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.

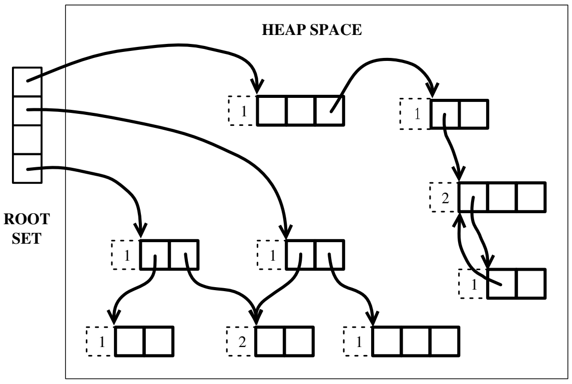

Figure 1: Reference counting.

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.

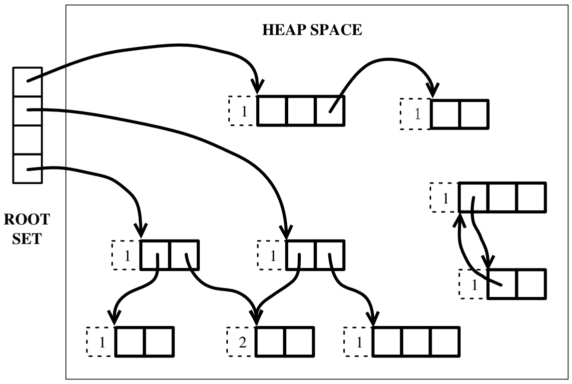

Figure 2: Reference counting with unreclaimable cycle.

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.

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-

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

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].

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

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

[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.

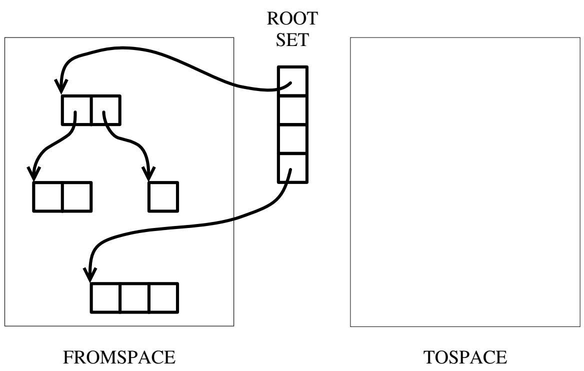

Figure 3: A simple semispace garbage collector before garbage collection.

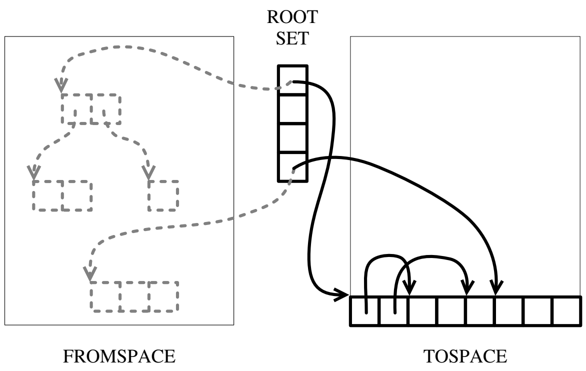

Figure 4: Semispace collector after garbage collection.

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

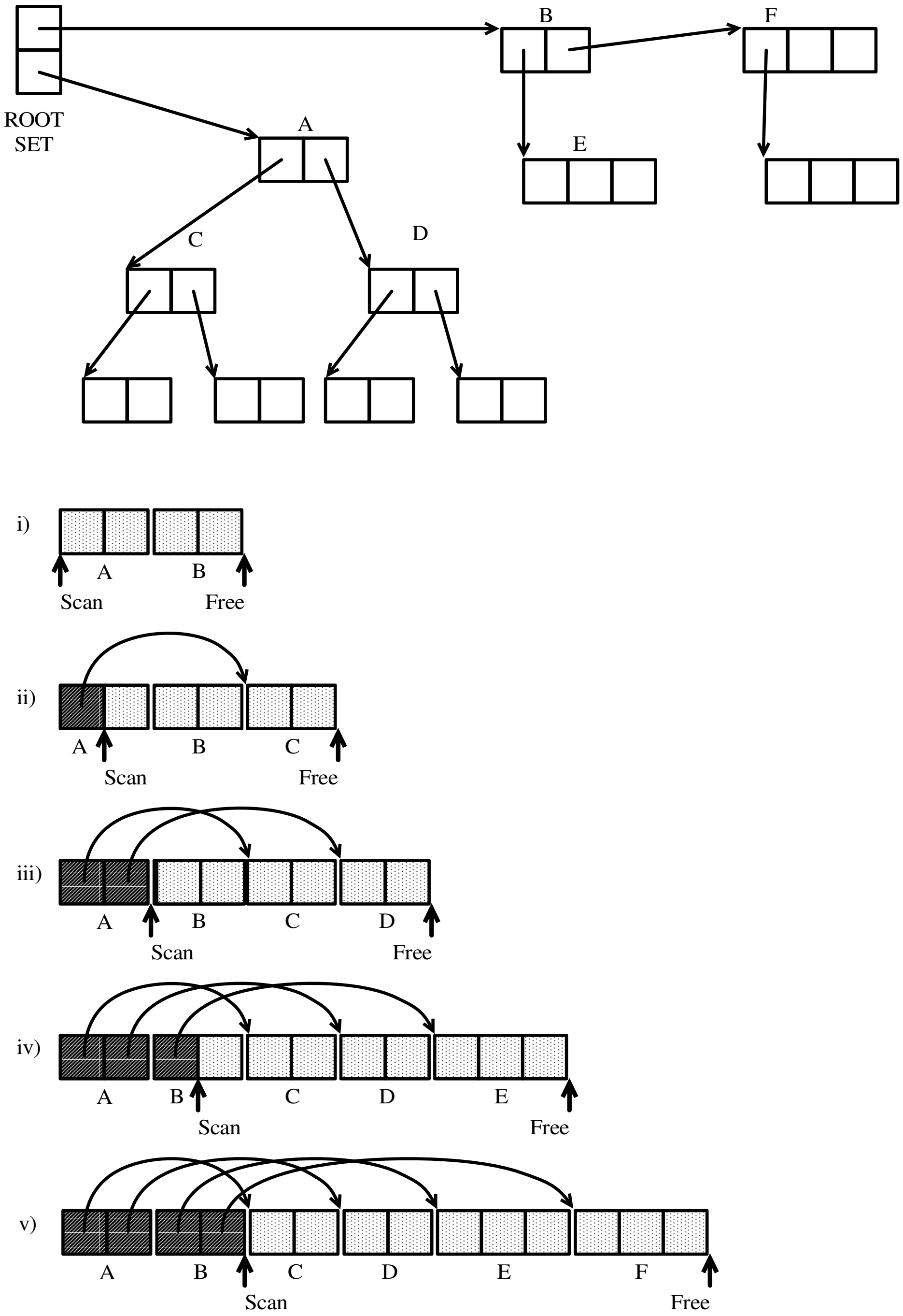

Figure 5: The Cheney algorithm's breadth-first copying.

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

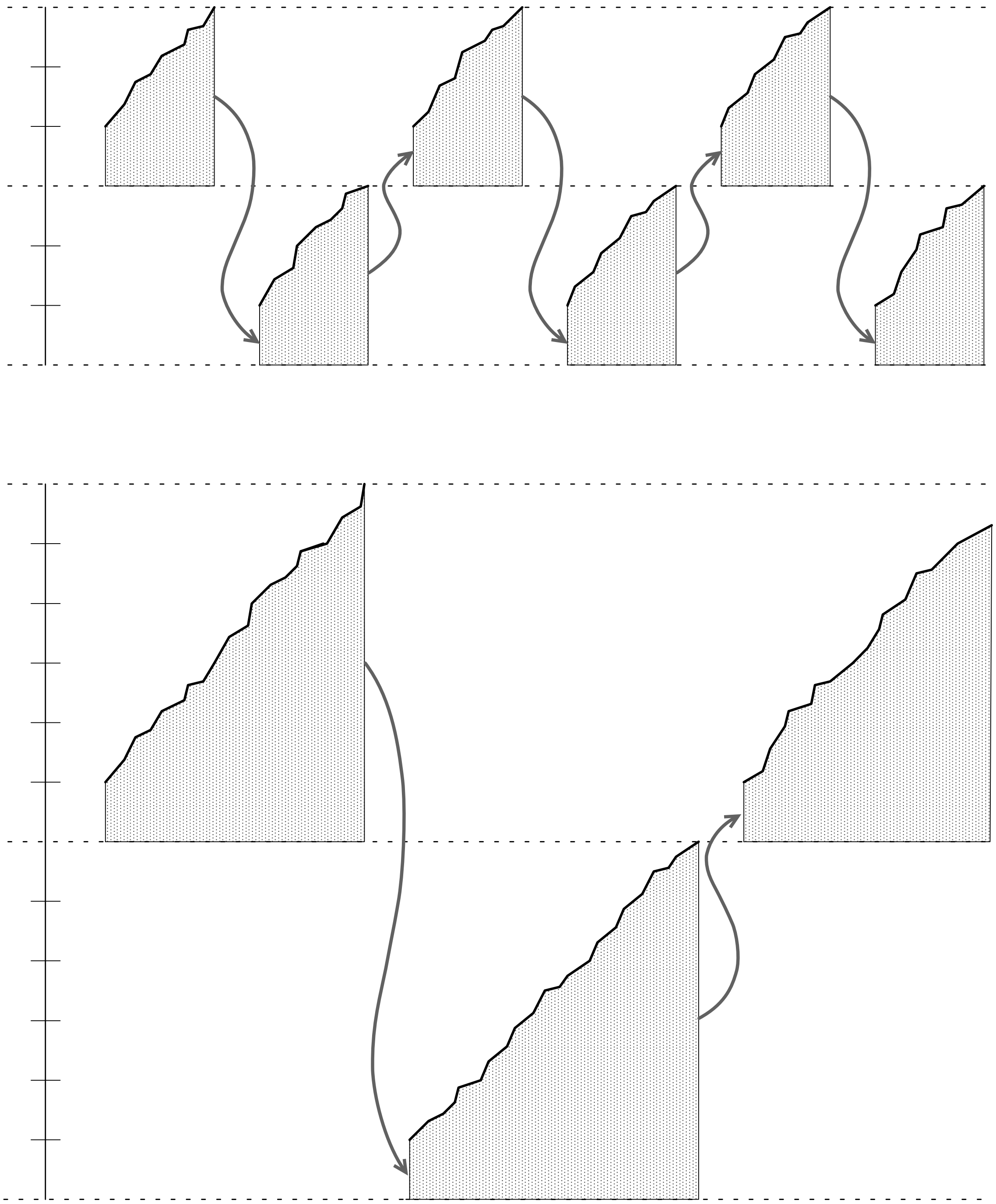

Figure 6: Memory usage in a semispace GC, with 3 MB (top) and 6 MB (bottom) semispaces

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 noncopying 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 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 Coherence and Conservatism

Incremental marking traversals must take into account changes to the reachability graph, made by the mutator during the collector's traversal. Incremental copying collectors pose more severe coordination problems—the *mutator* must also be protected from changes made by the garbage collector.

It may be enlightening to view these issues as a variety of *coherence* problems—having multiple processes attempt to share changing data, while maintaining some kind of consistent view [NOPH92]. (Readers unfamiliar with coherence problems in parallel systems should not worry too much about this terminology; the issues should become apparent as we go along.)

An incremental mark-sweep traversal poses a *multiple readers, single writer* coherence problem—the collector's traversal must respond to changes, but only the mutator can change the graph of objects. (Similarly, only the traversal can change the mark bits; each process can update values, but any field is writable by only one process. Only the mutator writes to pointer fields, and only the collector writes to mark fields.)

Copying collectors pose a more difficult problem—a *multiple readers, multiple writers* problem. Both the mutator and the collector may modify pointer fields, and each must be protected from inconsistencies introduced by the other.

Garbage collectors can efficiently solve these problems by taking advantage of the semantics of garbage collection, and using forms of *relaxed consistency*—that is, the processes needn't always have a consistent view of the data structures, as long as the differences between their views "don't matter" to the correctness of the algorithm.

In particular, the garbage collector's view of the reachability graph is typically *not* identical to the actual reachability graph visible to the mutator. It is only a safe, *conservative* approximation of the true reachability graph—the garbage collector may view some unreachable objects as reachable, as long as it doesn't view reachable objects as unreachable, and erroneously reclaim their space. Typically, some garbage objects go unreclaimed for a while; usually, these are objects that become garbage after being reached by the collector's traversal. Such floating garbage is usually reclaimed at the next garbage collection cycle; since they will be garbage at the *beginning* of that collection, the tracing process will not conservatively view them as live. The inability to reclaim floating garbage immediately is unfortunate, but may be essential to avoiding very expensive coordination between the mutator and collector.

The kind of relaxed consistency used—and the corresponding coherence features of the collection scheme—are closely intertwined with the notion of conservatism. In general, the more we relax the consistency between the mutator's and the collector's views of the reachability graph, the more conservative our collection becomes, and the more floating garbage we must accept. On the positive side, the more relaxed our notion of consistency, the more flexibility we have in the details of the traversal algorithm. (In parallel and distributed garbage collection, a relaxed consistency model also allows more parallelism and/or less synchronization, but that is beyond the scope of this survey.)

3.2 Tricolor Marking

The abstraction of *tricolor marking* is helpful in understanding incremental garbage collection [DLM+78]. 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, gray, 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 gray. When they are scanned and pointers to their offspring are traversed, they are blackened and the offspring are colored gray.

In a copying collector, the gray objects are the objects in the unscanned area of tospace—if a Cheney breadth-first traversal is used, that's the objects between the scan and free pointers. In a mark-sweep collector, the gray 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 gray 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 gray 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 gray objects and move the wavefront forward. If the mutator creates a pointer from a black object to a white one, it must somehow notify the collector that its assumption has been violated. This ensures 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 grayed. 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, B will be blackened, C will be reached again (from B), and D will never be reached at all, and hence will be erroneously deemed garbage and reclaimed.

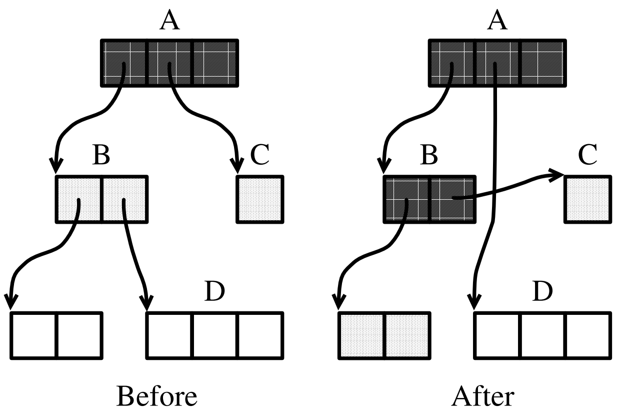

Figure 7: A violation of the coloring invariant.

3.2.1 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 gray; 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 gray 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 in a data structure "off to the side" so that the collector can find them. Thus no path to a white object can be broken without providing another path to the object for the garbage collector.

Incremental update collectors are 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 copies where they are stored, rather than noticing if the original is destroyed. 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 gray when the mutator "undoes" the collector's traversal [Ste75]. (Alternatively, the pointed-to object may be grayed immediately [DLM+78].) This ensures that the traversal is updated in the face of mutator changes.

Read barriers and write barriers are conceptually synchronization operations—before the mutator can perform certain operations, it must activate the garbage collector to perform some action. In practice, this invocation of the garbage collector only requires a relatively simple action, and the compiler can simply emit the necessary additional instructions as part of the mutator's own machine code. Each pointer read or write (depending on the incremental strategy) is accompanied by a few extra instructions that perform the collector's operations. Depending on the complexity of the read or write barrier, the entire barrier action may be compiled inline; alternatively, the barrier may simply be a hidden, out-of-line procedure call accompanying each pointer read or write. (Other strategies are possible, relying less on additional instructions in compiled code, and more on assistance from specialized hardware or virtual memory features.)

3.3 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.

3.3.1 Snapshot-at-beginning Algorithms

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.

The first snapshot-at-beginning algorithm was apparently that of Abrahamson and Patel, which used virtual memory copy-on-write techniques [AP87], but the same general effect can be achieved straightforwardly (and fairly efficiently) with a simple software write barrier.

Perhaps the simplest and best-known snapshot collection algorithm is Yuasa's [Yua90b]. 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 which are live at the beginning of 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 saved on a stack when it is overwritten with the pointer to C.

Snapshot-at-beginning schemes are very conservative, because they actually allow the tricolor "invariant" to be broken, temporarily, during incremental tracing. Rather than preventing the creation of pointers from black objects to white ones, a more global and conservative strategy prevents the loss of such white objects: the original path to the object can't be lost, because *all* overwritten pointer values are saved and traversed.

This implies that *no* objects can be freed during collection, because a pointer to any white object might have been stored into a reachable object. This includes objects that are created while the collection is in progress. Newly-allocated objects are therefore considered to be black, as though they had already been traversed. This short-circuits the traversal of new objects, which would fail to free any of them anyway.

The collector's view of the reachability graph is thus the set union of the graph at the beginning of garbage collection, plus all of those that are allocated during tracing.

An important feature to notice about snapshot-at-beginning algorithms is that since don't actually preserve Dijkstra's tricolor invariant, grey objects have a subtle role. Rather than guaranteeing that *each* path from a black object to a white object must go through a grey object, it is only guaranteed that for each such reachable white object there will be *at least one* path to the object from a grey object. A grey object therefore does not just represent the local part of the collector's traversal wavefront—it may also represent pointers elsewhere in the reachability graph, which cross the wavefront unnoticed.[14]

3.3.2 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.[15]

Perhaps the best known of these algorithms is due to Dijkstra *et al.* [DLM+78]. (This is similar to the scheme developed by Steele [Ste75], but simpler because it does not deal with compaction.) 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. More precisely, an object will not be reached by the collector if all paths to it are broken at a point that the garbage collector has not yet reached. If a pointer is obliterated *after* being reached by the collector, it is too late. (E.g., if the head of a list has already been reached and grayed, and then becomes garbage, the rest of the list will still be traversed.)

