Uniprocessor Garbage Collection Techniques

[Submitted to ACM Computing Surveys]

Paul R. Wilson

Abstract

We survey basic garbage collection algorithms, and variations such as incremental and generational collection; we then discuss low-level implementation considerations and the relationships between storage management systems, languages, and compilers. Throughout, we attempt to present a unified view based on abstract traversal strategies, addressing issues of conservatism, opportunism, and immediacy of reclamation; we also point out a variety of implementation details that are likely to have a significant impact on performance.

Contents

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[1] 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 surveys basic and advanced techniques in uniprocessor garbage collectors, especially those developed in the last decade. (For a more thorough treatment of older techniques, see [Knu69, Coh81].) While it does not cover parallel or distributed collection, it presents a unified taxonomy of incremental techniques, which lays the groundwork for understanding parallel and distributed collection. Our focus is on garbage collection for procedural and object-oriented languages, but much of the information here will serve as background for understanding garbage collection of other kinds of systems, such as functional or logic programming languages. (For further reading on various advanced topics in garbage collection, the papers collected in [BC92] are a good starting point.[3])

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A software routine operating on a data structure should not have to depend what

[1] We use the term "heap" in the simple sense of a storage management technique which allows any dynamically allocated object to be freed at any time—this is not to be confused with heap data structures which maintain ordering constraints.

[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.

[3] There is also a repository of papers in PostScript format available for anonymous Internet FTP from our FTP host (cs.utexas.edu:pub/garbage). Among other things, this repository contains collected papers from several garbage collection workshops held in conjunction with ACM OOPSLA conferences.

other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be 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 locally understandable and flexibly composable. This bookkeeping inhibits abstraction and reduces extensibility, because when new functionality is implemented, the bookkeeping code must be updated. The runtime cost of the bookkeeping itself may be significant, and in some cases it may introduce the need for additional synchronization in concurrent applications.

The unnecessary complications and subtle interactions created by explicit storage allocation are especially troublesome because programming mistakes often break 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 programming errors 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, after delivery, 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 the available memory and crash.[4]

Recently, tools have become available to help programmers find the source of leaked objects in languages with explicit deallocation [HJ92], and these can be extremely valuable. Unfortunately, these tools only find actual leaks during particular program runs, not possible leaks due to uncommon execution patterns. Finding the source of a leaked object does not always solve the problem, either: the programmer still must be able to determine a point where the object *should* be deallocated—if one exists. If one doesn't exist, the program must be restructured. (This kind of "garbage debugging" is better than nothing, but it is very fallible, and it must be repeated whenever programs change; it is desirable to actually eliminate leaks in general, rather than certain detectable leaks in particular.)

Explicit allocation and reclamation lead to program errors in more subtle ways as well. It is common for programmers to allocate a moderate number of objects statically, 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 programs, making them fail when those limitations are exceeded, possibly years later when computer 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. Because they are coded up for a one-shot application, these collectors are often both incomplete and buggy. 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.

It is widely believed that garbage collection is quite expensive relative to explicit heap management, but several studies have shown that garbage collection is sometimes cheaper [App87] than explicit deallocation, and is usually competitive with it [Zor93]. As we will explain later, a well-implemented garbage collector

[4] Long-running server programs are also especially vulnerable to leaks due to exception handling. Exception handling code may fail to deallocate all of the objects allocated by an aborted operation, and these occasional failures may cause a leak that is extremely hard to diagnose.

should slow running programs down by (very roughly) 10 percent, relative to explicit heap deallocation, for a high-performance system.[5] A significant number of programmers regard such a cost as unacceptable, but many others believe it to be a small price for the benefits in convenience, development time, and reliability.

Reliable cost comparisons are difficult, however, partly because the use of explicit deallocation affects the structure of programs in ways that may themselves be expensive, either directly or by their impact on the software development process.

For example, explicit heap management often motivates extra copying of objects so that deallocation decisions can be made locally—i.e., each module makes its own copy of a piece of information, and can deallocate it when it is finished with it. This not only incurs extra heap allocation, but undermines an object-oriented design strategy, where the identities of objects may be as important as the values they store. (The efficiency cost of this extra copying is hard to measure, because you can't fairly compare the same program with and without garbage collection; the program would have been written differently if garbage collection were assumed.)

