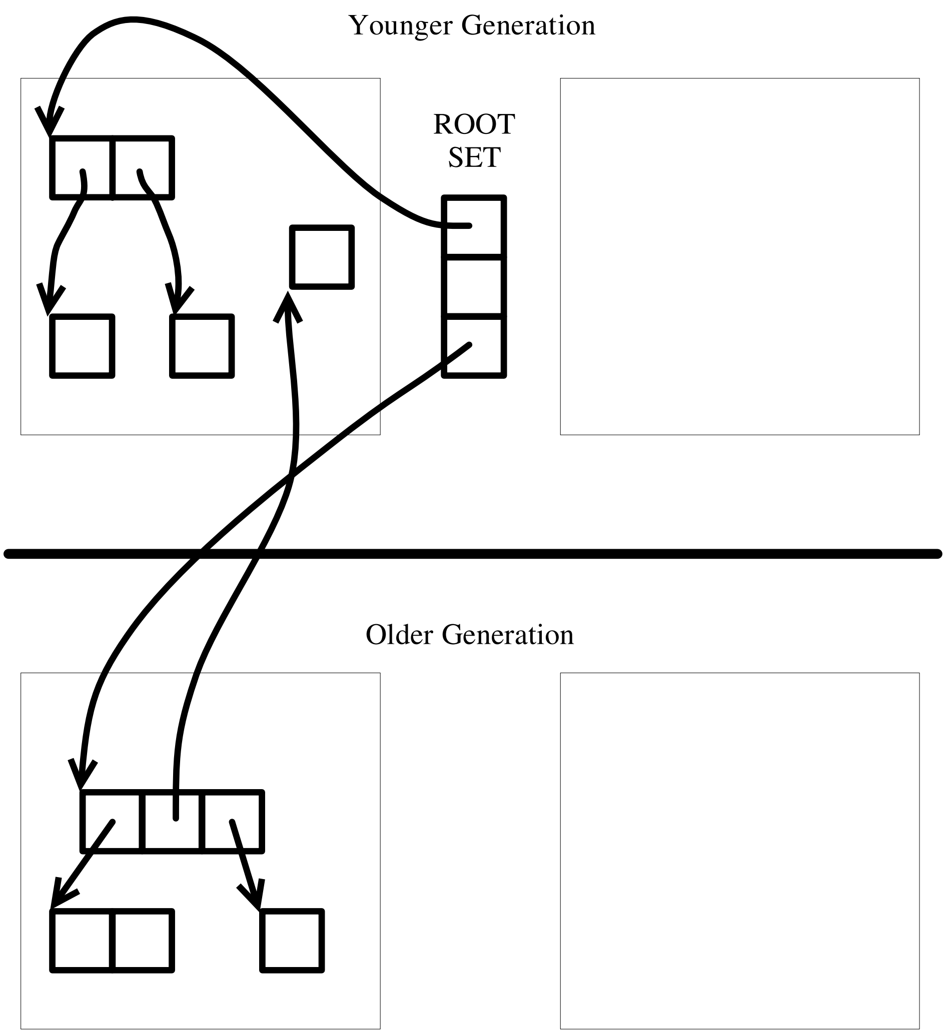

Fig. 9. A generational copying garbage collector before garbage collection.