To avoid the problem of pointers being hidden in reachable objects that have already been scanned, such copied pointers are caught when they are stored into the scanned objects. Rather than noticing when a pointer escapes *from* a location that hasn't been traversed, it notices when the pointer hides *in* 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. 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.[16]

Several variations of this incremental update algorithm are possible, with different implementations of the write barrier and different treatments of objects allocated during collection.

In the incremental update scheme of Dijkstra et al. [DLM+78], objects are optimistically assumed to be unreachable 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. In contrast, snapshot schemes must assume that such newly-created objects are live, because pointers to them might get installed into objects that have already been reached by the collector's traversal without being detected.

Dijkstra also chooses to allocate new objects white, on the assumption that new objects are likely to be short-lived and quickly reclaimed.

We believe that this has a potentially significant advantage over schemes that allocate black. 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 the snapshot scheme (or Baker's, described below) there's an extra computational cost—the newly-created objects that *are* still live at the end of collection must be traversed, and also any that became garbage too late to be reclaimed, because the traversal had already started along a path to them. As we will explain later, whether this is worthwhile may depend on several factors, such as the relative importance of average case efficiency and hard real-time response. Steele proposes a heuristic that allocates some objects white and other objects black, attempting to reclaim the short-live objects quickly while avoiding traversal of most other objects [Ste75].

[14] This nonlocal constraint poses significant problems for optimization of the garbage collection process, particularly when trying to make a *hierarchical* generational or distributed version of a snapshot algorithm, where multiple garbage collections of different scopes proceed concurrently [WJ93].

[15] Another probable reason is that the early papers on concurrent garbage collection addressed different concerns than those facing most language implementors. [DLM+78] stressed elegance of correctness proofs at the expense of efficiency, and readers may have missed the fact that trivial changes to the algorithm would make it vastly more practical. [Ste75] presented a complex algorithm with an optional incremental compaction phase; many readers doubtless failed to recognize that the incremental update strategy was itself simple, and orthogonal to the other features.

[16] The algorithm of [DLM+78] actually uses a somewhat more conservative technique, as we will explain shortly.

The effectiveness of this heuristic is unproven, and it appears to be difficult to implement efficiently on standard hardware.

Dijkstra's incremental update algorithm [DLM+78] (which apparently predates Steele's slightly) actually preserves the tricolor invariant by blackening the pointed-to white object, rather than reverting the stored-into black object to gray. Intuitively, this pushes the gray wavefront outward to preserve the tricolor invariant, rather than pushing it back. This is more conservative than Steele's strategy, because the pointer might later be overwritten, freeing the object. On the other hand, it appears to be simpler and faster in practice; it also makes it slightly easier to prove he correctness of the algorithm, because there is an obvious guarantee of forward progress.

3.4 Baker's Read Barrier Algorithms

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. 2.4, and uses a *read barrier* for coordination with the mutator. More recently, Baker has proposed a non-copying version of this algorithm, which shares many properties with the copying version [Bak91b].

3.4.1 Incremental Copying

Baker's original copying algorithm was an adaptation of the Cheney algorithm. 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, and mutator activity can also trigger copying, as needed, to ensure that the mutator's view of data structures is always consistent.

In Baker's system, a garbage collection cycle begins with an atomic *flip*, which conceptually invalidates all objects in fromspace, and copies to tospace all objects directly reachable from the root set. Then the mutator is allowed to resume. Any fromspace object that is accessed by the mutator must first be copied to tospace, and this copying-on-demand is enforced by the read barrier. (The read barrier is typically implemented as a few instructions emitted by the compiler, forming a wrapper around pointer-dereferencing read instructions.) The background scavenging process is also interleaved with normal program execution, to ensure that all reachable data are copied to tospace and the collection cycle completes before memory is exhausted.

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.[17]

In order to ensure that the collector 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 gray. 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 tracing, 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 gray. In effect, this advances the wavefront of graying just ahead of the actual references by the mutator, keeping the mutator inside the wavefront.[18] The preservation of the tricolor invariant is therefore indirect—rather than actually checking to see whether pointers to white objects are stored into black ones, the read barrier ensures

[17] 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, and older objects aren't interspersed with new ones.

[18] 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.

that the mutator can't see such pointers in the first place.

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.

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 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, extra code must be executed 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 [Zor89].

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 fromspace, its indirection pointer points to the new version. Unconditionally indirecting is cheaper than checking for indirections, but could still incur overheads in the tens of percent for a high-performance system [Ung84]. (A variant of this approach has been used by North and Reppy in a concurrent garbage collector [NR87]; another variant exploits immutable values in ML to allow reading of some data from fromspace [HL93]. 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].

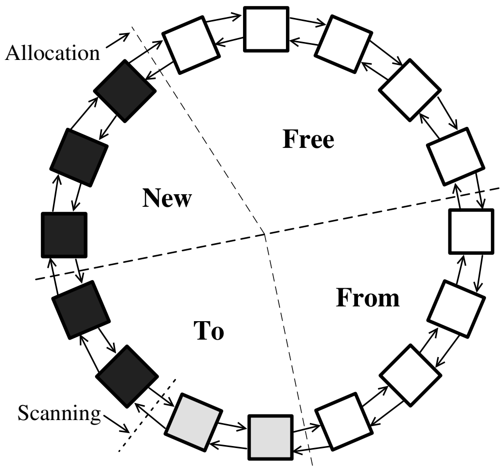

Figure 8: Treadmill collector during collection.

3.4.2 Baker's Incremental Non-copying Algorithm—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 [Bak91b, WJ93].[19]

This non-copying scheme preserves the essential efficiency advantage of copy collection, by reclaiming space implicitly. (As described in Sect. 2.5, unreached objects in the white set can be reclaimed in constant time 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 data structures, objects are moved from the from-list to the to-list. The to-list is initially empty,

[19] 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.

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 gray 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 to-list and advance it through the gray objects.[20]

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.[21]

The state is very similar to the beginning of the previous cycle, except that the segments have "moved" part way 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.

3.4.3 Conservatism of Baker's Read Barrier

Baker's garbage collectors use a somewhat conservative approximation of true liveness in two ways. 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 grayed, 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 gray 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, but only when gray objects have pointers overwritten. Overwriting pointers from black objects has no effect on conservatism, because their referents are already gray. The degree of conservatism (and floating garbage) thus depends on the details of the collector's traversal and of the program's actions.

3.4.4 Variations on the Read Barrier

Several garbage collectors have used slight variations of Baker's read barrier, where the mutator is only allowed to see *black* (i.e., completely scanned) objects. Recall that Baker's read barrier copies an object to tospace as soon as the mutator encounters a pointer to the object. This may be inefficient, because the checking costs are incurred at each reference in the general case, and because it costs something to trap to the scanning-and-copying routine (typically, a conditional branch and a subroutine call).

It may therefore be preferable to scan an entire object when it is first touched by the mutator, and update all of the object's pointer fields. This may be cheaper than calling the scanning-and-copying routine each time a field is first referenced; the compiler may also be able to optimize away redundant checks for multiple references to fields of the same object. (Juul and Jul's distributed garbage collector [JJ92] uses such an objectwise scanning technique, and combines some of the garbage collector's checking costs with those incurred for fine-grained object migration.)

Such a read barrier is coarser and more conservative than Baker's original read barrier. It enforces a stronger constraint—not only is the mutator not allowed to see *white* objects, it is only allowed to see *black* objects. Since an entire object is scanned when it is first touched, and its referents are grayed, the object becomes black before the mutator is allowed to see it. This advances the wavefront of the collector's traversal an extra step ahead of the mutator's pattern of references.

Such a "black only" read barrier prevents any data from becoming garbage, from the garbage collector's point of view, during a garbage collection—before any pointer can be overwritten, the object containing it will be scanned, and the pointer's referent will be grayed. In effect, this implements a "snapshot-at-beginning" collection, using a read barrier rather than

[20] Because the list structure is more flexible than a contiguous area of memory, it is even possible to implement a depth-first traversal with no auxiliary stack, in much the same way that the Cheney algorithm implements breadth-first [WJ93].

[21] 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.

a write barrier.

Appel, Ellis, and Li's concurrent incremental collector [AEL88] uses virtual memory primitives to implement a pagewise black-only read barrier. Rather than detecting the first reference to any grey object (in tospace), entire pages of unscanned data in tospace are access-protected, so that the virtual memory system will implicitly perform read barrier checks as part of the normal functioning of the virtual memory hardware. When the mutator accesses a protected page, a special 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., grayed) and access-protected. This avoids the need for continual software checks to implement the read barrier, and in the usual case is more efficient. (If the operating system's trap handling is slow, however, it may not be worth it.) Despite reliance on operating system support, this technique is relatively portable because most modern operating systems provide the necessary support.

Unfortunately this scheme fails to provide meaningful real-time guarantees in the general case [WM89, NS90, WJ93]. (It does support concurrent collection, however, and can 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.[22]

3.5 Replication Copying Collection

Recently, Nettles et al. [NOPH92, ONG93] have devised a new kind of incremental copying collection, *replication copying*, which is quite different from Baker's incremental copying scheme. Recall that in Baker's collector, garbage collection starts with a "flip," which copies the immediately-reachable data to tospace, and invalidates fromspace; from that moment on, the mutator is only allowed to see the new versions of objects, never the versions in fromspace.

Replication copying is almost the reverse of this. While copying is going on, the mutator continues to see the *fromspace* versions of objects, rather than the "replicas" in tospace. When the copying process is complete, a flip is performed, and the mutator then sees the replicas.

[22] Johnson has improved on this scheme by incorporating lazier copying of objects to tospace [Joh92]; this is essentially an application of Nilsen's lazy copying technique [Nil88] to the Appel-Ellis-Li collector. 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".

The consistency issues in replication copying are very different from those in Baker-style copying. The mutator continues to access the same versions of objects during the copying traversal, so it needn't check for forwarding pointers. This eliminates the need for a read barrier—conceptually, all objects are "forwarded" to their new versions at once, when the flip occurs.

On the other hand, this strategy requires a write barrier, and the write barrier must deal with more than just pointer updates. In Baker's collector, the mutator only sees the new versions of objects, so any writes to objects automatically update the current (tospace) version. In replication copying, however, the mutator sees the old version in fromspace; if an object has already been copied to tospace, and the fromspace version is then modified by the mutator, the new replica can have the wrong (old) values in it—it gets "out of synch" with the version seen by the mutator.

To avoid this, the write barrier must catch *all* updates, and the collector must ensure that all updates have been propagated when the flip occurs. That is, all of the modifications to old versions of objects must be made to the corresponding new versions, so that the program sees the correct values after the flip.

This write barrier appears to be expensive for most general-purpose programming languages, but not for functional languages, or "nearly-functional" languages (such as ML) where side effects are allowed but infrequently used.

3.6 Coherence and Conservatism Revisited

As we mentioned in Sect. 3.1, incremental collectors may take different approaches to coordinating the mutator with the collector's tracing traversal. If these quasi-parallel processes coordinate closely, their views of data structures can be very precise, but the coordination costs may be unacceptable. If they do not coordinate closely, they may suffer from using out-dated information, and retain objects which have become garbage during collection.

3.6.1 Coherence and Conservatism in Non-copying collection

The non-copying write-barrier algorithms we have described lie at different points along a spectrum of effectiveness and conservatism. Snapshot-at-beginning algorithms treat everything conservatively, reducing their effectiveness. Dijkstra et al.'s incremental update algorithm is less conservative than snapshot algorithms, but more conservative than Steele's algorithm.

In Steele's algorithm, if a pointer to a white object is stored into a black object, that white object is *not* immediately grayed—instead, the stored-into black object is reverted to gray, "undoing" the blackening done by the collector. This means that if the stored-into field is again overwritten, the white object may become unreachable and may be reclaimed at the end of the current collection. In contrast, Dijkstra's algorithm will have grayed that object, and hence will not reclaim it.

(It may seem that this is a trivial difference, but it is easy to imagine a scenario in which it matters. Consider a program that stores most of its data in stacks, implemented as linked lists hanging off of "stack" objects. If a stack object is reached and blackened by the collector's traversal, and then many objects are pushed onto and popped off of the stack, Dijkstra's algorithm will not reclaim *any* of the popped items—as the stack object's list pointer progresses through the list, repeatedly being overwritten with the pointer to the next item, each item will be grayed when the previous one is popped. Steele's algorithm, on the other hand, may reclaim almost all of the popped items, because the pointer field may be overwritten many times before the collector's traversal examines at it again.)

Note that this spectrum of conservatism (snapshot algorithms, Dijkstra's, Steele's) is only a linear ordering if the algorithms use the same traversal algorithm, scheduled in the same way relative to the program's actual behavior—and this is unlikely in practice. Details of the ordering of collector and mutator actions determine how much floating garbage will be retained. (Any of these collectors will retain any data reachable via paths that are traversed by the collector before being broken by the mutator.)

This suggests that the reachability graph might profitably be traversed *opportunistically*, i.e., total costs might be reduced by carefully ordering the scanning of gray objects. For example, it might be desirable to avoid scanning rapidly-changing parts of the graph for as long as possible, to avoid reaching objects that will shortly become garbage.

All other things being equal (i.e., in lieu of opportunism and random luck), snapshot-at-beginning is more conservative (hence less effective) than incremental update, and Dijkstra's incremental update is more conservative than Steele's.

3.6.2 Coherence and Conservatism in Copying Collection

Baker's read barrier algorithm does not fall neatly into the above spectrum. It is less conservative than snapshot-at-beginning, in that a pointer in a gray object may be overwritten and never traversed; it is more conservative than the incremental update algorithms, however, because anything reached by the mutator is grayed—objects cannot become garbage, from the collector's viewpoint, after simply being touched by the mutator during a collection.

Nettles, et al.'s replication copying algorithm (like an incremental update algorithm), *is* able to reclaim objects that become unreachable because a pointer can be overwritten before being reached by the collector. Their collector is less conservative than Baker's, in part because it can use a weaker notion of consistency. Because the mutator doesn't operate in tospace until after the copying phase is complete, the copies of data in tospace needn't be entirely consistent during incremental copying. (The changes made to fromspace data structures by the mutator must be propagated to tospace eventually, but the entire state only needs to be consistent at the end of collection, when the atomic "flip" is performed.) Like the other write-barrier algorithms, replication copying might benefit significantly from opportunistic traversal ordering.

3.6.3 "Radical" Collection and Opportunistic Tracing

The tracing algorithms we've described fall roughly into a spectrum of decreasing conservatism, thus:

- Snapshot-at-beginning write barrier
- Black-only read barrier
- Baker's read barrier
- Dijkstra's write barrier
- Steele's write barrier

In considering this quasi-spectrum, it is interesting to ask, *is there anything less conservative than Steele's algorithm?* That is, can we have a better-informed collector than Steele's, one which responds more aggressively to changes in the reachability graph? The answer is yes. Such a garbage collector would be willing to re-do some of the traversal it's already done, un-marking objects that were previously reached, to avoid conservatism. We refer to this as a "radical" garbage collection strategy. At first glance, such collectors may seem impractical, but under some circumstances, approximations of them may make sense.

The limiting case of decreasing conservatism is to respond fully to any change in the reachability graph, un-marking objects that have already been reached, so that all garbage can be detected. (We might call this a *fully radical* collector.)

One way of doing that is to perform a full trace of the actual reachability graph at every pointer write on the part of the application. Naturally, this is impractical because of its extreme cost. (A normal non-incremental collector can be viewed as an approximation of this; the graph is traversed "instantaneously" by stopping the mutator for the whole traversal, but that's only done occasionally.)

Another way of achieving fully radical collection would be to record all of the dependencies within the reachability graph, and update the dependency database at every pointer update. Whenever all paths keeping an object alive is broken, the object is known to be garbage. Again, a full implementation of this strategy would be impractical for general-purpose garbage collection, because the dependency database could be very large, and pointer updates would be very expensive.

Note, however, that approximations of this dependency information could be relatively cheap, and in fact, that's exactly what reference counts are. A reference count is a conservative approximation of the number of paths to an object, and when those paths are eliminated, the reference counts usually go to zero and allow the object to be reclaimed immediately. Some distributed garbage collection algorithms also perform somewhat radical collection, by frequently recomputing some local parts of the collector's traversal.

3.7 Comparing Incremental Techniques

In comparing collector designs, it is instructive to keep in mind the abstraction of tricolor marking—as distinct from concrete tracing mechanisms such as mark-sweep or copy collection. The choice of a read- or write-barrier (and strategy for ensuring correctness) is mostly independent of the choice of a tracing and reclamation mechanism.

For example, Brooks' copying collector [Bro84] (which we mentioned in Sect 3.4.1) is actually an incremental update write barrier algorithm, even though Brooks describes it as an optimization of Baker's scheme.[23] Similarly, Dawson's copying scheme (proposed in [Daw82]) is cast as a variant of Baker's, but it is actually an incremental update scheme, and objects are allocated in fromspace, i.e., white, as in Dijkstra's collector.

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. If appropriate virtual memory support is available, and hard real-time response is not required, a pagewise read barrier may be desirable.

Of write barrier schemes, snapshot-at-beginning algorithms are significantly more conservative than incremental update algorithms. This advantage of incremental update might be increased 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.

While incremental update schemes increase effectiveness, they may also increase costs. In the worst case, everything that becomes garbage during a collection "floats," i.e., it becomes unreachable too late, and is traversed and retained anyway. If new objects are allocated white (subject to reclamation), incremental update algorithms may be considerably *more* expensive than snapshot-at-beginning algorithms in the worst case—it is possible that all of the newly-allocated objects will float and require traversal, with no increase in the amount of memory reclaimed. We will discuss this in more detail in Sect. 3.8.2.

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.)

[23] 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. Brooks' algorithm uses a simple write barrier to protect the mutator from the collector, and a simple read barrier to protect the collector from the mutator.

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.