In the long run, poor program structure may incur extra development and maintenance costs, and may cause programmer time to be spent on maintaining inelegant code rather than optimizing time-critical parts of applications; even if garbage collection costs more than explicit deallocation, the savings in human resources may pay for themselves in increased attention to other aspects of the system.[6]

For these reasons, garbage-collected languages have long been used for the programming of sophisticated algorithms using complex data structures. Many garbage-collected languages (such as Lisp and Prolog) were originally popular for artificial intelligence programming, but have been found useful for general-purpose programming. Functional and logic programming languages generally incorporate garbage collection, because their unpredictable execution patterns make it especially difficult to explicitly program storage deallocation. The influential object-oriented programming language Smalltalk incorporates garbage collection; more recently, garbage collection has been incorporated into many general-purpose languages (such as Eiffel, Self and Dylan), including those designed in part for low-level systems programming (such as Modula-3 and Oberon). Several add-on packages also exist to retrofit C and C++ with garbage collection.

In the rest of this paper, we focus on garbage collectors that are built into a language implementation, or grafted onto a language by importing routines from a library. The usual arrangement is that the heap allocation routines perform special actions to reclaim space, as necessary, when a memory request is not easily satisfied. Explicit calls to the "deallocator" are unnecessary because calls to the collector are implicit in calls to the allocator—the allocator invokes the garbage collector as necessary to free up the space it needs.

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 compiler's code generator, to emit certain extra information at compile time, and perhaps execute different instruction sequences at run time [Boe91, WH91, BC91, 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 (*garbage detection*), and
2. Reclaiming the garbage objects' storage, so that the running program can use it (*garbage reclamation*).

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

[5] This is an estimate on our part, and in principle we think garbage collection performance could be somewhat better; in practice, it is sometimes worse. Reasons for (and limitations of) such an estimate will be discussed in Sect. 8. One practical problem is that state-of-the-art garbage collectors have not generally been available for most high-performance programming systems.

[6] For example, Rovner reports that an estimated 40% of developer effort in the Mesa system was spent dealing with difficult storage management issues [Rov85].

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 moment the garbage collector is invoked, the active variables are considered live. Typically, this includes statically-allocated global or module variables, as well as local variables in activation records on the activation stack(s), and any variables currently in registers. These variables form the *root set* for the traversal. 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.

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

In most of 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.) Such information can easily be generated by the compiler, which must have the information to generate correct code for references to objects' fields.

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. (This will be discussed in Sect 6.1.) 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.)

(Garbage collectors using *conservative pointer finding* [BW88] are usable with little or no cooperation from the compiler—not even the types of named variables—but we will defer discussion of these collectors until Sect 6.2.)

1.4 Overview of the Paper

The remainder of this paper will discuss basic and advanced topics in garbage collection.

The basic algorithms include reference counting, mark-sweep, mark-compact, copying, and non-copying implicit collection; these are discussed in Sect. 2.

Incremental techniques (Sect. 3) allow garbage collection to proceed piecemeal while applications are running. These techniques can reduce the disruptiveness of garbage collection, and may even provide *real-time* guarantees. They can also be generalized into concurrent collections, which proceed on another processor, in parallel with actual program execution.

Generational schemes (Sect. 4) improve efficiency and/or locality by garbage collecting a smaller area more often, while exploiting typical lifetime characteristics to avoid undue overhead from long-lived objects. Because most collections are of a small area, typical pause times are also short, and for many applications this is an acceptable alternative to incremental collection.

Section 5 discusses locality properties of garbage-collected systems, which are rather different from those of conventional systems. Section 6 explores low-level implementation considerations, such as object formats and compiler cooperation; Section 7 describes language-level constraints and features for garbage-collected systems. Section 9 presents the basic conclusions of the paper and sketches research issues in

garbage collection of parallel, distributed, and persistent systems.

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 two ways: by *reference counting*, or by *tracing*. (The general term "tracing," used to include both marking and copying techniques, is taken from [LD87].) Reference counting garbage collectors maintain counts of the number of pointers to each object, and this count is used as a local approximation of determining true liveness. Tracing collectors determine liveness more directly, by actually traversing the pointers that the program could traverse, to find all of the objects the program might reach. There are several varieties of tracing collection: mark-sweep, mark-compact, copying, and non-copying implicit reclamation.[7] Because each garbage detection scheme has a major influence on reclamation and on reuse techniques, we will introduce reclamation methods as we go.