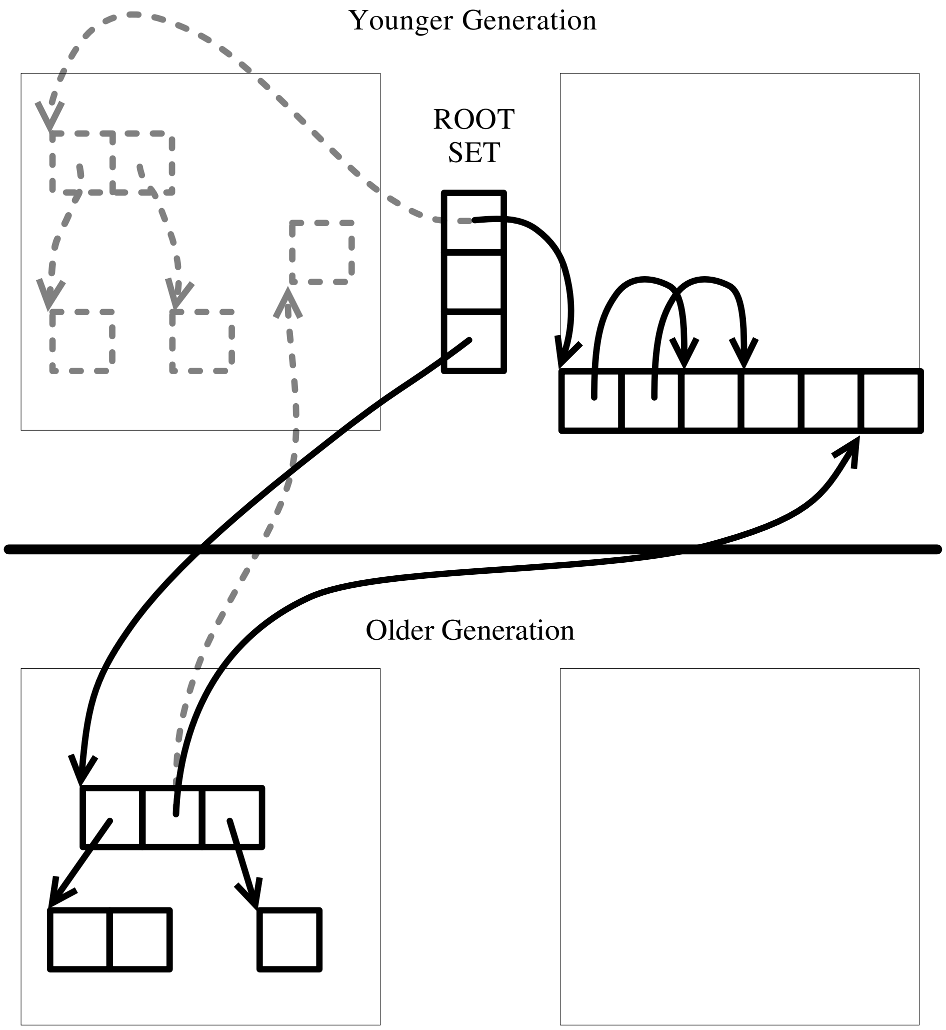

Fig. 10. Generational collector after garbage collection.

fill much more slowly than new. Eventually, old memory will fill up and have to be garbage collected as well. Figure 11 shows the general pattern of memory use in this simple generational scheme. (Note the figure is not to scale—the younger generation is typically several times smaller than the older one.)

The number of generations may be greater than two, with each successive generation holding older objects and being scavenged considerably less often. (Tektronix 4406 Smalltalk is such a generational system, using semispaces for each of eight generations [CWB86].)

4.2 Detecting Intergenerational References

In order for this scheme to work, it must be possible to scavenge the younger generation(s) without scavenging the older one(s). Since liveness of data is a global property, however, old-memory data must be taken into account. For example, if there is a pointer from old memory to new memory, that pointer must be found at scavenge time and used as one of the roots of the traversal. (Otherwise, an object that is live may not be preserved by the garbage collector, or the pointer may simply not be updated appropriately when the object is moved. Either event destroys the integrity and consistency of data structures in the heap.)

In the original generational collection scheme [LH83] scheme, no pointer in old memory may point directly to an object in new memory; instead it must point to a cell in an indirection table, which is used as part of the root set. Such indirections are transparent to the user program. This technique was implemented on Lisp machines such as the MIT machines [Gre84] and Texas Instruments Explorer [Cou88]. (There are minor differences between the two, but the principles are the same.[21])

Note that other techniques are often more appropriate, especially on stock hardware. Using indirection tables introduces overhead similar to that of Baker's read barrier. A *pointer recording* technique can be used instead. Rather than indirecting pointers from old objects to young ones, normal (direct) pointers are allowed, but the locations of such pointers are noted so that they can be found at scavenge time. This requires something like a write barrier [Ung84, Moo84]; that is, the running program cannot freely modify the reachability graph by storing pointers into objects in older generation.