3.8 Real-time Tracing Collection

Incremental collectors are often designed to be *real-time*, i.e., to impose strictly limited delays on program execution, so that programmers can guarantee that their garbage-collected programs will meet real-time deadlines. Real-time applications are many and varied, including industrial process controllers, testing and monitoring equipment, audiovisual processing, fly-by-wire aircraft controls, and telephone switching equipment. Real-time applications can usually be classified as *hard* real time, where computations must complete with strictly-limited time bounds, and *soft* real-time, where it is acceptable for some tasks to miss their schedules some of the time, as long as it doesn't happen "too often".[24]

The criterion for real time garbage collection is often stated as *imposing only small and bounded delays on any particular program operation.* For example, traversing a pointer might never take more than a microsecond, heap-allocating a small object might never take more than a few microseconds, and so on.

There are two problems with this kind of criterion. One problem is that the appropriate notion of a "small" delay is inevitably dependent on the nature of an application. For some applications, it is acceptable to have responses that are delayed by a significant fraction of a second, or even many seconds. For other applications, a delay of a millisecond or two is not a problem, while for others delays of more than a few microseconds could be fatal. (On one hand, consider a music synthesizer controller, where humans' own imprecision will swamp a delay of a millisecond and be unnoticeable; on the other, consider a high-precision guidance system for anti-missile missiles.)

Another problem with this kind of criterion is that it unrealistically emphasizes the smallest program operations. When you press a key on a musical keyboard, the controller may be required to execute thousands of program statements (e.g., to decide which note is being played, what the corresponding pitch is given the current tunings, how loud to play it, which sound components to mix in what proportions to achieve the right timbre for the given pitch, and so on).

For most applications, therefore, a more realistic requirement for real time performance is that the application always be able to use the CPU for a given fraction of the time *at a timescale relevant to the application.* (Naturally, the relevant fraction will depend on both the application and the speed of the processor.)

For a chemical factory's process control computer, it might be sufficient for the controlling application to execute for at least one second out of every two, because the controller must respond to changes (e.g., in vat temperatures) within two seconds, and one second is enough to compute the appropriate response. On the other hand, a controller for a musical synthesizer might require the CPU to run its control program for half a millisecond out of every two milliseconds, to keep delays in the onset of individual notes below the threshold of noticeability.

Note that either of these applications can function correctly if the garbage collector sometimes stops the application for a quarter of a millisecond. Provided that these pauses aren't too frequent, they're too short to be relevant to the applications' real-time deadlines.

But suppose these pauses are clustered in time; if they happen frequently enough they will destroy the application's ability to meet deadlines, simply by soaking up too large a fraction of the CPU time. If the application only executes for a sixteenth of a millisecond between quarter-millisecond pauses, it can't get more than a fifth of the CPU time. In that case, either of the above programs would fail to meet its real-time requirements, even the process control system that only needs to respond within two seconds.

As we described above, some copy collectors use virtual memory protections to trigger pagewise scanning, and this coarseness may fail to respect real-time guarantees. In the worst case, traversing a list of a thousand elements may cause a thousand pages to be scanned, performing considerable garbage collection work and incurring trap overheads as well. (In this way, a list traversal that would normally take a few thousand instructions may unexpectedly take millions, increasing the time to traverse the list by several orders of magnitude.) Locality of reference may make such situations improbable, but the probability of bad cases is not negligible.

[24] For example, in a digital telephone system, making a connection might be a soft real-time task, but once a connection is established, delivering continuous audio may be a hard real-time task. In this section, we will deal primarily with hard real-time issues.

Unfortunately, using a fine-grained incremental collector may not fix this problem, either [Nil88, Wit91]. Consider Baker's copying technique. The time to traverse a list depends on whether the list elements require relocation to tospace. Traversing a single pointer may require an object to be copied; this may increase the cost of that memory reference by an order of magnitude *even if objects are small and hardware support is available*. (Consider copying a Lisp cons cell consisting of a header, a CAR field, and a CDR field. At least three memory reads and three memory writes are required for the actual copy, plus extra instructions to install a forwarding pointer, adjust the free-space pointer, and probably to branch to and from the garbage collector routine that does this work.) In such cases, it is possible for the garbage collector overheads to consume over 90% of the CPU time, reducing the available computing power—that is, the power guaranteed to be available for meeting real-time deadlines—by an order of magnitude.

(In Baker's original incremental copying scheme, the worst-case cost is even worse, because any pointer traversal may force the copying of a *large* object. Arrays are treated specially, and copied lazily, i.e., only when they are actually touched. Nilsen reduces the worst-case by extending this lazy copying to all types of objects. When a pointer to a tospace object is encountered by the mutator, space is simply reserved in tospace for the object, rather than actually copying it. The actual copying occurs later, incrementally, when the background scavenger scans that part of tospace [Nil88].)

In deciding on a real-time tracing strategy, therefore, it is important to decide what kind of guarantees are necessary, and at what timescales. While Baker's is the best-known incremental algorithm, it may not be the most suitable for most real-time applications, because its performance is very unpredictable at small timescales. Algorithms with a weaker coupling between the mutator and the collector (such as most write-barrier algorithms) may be more suitable [WJ93]. It may be easier for programmers to reason about real-time guarantees if they know that pointer traversals always take a constant time, independent of whether the pointer being traversed has been reached by the garbage collector yet. (Write barrier algorithms require more work per pointer store, but the work per program operation is less variable, and most of it need not be performed immediately to maintain correctness.)

Unfortunately, while non-copying algorithms have the convenient property that their time overheads are more predictable, their space costs are much more difficult to reason about. A copying algorithm generally frees a large, contiguous area of memory, and requests for objects of any size can be satisfied by a constant-time stack-like allocation operation. Non-copying algorithms are subject to fragmentation—memory that is freed may not be contiguous, so it may not be possible to allocate an object of a given size even if there is that much memory free.

The following sections discuss techniques for obtaining real-time performance from an incremental tracing collector. We assume that the system is purely hard real time—that is, the program consists only of computations which *must* complete before their deadlines; we also assume that there is only one timescale for real-time deadlines. In such a system, the main goal is to make the worst-case performance as good as possible, and further increases in expected-case performance do no good. (Later, we will briefly discuss tradeoffs in systems with soft real-time schedules, where differences in expected-case performance may also be important.) We also assume that either a copying algorithm is used, or all objects are of a uniform size.[25] This allows us to assume that any memory request can be satisfied by any available memory, and ignore possible fragmentation of free storage.

3.8.1 Root Set Scanning

An important determinant of real-time performance is the time required to scan the root set. Recall that in Baker's incremental collector, the root set is updated, and immediately-reachable objects are copied to tospace, in a single atomic operation, uninterrupted by mutator execution. This means that there will occasionally be a pause of a duration roughly proportional to the size of the root set. This pause is likely to be much larger than a pause for a normal increment of tracing, and may be the main limitation on real-time guarantees.

Similar pauses occur in incremental update tracing algorithms when attempting to terminate a collection. Before a collection can be considered finished, the root set must be scanned (along with any gray objects recorded by the write barrier, in the case of an algorithm like Steele's), and all reachable data must be traversed and blackened atomically. (This ensures

[25] In some systems, it is feasible to transparently fragment language-level objects into easily-managed chunks, to make garbage collection easier and reduce or eliminate fragmentation problems.

that no pointers have been hidden from the collector by storing them in roots after those roots were last scanned.) If this work cannot be accomplished within the time allowed by the real-time bounds, the collector must be suspended and the mutator resumed, and the entire termination process must be tried again later. (Snapshot at beginning algorithms don't pose as difficult a problem for termination detection, since no paths can be hidden from the collector.)

One way to bound the work required for a flip or for termination is to keep the root set small. Rather than considering all local and global variables to be part of the root set, some or all of them may be treated like objects on the heap. Reads or writes to these variables will be detected by the read barrier or write barrier, and the collector will therefore maintain the relevant information incrementally.

The problem with keeping the root set small is that the cost of the read or write barrier goes up correspondingly—a larger number of variables is protected by a read or write barrier, incurring overhead each time they are read or written. One possible tradeoff is to avoid the read or write-barrier cost for register-allocated variables, and to scan (only) the register set atomically when necessary. If there are too many operations on stack-allocated local variables, however, this will slow execution significantly. In that case, the entire stack may be scanned atomically instead. While this may sound expensive, most real-time programs never have deep or unbounded activation stacks, and the cost may be negligible at the scale of the program's intended response times. Similarly, for small systems using fast processors (or with relatively large timescales for real-time requirements), it may be desirable to avoid the read or write barrier for all global variables, and scan them atomically as well. Intermediate strategies are possible, treating some variables one way and others another, perhaps based on profiling information.

3.8.2 Guaranteeing Sufficient Progress

The preceding section focused on ensuring that the collector does not use too much CPU time, at the relevant timescale, keeping the processor from being able to meet its real-time deadlines. Conversely, the collector has a real-time deadline of its own to meet: it must finish its traversal and free up more memory before the currently-free memory is exhausted. If it doesn't, the application will have to halt and wait for the collection to complete and free up more memory.

For hard real-time programs, then, there must be some way of ensuring that the collector gets enough CPU time to complete its task before free memory is exhausted, even in the worst possible case. To provide such a guarantee, it is necessary to quantify the worst case—that is, to put some bound on what the collector could be expected to do. Since a tracing collector must traverse live data, this requires putting a bound on the amount of live data. In general, the programmer of an application must ensure that the program doesn't have more than a certain amount of live data to traverse, and the collector can then determine how fast it must operate in order to meet its deadline. It can then determine whether this requires more CPU time than it is allowed to consume. Naturally, this generally allows some tradeoffs to be made in the parameter settings. If more memory is available, the collector generally needs a smaller fraction of the CPU time to guarantee that it finishes before memory is exhausted.

The usual strategy for ensuring that free memory is not exhausted before collection is finished is to use an *allocation clock*—for each unit of allocation, a corresponding unit of collection work is done, and the latter unit is large enough to ensure that the traversal is completed before the free space is exhausted [Bak78]. The simplest form of this is to key collection work directly to allocation—each time an object is allocated, a proportional amount of garbage collection work is done. This guarantees that no matter how fast a program uses up memory, the collector is accelerated correspondingly. (In actual implementations, the work is usually batched up into somewhat larger units over several allocations, for efficiency reasons.)

In the rest of this section, we show how to compute the minimum safe tracing rate, starting with a non-copying snapshot-at-beginning collector, which allocates objects black (i.e., not subject to collection). We make the simplifying assumption that all objects are of a uniform size, so that there is a single pool of memory that is allocated from and reclaimed. After describing this simple case, we will explain how the safe tracing rate differs for other incremental tracing algorithms.

For a snapshot-at-beginning algorithm, all of the live data at the beginning of a collection must be traversed by the end of the collection. Other algorithms must do this too, in the worst case, because objects may have to be traversed even if they are freed during collection. In the absence of any other information from the programmer, the collector must generally assume that at the beginning of collection, the maximum amount of live data is in fact live.

Since we assume that objects created during collection are allocated black, i.e., not subject to reclamation, we need not traverse them—those objects will be ignored until the next garbage collection cycle.

At first glance, it might appear that for a maximum amount of live data L and a memory size of M, we would have $(M - L)$ memory available to allocate—this would imply a minimum safe tracing rate of $(M - L)/L$, to trace L data before this "headroom" is exhausted. Unfortunately, though, we also have to deal with floating garbage. The data that are live at the beginning of collection may become garbage during collection, but too late to be reclaimed at this garbage collection cycle. The data we've allocated may also be garbage, but since we allocate black we don't know that yet. If we were to use up $(M - L)$ memory, we might not get *any* space back at this garbage collection cycle, and we would have no headroom left to try another collection. the maximum data we should allocate is therefore only half the headroom, or $(M - L)/2$. The minimum safe tracing rate allows us to allocate that in the time it takes to traverse the maximum live data, so the safe tracing rate is $((M - L)/2)/L$, or $(M - L)/2L$. This is sufficient for the worst case, in which all garbage floats for an entire garbage collection cycle, but is reclaimed at the next cycle.

As mentioned above, the situation is essentially the same for other incremental tracing algorithms, so long as they allocate new objects black, because in the worst case they retain all of the same objects as a snapshot-at-beginning algorithm. The minimum safe tracing rate is proportional to the amount of live data and inversely proportional to the amount of free memory; it therefore approaches zero as memory becomes very large relative to the maximum amount of live data.

For allocating white, however, the situation is considerably worse. When allocating white, we are gambling that newly-allocated data will be short-lived; we therefore make them subject to garbage collection in hopes of reclaiming their space at the current cycle. This obliges us to traverse reachable white objects, and in the worst case we traverse *everything we allocate* before it becomes garbage. Even though we assume that there is a bound on the amount of live data (provided by the programmer), we must take into account the conservatism of the traversal process, and the fact that any pointer may be traversed by the collector before it's broken by the mutator.

When allocating white, therefore, the worst-case safe traversal rate does not approach zero as memory becomes very large—it approaches the allocation rate; the traversal must keep up with the allocation rate, and go at least a little faster, to ensure that it eventually catches up. If we increase the amount of memory relative to the amount of live data, we reach a point of diminishing returns—we must always trace at least as fast as we allocate.

The above analysis applies to non-copying collectors for uniform-sized objects. In copying collectors, more memory is required to hold the new versions of objects being copied; there must be another L units of memory available in the worst case to ensure that tospace is not exhausted before fromspace is reclaimed. This a major space cost if L is large relative to the actual amount of memory available. In non-copying collectors for nonuniform-sized objects, fragmentation must be taken into account. Fragmentation reduces the effective memory available, requiring faster tracing to complete collection in bounded memory. Computations of worst-case fragmentation are intrinsically program-specific [WJ93]; due to space limitations, we will not discuss them here.

3.8.3 Trading worst-case performance for expected performance

When a collection phase is complete, the collector can often determine a less conservative traversal rate, slowing down the collection process and yielding more CPU cycles to the mutator. This is possible because at the end of the collection, the collector can determine how much live data was in fact traced, and revise downward its worst-case estimate of what could be live at the next collection. This may improve performance somewhat, but usually not dramatically.

Alternatively, when the collector can determine that it has less than the worst-case amount of work to do, it may avoid GC activity entirely for a while, then re-activate the collector in time to ensure that it will meet its deadline. This is an attractive option if the read or write barrier can be efficiently disabled on the fly.

3.8.4 Discussion

The foregoing analysis assumes a fairly simple model of real-time performance, with a single timescale for hard real-time deadlines. More complex schemes are certainly possible, in systems with mixed hard and soft deadlines, or systems which have multiple timescales for different kinds of goals. For example, the collector's infrequent but relatively expensive operations (like root-set scanning) might be scheduled along with the application's own longer-term deadlines, in a complementary pattern. This could achieve higher performance overall while providing tight real-time guarantees where necessary.[26]

We have also assumed a fairly simple model of garbage collection, in that there is a single pool of memory available for all memory requests. In a non-copying system with objects of widely differing sizes, this will not be the case, because freeing several small objects does not necessarily make it possible to allocate a larger one. On the other hand, it appears that many applications' memory usage is dominated by a very few sizes of objects; reasoning about real-time collection may not be as hard as it appears at first glance, for the majority of programs [WJ93]. Still, such reasoning must be done on application-by-application basis. For some programs, guaranteeing real-time performance may cost considerable memory due to possible fragmentation, unless application-level objects can be split into more uniform chunks. Another possibility is to statically-allocate the most troublesome datatypes, as is usually done in real-time systems anyway, but rely on the garbage collector to manage most of the objects automatically.

For fully general real-time garbage collection, with reasonable worst-case memory usage, it appears that fine-grained copying collection is required [Nil88]. As mentioned above, copying collection can be quite expensive in the worst case, even if Lisp-machine style hardware support is available to speed up the read barrier [EV91, Wit91]. Nilsen and Schmidt have designed and simulated hardware support which *will* guarantee usefully real-time performance [NS92], but it is significantly more complex.[27]

3.9 Choosing an Incremental Algorithm

In choosing an incremental strategy, it is important to prioritize overall average performance and worst-case performance. Algorithms that are "less conservative" may not be more attractive than others, because the "less conservative" algorithms are just as conservative in the worst case.

Even in the usual case, the less conservative algorithms may not be desirable, because they may simply be slower (e.g., because their write barriers require more instructions.) Paradoxically, this can make a "less conservative" algorithm *more conservative* in practice, because its cost may keep it from being run as often. Because of the higher overhead, the reduced conservatism in terms of incremental strategies may introduce greater conservativeness in how frequently garbage is collected at all.

Overall system design goals are therefore important to the choice of any garbage collection algorithm. As we will explain in the next section, generational techniques make the overheads of incremental collection unnecessary for many systems, where hard real-time response is not necessary, and it is sufficient for the collector to be "nondisruptive" in typical operation. For other systems, it may be desirable to combine incremental and generational techniques, and careful attention should be paid to how they are combined.

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 from program to program, usually between 80 and 98 percent of all newly-allocated heap 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 implementation 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.[28] It is also appropriate to speak

[26] Consider an autonomous robot, which might need to revise its overall high-level planning only every second or so, but might also need to respond "reflexively" to changes in its environment within a few milliseconds. The low-level vision and reactive adjustments might consume a fixed percentage of CPU time on the scale of a few milliseconds, with the remainder available alternately to the high-level planning functions and to the GC, alternating every half-second.

[27] Nilsen's approach is interesting in that it requires relatively complex memory controllers, but it is compatible with off-the-shelf high-performance microprocessors.

[28] One must be careful, however, not to interpret it as the ideal abstract measure. For example, rates of heap allocation

in terms of amounts allocated because the time between garbage collections is largely determined by the amount of memory available.[29] 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 collection, 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 collecting areas containing older objects less often than the younger ones. Once objects have survived a small number of collections, they are moved to a less frequently collected area. Areas containing younger objects are collected 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 collections, to keep copying costs down.

For stop-and-collect (non-incremental) garbage collection, generational garbage collection has an additional benefit in that most collections take only a short time—collecting just the youngest generation is much faster than a full garbage collection. This reduces the frequency of disruptive pauses, and for many programs without real-time deadlines, this is sufficient for acceptable interactive use. The majority of pauses are so brief (a fraction of a second) that they are unlikely to be noticed by users [Ung84]; the longer pauses for multi-generation collections can often be postponed until the system is not in use, or hidden within noninteractive compute-bound phases of program operation [WM89]. Generational techniques are often used as an

are typically 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.

[29] 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.