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 pointed-to object's count is incremented. When an existing reference to an object is eliminated, the count is 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 cannot 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.

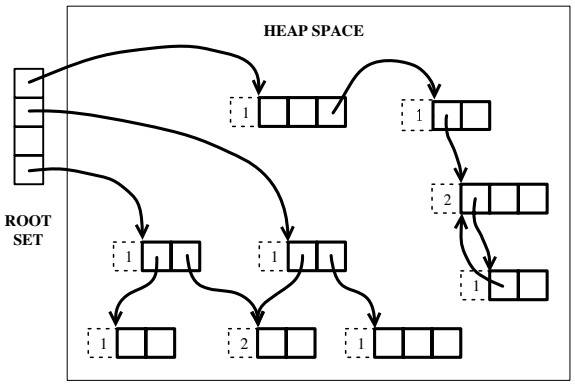

Figure 1: Reference counting.

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.

This incremental collection can easily satisfy "real time" requirements, guaranteeing that memory management operations never halt the executing program

[7] 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."

for more than a very brief period. This can support 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. (Subtleties of real-time requirements will be discussed in the context of tracing collection in Sect. 3.8.)

One minor problem with reference counting systems is that the reference counts themselves take up space. In some systems, a whole machine word is used for each object's reference count field, actually allowing it to represent any number of pointers that might actually exist in the whole system. In other systems, a shorter field is used, with a provision for overflow—if the reference count reaches the maximum that can be represented by the field size, its count is fixed at that maximum value, and the object cannot be reclaimed. Such objects (and other objects reachable from them) must be reclaimed by another mechanism, typically by a tracing collector that is run occasionally; as we will explain below, such a fall-back reclamation strategy is usually required anyway.

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

2.1.1 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 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, as well as when the application-domain semantics of data are most naturally expressed with cycles.

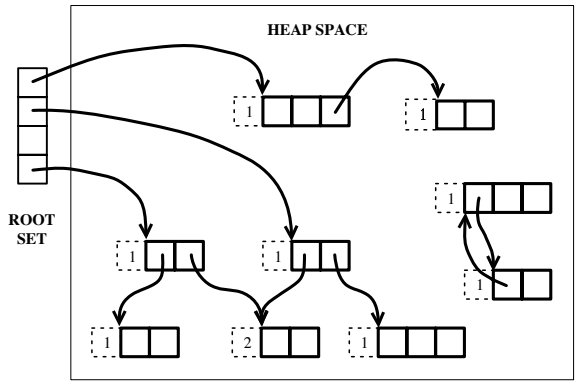

Figure 2: Reference counting with unreclaimable cycle.

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.[8] 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.

2.1.2 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

[8] [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.

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.

2.1.3 Deferred Reference Counting.

Much of this cost can be optimized away by special treatment of local variables [DB76, Bak93b]. 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 as well.

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.

2.1.4 Variations on Reference Counting

Another optimization of reference counting is to use a very small count field, perhaps only a single bit, to avoid the need for a large field per object [WF77]. Given that deferred reference counting avoids the need to continually represent the count of pointers from the stack, a single bit is sufficient for most objects; the minority of objects whose reference counts are not zero or one cannot be reclaimed by the reference counting system, but are caught by a fall-back tracing collector. A one-bit reference count can also be represented in each pointer to an object, if there is an unused address bit, rather than requiring a header field [SCN84].

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, from which the program's allocation requests are satisfied. (Other strategies will be discussed in the context of mark-sweep collection, in Sect. 2.2.) Objects' pointer fields must also be examined so that their referents can be freed.

It is difficult to make these reclamation operations take less than a few tens of 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. Still, reference counting has its advantages. The immediacy of reclamation can have advantages for overall memory usage and for locality of reference [DeT90]; 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.[9] It can also be useful for *finalization*, that is, performing "clean-up" actions (like closing files) when objects die [Rov85]; this will be discussed in Sect. 7.