The write barrier may do checking at each store, or it may be as simple as maintaining dirty bits and scanning dirty areas at collection time [Sha88, Sob88, WM89, Wil90, HMS92].[22]; the same mechanism might support real-time incremental collection as well.

[21] The main difference is that the original scheme used per-generation *entry* tables, indirecting and isolating the pointers into a generation. The Explorer used *exit* tables, indirecting the pointers *out of* each generation; for each generation, there is a separate exit table for pointers into *each* younger generation[Cou88].

[22] Ungar and Chambers' improvement [Cha92], of our "card marking" scheme [WM89, Wil90] decreases the cost per heap write by using whole bytes as dirty bits. Given the byte write instructions available on common architectures, the overhead is only three instructions per potential pointer store, at an increase in bitmap size and per-garbage collection scanning cost.

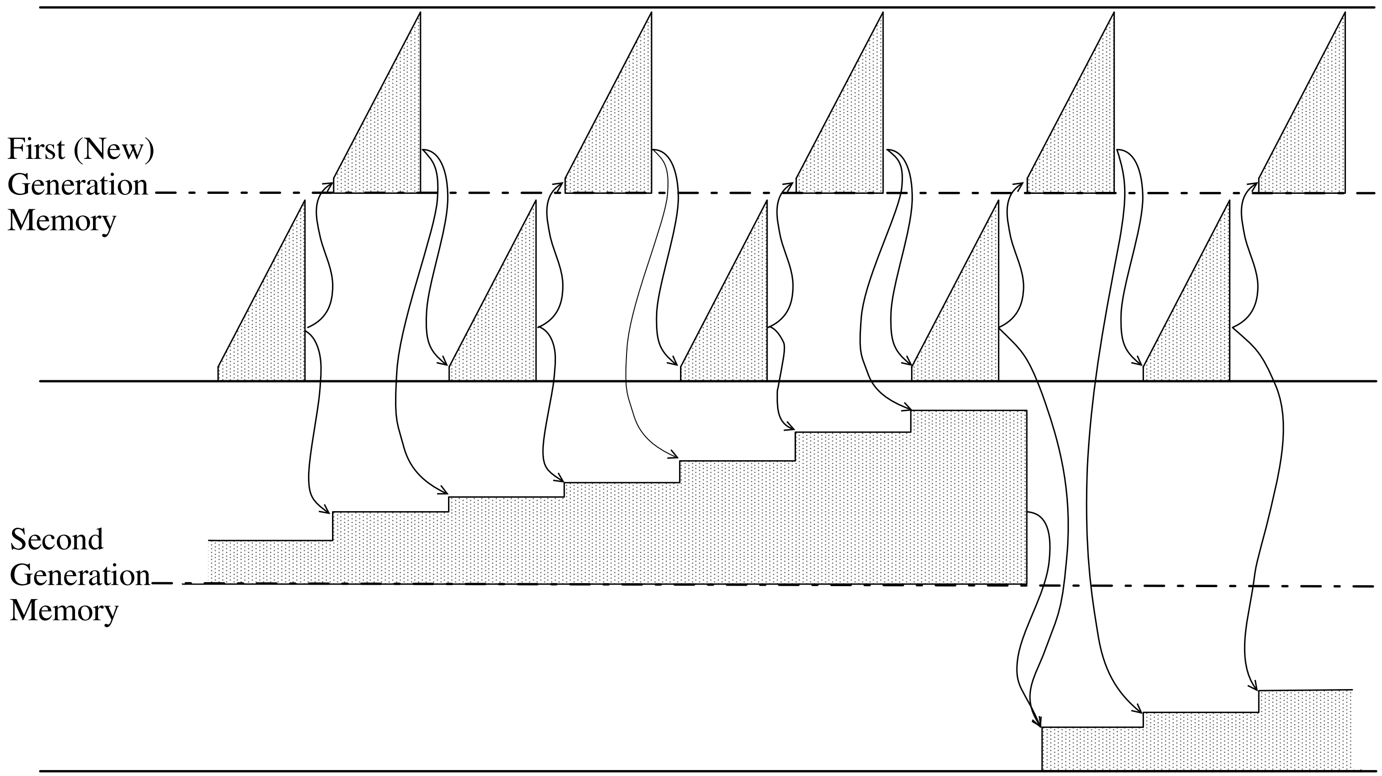

Fig. 11. Memory use in a generational copy collector with semispaces for each generation.