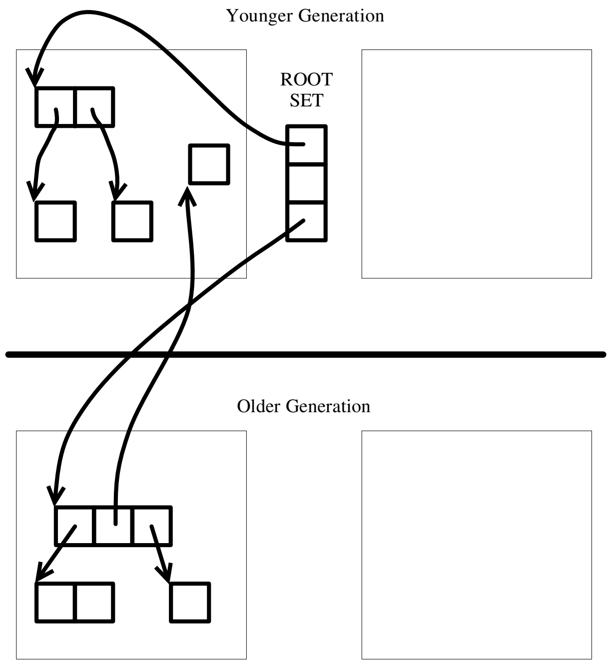

Figure 9: A generational copying garbage collector before garbage collection.

acceptable substitute for more expensive incremental techniques, as well as to improve overall efficiency.

(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 Collection 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 collected, 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 semis-

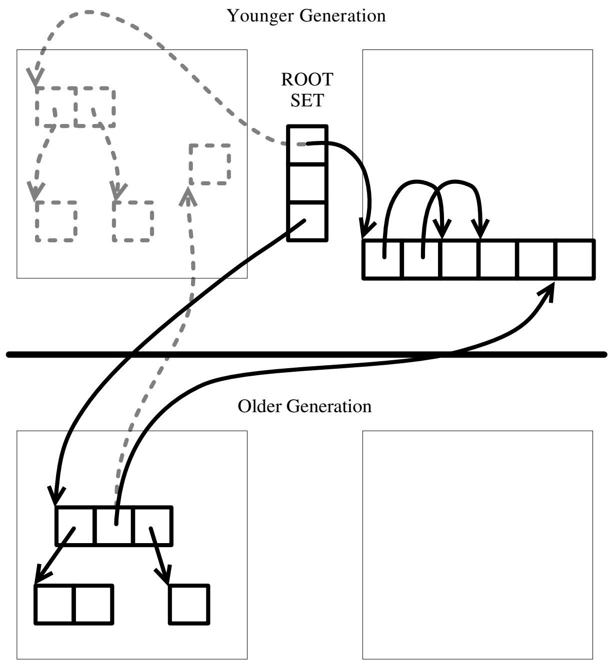

Figure 10: Generational collector after garbage collection.

pace. This removes it from consideration by single-generation collections, so that it is no longer copied at every collection. Since relatively few objects live this long, old memory will 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 collected considerably less often. (Tektronix 4406 Smalltalk is such a generational system, using semispaces for each of eight generations [CWB86].)

In order for this scheme to work, it must be possible to collect the younger generation(s) without collecting 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 collection 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.)

Ensuring that the collector can find pointers into young generations requires the use of something like the "write barrier" of an incremental collector—the running program can't freely store pointers into heap objects at will. Each potential pointer store must be accompanied by some extra bookkeeping, in case an intergenerational pointer is being created. As in an incremental collector, this is usually accomplished by having the compiler emit a few extra instructions along with each store of a (possible) pointer value into an object on the heap.

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]. The important point is that all references from old to new memory must be located at collection 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 copying collector, it also 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 collected. 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 newer objects into older objects, allowing older objects to be collected independently of newer ones. This is more costly, however, because there are typically many more pointers from new to old than from old to new. Such flexibility 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 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

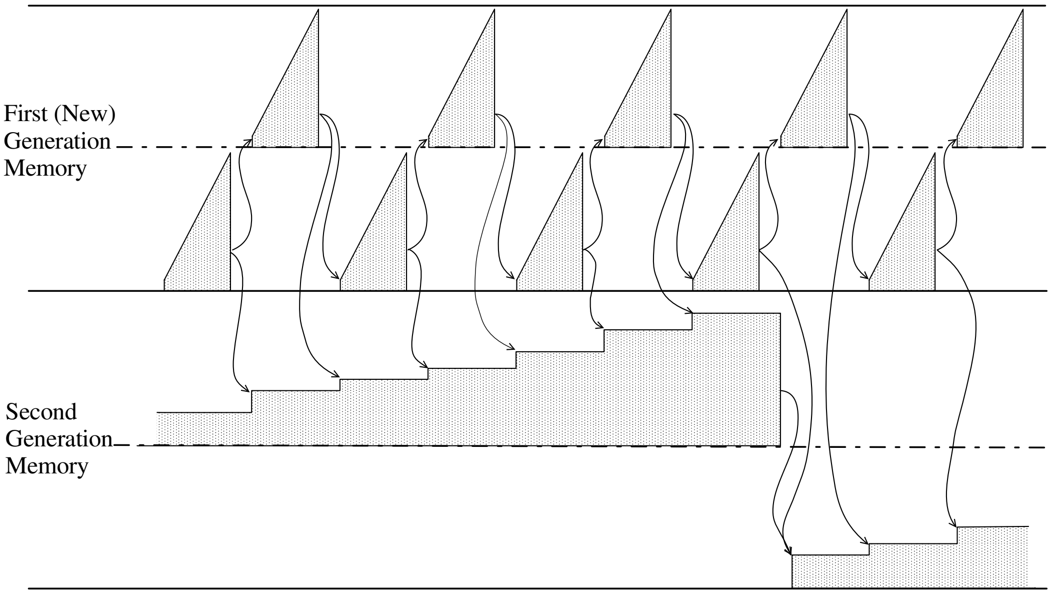

Figure 11: Memory use in a generational copy collector with semispaces for each generation.

youngest generation can't create pointers into younger ones.

Even if young-to-old pointers are not recorded, it may still be feasible to collect a generation without collecting younger ones. In this case, *all* data in the younger 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 younger generations, each generation is usually considerably smaller than the next, and the cost may be small relative to the cost of actually collecting the older generation. (Scanning the data in the younger generation may be preferable to collecting both generations, because scanning is generally faster than tracing and 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.

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

- *Advancement policy.* How long must an object survive in one generation before it is advanced to the next?

- *Heap organization.* How should storage space be divided and used between generations, and within a generation? How does the resulting reuse pattern affect locality at the virtual memory level, and at the level of high-speed cache memories?

- *Collection scheduling.* For a non-incremental collector, how might we avoid or mitigate the effect of disruptive pauses, especially in interactive applications? Can we improve efficiency by careful "opportunistic" scheduling? Can this be adapted to incremental schemes to reduce floating garbage?

- *Intergenerational references.* Since it must be possible to collect younger generations without collecting the older ones, we must be able to find the live pointers from older generations into the ones we're collecting. What is the best way to do this?

4.2 Advancement Policies

The simplest advancement policy is to simply advance all live data into the next generation whenever they are traversed. This has an advantage of ease of implementation, because it is not necessary to be able to distinguish between objects of different ages within a generation. In a copying collector, this allows the use of a single contiguous area for a generation, with no division into semispaces, and it does not require any header fields to hold age information.

An additional advantage of advancing everything out of a generation at the first traversal is that it avoids the buildup of long-lived objects within a generation—a long-lived object cannot be copied repeatedly at the same timescale, because it will be quickly advanced to the next older generation, which is collected less often.

The problem here is that objects may be advanced *too* fast—short-lived objects allocated shortly before a collection will be advanced to the next generation, even though they are quite young and likely to die almost immediately [Ung84, WM89]. This will cause the older generation to fill up more quickly and be collected more often. The problem of very short-lived objects may be alleviated by delaying the advancement of objects by just one garbage collection cycle; this ensures that all objects are roughly the same age (within a factor of two) when they are advanced to an older generation. (In an incremental generational collector, allocating black can have a similar effect if incremental collection phases are of a sufficient duration [WJ93].)

It is unclear whether keeping objects within a generation for more than two collection cycles is worth the extra copying cost. Under most conditions, it appears that successive copies do not greatly reduce the amount of data advanced [WM89, Zor89], although this is highly dependent on the nature of the application; it may also be desirable to vary the advancement policy dynamically [WM89, UJ88].

The desirability of keeping data in a generation for multiple collection cycles is also affected by the number and size of older generations. In general, if there are very few generations (e.g., two, as in Ungar's *Generation Scavenging* system), it is more desirable to retain data longer, to avoid filling up older generations. If intermediate generations are available, it is usually preferable to advance things more quickly, because they are likely to die in an in-between generation and never be advanced to the oldest generation [WM89, Zor89].

4.3 Heap Organization

A generational collector must treat objects of different ages differently. While tracing, it must be able to tell which generation an object belongs to, in order to decide whether to trace its offspring, and whether to advance it to another generation. The write barrier must also be able to determine objects' generations, to detect whether a pointer to a younger object is being stored into an older object.

In a copying collector, this is usually done by keeping objects of different ages in different areas of memory. In many systems, these are contiguous areas; an object's generation can therefore be determined by simple address comparisons. In other systems, the "areas" may be noncontiguous sets of pages—an object's generation can be determined by using the page number part of its address to index into a table that says which generation that page belongs to.

In other systems, such as non-copying collectors, each object belongs to a generation, but objects of different generations may be interspersed in memory. Typically, each object has a header field indicating which generation it belongs to.

4.3.1 Subareas in copying schemes

Generational copying collectors divide each generations' space into several areas. For example, each generation may consist of a pair of semispaces, so that objects can be copied back and forth from one space to another, to retain them within a generation over multiple collections. (If only one space is used, objects must be advanced to another generation immediately because there's nowhere to copy them to within the same generation.)

The locality of semispace memory usage is poor—only half of the memory of a generation can be in use at a given time, yet both of the spaces are touched in their entirety every two collection cycles. Lisp machine garbage collectors [Moo84, Cou88] avoid this problem by using only a single space per generation. Rather than copying objects from one semispace to the other until they are advanced, garbage collection of a generation advances *all* objects into the next generation. This avoids the need for a pair of semispaces, except in the oldest generation, which has no place to copy things to. Unfortunately, it has the drawback that relatively young objects may be advanced along with relatively old ones—objects allocated shortly before a collection are not given much time to die before being advanced. These relatively young objects are likely to die shortly after being advanced, needlessly taking up space in the next generation and forcing it to be collected again sooner.

Ungar's solution to this problem in the Young generation (of his Generation Scavenging collector) is to use *three* spaces instead of just two, with all objects being initially allocated in the third space. The newly-created objects in this third space are copied into a semispace, along with the objects from the other semispace. The third space is emptied at every garbage collection cycle, and can therefore be reused immediately each time. It therefore has locality characteristics similar to those of a single-space-per-generation system. It might seem that this third space would *increase* memory usage, since semispaces are still required in that generation so that objects can be kept in the generation for multiple collections. The creation area is used in its entirety at each allocation-and-collection cycle, while each semispace is used to hold the survivors at every other collection cycle. Typically only a small minority of new objects typically survives even a first collection, so only a small part of each semispaces is actually used most of the time, and the overall memory usage is lower.

Wilson's Opportunistic Garbage Collector [WM89] uses a variation on this scheme, with the subareas within a generation used for the additional purpose of deciding when to advance an object from one generation to another—objects are advanced out of the semispaces to the next generation at each cycle, rather than being copied back and forth from one semispace to the other at successive collections. In effect, this is a simple "bucket brigade" advancement mechanism [Sha88], using the segregation of objects into subareas to encode their ages for the advancement policy. It avoids the need for age fields in object headers, which may be advantageous in some systems, where some objects do not have headers at all.[30] It does provide a guarantee that objects will not be advanced out of a generation without surviving for at least one (and up to two) collection cycles; this is sufficient to avoid premature advancement of very short-lived data.

In several generational copying collection systems, the oldest generation is treated specially. In the Lisp machine collectors, this is necessitated by the fact that most generations are emptied at every collection cycle, and their contents copied to the next generation—for the oldest generation there isn't an older generation

[30] For example, in some high-performance Lisp systems, a special pointer tag signifies a pointer to a cons cell, and the cons cell itself has no header.

to copy things into. The oldest generation ("dynamic space") is therefore structured as a pair of semispaces, which are used alternately. A further enhancement is to provide a special area, called "static space," which is not garbage collected at all during normal operation. This area holds system data and compiled code that are expected to change very rarely.

Some copying collectors based on Ungar's Generation Scavenging system treat the oldest generation specially by structuring it as a single space and using a mark-compact algorithm. In these systems, all generations typically reside in RAM during normal execution, and the use of a single space reduces the RAM required to keep the oldest generation memory resident. While the mark-compact algorithm is more expensive than a typical copy collection, the ability to perform a full collection without paging makes it worthwhile for the oldest generation. Non-copying techniques can be used for the same purpose, although they are more subject to fragmentation problems.

4.3.2 Generations in Non-copying Schemes

In our discussion of generational collection thus far, we have focused primarily on copying garbage collection schemes, where generations can be viewed as "areas" of memory holding objects of different ages. This is unnecessary, however, as long as it is possible to distinguish objects of different ages and treat them differently. Just as incremental collection algorithms are best understood in terms of the abstraction of tricolor marking, generational algorithms are best understood in terms of sets of objects which are garbage collected at different frequencies. (Each of these age sets, in turn, can be divided into shaded and unshaded sets for the purposes of the tracing traversal.)

The Xerox PARC PCR (Portable Common Runtime) garbage collector is a generational mark-sweep collector, with a header field per object indicating the object's age. Objects of different generations may be allocated in the same page, although the system uses a heuristic to minimize this for locality reasons[31]. When garbage collecting only young data, the PCR collector scans the root set and traverses objects whose age fields signify that they are subject to collection. This tracing continues transitively in the usual way, tracing all reachable young objects. The generational write barrier uses pagewise dirty bits maintained by virtual memory access protection techniques; as will be explained in Sect. 4.4.3, pages of older generations dirtied since the previous collection are scanned in their entirety, and any pointers to young generations are noted and used as part of the root set.

[31] Pages containing old objects are not used to hold young objects unless they are more than half empty; this tends to avoid gratuitously mixing older and younger data.

4.3.3 Discussion

Many variations on generational collection are possible, and hybrids are common, to allow various tradeoffs to be adjusted.

It is common for copying collectors to manage large objects differently, storing them in a special *large object area* and avoiding actually copying them [CWB86]. This essentially combines copying of small objects (where it's cheap) with mark-sweep for large objects, to avoid the larger space and time overheads of copying them. (Commonly, large objects are actually represented by a small copy-collected proxy object, which holds an indirection pointer to the actual storage for the object's data fields.)

Objects known not to contain pointers may also be segregated from other objects, to optimize the tracing process and/or the scanning involved in some schemes for tracking intergenerational pointers [Lee88]; this may also improve locality of reference during tracing if copy-collected proxies are used, because the actual storage for non-pointer objects needn't be touched at all.

The ParcPlace Smalltalk-80 garbage collector combines stop-and-copy collection of the young generation (where the worst-case pause is not large) with incremental mark-sweep collection of older data.[32]

4.4 Tracking Intergenerational References

Generational collectors must detect pointers from older to younger generations, requiring a *write barrier* similar to that used by some incremental tracing algorithms. That is, a program cannot simply store pointers into heap objects—the compiler and/or hardware must ensure that each potential store is accompanied by checking or recording operations, to ensure that if any pointers to younger generations are created, they can be found later by the collector. Typically, the compiler emits additional instructions along with each potential pointer store instruction, to perform the required write barrier operations.

For many systems, this may be the largest cost of generational garbage collection. For example, in

[32] David Ungar, personal communication, 1992.

a modern Lisp system with an optimizing compiler, pointer stores typically account for one percent or so of the total instruction count [SH87, Zor89]. If each pointer store requires twenty instructions for the write barrier, performance will be degraded by roughly twenty percent. Optimizing the write barrier is therefore very important to overall garbage collector performance, and significantly faster write barriers have been developed. Because of their key role in the overall performance of a generational collector, we will discuss write barriers in some detail.

Many write barrier techniques have been used, with different performance tradeoffs on different hardware, and for different languages and language implementation strategies.

Some systems use a collection strategy that is "almost generational" but without using a write barrier, to get some of the benefit of generational collection. Rather than keeping track of pointers from old data to young data as they are created, old data are simply scanned for such pointers at collection time. This requires more scanning work, and has worse locality than true generational collection, but it may be considerably faster than tracing all reachable data. (Scanning is typically several times faster than tracing, and has strong spatial locality of reference.) The "Stratified Garbage Collector" for Austin Kyoto Common Lisp (an enhancement of Kyoto Common Lisp) uses such a scheme to avoid the overhead of a write barrier.[33] Bartlett has used a similar technique in a collector designed to work with off-the-shelf compilers which do not emit write barrier instructions [Bar89].

4.4.1 Indirection Tables

The original generational collectors for Lisp Machines [LH83] used specialized hardware and/or microcode to speed up the checks for pointers into younger generations, and the pointers that were found were indirected (by a microcoded routine) through an *entry table*. No pointers directly into a younger generation were allowed, only pointers to a table entry holding the actual pointer. Each generation had its own entry table holding the actual pointers to objects. When the mutator executed a store instruction, and attempted to create a pointer into a younger generation, the store instruction trapped to microcode. Rather than actually creating a pointer directly into the younger generation, an *invisible forwarding pointer* was created and stored instead. The Lisp Machine hardware and microcode detected and dereferenced the forwarding pointers automatically, making the indirections invisible to running programs.

[33] William Schelter, personal communication 1991.

When garbage collecting a particular generation, it was therefore only necessary to use the entry table as an additional set of roots, rather than actually finding the pointers in other generations and updating them.

A somewhat different scheme was used in the TI Explorer system; rather than having a table of incoming pointers per generation, a separate table of outgoing pointers was maintained for each combination of older and younger generation. (So, for example, the oldest generation had a separate exit table for each younger generation, holding its indirected pointers into each of those generations.) This allowed the scanning of tables to be more precise (i.e., only scanning the pointers relevant to the generations being collected), and made it simpler to garbage collect the table entries themselves.