The inability to reclaim cyclic structures is not a problem in some languages which do not allow the construction of cyclic data structures at all (e.g., purely functional languages). Similarly, the relatively high cost of side-effecting pointers between heap objects is not a problem in languages with few side-effects. Ref-

[9] As [WLM92] shows, generational techniques can recapture some of this locality, but not all of it.

erence 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 can benefit from the local nature of garbage collection, compared to global tracing. (In some configurations the cost of reference counting is only incurred for pointers to objects on other nodes; tracing collection is used within a node and to compute changes to reference counts between nodes.) Future systems may find other uses for reference counting, perhaps in hybrid collectors also involving other techniques, or when augmented by specialized hardware [PS89, Wis85, GC93] to keep CPU costs down.

While reference counting is out of vogue for high-performance implementations of general-purpose programming languages, it is quite common in other applications, where acyclic data structures are common. Most file systems use reference counting to manage files and/or disk blocks. Because of its simplicity, simple reference counting is often used in various software packages, including simple interpretive languages and graphical toolkits. Despite its weakness in the area of reclaiming cycles, reference counting is common even in systems where cycles may occur.

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.[10]

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.

[10] More detailed descriptions of traversal and marking algorithms can be found in [Knu69] and [Coh81].

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 (non-generational) 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. The clus-

tered 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.

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 copying 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 Daniel J. Edwards' *two-pointer algorithm*,[11] 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 holes to put them 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 [CN83].

[11] Described in an exercise on page 421 of [Knu69].

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 mark-compact 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.

2.4.1 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.[12]

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

[12] 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 on disk, 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 times slower than memory operations.

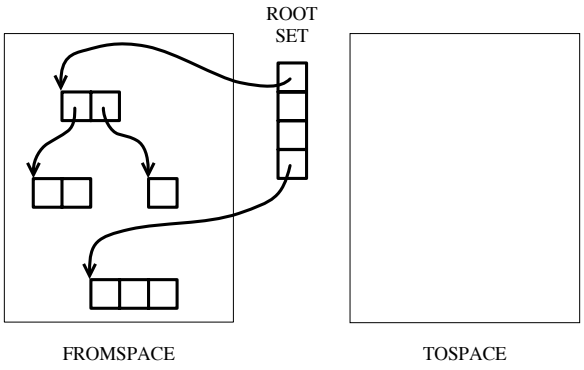

Figure 3: A simple semispace garbage collector before garbage collection.

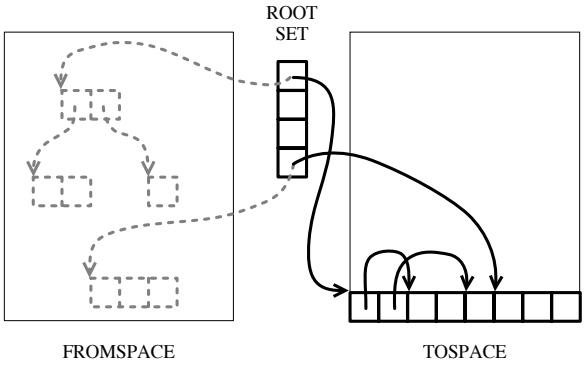

Figure 4: Semispace collector after garbage collection.

as demanded by the executing program. As with a mark-compact collector, the ability to allocate from a large, contiguous free space makes allocation simple and fast, much like allocating on a stack; 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.)

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 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 by which it was reached 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.