The important point is that all references from old to new memory must be located at scavenge time, and used as roots for the copying traversal.

Using these intergenerational pointers as roots ensures that all reachable objects in the younger generation are actually reached by the collector; in the case of a copy collector, it ensures that all pointers to moved objects are appropriately updated.

As in an incremental collector, this use of a write barrier results in a *conservative approximation* of true liveness; any pointers from old to new memory are used as roots, but not all of these roots are necessarily live themselves. An object in old memory may already have died, but that fact is unknown until the next time old memory is scavenged. Thus some garbage objects may be preserved because they are referred to from objects that are floating (undetected) garbage. This appears not to be a problem in practice [Ung84, UJ88].

It would also be possible to track all pointers from new memory into old memory, allowing old memory to be scavenged independently of new memory. This is more costly, however, because there are typically many more pointers from new to old than from old to new. This is a consequence of the way references are typically created—by creating a new object that refers to other objects which already exist. Sometimes a pointer to a new object is installed in an old object, but this is considerably less common. This asymmetrical treatment allows allows object-creating code (like Lisp's frequently-used **cons** operation) to skip the recording of intergenerational pointers. Only non-initializing stores into objects must be checked for intergenerational references; writes that initialize objects in the youngest generation can't create pointers into younger ones.

Even if new-to-old pointers are not recorded, it may still be feasible to scavenge a generation without scavenging newer ones. In this case, *all* data in the newer generations may be considered possible roots, and they may simply be scanned for pointers [LH83]. While this scanning consumes time proportional to the amount of data in the newer generations, each generation is usually considerably smaller than the next, and the cost may be small relative to the cost of actually scavenging the older generation. (Scanning the data in the newer generation may be preferable to scavenging both generations, because scanning is generally faster than copying; it may also have better locality.)

The cost of recording intergenerational pointers is typically proportional to the rate of program execution i.e., it's not particularly tied to the rate of object creation. For some programs, it may be the major cost of garbage collection, because several instructions must be executed for every potential pointer store into the heap. This may slow program execution down by several percent. (It is interesting to note that this pointer recording is essentially the same as that required for a write barrier incremental scheme; the same cost may serve both purposes.)

Within the framework of the generational strategy we've outlined, several important questions remain:

1. *Advancement policy.* How long must an object survive in one generation before it is advanced to the next? [Ung84, WM89]
2. *Heap organization.* How should storage space be divided and used between generations, and within a generation [Moo84, Ung84, Sha88, WM89]? How does the resulting reuse pattern affect locality at the virtual memory level [Ung84, Zor89,

WM89], and at the level of high-speed cache memories [Zor91, WLM92]?

3. *Traversal algorithms.* In a tracing collector, the traversal of live objects may have an important impact on locality. In a copying collector, objects are also reordered in memory as they are reached by the copy collector. What affect does this have on locality, and what traversal yields the best results [Bla83, Sta84, And86, WLM91]?
4. *Collection scheduling.* For a non-incremental collector, how might we avoid or mitigate the effect of disruptive pauses, especially in interactive applications [Ung84, WM89]? Can we improve efficiency by careful "opportunistic"scheduling [WM89, Hay91]? Can this be adapted to incremental schemes to reduce floating garbage?
5. *Intergenerational references.* Since it must be possible to scavenge younger generations without scavenging the older ones, we must be able to find the live pointers from older generations into the ones we're scavenging. What is the best way to do this [WM89, BDS91, App89b, Wil90]?