Unfortunately, the indirection table schemes were not fast or efficient, especially on stock hardware with no support for transparently dereferencing forwarded pointers. (As with Baker-style incremental copying, the cost of common pointer operations is greatly increased if pointers must be checked for indirections.) Recent generational collectors have therefore avoided indirections, and allowed pointers directly from any generation into any other. Rather than requiring such pointers to be localized, they simply keep track of where such pointers are, so that they can be found at collection time. We refer to such schemes as *pointer recording* schemes, because they simply record the location of pointers.

4.4.2 Ungar's Remembered Sets

Ungar's *Generation Scavenging* collector used an objectwise pointer-recording scheme, recording which objects had pointers to younger generations stored into them. At each potential pointer store, the write barrier would check to see if an intergenerational pointer was being created—by checking to see if the stored value was in fact a pointer, pointed into the young generation, and was being stored into an object in the old generation. If so, the stored-into object was added to the *remembered set* of objects holding such pointers, if it was not already there. (Each object had a bit in its header saying whether it was already in the remembered set, so that duplicate entries could be avoided. This makes the collection-time scanning cost dependent on the number and size of the stored-into objects, not the actual number of store operations.)

In the usual case, this scheme worked quite well for a Smalltalk virtual machine. Unfortunately, in the worst case, this checking and recording incurred tens of instructions at a pointer store, and the relative cost would have been appreciably higher in a higher-performance language implementation.

A significant drawback of this scheme was that the remembered set of objects must be scanned in its entirety at the next garbage collection, which could be expensive for two reasons. Some of the checking cost was repeated, because a stored-into location might be stored into many times between collections, being checked each time, and because stored-into objects had to be scanned again at collection time. Worse, very large objects might be stored into regularly, and have to be scanned in their entirety at each collection. (The latter was observed to cause large amounts of scanning work—and even thrashing—for some programs running in Tektronix Smalltalk.[34])

4.4.3 Page Marking

Moon's Ephemeral Garbage Collector for Symbolics Lisp machines used a different pointer-recording scheme [Moo84]. Rather than recording which objects had intergenerational pointers stored into them, it recorded which virtual memory pages were stored into. The use of the page as the granularity of recording avoided the problem of scanning very large objects, although it increased costs for sparse writes to small objects, because the entire page would still be scanned. The scanning cost was not large on the Symbolics hardware, because it had special tag support to make generation checking very fast, and because pages were fairly small. Much of the write-barrier was also implemented directly in hardware (rather than by additional instructions accompanying each pointer write), so the time cost at each pointer store was small. In this system, the information about pointers into younger generations is held in a pagewise table. (This has the advantage that the table implicitly eliminates duplicates—a page may be stored into any number of times, and the same bit set in the table, but the page will only be scanned once at the next garbage collection. This duplicate elimination is equivalent to Ungar's use of bits in object headers to ensure uniqueness of entries in the remembered set. The time required to scan the recorded items at a garbage collection is therefore proportional to the number of stored-into pages, and to the page size, but not the number of actual store operations.)

Unfortunately, this scheme would be considerably slower if implemented on stock hardware, with larger pages and no dedicated hardware for page scanning or write checking. It also requires the ability to scan an arbitrary stored-into page from the beginning, which is more complicated on standard hardware than on the Symbolics machines, in which every machine word had extra bits holding a type tag.

More recently, virtual memory *dirty bits* have been used as a coarse write barrier [Sha88]. The underlying virtual memory system typically maintains a bit per page that indicates whether the page has been dirtied (changed in any way) since it was last written out to disk. Most of the work done to maintain these bits is done in dedicated memory hardware, so from the point of view of the language implementor, it is free. Unfortunately, most operating systems do not provide facilities for examining dirty bits, so operating system kernel modifications are required. Alternatively, virtual memory protection facilities can be used to simulate dirty bits, by write-protecting pages so that writes to them can be detected by the hardware and invoke a trap handler [BDS91]; this technique is used in the Xerox Portable Common Runtime garbage collector. The trap handler simply records that the page has been written to since the last garbage collection, and un-protects the page so that program execution can resume. (In the PCR collector, objects of different generations may reside in the same page, so when a dirtied page is scanned at collection time, objects of irrelevant generations are skipped.) As with the Appel, Ellis, and Li collector, the use of virtual memory protections makes it impossible to satisfy hard real-time requirements, and may incur significant trap overhead; scanning costs may also be relatively high if write locality is poor. As we will explain in Sect. 6.2, however, this kind of write barrier has advantages when dealing with compilers that are uncooperative and do not emit write barrier instructions.

4.4.4 Word marking

In adapting Moon's collector for standard hardware, Sobalvarro avoided the cost of scanning large pages by the use of a *word marking* system, which used a bitmap to record which particular machine words of memory actually had pointers stored into them [Sob88]. This avoided the need to be able to scan an arbitrary page for pointers, because the locations of the relevant pointers were stored exactly.

Sobalvarro also optimized the scheme for standard

[34] Patrick Caudill, personal communication 1988.

hardware by making the write barrier simpler—most write-barrier checking was eliminated, and deferred until collection time. The stored-into locations are checked at collection time to see whether the stored items are intergenerational pointers. While this is less precise than Moon's or Ungar's checking, and may cause more words to be examined at collection time, it also has the benefit that implicit duplicate elimination is performed first, and the other checks only need to be performed once per stored-into word.

The drawback of Sobalvarro's scheme is that for a reasonably large heap, the table of bits is fairly large, about three percent of the total size of memory. Scanning this table would be relatively expensive if it were represented as a simple linear array of bits. (Storing individual bits would also make the write barrier expensive on some architectures, where sub-word write instructions are slow, or must be synthesized using several other instructions.) Sobalvarro's solution to this was to use a sparse (two-level) representation of the table; this incurred an additional write-barrier cost, because operations on the sparse array are significantly slower than operations on a contiguous array.

4.4.5 Card Marking

An alternative to marking pages or words is to conceptually divide memory into intermediate-sized units called *cards* [Sob88]. The use of relatively small cards has the advantage that a single store operation can only cause a small amount of scanning at collection time, making the cost smaller than page-marking on average. As long as the cards aren't extremely small, the table used for recording stored-into cards is much smaller than the corresponding table for word marking. For most systems, this makes it feasible to represent the table as a contiguous linear array, keeping the write barrier fast.

One problem of using card marking on standard hardware is that it requires that cards be scanned for pointers, even if the card does not begin with the beginning of an object. Wilson's Opportunistic Garbage Collector addresses this by maintaining a *crossing map*, recording which cards begin with an unscannable part of an object [WM89]. In the case of a card that's not scannable from the beginning, the map can be used to find a previous card that is scannable, locate an object header on that card, and skip forward object by object until it finds the headers of the objects on the card to be scanned. (This is a refinement of the crossing maps used by Appel Ellis and Li to support pagewise scanning in their incremental copying collector [AEL88]). In Wilson's scheme, the bit corresponding to a card is left set if the card contains a pointer into a younger generation. Such cards must be scanned again at the next garbage collection, even if they are not stored into again.

Ungar, Chambers, and Hölzle have further refined this card-marking scheme in a garbage collector for the Self language. Rather than using a table of bits to record stored-into cards, it uses a table of bytes—even though only a bit is needed, a byte is used because byte stores are fast on most architectures. This allows the write barrier to consist of only three instructions, which unconditionally store a byte into a byte array. This comes at an increase in scanning costs, because the byte array is eight times larger than a bit array, but for most systems the decrease in write-barrier cost is well worth it [Cha92, DMH92]. Hölzle has further refined this by relaxing the precision of the write barrier's recording, bringing the cost per store down to two instructions (on a Sun SPARC processor) with a slight increase in scanning costs [H9̈3]. (As we will explain in the next section, card marking can also be combined with *store lists* to reduce scanning of cards which hold pointers into younger generations, but aren't stored into again.)

4.4.6 Store Lists

The simplest approach to pointer recording is simply to record each stored-into address in a list of some sort. This might be a linked list, or a pre-allocated array that has successive locations stored into, much like pushing items on a linear-array stack.

Appel's very simple (500 lines of C code) generational collector for Standard ML of New Jersey [App89b] uses such a list, which is simply scanned at each collection, with good performance for typical ML programs.

Simple store lists have a disadvantage for many language implementations, however, in that they implement bags (multisets) of stored-into locations, not sets. That is, the same location may appear in the list many times if it is frequently stored into, and the garbage collector must examine each of those entries at collection time. The collection-time cost is therefore proportional to the number of pointer stores, rather than to the number of stored-into locations. This lack of duplicate elimination can also lead to excessive space usage if pointer stores are very frequent. (For ML, this is generally not a problem, because side effects are used relatively infrequently.)

Moss et al. have devised a variation of the store list technique which allows a bounded number of entries in a special kind of list called a *static store buffer*, and calls a special routine when this buffer is full. The special routine processes the list, using a fast hash table to remove duplicates. This technique [HMS92]reduces space costs, and avoids doing all of the store-list processing at garbage collection time, but it does not have the same duplicate elimination advantage as the table-based schemes—duplicates are eliminated, but only after they've already been put in the store list. Each pointer store creates an entry in the store list, which must be fetched and examined later.[35]

Hosking and Hudson [HH93] have combined some of the best features of card marking and store lists. Pointer stores are recorded by a card-marking write barrier in the usual way, but when a card is scanned, the individual locations containing pointers into younger generations are recorded. This allows subsequent collections to avoid re-scanning whole cards if they are not stored into again.

4.4.7 Discussion

In choosing a write barrier strategy for a generational collector, it is important to take into account interactions with other aspects of the system's implementation. For example, the Xerox PARC "mostly-parallel" collector uses virtual memory techniques (to implement pagewise dirty bits) partly because it is designed to work with a variety of compilers which may not cooperate in the implementation of the write barrier—e.g., off-the-shelf C compilers do not emit write-barrier instructions along with each pointer store. In other systems, especially systems with static type systems and/or type inference capabilities, the compiler can significantly reduce the cost of the write barrier, by omitting the write barrier checks that can be done statically by the compiler.

Another issue is whether real-time response is required. Table-based schemes such as page marking and card marking may make it difficult to scan the recorded pointers incrementally in real-time. Store lists are easier to process in real time; in fact, the work done for the write barrier may be similar to the work done for an incremental update tracing technique, allowing some of the costs to be optimized away [WJ93] by combining the two write barriers.

[35] Ungar's technique of using flag bits (to signify whether an object is already in a set) could conceivably be used, but it is probably not worthwhile to use a bit per memory word, as opposed to a bit per object header, unless there is hardware support to make it fast.

The actual cost of write barriers is somewhat controversial. Several studies have measured write barrier overheads for interpreted systems (e.g., [Ung84, HMS92]), making them hard to relate to high-performance systems using optimizing compilers [Moo84, Ḧ93]. It may be more reasonable to combine measurements of high-performance systems with an analytic understanding of garbage collector costs, to infer what the approximate cost of a well-implemented collector would be for a well-implemented systems.

As mentioned earlier, compiled Lisp systems appear to execute roughly one pointer store into a heap object per hundred instructions; a card-marking write barrier should only slow such a system down by about four or five percent, executing two or three instructions at the time of each pointer store, plus a smaller card-scanning cost at each collection. For many programs (with little live data, or lifetime distributions favorable to generational collection), the tracing and reclamation cost will be similarly low, and the cost of garbage collection should be under ten percent.

This figure can vary considerably, however—and often upward—based on the workload, type information in the programming language, data representations, and optimizations used by the compiler. (Several implementation choices will be considered in later sections.)

If a compiler generates faster code, the write barrier cost may become a larger fraction of the (smaller) overall running time. On the other hand, the compiler may also be able to reduce write barrier costs by inferring that some pointer recording is redundant, or that some dynamically-typed values will never be pointers (Sect. 6.4.3).

Unfortunately, the cost of write barriers in conventional imperative statically-typed systems is poorly understood. Static type systems generally distinguish pointer and nonpointer types, which may help the compiler, but type declarations may improve other areas of the system's performance even more, making the relative performance of the garbage collector worse. On the other hand, conventional statically- and strongly-typed languages often have lower overall rates of heap object allocation and mutation, reducing both the write barrier and tracing costs as a fraction of overall program running time.

Programming style can also have a significant impact on write-barrier costs. In many languages designed for use with garbage collection, allocation routines are primitives which take values as arguments, and initialize fields of objects. In other languages, the programmer may be expected to initialize a new objects' fields explicitly. In the former case, the language implementation can omit the write barrier for initializing writes to pointer fields, but in the latter it generally cannot. It is not clear whether this is a problem; programs in such languages may typically have lower rates of heap allocation and fewer pointer fields in objects.

4.5 The Generational Principle Revisited

Generational garbage collectors exploit the fact that heap-allocated objects are typically short-lived. Longer-lived objects are collected less often, on the assumption that the minority objects which live for a significant period are likely to live a while longer still. This simple idea is widespread, but it is not obvious that it is true in any strong sense [Hay91, Bak93a]; it is also unclear that it must be true to make generational collection worthwhile in practice.

Consider a system in which this property does not hold—i.e., the probability that an object will die at a particular moment is *not* correlated with its age. Lifetimes distributions in such a system may still be heavily skewed toward short-lived objects. A simple example of such a system is one with a random *exponential decay* property, where a fixed fraction of the objects die in a fixed period of time, much like the "half-life" property of radioactive isotopes.

In such a system, the lifetime distribution may appear ideally suited to generational collection, because young objects die young. On closer examination, however, it turns out that picking *any* subset of the objects will yield an equal proportion of live and dead objects over a given period of time. In that case, any advantage of generational collection would be due to restricting the scope of collection, not to a higher mortality rate among the objects subject to collection.

This analogy appears to undermine the notion that generational collection is a good idea, but in fact it may not. Even under an exponential decay model, generational collection may improve *locality*, despite the fact that it won't directly improve algorithmic efficiency—reclaiming and reusing recently-allocated space improves locality, compared to reusing memory that has been idle for a significant period. Traversing live objects is likely to be cheaper if the objects were allocated—hence touched—recently; reclaiming and reusing space of garbage objects is also likely to be cheaper because that space will have been touched recently as well.

4.6 Pitfalls of Generational Collection

Generational collection attempts to improve performance heuristically, taking advantage of characteristics of typical programs; naturally, this cannot be successful for all programs. For some programs generational collection will fail to improve performance, and may decrease it.

4.6.1 The "Pig in the Snake" Problem

One problematic kind of data for generational collection is a cluster of relatively long-lived objects, which are created at about the same time and persist for a significant period. This often occurs because data structures are built during one phase of execution, then traversed during subsequent phases of execution, and become garbage all at once (e.g., when the root of a large tree becomes garbage). This kind of data structure will be copied repeatedly, until the advancement policy succeeds in advancing it to a generation large enough to hold the whole cluster of related data objects. This increases traversal costs first in the youngest generation, then in the next generation, and so on, like the bulge in a snake advancing along the snake's body after a large meal.[36] Until the bulge is advanced to a generation that will hold it until it dies, the collector's age heuristic will fail and cause additional tracing work.

The pig-in-the-snake problem therefore favors the use of relatively rapid advancement from one generation to the next, which must be balanced against the disadvantage of advancing too much data and forcing the next generation to be collected more often than necessary. Ungar and Jackson vary the advancement policy dynamically in an attempt to advance large clusters of data out of the youngest generation before they incur too much copying cost; this appears to work well for most programs, but may cause the older generation to fill rapidly in some cases. Wilson advocates the use of more generations to alleviate this problem, along with careful "opportunistic" scheduling of garbage collections in an attempt to collect when little data is live [WM89]. Hayes's "key object" opportunism refines this by using changes in the root set to

[36] Incidentally, this term was inspired by a similar description of the baby boom generation's use of resources through its life cycle—first requiring more kindergartens, then elementary schools, and so on, and ultimately causing an increase in demand for retirement homes and funeral directors. (Jon L. White, personal communication 1989.)

influence garbage collection policy [Hay91]. (These techniques appear to be beneficial but more experimentation is needed in larger systems.)

4.6.2 Small Heap-allocated Objects

One of the assumptions behind the generational heuristic is that there will be few pointers from old objects to young ones; some programs may violate this assumption, however. One example is programs that use large arrays of pointers to small, heap-allocated floating point numbers. In many dynamically-typed systems, floating point numbers do not fit in a machine word, and in the general case must be represented as tagged pointers to heap-allocated objects. Updating the values in an array of floating point numbers may actually cause new floating point objects to be allocated, and pointers to them to be installed in the array. If the array is large and not short-lived, it is likely to reside in an older generation, and each update of a floating point value will create a young object and an intergenerational pointer. If a large number of elements of the array are updated (e.g., by a sequential pass through the whole array), each floating point number object is likely to live a significant period of time—long enough to be traced by several youngest-generation collections, and advanced to an older generation. Thus these large numbers of intermediate-lifetime number objects will cause considerable overhead, both in the write barrier and in tracing. (In the case of systems using Ungar's original remembered set scheme, the remembered set scanning costs may be very large if large objects hold intergenerational pointers.)

To avoid these costs, two approaches are common—the use of short floating-point formats which can be represented as tagged immediate values, and the use of arrays with typed fields, which can contain raw floating point values rather than pointers to heap-allocated objects.

The problem with short floating point values is that they may not have the desired numerical characteristics. In the first place, numbers short enough to fit in a machine word may not have sufficient precision for some applications. In addition, some bits must be sacrificed for the tag field, further reducing the precision or the range of the number. The simplest scheme is to sacrifice bits of precision by removing bits from the mantissa, but this has the problem that the resulting number does not map well onto typical floating-point hardware, which has very carefully designed precision and rounding characteristics. (Without the expected rounding properties, some algorithms may compute incorrect answers, or even fail to terminate. Simulating these characteristics in software is extremely expensive.) The alternative is to sacrifice bits from the exponent part of the hardware-supported format, and restrict the range of numbers that can be represented. This introduces several instructions extra overhead in converting between the hardware-supported format and the tagged format [Wil90, Cha92].