2.4.2 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

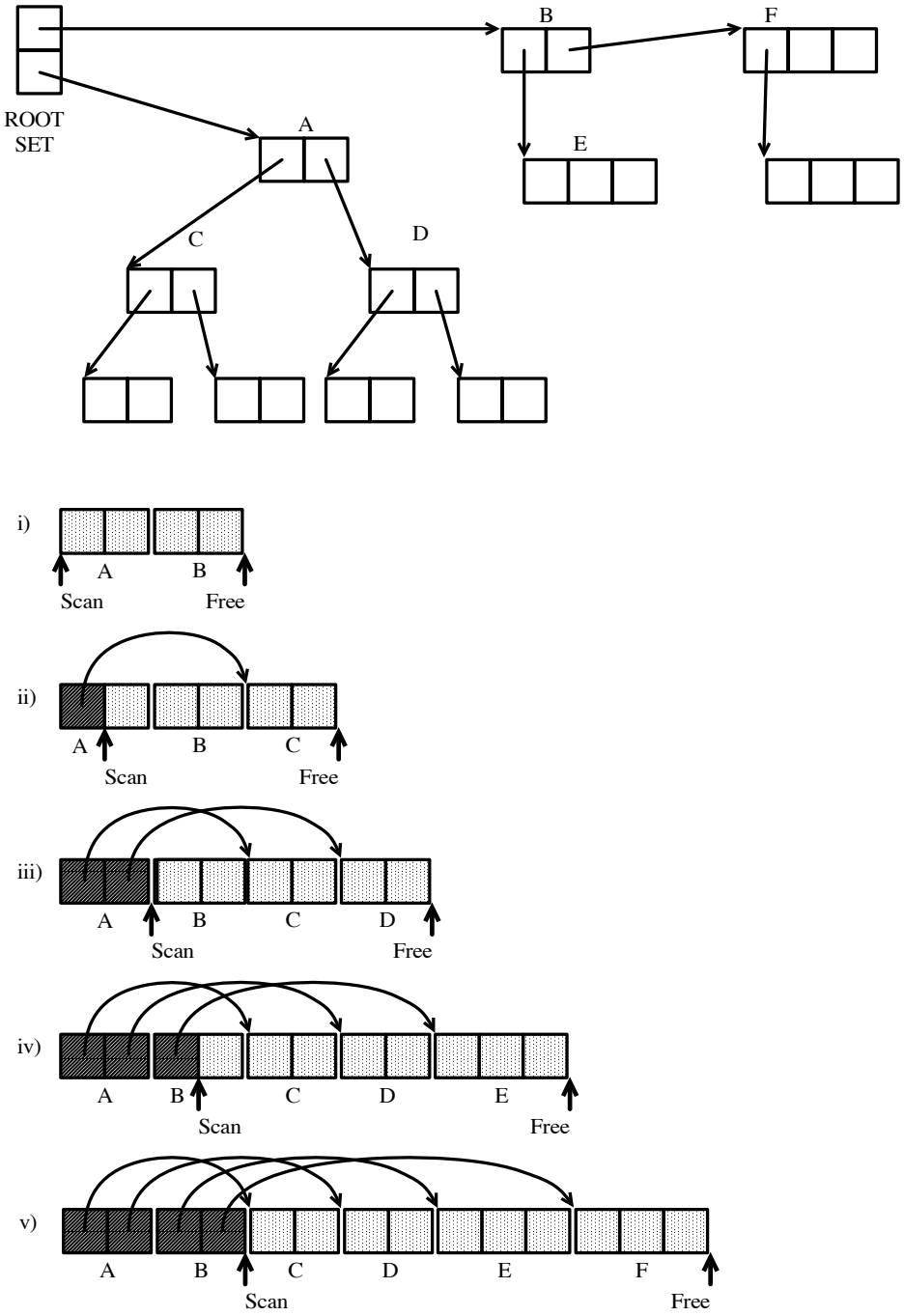

Figure 5: The Cheney algorithm's breadth-first copying.

of data is live at any given time during the program's execution, decreasing the frequency of garbage collections 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. (For the moment, we ignore virtual memory paging costs, assuming that the larger heap area can be cached in RAM rather than paged to disk. As we will explain in Sect. 2.7, paging costs may make the use of a larger heap area impractical if there is not a correspondingly large amount of RAM.)

2.5 Non-Copying Implicit Collection

Recently, Wang [Wan89] and Baker [Bak91b] have (independently) proposed a new kind of non-copying collector with some of the efficiency advantages of a copying scheme. Their insight is that in a copying collector, the "spaces" of the collector are really just a particular implementation of sets. The tracing process removes objects from the set subject to garbage collection, and when tracing is complete, anything remaining in the set is known to be garbage, so the set can be reclaimed in its entirety. 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. (In a copying collector, the set is an area of memory, but in a non-copying collector it can be any kind of set of chunks of memory that formerly held live objects.)