5 Conclusions

Recent advances in garbage collection technology make automatic storage reclamation affordable for use in high-performance systems. Even relatively simple garbage collectors' performance is often competitive with conventional explicit storage management [App87, Zor92]. Generational techniques reduce the basic costs and disruptiveness of collection by exploiting the empirically observed tendency of objects to die young; stock hardware incremental techniques may even make this relatively inexpensive for hard real-time systems.

We have discussed the basic operation of several kinds of garbage collectors, to provide a framework for understanding current research in the field. A key point is that standard textbook analyses of garbage collection algorithms usually miss the most important characteristics of collectors—namely, the constant factors associated with the various costs, including locality effects. These factors require garbage collection designers to take detailed implementation issues into account, and be very careful in their choices of features.

Features also interact in important ways. Fine-grained incremental collection is unnecessary in most systems without hard real-time constraints. Coarser incremental techniques may be sufficient, because the modest pause times are acceptable [AEL88, BDS91], and the usually-short pauses of a stop-and-collect generational system may be acceptable enough for many systems [Ung84, WM89]. (On the other hand, the write barrier support for generational garbage collection could also support an incremental update scheme for incremental collection; if this recording is cheap and precise enough, it might support fine-grained real-time collection at little cost.)

In this introductory survey, we have not addressed the increasingly important areas of parallel [Ste75, KS77, DLM+78, NR87, AEL88, SS91] and distributed [LQP92, RMA92, JJ92, PS92] collection; we have also given insufficient coverage of conservative collectors, which can be used with systems not originally designed for garbage collection [BW88, Bar88, Ede90, Wen90, WH91]. These developments have considerable promise for making garbage collection widely available and practical; we hope that we've laid a proper foundation for discussing them, by clarifying the basic issues.

Acknowledgments

I am grateful to innumerable people for enlightening discussions of heap management over the last few years, including David Ungar, Eliot Moss, Henry Baker, Andrew Appel, Urs Hoelzle, Mike Lam, Tom Moher, Henry Lieberman, Patrick Sobalvarro, Doug Johnson, Bob Courts, Ben Zorn, Mark Johnstone and David Chase. Special thanks to Hans Boehm, Joel Bartlett, David Moon, Barry Hayes, and especially to Janet Swisher for help in the preparation of this paper.

References

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 11–20. SIGPLAN ACM Press, June 1988. Atlanta, Georgia.

[AL91] Andrew W. Appel and Kai Li. Virtual memory primitives for user programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, April 1991. Santa Clara, CA.

[And86] David L. Andre. Paging in Lisp programs. Master's thesis, University of Maryland, College Park, Maryland, 1986.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[App89a] Andrew W. Appel. Runtime tags aren't necessary. *Lisp and Symbolic Computation*, 2:153–162, 1989.

[App89b] Andrew W. Appel. Simple generational garbage collection and fast allocation. *Software Practice and Experience*, 19(2):171–183, February 1989.

[App91] Andrew W. Appel. Garbage collection. In Peter Lee, editor, *Topics in Advanced Language Implementation*, pages 89–100. MIT Press, Cambridge, Massachusetts, 1991.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bak92] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. *ACM SIGPLAN Notices*, 27(3):66–70, March 1992.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation Western Research Laboratory, Palo Alto, California, February 1988.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157–164, June 1991. Toronto, Ontario, Canada.

[Bla83] Ricki Blau. Paging on an object-oriented personal computer for Smalltalk. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, August 1983. Minneapolis, MN. Also appears as Technical Report UCB/CSD 83/125, University of California at Berkeley, Computer Science Division (EECS), Berkeley, California, August 1983.

[Bob80] Daniel G. Bobrow. Managing reentrant structures using reference counts. *ACM Transactions on Programming Languages and Systems*, 2(3):269–273, July 1980.

[Boe91] Hans-Juergen Boehm. Hardware and operating system support for conservative garbage collection. In *International Workshop on Memory Management*, pages 61–67, Palo Alto, California, October 1991. IEEE Press.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 108–113, August 1984. Austin, Texas.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software Practice and Experience*, 18(9):807–820, September 1988.

[CG77] Douglas W. Clark and C. Cordell Green. An empirical study of list structure in LISP. *Communications of the ACM*, 20(2):78–87, February 1977.