Using arrays with typed fields introduce irregularities into dynamically-typed systems (e.g., most arrays can hold any kind of data, but some can't), but this strategy is easy to implement efficiently, and is frequently used in Lisp systems.

Unfortunately, neither of these solutions fixes the problem in the general case, because floating point numbers are not the only possible data type that can cause this problem. Consider complex numbers, or 3D point objects: it's unlikely that such objects are going to be crammed into a machine word. Similarly, the problem doesn't only occur with arrays—any aggregate data structure (such as a binary tree) can exhibit the same problem. In such cases, the programmer may choose to use a different representation (e.g., parallel arrays of real and imaginary components, rather than an array of complex numbers) to avoid unnecessary garbage collection overhead.

4.6.3 Large Root Sets

Another potential problem with generational collection is the handling of root sets. (This is essentially the same problem that occurs for incremental collection.) Generational techniques reduce the scope of tracing work at most collections, but that does not in itself reduce the number of roots that must be scanned at each collection. If the youngest generation is small and frequently collected, and global variables and the stack are scanned each time, that may be a significant cost of garbage collection in large systems. (Large systems may have tens of thousands of global or module variables.)

An alternative is to consider fewer things to be part of the usual root set, and treat most variables like heap objects, with a write barrier. Stores into such objects that may create pointers into young generations are then recorded in the usual way, so that the pointers can be found at collection time. The problem with this approach is that it may significantly increase the cost of stores into local variables. (This cost can be reduced if the compiler can determine types at compile time, and omit the write barrier code for nonpointer writes.)

Treatment of local variables is more complicated in languages that support closures—procedures which can capture local variable binding environments, forcing local variables to be allocated on the garbage collected heap, rather than a stack. If all variable bindings are allocated on the heap, this requires that pointer stores into local variables use a write barrier; this may significantly increase write barrier costs for many programs, where side-effects to local variables are relatively common. For such systems, it is desirable to have compiler optimizations which avoid the heap allocation of local variables that are never referenced from closures and hence can be allocated on a stack or in registers. (Such techniques are valuable in themselves for improving the speed of code operating on those variables [Kra88], as well as for reducing overall heap allocation.)

In most application programs, root set scanning time is negligible, because there are only a few thousand global or module variables. In large, integrated programming environments, however, this root set may be very large; to avoid large amounts of scanning at every garbage collection, it may be desirable to use a write barrier for some variables, so that only the stored-into variables are actually scanned at collection time. It is easy to imagine a large integrated development system which uses a write barrier for most variables, but which can also generate stripped-down standalone application code for which the root set is scanned atomically to avoid the write-barrier cost when programs are distributed.

4.7 Real-time Generational Collection

Generational collection can be combined with incremental techniques, but the marriage is not a particularly happy one [WJ93]. Typically, real-time garbage collection is oriented toward providing absolute worst-case guarantees, while generational techniques improve expected performance at the expense of worst-case performance. If the generational heuristic fails, and most data are long-lived, garbage collecting the young generation(s) will be a waste of effort, because no space will be reclaimed. In that case, the full-scale garbage collection must proceed just as fast as if the collector were a simple, non-generational incremental scheme.

Real-time generational collection may still be desirable for many applications, however, provided that the programmer can supply guarantees about object lifetimes, to ensure that the generational scheme will be effective. Alternatively, the programmer may supply weaker "assurances," at the risk of a failure to meet a real-time deadline if an assurance is wrong. The former reasoning is necessary for mission-critical hard-real time systems, and it is necessarily application-specific. The latter "near-real-time" approach is suitable for many other applications such as typical interactive audio and video control programs, where the possibility of a reduction in responsiveness is not fatal.

When it is desirable to combine generational and incremental techniques, the details of the generational scheme may be important to enabling proper incremental performance. For example, the (Symbolics, LMI, and TI) Lisp machines' collectors are the best-known "real-time" generational systems, but the interactions between their generational and incremental features turn out to have a major effect their worst-case performance.

Rather than garbage collecting older generations slowly over the course of several collections of younger generations, only one garbage collection is ongoing at any time, and that collection collects only the youngest generation, or the youngest two, or or the youngest three, etc. That is, when an older generation is collected, it and all younger generations are effectively regarded as a single generation, and garbage collected together. This makes it impossible to benefit from younger generations' generational effect *while* garbage collecting older generations; in the case of a full garbage collection, it effectively degenerates into a simple non-generational incremental copying scheme.

During such large-scale collections, the collector must operate fast enough to finish tracing before the available free space is exhausted—there are no younger generations that can reclaim space and reduce the safe tracing rate. Alternatively, the collection speed can be kept the same, but space requirements will be much larger during large scale collections. For programs with a significant amount of long-lived data, therefore, this scheme can be expected to have systematic and periodic performance losses, even if the program has an object lifetime distribution favorable to generational collection, and the programmer can provide the appropriate guarantees or assurances to the collector. Either the collector must operate at a much higher speed during full collections, or memory usage will go up dramatically. The former typically causes major performance degradation because the collector uses most of the CPU cycles; the latter either requires very large amounts of memory—negating the advantage of generational collection—or incurs performance degradation due to virtual memory paging.

5 Locality Considerations

Garbage collection strategies have a major effect on the way memory is used and reused; naturally, this has a significant effect on locality of reference.

5.1 Varieties of Locality Effects

The locality effects of garbage collection can be roughly divided into three classes:

- Effects on programming style which change the way data structures are created and manipulated
- Direct effects of the garbage collection process itself, and
- Indirect effects of garbage collection, especially patterns of reallocation of free memory and clustering of live data.

The first of these—effects of programming style—is poorly understood. In systems with an efficient garbage collector, programmers are likely to adopt a programming style that is appropriate to the task at hand, often an object-oriented or functional approach. New data objects will be dynamically allocated to hold newly computed data, and the objects will be discarded when the data are no longer interesting. Ideally, the programmer expresses the computation in the most natural form, with application-level data mapping fairly directly onto language-level data objects.

In contrast, explicit allocation and deallocation often encourage a distorted programming style where the programmer reuses language-level objects to represent conceptually distinct data over time, simply because it's too expensive to deallocate an object and reallocate another one. Similarly, in systems with inefficient garbage collection (such as many older Lisp implementations) programmers often resort to similar language-level object reuse, for example destructively side-effecting list structures to avoid allocating new list elements, or allocating a single large array used to hold several sets of data over time. Explicit deallocation often leads to distortions of the opposite variety, as well—mapping a single conceptual data object onto multiple language-level objects. Programmers may allocate many *extra* objects to simplify explicit deallocation. Any module interested in a data structure may copy the data structure, so that it can make a local decision as to when its memory can be reclaimed. Such distortions make it extremely difficult to compare the locality of garbage collected and non-garbage-collected systems directly. (Typical studies that attempt to do so compare use programs written without garbage collection in mind, in both their original form and with explicit deallocation replaced by garbage collection. This implicitly hides the effects of distorted programming style, because the garbage collected version of the program inherits the distortions.)

The second category of locality effects—locality of the garbage collection process itself—is the one that usually comes immediately to mind. It is sometimes quite significant, although it may be the least important of the three. For a full garbage collection, all live data must be traced, and this can interact very poorly with conventional memory hierarchies. Most live objects will be touched only once during tracing, so there will be little *temporal* locality, i.e., few repeated touches to the same data over a short period of time. On the other hand, there may be considerable *spatial* locality—touches to several different nearby areas of memory (e.g., within the same virtual memory page) over short periods of time.

The third category of effects is probably the most important, although its significance is not widely appreciated. The strategy for memory *reallocation* imposes locality characteristics on the way *memory* is touched repeatedly, *even if the objects themselves die quickly and are therefore never touched again.* This is, of course, one of the main reasons for generational garbage collection—to reuse a small area of memory (the youngest generation) repeatedly by allocating many short-lived objects there. It is also the reason that activation stacks typically have excellent locality of reference—near the top of the stack, memory is reused very frequently by allocating many very short-lived activation records there. (A generational garbage collector can be seen as imposing a roughly stack-like memory reuse pattern on a heap, by exploiting the fact that the lifetime distributions are roughly similar.)

As we pointed out earlier, a simple, non-generational garbage collector has very poor locality if allocation rates are high, simply because too much memory is touched in any reasonable period of time. As Ungar has pointed out [Ung84], simply paging out the resulting garbage (to make room in memory for new data) would typically be an unacceptable cost in a high-performance system. A generational garbage collector restricts the scope of this locality disaster to a manageable area—the youngest generation or two. This allows the "real" locality characteristics (of repeated access to longer-lived data) to show up in the older generation(s). Because of this effect, the overall locality characteristics of garbage collected systems appear to be roughly comparable to that of non-garbage collected systems—the youngest generation filters out most of the heap-allocated data that might be stack-allocated in other languages.

Direct comparisons are difficult, however, because of the large number of design choices involved in both garbage collectors and explicit heap management.

With copying garbage collection, there is obviously another indirect locality effect—by moving data objects around in memory, the collector affects the locality of the program's own accesses—that is, it affects the mapping between a program's logical references to language-level data and the resulting references to particular memory locations.

This kind of effect is not restricted to copying collection. Noncopying collectors also have considerable latitude in deciding how to map language-level objects onto the available free memory. When the program requests a piece of memory of a given size, the allocator is free to return *any* suitably-sized piece of free memory. It is unclear what rules should govern this decision.[37]

The importance of different locality effects is, naturally, dependent on the relative size of the data used by a system and the main memory of the hardware it runs on. For many systems, the full system heap fits in main memory, including an editor, a browser, a compiler, and application code and data; the normal mode of such systems is to have enough RAM that programs typically do not page at all. In other systems, however, the oldest generation or two is too large to fit in typical main memories, either because the system itself includes large complex software, or because application data or code are very large.

In a copying collector, there is therefore a binary distinction between generations that are effectively memory-resident and those that are so large they must truly rely on virtual memory caching. For the former, which may include the whole system, locality of allocation and collection are essentially irrelevant to virtual memory performance—the space cost is the fixed cost of keeping everything in RAM. For the latter, in contrast, locality at the level of virtual memory may be crucial.

In a nonmoving collector, the situation is somewhat different—the space cost of the youngest generation also depends on the degree of fragmentation and how data from various generations are intermingled in memory. The extent of these problems is not well understood, but they may be less serious than is widely believed [Hay91, BZ93, Boe93].

5.2 Locality of Allocation and Short-lived objects

As noted above, the pattern of allocation often has the most important effect on locality in a simple collector. In a generational copy collector, this effect is much reduced from the point of view of virtual memory—the pages that make up the youngest generation or two are reused so frequently that they simply stay in RAM and effectively incur a fixed space cost. On the other hand, the frequent reuse of the whole youngest generation may have a deleterious effect on the next smaller level of the memory hierarchy—high-speed CPU caches. The cycle of memory reuse has been made much smaller, but if the cycle does not fit in cache, the cache will suffer extra misses in much the same way that main memory does for a simple collector. The effects of such misses are not as dramatic as those for virtual memory, however, because the ratio of cache to main memory speeds is not nearly as large as the ratio of main memory to disk speeds. Also, in copying collection at least, the pattern of reallocation is so strongly sequential that misses can be reduced considerably by simple prefetching strategies, or even just the use of large block sizes. On current processors, the cost appears to be no more than a few percent of overall run time, even when allocation rates are relatively high. Faster processors may suffer more from this effects, however, if cache-to-memory bandwidths do not scale with processor speeds, or if bus bandwidth is at a premium as in shared-bus multiprocessors.

Zorn has shown that relatively large caches can be quite effective for generationally garbage-collected systems [Zor89]. Wilson has shown that the relative sizes of the cache and the youngest generation are especially important, the particulars of the cache replacement policy may be important as well, due to peculiarities in the access patterns due to reallocation [WLM92].[38]

[37]This occurs in explicit heap management systems as well, of course, but it has not been systematically studied there either. Most studies of explicit techniques have studied fragmentation and CPU costs, but have ignored effects on caching in hierarchical memories.

[38]The effect of associativity is very dependent of the ratio of cache size to youngest generation size, and lower associativities

Several researchers have suggested optimizations to avoid the fetching of garbage data in areas about to be re-allocated; this can cut cache-to-memory bandwidth requirements nearly in half. Koopman *et al.* first illustrated this in a functional programming language implementation based on combinator reduction [KLS92], and showed that some conventional cache designs can achieve this effect.[39] Tarditi and Diwan show that the same effect can be achieved in a more conventional language implementation using generational garbage collection, and demonstrate the value of a cache-to-memory interface supporting high write rates [DTM93].

The difference in locality between moving and non-moving collectors does not appear to be large at the scale of high-speed cache memories—the type of collector is not as important as rate of allocation and the size of the youngest generation, i.e., how quickly memory is used, reclaimed and reused in the usual case. [Zor90] shows that a non-copying collector can have better locality than a copying collector using semi-spaces, simply because it only needs a single space per generation; [WLM92] shows that Ungar's technique of using a special creation area can yield similar benefits at the level of large caches, just as it does at the level of virtual memory. [WLM92] also shows that the allocation of variable binding environments and activation records on the heap can greatly exacerbate cache-level locality problems due to a youngest generation that won't fit in the cache. This is borne out by simulation studies of Standard ML of New Jersey [DTM93] on high-performance processors. It suggests that activation information and binding environments should be allocated on a stack using compile time analysis [KKR+86] or in a software stack cache [CHO88, WM89, Kel93]. (Software stack caches can be used in languages like ML and Scheme, where binding environments may be captured by first-class procedures and/or activation chains may be captured with first-class continuations. The cache takes advantage of the fact that while environments and continuations can be captured, the vast majority of them are not and needn't be put on the heap.)

may actually perform better when the two are nearly equal.

[39] The essential feature is the use a of a *write-allocate* policy, in combination with *sub-block placement*. Write-allocate means that if a block is written to without first writing, it is allocated a block of cache memory, rather than having the write bypass the cache and simply update the block out in main memory. This ensures that when objects are allocated (and hence written), they will be in cache when they are referenced shortly thereafter. Sub-block placement means that the cache block is divided into several independent lines which can be valid or invalid in the cache. This allows writes to one word to avoid triggering a fetch of the rest of the block. Objects can thus be allocated and initialized without stalling the processor and fetching the old contents of the storage they occupy.

5.3 Locality of Tracing Traversals

The locality of GC tracing traversals is difficult to study in isolation, but it appears to have some obvious characteristics. Most objects in the generation(s) being collected will be touched exactly once, because most objects are pointed to by exactly one other object [Rov85, DeT90]—typical data structures do not contain a large number of cycles, and many cycles are small enough to have little impact on traversal locality.

Given this, the main characteristic of the traversal is to exhaustively touch all live data, but for the most part very briefly. There is very little *temporal* locality of reference i.e., repeated touching of the same data. (Most objects are referenced by exactly one pointer at any given time, and will therefore only be reached once by the tracing traversal.) The major locality characteristic that can be exploited is the *spatial* locality of data structures layouts in memory—if closely-linked objects are close to each other in memory, touching one object may bring several others into fast memory shortly before they are traversed.

Experience with the Xerox PCR system indicates that even in a non-copying collector (i.e., without compaction) there is useful locality in objects' initial layouts in memory; related objects are often created and/or die at about the same time, so simple allocation strategies result in useful clustering in memory. The PCR collector enhances this by sorting its root set before traversing data, so that root pointers into the same area are traversed at about the same time. This has been observed to significantly reduce paging during full garbage collections.[40]

In a copying collector, it would seem that traversals usually have good spatial locality, in that objects are typically organized by the traversal ordering when they are first copied, and then are traversed in the same order by subsequent traversals. (At the first traversal, of course, the match between objects' initial allocation layout and the traversal order may also be significant.)

Because of the high spatial locality and low temporal locality, it may be desirable to limit the memory used by a tracing traversal, to avoid needlessly displacing the contents of memory by large amounts

[40] Carl Hauser, personal communication 1991.

of data that are only briefly touched during tracing [Wil90, Bak91a]. Incremental tracing may yield some of the same benefit, by allowing the mutator to touch data during the tracing phase, keeping the most active data in fast memory.[41]

5.4 Clustering of Longer-Lived Objects

Several studies have addressed copying collection's indirect effect on locality—i.e., the effect of reorganizing the data which are subsequently accessed by the running program.

5.4.1 Static Grouping

Stamos [Sta82, Sta84] Blau [Bla83] studied the effects of using different copying traversal algorithms to reorganize long-lived system data and code in Smalltalk systems. While these algorithms reorganize data during program execution (i.e., at garbage collection time), they are referred to as *static grouping* algorithms because they reorganize data according to how objects are linked at the time garbage collection occurs—clustering is not based on the dynamic pattern of the program's actual accesses to data objects.

Both studies concluded that depth-first reorganization was preferable to breadth-first reorganization, but not by a large margin; there is useful locality information in the topology of data structures, but any reasonable traversal will do a decent job of organizing the data. (Breadth- and depth-first traversals both dramatically outperform a random reorganization.)

Wilson *et al.* performed a similar study for a Lisp system [WLM91], and showed that traversal algorithms can make an appreciable difference. The most important difference in those experiments was not between traversal algorithms *per se*, however, but in how large hash tables were treated. System data are often stored in hash tables which implement large variable binding environments, such as a global namespace, or a package. Hash tables store their items in pseudo-random order, and this may cause a copying collector to reach and copy data structures in a pseudo-random fashion. This greatly reduces locality, but is easy to avoid by treating hash tables specially.[42] Once hash tables are treated properly, further locality gains can be made by using an algorithm that clusters data structures hierarchically. (Similar results were reported for the Symbolics Lisp Machine system, in a master's thesis by D.L. Andre [And86]. While those experiments were not particularly well-controlled, the results were strikingly positive, and this is particularly significant because the results were obtained for a large, commercial system.)

5.4.2 Dynamic Reorganization

In 1980, White proposed a system in which garbage collection was deferred for long periods of time, but in which Baker's read-barrier incremental copier was enabled to copy data for its locality effects [Whi80]; the goal was not to reclaim empty space, but instead to simply cluster the active data so that it could be kept in fast memory. One interesting property of this scheme is that it reorganizes data in the order in which they are touched by the mutator, and if subsequent access patterns are similar, it should greatly improve locality.