The 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. (Wang therefore refers to this as a "fake copying" collector.) Chunks of free space are initially linked to form a doubly-linked list, while chunks holding allocated objects are linked together into another list.

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 color field, and linking it into the toset's doubly-linked list.

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 3.4.2, 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 objects are all reclaimed in small constant time.

This scheme can be optimized in ways that are analogous to those used in a copying collector—allocation can be fast because the allocated and free lists can be contiguous, and separated only by an allocation pointer. Rather than actually unlinking objects from one list and linking them into another, the allocator can simply advance a pointer which points into the list and divides the allocated segment from the free segment. Similarly, a Cheney-style breadth-first traversal can be implemented with only a pair of pointers, and the scanned and free lists can be contiguous, so that advancing the scan pointer only requires advancing the pointer that separates them.

This scheme has both advantages and disadvantages

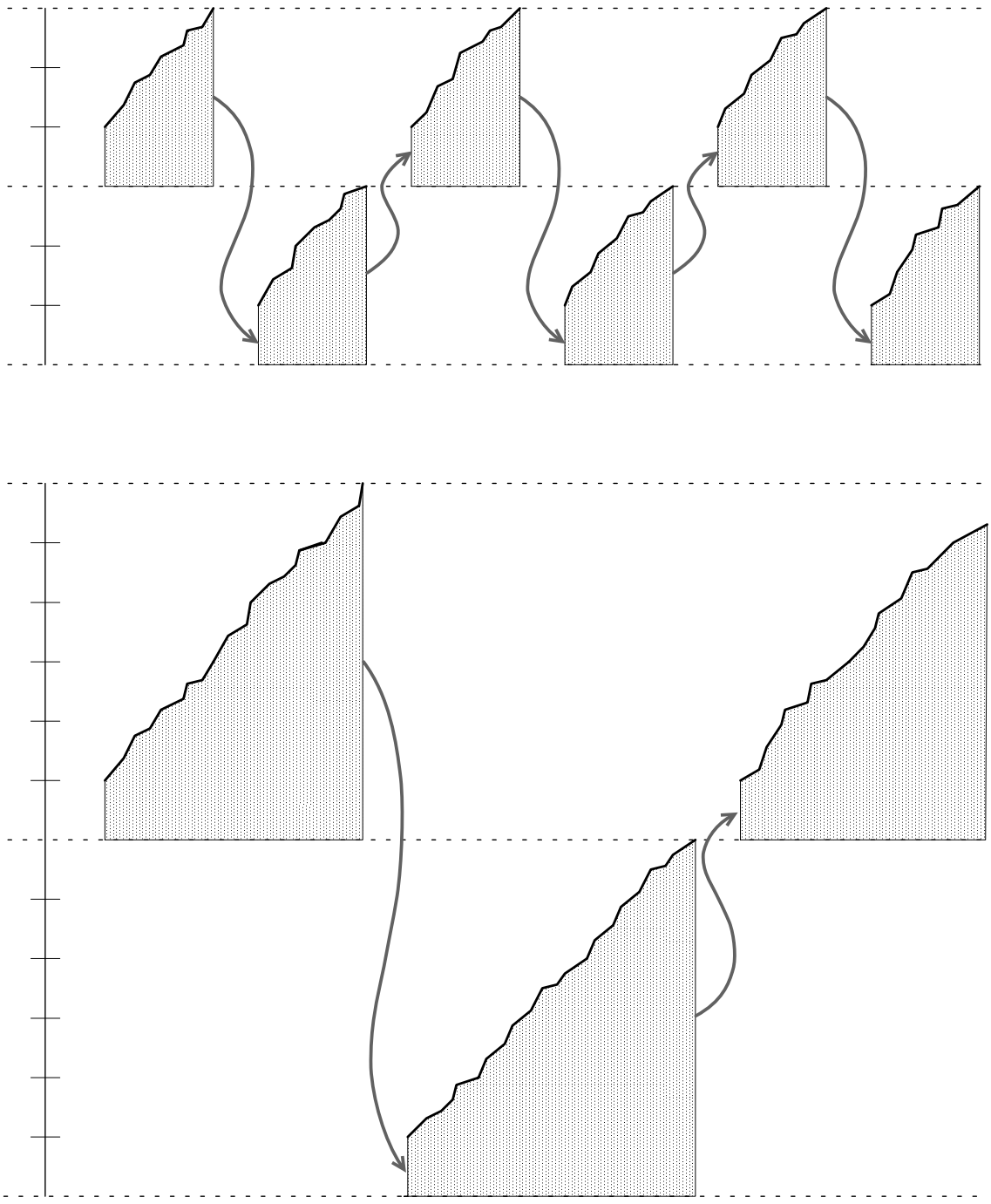

Figure 6: Memory usage in a semispace GC, with 3 MB (top) and 6 MB (bottom) semispaces

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.