[Cha92] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for an Object-Oriented Programming Language*. PhD thesis, Stanford University, March 1992.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[Cla79] Douglas W. Clark. Measurements of dynamic list structure use in Lisp. *IEEE Transactions on Software Engineering*, 5(1):51–59, January 1979.

[CN83] Jacques Cohen and Alexandru Nicolau. Comparison of compacting algorithms for garbage collection. *ACM Transactions on Programming Languages and Systems*, 5(4):532–553, October 1983.

[Coh81] Jacques Cohen. Garbage collection of linked data structures. *Computing Surveys*, 13(3):341–367, September 1981.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 2(12):655–657, December 1960.

[Cou88] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[CWB86] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In Norman Meyrowitz, editor, *ACM SIGPLAN 1986 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '86)*, pages 119–130, September 1986. Also published as *ACM SIGPLAN Notices 21*(11):119-130, November, 1986.

[Daw82] Jeffrey L. Dawson. Improved effectiveness from a real-time LISP garbage collector. In *SIGPLAN Symposium on LISP and Functional Programming*, pages 159–167, August 1982.

[DB76] L. Peter Deutsch and Daniel G. Bobrow. An efficient, incremental, automatic garbage collector. *Communications of the ACM*, 19(9):522–526, September 1976.

[DeT90] John DeTreville. Experience with concurrent garbage collectors for Modula-2+. Technical Report 64, Digital Equipment Corporation Systems Research Center, Palo Alto, California, August 1990.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[DMH92] Amer Diwan, Eliot Moss, and Richard Hudson. Compiler support for garbage collection in a statically-typed language. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992.

[DWH+90] Alan Demers, Mark Weiser, Barry Hayes, Daniel Bobrow, and Scott Shenker. Combining generational and conservative garbage collection: Framework and implementations. In *Conf. Record of the Seventeeth Annual ACM Symposium on Principles of Programming Languages*, pages 261–269, January 1990. Las Vegas, Nevada.

[Ede90] Daniel Ross Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A LISP garbage-collector for virtual-memory computer systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly-typed programming languages. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 165–176, June 1991. Toronto, Ontario, Canada.

[Gre84] Richard Greenblatt. The LISP machine. In D.R. Barstow, H.E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*. McGraw Hill, 1984.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 33–46, Phoenix, Arizona, October 1991. ACM Press.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In *ACM SIGPLAN 1992 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, Vancouver, British Columbia, Canada, October 1992. To appear.

[JJ92] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science series.

[Joh91] Douglas Johnson. The case for a read barrier. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 96–107, Santa Clara, California, April 1991.

[Joh92] Ralph E. Johnson. Reducing the latency of a real-time garbage collector. *ACM Letters on Programming Languages and Systems*, 1(1):46–58, March 1992.

[Knu69] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, chapter 2.3.5, pages 406–422. Addison-Wesley, Reading, Massachusetts, 1969.

[KS77] H.T. Kung and S.W. Song. An efficient parallel garbage collection system and its correctness proof. In *IEEE Symposum on Foundations of Computer Science*, pages 120–131, Providence, Rhode Island, October 1977.

[Lar77] R. G. Larson. Minimizing garbage collection as a function of region size. *SIAM Journal on Computing*, 6(4):663–667, December 1977.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[LQP92] Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *ACM Symposium on Principles of Programming*, pages 39–50, Albuquerque, New Mexico, January 1992.

[McB63] J. Harold McBeth. On the reference counter method. *Communications of the ACM*, 6(9):575, September 1963.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Communications of the ACM*, 3(4):184–195, April 1960.

[Min63] Marvin Minsky. A LISP garbage collector algorithm using serial secondary storage. A.I Memo 58, Project MAC, MIT, Cambridge, Massachusetts, 1963.

[Moo84] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 235–246, Austin, Texas, August 1984.

[Nil88] Kelvin Nilsen. Garbage collection of strings and linked data structures in real time. *Software, Practice and Experience*, 18(7):613–640, July 1988.