This scheme is impractical in its original form (because the sheer volume of garbage would swamp the write bandwidth of typical disks if memory were not reclaimed [Ung84]), but the same basic idea has been incorporated into the garbage collector of the Texas Instruments Explorer Lisp machines [Cou88, Joh91]. This collector avoids performing exhaustive background scavenging until toward the end of the garbage collection cycle, to enhance the odds that objects will be reached first by the mutator, and copied in a locality-enhancing order.

A similar approach has been used in simulations by the MUSHROOM project at the University of Manchester [WWH87]. Rather than relying on the garbage collector, however, this system is triggered by cache misses; it can therefore respond more directly to the locality characteristics of a program, rather than to the interaction between the program and the garbage collector.

Unfortunately, such schemes rely on specialized hardware to be worthwhile. The Explorer system exploits Lisp machines' hardware support for a Baker-style read barrier, and the MUSHROOM system is

[41] This benefit may not be large relative to the additional space cost of incremental collection—the deferred reuse of memory that can't be reclaimed until the end of the incremental tracing phase.

[42] One technique is to modify the hash tables' structure to record the order in which entries are made, or impose some other nonrandom ordering, and then modify the collector to use this information to order its examination of the entries. Another technique is to make hash tables indirect indexes into an ordered array of entries; this has the advantage that it can be implemented without modifying the collector, and can therefore be used for user-defined table structures.

based on a novel "object-oriented" architecture.

Wilson [Wil91] casts these techniques as a form of adaptive prefetching, and argues that as memories continue to grow, such fine-grained reorganization may be overkill; reorganization of virtual memory pages within larger units of disk transfer may yield good results on stock hardware. (This is supported by data such as those from the LOOM object-oriented virtual memory for Smalltalk [Sta82], which show that finer-grained caching strategies are helpful primarily when memories are very small. As memories get larger, the optimal unit of caching gets larger as well, and pagewise schemes tend to work roughly as well as objectwise schemes.) Wilson also argues that the improvements due to fine-grained reorganization may be mostly due to deficiencies in the static-graph algorithms used for comparison—in particular, treatment of large hash tables. However, Llames has reported [Lla91] that dynamic reorganization can significantly improve locality, even after roots are treated appropriately and a good background scavenging traversal is used.[43]

5.4.3 Coordination with Paging

Several systems have coordinated the garbage collector with the virtual memory system to improve paging performance.

The Symbolics Lisp machines have had perhaps the most comprehensive coordination of garbage collection with virtual memory. The Symbolics allocator could notify the virtual memory system when a page no longer contained any useful data, allocate pages of virtual memory without paging their old (garbage) contents into memory when the page was first touched.

Virtual memory cooperation was also used in the intergenerational pointer recording mechanism. Before paging out a page holding pointers to younger generations, the page was scanned and the intergenerational pointers found [Moo84]. This allowed the garbage collector to avoid paging in data just to scan them for pointers into younger generations. Virtual memory mapping techniques were also used to optimize the copying of large objects—rather than actually copying the data within a large object, the pages holding the data could simply be mapped out of the old range of virtual addresses and into the new range [Wit91].

[43] Llames added dynamic grouping to Moon's Ephemeral Garbage Collector, which uses the static grouping techniques described in [And86].

More recently, microkernel operating systems have offered the ability to modify virtual memory policies without actually modifying the kernel. The kernel calls user-specified routines to control the paging of a process, rather than hard-coding the entire paging policy into the kernel itself. In the Mach system, for example *external pager* processes can be used to control paging activity; this feature has been used to reduce paging for Standard ML of New Jersey in ways similar to those used in the Symbolics system [Sub91].

6 Low-level Implementation Issues

So far, we have mostly discussed basic issues of garbage collector design, and basic performance tradeoffs. In addition to these primary considerations, a garbage collector designer is faced with many design choices which can have a significant impact on the ultimate performance of the system, and on how easily the garbage collector can be integrated with other components of the system. In this section we discuss these low-level implementation questions in more detail.

6.1 Pointer Tags and Object Headers

For most of this paper, we have assumed that pointers are tagged with small tags and that pointed-to objects have header fields that encode more specific type information; this specific type information can be used to determine objects' layouts, including the locations of embedded pointer fields. This is the most common scheme in dynamically-typed languages such as Lisp and Smalltalk. It is common in such languages that objects are divided into two major categories: objects that can be stored within a machine word as tagged immediate values, and objects which are allocated on the heap and referred to via pointers. Heap-allocated objects are often further divided into two categories: those which contain only tagged values (immediates and pointers) which must be examined by the collector to find the pointers, and those which contain only nonpointer fields which can be ignored by the garbage collector.

Another possibility is that each field contains both the object (if it's a small immediate value) or pointer, *and* the detailed type information. This generally requires fields to be two words—one word long enough to hold a raw pointer or raw immediate, and another to hold a bit pattern long enough to encode all of the types in the system. Generally, a whole word is used for the latter field, because of alignment constraints for load and store operations.[44] Despite the waste of space, this scheme may be attractive on some architectures, especially those with wide buses.

For a language with a static type system, still another possibility is that all objects have headers, and that pointer fields contain no tags. (This requires a static type system which ensures that immediate values can't be stored in the same fields as pointers—if they could, it would require a tag to tell the difference.) Simply knowing which fields of objects may contain pointers is sufficient, if the pointed-to objects have headers to decode *their* structure. Some dynamically-typed systems use this representation as well, and avoid having immediate values within a word—even short integers are represented as objects with headers.[45]

If strictly static typing is used, even the headers can be omitted—once a pointer field is found, and the type of the pointer is known, the type of the object it points to is obvious [App89a, Gol91]. To allow tracing traversals, it is only necessary that the types of the root pointers (e.g., local variables in an activation stack) be known. (This can be accomplished in several ways, as we will explain later.) From there, we can determine the types of their referents, and thus their referents' pointer fields, and so on transitively. Still, some systems with static typing put headers on objects anyway, because the cost is not that large and it simplifies some aspects of the implementation.[46]

The choice of tag and pointer schemes is usually made with some regard to garbage collection, but most often the main consideration is making normal program operations efficient. Tagging schemes are usually chosen primarily to make type dispatching, arithmetic operations, and pointer dereferencing as fast as possible; which scheme is best depends largely on the language semantics and the strategy for ensuring that the most frequent operations are fast [KKR+86, GG86, SH87, Ros88, Gud93].

In some systems, individual objects do not have headers, and type information is encoded by segregating objects of particular types into separate sets of pages. This "big bag of pages" or *BiBOP* technique associates types with pages, and constrains the allocator to allocate objects in the appropriate pages. A type test requires masking and shifting a pointer to derive the page number, and a table lookup to find the type descriptor for objects in that page. BiBOP encoding can save space by letting a tag per page suffice to encode the types of many objects.

Another variation on conventional tagging schemes is to avoid putting object headers directly on the objects, and to store them in a parallel array; this may have advantages for locality of reference by separating out the data relevant to normal program operation from those that are only of interest to the collector and allocator.

6.2 Conservative Pointer Finding

An extreme case of catering to other aspects of a language implementation is *conservative pointer-finding*, which is a strategy for coping with compilers that don't offer *any* support for runtime type identification or garbage collection [BW88].[47] In such a system, the collector treats anything that *might* be a pointer as a pointer—e.g., any properly-aligned bit pattern that could be the address of an object in the heap. The collector may mistake other values (such as an integer with the same bit pattern) for pointers, and retain objects unnecessarily, but several techniques can be used to make the probability of such mistakes very small. Surprisingly, these techniques are effective enough that most C programs can be garbage collected fairly efficiently, with little or no modification [Boe93]. This simplifies the garbage collection of programs written without garbage collection in mind, and programs written in multiple languages, some of which are uncooperative [WDH89].

(Making such a collector generational requires special techniques, due to the lack of compiler cooperation in implementing a write barrier to detect intergenerational pointers. Virtual memory dirty bits or access-protection traps can be used to detect which pages are written to, so that they can be scanned at collection time to detect pointers into younger generations [DWH+90].)

Conservative pointer finding imposes additional constraints on the garbage collector. In particular, the collector is not free to move objects and update pointers, because a non-pointer might be mistaken for a pointer and mistakenly updated. (This could result in

[44] On many architectures, normal loads and stores must be aligned on word boundaries, and on others there is a time penalty for unaligned accesses.

[45] This typically requires clever implementation strategies to optimize away the heap allocation of most integers [Yua90a].

[46] For example, if using page marking or card marking for generational collection, headers make it much simpler to scan pages.

[47] These techniques are usually associated with Boehm and his associates, who have developed them to a high degree, but similar techniques appear to have been used earlier in the Kyoto Common Lisp system and perhaps elsewhere.

mysterious and unpredictable changes to nonpointer data like integers and character strings.) Conservative collectors therefore can't use a straightforward copying traversal algorithm.

Conservative pointer finding can be combined with other techniques to cope with language implementations that are only partly cooperative. For example, Barlett's and Detlefs' "mostly-copying" collectors use headers to decode fields of objects in the heap, but rely on conservative techniques to find pointers from the activation stack [Bar88, Det91]. This supports copying techniques that relocate and compact most (but not all) objects. Objects conservatively identified as being pointed to from the stack are "pinned" in place, and cannot be moved.[48]

The choice of conservative stack scanning and more precise heap tracing is often reasonable, because it is usually easier to retrofit object headers into a language than it is to modify the compiler to make stack frame formats decodable. Headers can be added to objects by a heap allocation routine, which may simply be a library routine that can be changed or substituted easily. Compilers often record enough information to decode record layouts, for debugging purposes, and that information can be captured and massaged into runtime type identification for heap-allocated objects [WJ93].

Conservative pointer-finding can be defeated by language-level facilities such as the ability to cast pointers to integers, destroy the original pointer, and perform arbitrary arithmetic on the integer value. If the original value is then restored and cast back to a pointer, the referred-to object may no longer exist—the garbage collector may have reclaimed the object because it couldn't tell that the integer value "pointed to" the object, even when viewed as a pointer value. Fortunately, most programs do not perform this sequence of operations—they may cast a pointer to an integer, but the original pointer is likely to still be present, and the object will therefore be retained.

Compiler optimizations can perform similar operations on pointers, and this is unfortunately harder for the programmer to avoid—the compiler may use algebraic transformations on pointer expressions, disguising the pointers, or they may perform operations on subword parts of a pointer, with temporary inconsistencies in the state of the pointer. While most compilers don't perform these optimizations very often, they do occur, and a few compilers do them regularly. For most compilers, it is sufficient to turn off the highest levels of optimization to avoid such pointer-mangling optimizations, but this is not reliable across compilers, and typically costs a few percent in runtime efficiency due to missed optimizations. Since most compilers do not provide the ability to selectively turn off only the optimizations that are troublesome for garbage collectors, it is usually necessary to turn off several optimizations, i.e., the "high level" optimizations. To avoid this problem, garbage collector designers have proposed a set of constraints that compilers can preserve to ensure that garbage collection is possible; these constraints do not require major changes to existing compilers [Boe91, BC91].

Similarly, programming guidelines have been proposed to ensure that programmers in C++ avoid constructions that make correct garbage collection impossible [ED93]; by programming in a very slightly restricted subset of C++, it is possible to ensure that a cooperative compiler can support correct garbage collection. By using a slightly more restricted ("safe") subset of C++, it is possible to ensure that even buggy programs do not break the basic memory abstractions and produce hard-to-diagnose errors.

Some people object to conservative pointer-finding techniques because they are known not to be "correct"—it is possible, for example, for an integer to be mistaken for a pointer, causing garbage to be retained. In the worst case, this may cause considerable garbage to go unreclaimed, and a program may simply run out of memory and crash. Advocates of conservative pointer-finding counter that the probability of such an occurrence can be made very small, and that such errors are much less likely than fatal errors due to programmers' mistakes in explicit heap management. In practice, therefore, the use of an "incorrect" technique may be better than having programmers write programs that are even less correct. In the long run, it is hoped, conservative pointer finding will make garbage collection widely usable, and once it's widely used, compiler vendors will provide some degree of support for more precise techniques.

[48] Such objects are pinned in place in memory, but can be advanced to an older generation by changing the set to which a page belongs—that is, objects belong to pages, and pages belong to generations, but an entire page can be moved to a different generation simply by changing the tables that record which pages belong to which generations. In essence, objects' ages are encoded using something like a BiBOP tagging scheme.

6.3 Linguistic Support and Smart Pointers

Another approach to retrofitting garbage collection into existing systems is to use the extension facilities provided by the language, and implement garbage collection within the language itself. The most common form of this is to implement a special set of garbage-collected data types with a restricted set of access functions that preserve the garbage collector constraints. For example, a reference-counted data type might be implemented, and accessor functions (or macros) used to perform assignments—the accessor functions maintain the reference counts as well as performing the actual assignments.

A somewhat more elegant approach is to use extension mechanisms such as operator overloading to allow normal program operators to be used on garbage collected types, with the compiler automatically selecting the appropriate user-defined operation for given operands. In C++, it is common to define "smart pointer" classes that can be used with garbage-collectible classes [Str87], with appropriately defined meanings for pointer-manipulating operations (such as `*` and `->`) and for the address-taking operation (`&`) on collectible objects. Unfortunately, these user-defined pointer types can't be used in exactly the same ways as built-in pointer types, for several reasons [Ede92]. One reason is that there is no way to define all of the automatic coercions that the compiler performs automatically for built-in types. Another problem is that not all operators can be overloaded in this way. C++ provides most, but not all, of the extensibility necessary to integrate garbage collection into the language gracefully. (It is apparently easier in Ada [Bak93b], because the overloading system is more powerful and the builtin pointer types have fewer subtleties which must be emulated.) Yet another limitation is that it is impossible to re-define operations on built-in classes, making it difficult to enhance the existing parts of the language—only user-defined types can be garbage collected gracefully.

Still another limitation is that garbage collection is difficult to implement *efficiently* within the language, because it is impossible to tell the compiler how to compile for certain important special cases [Det92, Ede92] For example, in C++ or Ada, there is no way to specialize an operation for objects that are known at compile time not to be allocated in the heap.[49]

Recent work in *reflective* systems has explored languages with very powerful and regular extension mechanisms, and which expose some of the underlying implementation to allow efficient reimplementation of existing language features [KdRB91, MN88, YS92]. While most existing reflective languages supply garbage collection as part of the base language, it is possible to imagine implementing garbage collection within a small, powerful reflective language. As in any reflective system, however, it is important to expose only the most important low-level issues, to avoid limiting the choices of the base language implementor; this is an area of active research.

6.4 Compiler Cooperation and Optimizations

In any garbage-collected system, there must be a set of conventions used by both the garbage collector and the rest of the system (the interpreter or compiled code), to ensure that the garbage collector can recognize objects and pointers. In a conservative pointer-finding system, the "contract" between the compiler and collector is very weak indeed, but it's still there—if the compiler avoids strange optimizations that can defeat the collector. In most systems, the compiler is much more cooperative, ensuring that the collector can find pointers in the stack and from registers.

6.4.1 GC-Anytime vs. Safe-Points Collection

Typically, the contract between the collector and running code takes one of two forms, which we call the *gc-anytime* and *safe-points* strategies. In a gc-anytime system, the compiler ensures that running code can be interrupted at any point, and it will be safe to perform a garbage collection—information will be available to find all of the currently active pointer variables, and decode their formats so that reachable objects can be found.

In a *safe-points* system, the compiler only ensures that garbage collection will be possible at certain selected points during program execution, and that these points will occur reasonably frequently and regularly. In many systems, procedure calls and backward branches are guaranteed to be safe points, ensuring that the program cannot loop (or recurse) indefinitely without reaching a safe point—the longest possible time between safe points is the time to take the longest noncalling forward path through any pro-

[49] In C++ terminology, operators can only be specialized on the types of their arguments, not the arguments' *storage class*.

cedure. (Finer-grained responsiveness can be guaranteed by introducing intermediate safe points, if necessary.)

The advantage of a safe-points scheme is that the compiler is free to use arbitrarily complex optimizations within the unsafe regions between safe points, and it is not obligated to record the information necessary to make it possible to locate and de-optimize pointer values.[50] One disadvantage of a safe points scheme is that it restricts implementation strategies for lightweight processes (threads). If several threads of control are executing simultaneously in the same garbage-collected heap, and one thread forces a garbage collection, the collection must wait until all threads have reached a safe point and stopped.

One implementation of this scheme is to mask hardware interrupts between safe points; a more common one is to provide a small routine which can handle the actual low-level interrupt at any time by simply recording basic information about it, setting a flag, and resuming normal execution. Compiled code checks the flag at each safe point, and dispatches to a high-level interrupt handler if it is set. This introduces a higher-level notion of interrupts, somewhat insulated from actual machine interrupts—and with a somewhat longer latency.

With either a safe-points or a gc-anytime strategy, there are many possible conventions for ensuring that pointers can be identified for garbage collection; with a safe-points system, however, the compiler is free to violate the convention in between safe points.

6.4.2 Partitioned Register Sets vs. Variable Representation Recording

In many systems, the compiler respects a simple convention as to which registers can be used for holding which kinds of values. In the T system (using the Orbit compiler [KKR+86, Kra88]), for example, some registers are used only for tagged values, and others only for raw nonpointer values. The pointers are assumed to be in the normal format—direct pointers to known offset within the object, plus a tag; headers can thus be extracted from the pointed-to objects by a simple indexed load instruction.

Other register set partitionings and conventions are possible. For example, it would be possible to have registers holding raw pointers, perhaps pointers anywhere within an object, if the collector ensures that headers of objects can be derived from them using alignment constraints. If a non-copying collector is used, "untraced" registers might be allowed to hold optimized pointers (which might not actually point within an object at all, due to an algebraic transformation), as long as an unoptimized pointer to the same object is in a "traced" register in a known format.

The main problem with partitioned register sets is that it reduces the compiler's freedom to allocate program variables and temporaries in whichever registers are available. Some code sequences might require several pointer registers and very few nonpointer registers, and other code sequences might require the opposite. This would mean that more variables would have to be "spilled," i.e., allocated in the stack or heap instead of in registers, and code would run more slowly.