The space costs of this technique are usually roughly comparable to those of a copying collector. Two pointer fields are required per object, but live objects being traced do not require space for both fromspace and tospace versions. In most cases, this appears to make the space cost smaller than that of a copying collector, but in some cases fragmentation costs (due to the inability to compact data) may outweigh those savings.

2.6 Choosing Among Basic Tracing 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. A common criterion for "high performance" garbage collection is that the cost of garbage collecting objects be comparable, on average, to the cost of allocating objects.

Any efficient tracing collection scheme 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 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 initial work is usually relatively fixed for a particular program, by the size of the root set. The work done at allocation is generally proportional to the number of objects allocated, plus an initialization cost proportional to their sizes. The garbage detection cost is proportional to the amount of live data that must be traced.

The latter two costs are usually similar, in that the amount of live data traced is usually some significant percentage of the amount of allocated memory. 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 is 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.

In systems where memory is not much larger than the expected amount of live data, nonmoving collectors have an an advantage over copying collectors in that they don't need space for two versions of each live object (the "from" and "to" versions). When space is very tight, reference counting collectors are particularly attractive because their performance is essentially independent of the ratio of live data to total storage.

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. Other hybrids may use non-copying techniques most of the time, but occasionally compact some of the data using copying techniques to avoid permanent fragmentation (e.g., [LD87]).

A major point in favor of in-place collectors is the ability to make them *conservative* with respect to data values that may or may not be pointers. 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. (Conservative pointer-finding techniques will be discussed in more detail in Sect. 6.2.)

Similarly, the choice of a non-moving collector can greatly simplify the interfaces between modules written in different languages and compiled using different compilers. It is possible to pass pointers to garbage-collectible objects as arguments to foreign routines that were not written or compiled with garbage collection in mind. This is not practical with a copying collector, because the pointers that "escape" into foreign routines would have to be found and updated when their referents moved.

2.7 Problems with Simple Tracing Collectors

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 partly 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* efficient 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.

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

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 typically swamped by the effects 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, 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.[13]

The only way to have good locality is to ensure that memory is large enough to hold the regularly-reused

[13] 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.

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 non-copying collectors are slightly harder to construct, but they do exist and are quite practical [DWH+90, WJ93].)

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.

2.8 Conservatism in Garbage Collection

An ideal garbage collector would be able to reclaim every object's space just after the last use of the object. Such an object is not implementable in practice, of course, because it cannot in general be determined when the last use occurs. Real garbage collectors can only provide a reasonable approximation of this behavior, using conservative approximations of this omniscience. The art of efficient garbage collector design is largely one of introducing small degrees of conservatism which significantly reduce the work done in detecting garbage. (This notion of conservatism is very general, and should not be confused with the specific pointer-identification techniques used by so-called "conservative" garbage collectors. All garbage collectors are conservative in one or more ways.)

The first conservative assumption most collectors make is that any variable in the stack, globals, or registers is live, even though the variable may actually never be referenced again. (There may be interactions between the compiler's optimizations and the garbage collector's view of the reachability graph. A compiler's data and control flow analysis may detect dead values and optimize them away entirely. Compiler optimizations may also extend the effective lifetime of variables, causing extra garbage to be retained, but this is not typically a problem in practice.)

Tracing collectors introduce a major *temporal* form of conservatism, simply by allowing garbage to go uncollected between collection cycles. Reference counting collectors are conservative topologically, failing to distinguish between different paths that share an edge in the graph of pointer relationships.

As the remainder of this survey will show, there are many possible kinds and degrees of conservatism with different performance tradeoffs.

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 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 much 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 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 tech-