[NR87] S. C. North and J. H. Reppy. *Concurrent Garbage Collection on Stock Hardware*, pages 113–133. Number 274 in Lecture Notes in Computer Science. Springer-Verlag, September 1987.

[PS92] David Plainfosse and Marc Shapiro. Experience with fault tolerant garbage collection in a distributed Lisp system. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science series.

[RMA92] G. Ringwood, E. Miranda, and S. Abdullahi. Distributed garbage collection. In *International Workshop on Memory Management*, St. Malo, France, September 1992. Springer-Verlag Lecture Notes in Computer Science series.

[Rov85] Paul Rovner. On adding garbage collection and runtime types to a strongly-typed, statically checked, concurrent language. Technical Report CSL-84-7, Xerox Palo Alto Research Center, Palo Alto, California, July 1985.

[Sha88] Robert A. Shaw. *Empirical Analysis of a Lisp System.* PhD thesis, Stanford University, Stanford, California, February 1988. Also appears as Technical Report CSL-TR-88-351, Stanford University Computer Systems Laboratory, 1988.

[Sob88] Patrick G. Sobalvarro. A lifetime-based garbage collector for LISP systems on general-purpose computers. B.S. thesis, Massachusetts Institute of Technology, Electrical Engineering and Computer Science Department, Cambridge, Massachusetts, 1988.

[SS91] Ravi Sharma and Mary Lou Soffa. Parallel generational garbage collection. In *ACM SIGPLAN 1991 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, pages 16–32, Phoenix, Arizona, October 1991.

[Sta84] James William Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Transactions on Programming Languages and Systems*, 2(2):155–180, May 1984.

[Ste75] Guy L. Steele Jr. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In *ACM SIGPLAN 1988 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 1–17, San Diego, California, September 1988. ACM. Also published as *ACM SIGPLAN Notices 23*(11):1–17, November, 1988.

[Ung84] David M. Ungar. Generation scavenging: A non-disruptive high-performance storage reclamation algorithm. In *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, Pennsylvania, April 1984. Also distributed as *ACM SIGPLAN Notices 19*(5):157–167, May, 1987.

[Wen90] E.P. Wentworth. Pitfalls of conservative garbage collection. *Software, Practice and Experience*, 20(7):719–727, July 1990.

[WH91] Paul R. Wilson and Barry Hayes. The 1991 OOPSLA Workshop on Garbage Collection in Object Oriented Systems (organizers' report). In *Addendum to the proceedings of OOPSLA '91*, Phoenix, Arizona, 1991.

[Wil90] Paul R. Wilson. Some issues and strategies in heap management and memory hierarchies. In *OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems*, Ottawa, Ontario, Canada, October 1990. Also in *SIGPLAN Notices 23*(1):45–52, January 1991.

[Wil91] Paul R. Wilson. Operating system support for small objects. In *International Workshop on Object Orientation in Operating Systems*, Palo Alto, California, October 1991. IEEE Press. Revised version to appear in *Computing Systems*.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 177–191, Toronto, Canada, June 1991.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *SIGPLAN Symposium on LISP and Functional Programming*, San Francisco, California, June 1992.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the opportunistic garbage collector. In *ACM SIGPLAN 1989 Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89)*, pages 23–35, New Orleans, Louisiana, October 1989.

[Yua90] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11:181–198, 1990.

[Zor89] Benjamin Zorn. *Comparative Performance Evaluation of Garbage Collection Algorithms*. PhD thesis, University of California at Berkeley, Electrical Engineering and Computer Science Department, Berkeley, California, December 1989. Also appears as Technical Report UCB/CSD 89/544, University of California at Berkeley.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *1990 ACM Conference on Lisp and Functional Programming*, pages 87–98, Nice, France, June 1990.

[Zor91] Benjamin Zorn. The effect of garbage collection on cache performance. Technical Report CU-CS-528-91, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, May 1991.

[Zor92] Benjamin Zorn. The measured cost of conservative garbage collection. Technical report, University of Colorado at Boulder, Dept. of Computer Science, Boulder, Colorado, 1992.

This article was processed using the LaTeX macro package with LLNCS style