An alternative to partitioned register sets is to give the compiler more freedom in its use of registers, but require it to communicate more information to the garbage collector—i.e., to tell it where the pointers are and how to interpret them properly. We call this strategy *variable representation recording*, because the compiler must record its decisions about which registers (or stack variables) hold pointers, and how to recover the object's address if the pointer has been transformed by an optimization. For each range of instructions where pointer variable representations differ, the compiler must emit an annotation. This information is similar to that required for debugging optimized code, and most optimizations can be supported with little overhead. The space requirements for this additional information (which essentially annotates the executable code) may not be negligible, however, and it may be desirable in some cases to change the compiler's code generation strategy slightly [DMH92].

6.4.3 Optimization of Garbage Collection Itself

While garbage collectors can be constrained by their relationship to the compiler and its optimizations, it is also possible for the compiler to assist in making the garbage collector efficient. For example, an optimizing compiler may be able to optimize away unnecessary or redundant read-or write-barrier operations, or to detect that a heap object can be safely stack-allocated instead.

In most generational and incremental algorithms, the write barrier is only necessary when a pointer is being stored, but it may not be obvious at compile

[50] This is particularly important when using a high-level language such as C as an intermediate language, and it is undesirable or impossible to prevent the C compiler from using optimizations that mask pointers.

time whether a value will be a pointer or not. In dynamically typed languages, variables may be able to take on either pointer or nonpointer values, and even in statically typed languages, pointer variables may be assigned a null value. Compilers may perform dataflow analysis which may allow the omission of a write barrier for more non-pointer assignment operations. In the future, advanced type-flow analysis such as that used in the Self compiler [CU89, CU91] may provide greater opportunities for read and write barrier optimizations.

The compiler may also be able to assist by eliminating redundant checks or marking when a write barrier tests the same pointer or marks the same object multiple times. (To the best of our knowledge, no existing compilers currently do this.) For this to be possible, however, the optimizer must be able to assume that certain things are not changed by the collector in ways that aren't obvious at compile time. For example, with a gc-anytime collector and multiple threads of control, a thread could be pre-empted at any time, and a garbage collection could occur before the thread is resumed. In such a system, there are fewer opportunities for optimization because the collector can be invoked between any two consecutive instructions. With a safe-points system, however, the optimizer may be able to perform more optimizations across sequences of instructions that execute atomically with respect to garbage collection.

Several researchers have investigated compiler optimizations related to heap-allocated structures, both to detect potential aliasing and to allow heap objects to be stack allocated when it is possible to infer that they become garbage at a particular point in a program [Sch75a, Sch75b, Hud86, JM81, RM88, LH88, HPR89, CWZ90, Bak90]. [Cha87] discusses interactions between conventional optimizations and garbage collection, and when garbage collection-oriented optimizations are safe. These topics are beyond the scope of this survey, and to the best of our knowledge no actual systems use these techniques for garbage collection optimizations; still, such techniques may lead to important improvements in garbage collector performance by shifting much of the garbage detection work to compile time.

Certain restricted forms of lifetime analysis—for local variable binding environments only—can be simple but effective in avoiding the need to heap-allocate most variable bindings in languages with closures [Kra88].

6.5 Free Storage Management

Nonmoving collectors must deal with the fact that freed space may be distributed through memory, interspersed with the live objects. The traditional way of dealing with this is to use one or more free lists, but it is possible to adapt any of the free storage management techniques which have been developed for explicitly-managed heaps [Knu69, Sta80].

The simplest scheme, used in many early Lisp interpreters, is to support only one size of heap-allocated object, and use a single free list to hold the freed items. When garbage is detected (e.g., during a sweep phase or when reference counts go to zero), the objects are simply strung together into a list, using one of the normal data fields to hold a list pointer.

When generalizing this scheme to support multiple object sizes, two basic choices are possible: to maintain separate lists for each object size (or approximate object size), or to keep a single list holding various sizes of freed spaces. Techniques with separate lists for different-sized objects include *segregated storage* and *buddy systems* [PN77]. Systems with a unified free list include *sequential fit* methods and *bitmapped* techniques. An intermediate strategy is to use a single data structure, but use a tree or similar structure sorted by the sizes (and/or addresses) of the free spaces [Ste83] to reduce search times. Most of these systems and several hybrids (e.g., [BBDT84, OA85, WW88, GZ93]) are already described in the literature on memory management, and we will not describe them here. An exception to this is bitmapped memory management, which has been used in several garbage collectors, but is not usually discussed in the literature.

Bitmapped memory management simply maintains a bitmap corresponding to units of memory (typically words, or pairs of words if objects are always aligned on two-word boundaries), with the bit's value indicating whether the unit is in use or not. The bitmap is updated when objects are allocated or reclaimed. The bitmap can be scanned to construct a free list (as in [BDS91]) or it can be searched at allocation time in a manner analogous to the search of a free list in a sequential fit algorithm.

6.6 Compact Representations of Heap Data

Representations of heap data are often optimized for speed, rather than for space. In dynamically typed languages, for example, most fields of most objects are typically of a uniform size, large enough to hold a tagged pointer. Pointers, in turn, are typically represented as full-size virtual addresses in a flat (nonsegmented) address space. Much of this space is "wasted" in some sense, because many nonpointer values could be represented in fewer bits, and because pointers typically contain very little information due to limited heap sizes and locality of reference.

At least two mechanisms have been developed to exploit the regularities in data structures and allow them to be stored in a somewhat compressed form most of the time, and expanded on demand when they are operated on. One of these is a fine-grained mechanism called *cdr coding*, which is specific to list cells such as Lisp cons cells [Han69, BC79, LH86]. The other is *compressed paging*, a more general-purpose mechanism that operates on virtual memory pages [Wil91, Dou93, WB94]. Both mechanisms are invisible at the language level.

Cdr-coding was used in many early Lisp systems, when random-access memory was very expensive. Unfortunately, it tended to be rather expensive in CPU time, because the changeable representations of list complicate basic list operations. Common operations such as CDR require extra instructions to check to see what kind of list cell they are traversing—is it a normal cell, or one that has been compressed?

The compressed representation of a list in a cdr-coded system is really an array of items corresponding to the items of the original list. The cdr-coding system works in concert with the garbage collector, which linearizes lists and packs consecutive items into arrays holding the CAR values (list items); the CDR values—the pointers that link the lists—are omitted. To make this work, it is necessary to store a bit somewhere (e.g., a special extra bit in the tag of the field holding the CAR value) saying that the CDR value is implicitly a pointer to the next item in memory. Destructive updates to CDR values require the generation of actual cons cells on demand, and the forwarding of references from the predecessor part of the array.

This scheme is really only worthwhile with special hardware and/or microcoded routines, as found on Lisp Machines. On current general-purpose processors, it is seldom worth the time for the savings that can be gained. (Roughly half the data in a Lisp system consists of cons cells, so compressing them from two words to one can only save about 25% at best.)

More recently, *compressed paging* has been proposed as a means of reducing memory requirements on stock hardware. The basic idea is to devote some fraction of main memory to storing pages in a compressed form, so that one main memory page can hold the data for several virtual memory pages. Normal virtual memory access-protection hardware is used to detect references to compressed pages, and trap to a routine that will uncompress them. Once touched, a page is cached in normal (uncompressed) form for a while, so that the program can operate on it. After a page has not been touched for some time, it is re-compressed and access protected. The compression routine is not limited to compressing CDR fields—it can be designed to compress other pointers, and non-pointer data (such as integers and character strings, or executable code).

In effect, compressed paging adds a new level to the memory hierarchy, intermediate in cost between normal RAM and disk. This may not only decrease overall RAM requirements, but actually improve speed by reducing the number of disk seeks. Using simple and fast (but effective) compression algorithms, heap data can typically be compressed by a factor of two to four—in considerably less time than it would take to do a disk seek, even on a relatively slow processor [WB94]. As processor speed improvements continue to outstrip disk speed improvements, compressed paging becomes increasingly attractive.

7 GC-related Language Features

The main use of garbage collection is to support the simple abstraction that infinite amounts of uniform memory are available for allocation, so that objects can simply be created at will and can conceptually "live forever". Sometimes, however, it is desirable to alter this view. It is sometimes desirable to have pointers which do not prevent the referred-to objects from being reclaimed, or to trigger special routines when an object is reclaimed. It can also be desirable to have more than one heap, with objects allocated in different heaps being treated differently.

7.1 Weak Pointers

A simple extension of the garbage collection abstraction is to allow programs to hold pointers to objects *without* those pointers preventing the objects from being collected. Pointers that do not keep objects from being collected are known a *weak pointers*, and they are useful in a variety of situations. One common application is the maintenance of tables which make it possible to enumerate all of the objects of a given kind. For example, it might be desirable to have a table of all of the file objects in a system, so that their buffers could be flushed periodically for fault tolerance. Another common application is the maintenance of collections of auxiliary information about objects, where the information alone is useless and should not keep the described objects alive. (Examples include property tables and documentation strings about objects—the usefulness of the description depends on the described object being otherwise interesting, not vice versa.)

Weak pointers are typically implemented by the use of a special data structure, known to the garbage collector, recording the locations of weak pointer fields. The garbage collector traverses all other pointers in the system first, to determine which objects are reachable via normal paths. Then the weak pointers are traversed; if their referents have been reached by normal paths, the weak pointers are treated normally. (In a copying collector, they are updated to reflect the new location of the object.) If their referents have not been reached, however, the weak pointers are treated specially, typically replaced with a nonpointer value (such as null) to signal that their referents no longer exist.

7.2 Finalization

Closely related to the notion of weak pointers is the concept of *finalization*, i.e., actions that are performed automatically when an object is reclaimed. This is especially common when an object manages a resource other than heap memory, such as a file or a network connection. For example, it may be important to close a file when the corresponding heap object is reclaimed In the example of annotations and documentation, it is often desirable to delete the description of an object once the object itself is reclaimed. Finalization can thus generalize the garbage collector, so that other resources are managed in much the same way as heap memory, and with similar program structure. This makes it possible to write more general and reusable code, rather than having to treat certain kinds of objects very differently than "normal" objects. (Consider a routine that iterates over a list, applying an arbitrary function to each item in the list. If file descriptors are garbage collected, the very same iteration routine can be used for a list of file descriptors as for a list of heap objects. If the list becomes unreachable, the garbage collector will reclaim the file descriptors along with the list structure itself.)

Finalization is typically implemented by marking finalizable objects in some way and registering them in a data structure much like that used for weak pointers. (They may in fact use the same data structure.) Rather than simply nilling the pointers if the objects aren't reached by the primary traversal, however, the pointers are recorded for special treatment after the collection is finished. After the collection is complete and the heap is once again consistent, the referred-to objects have their finalization operations invoked.

Finalization is useful in a variety of circumstances, but it must be used with care. Because finalization occurs asynchronously—i.e., whenever the collector notices the objects are unreachable and does something about it—it is possible to create race conditions and other subtle bugs. For a more thorough discussion of both weak pointers and finalization, see [Hay92].

7.3 Multiple Differently-Managed Heaps

In some systems, a garbage collected heap is provided for convenience, and a separate, explicitly-managed heap is provided to allow very precise control over the use of memory.

In some languages, such as Modula-3 [CDG+89] and an extended version of C++ [ED93], a garbage collected heap coexists with an explicitly-managed heap. This supports garbage collection, while allowing programmers to explicitly control deallocation of some objects for maximum performance or predictability. Issues in the design of such multiple-heap systems are discussed in [Del92] and [ED93].

In other systems, such as large persistent or distributed shared memories, it may also be desirable to have multiple heaps with different policies for distribution (e.g., shared vs. unshared), access privileges, and resource management [Mos89, Del92].

Many language implementations have had such features internally, more or less hidden from normal programmers, but there is little published information about them, and little standardization of programmer interfaces. (The terminology is also not standardized, and these heaps are variously called "heaps," "zones," "areas," "arenas," "segments," "pools," "regions," and so on.)

8 Overall Cost of Garbage Collection

The costs of garbage collection have been studied in several systems, especially Lisp and Smalltalk systems. Most of these studies have serious limitations, however. A common limitation is the use of a slow language implementation, e.g., an interpreter, or an interpreted virtual machine, or a poor compiler. When programs execute unrealistically slowly, the CPU costs of garbage collection appear very low. Another common limitation of cost studies is the use of "toy" programs or synthetic benchmarks. Such programs often behave very differently from large, real-world applications. Often, they have little long-lived data, making tracing collection appear unrealistically efficient. This may also affect measurements of write barrier costs in generational collectors, because programs with little long-lived data do not create many intergenerational pointers.

Yet another difficulty is that most studies are now somewhat dated, because of later improvements in garbage collection implementations. The most valuable studies are therefore the ones with detailed statistics about various events, which can be used to infer how different strategies would fare.

Perhaps the best study to date is Zorn's investigation of GC cost in a large commercial Common Lisp system, using eight large programs [Zor89]. Zorn found the time cost of generational garbage collection to be 5% to 20%. (We suspect those numbers could be improved significantly with the use of a fast card-marking write barrier.) Shaw's thesis provides similar performance numbers [Sha88], and Steenkiste's thesis contains similar relevant statistics [Ste87].

Zorn has also studied simple non-generational collector using conservative pointer-finding [Zor93]; this allowed him to study garbage collection using a high-performance implementation of the C language, and compare the costs of garbage collection to those of various implementations of explicit deallocation. Unfortunately, the garbage collector used was not state-of-the art, partly due to the lack of compiler cooperation. Zorn found that, when compared to using a well-implemented `malloc()` and `free()`, programs using a simple conservative GC used between 0% and 36% more CPU time, and between 40% and 280% more memory. Zorn's test programs were unusually heap allocation-intensive, however, and the costs would presumably be lower for a more typical workload. We also believe these figures could be improved considerably with a state-of-the art generational collector and compiler cooperation.

Tarditi and Diwan [TD93] studied garbage collection costs in Standard ML[51] of NJ and found the total time cost of garbage collection between 22% and 40%. While this is a very interesting and informative study, we believe these numbers to be unnecessarily high, due to extreme load on the garbage collector, caused by allocating tremendous amounts of activation information and binding environments on the heap, rather than on a stack (Sect. 5.2). We also believe their write barrier could be improved.

Our own estimate of the typical cost of garbage collection in a well-implemented non-incremental generational system (for a high-performance language implementation) is that it *should* cost roughly ten percent of running time over the cost of a well-implemented explicit heap management system, with a space cost of roughly a factor of two in data memory size. (Naturally, increased CPU costs could be traded for reduced space costs.) This must be taken as an educated guess, however, compensating for what we perceive as deficiencies in existing systems and limitations of the studies to date.

Clearly, more studies of garbage collection in high-performance systems are called for, especially for good implementations of strongly-typed languages.

9 Conclusions and Areas for Research

We have argued that garbage collection is an essential for fully modular programming, to allow flexible, reusable code and to eliminate a large class of extremely dangerous coding errors.

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, Zor93]. Generational techniques reduce the basic costs and disruptiveness of collection by exploiting the empirically-observed tendency of objects to die young. Incremental techniques may even make garbage collection relatively attractive for hard real-time systems.

We have discussed the basic operation of several kinds of garbage collectors, to provide a framework

[51] ML is a statically typed, general-purpose mostly-functional language.

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, such as write barrier overhead and locality effects. Similarly, "real-time" garbage collection is a more subtle topic than is widely recognized. These factors require garbage collection designers to take detailed implementation issues into account, and be very careful in their choices of features. Pragmatic decisions (such as the need to interoperate with existing code in other languages) may also outweigh small differences in performance.

Despite these complex issues, many systems actually have fairly simple requirements of a garbage collector, and can use a collector that consists of a few hundred lines of code. Systems with large, complex optimizing compilers should have more attention paid to garbage collection, and use state-of-the-art techniques. (One promising development is the availability of garbage collectors written in portable high-level languages (typically C) and adaptable for use with various implementations of various languages.[52])

Garbage collector designers must also keep up with advances in other aspects of system design. The techniques described in this survey appear to be sufficient to provide good performance in most relatively conventional uniprocessor systems, but continual advances in other areas introduce new problems for garbage collector design.

Persistent object stores [ABC+83, DSZ90, AM92] allow large interrelated data structures to be saved indefinitely without writing them to files and re-reading them when they are needed again; by automatically preserving pointer-linked data structures, they relieve the programmer of tedious and error-prone coding of input/output routines. Large persistent object stores can replace file systems for many purposes, but this introduces problems of managing large amounts of long-lived data. It is very desirable for persistent stores to have garbage collection, so that storage leaks do not result in a large and permanent accumulation of unreclaimed storage. Garbage collecting a large persistent store is a very different task from garbage collecting the memory of a single process of bounded duration. In effect, a collector for a conventional system can avoid the problem of very long-lived data, because data written to files "disappear" from the collector's point of view. A persistent store keeps those data within the scope of garbage collection, offering the attractive prospect of automatic management of long-lived data—as well as the challenge of doing it efficiently.

Parallel computers raise new issues for garbage collectors, as well. It is desirable to make the collector *concurrent*, i.e., able to run on a separate processor from the application, to take advantage of processors not in use by the application. It is also desirable to make the collector itself parallel, so that it can be sped up to keep up with the application. Concurrent collectors raise issues of coordination between the collector and the mutator which are somewhat more difficult than those raised by simple incremental collection. Parallel collectors also raise issues of coordination of different parts of the garbage collection process, and of finding sufficient parallelism, despite potential bottlenecks due to the topologies of the data structures being traversed.

Distributed systems pose still more problems [AMR92]; the limitations on parallelism are particularly severe when the collection process must proceed across multiple networked computers, and communication costs are high. In large systems with long-running applications, networks are typically unreliable, and distributed garbage collection strategies for such systems—like the applications running on them—must be robust enough to tolerate computer and network failures.

In large persistent or distributed systems, data integrity is particularly important; garbage collection strategies must be coordinated with checkpointing and recovery, both for efficiency and to ensure that the collector itself does not fail [Kol90, Det91, ONG93].

As high-performance graphics and sound capabilities become more widely available and economical, computers are likely to be used in more graphical and interactive ways. Multimedia and virtual reality applications will require garbage collection techniques that do not impose large delays, making incremental techniques increasingly desirable. Increasing use of embedded microprocessors makes it desirable to facilitate programming of hard real-time applications, making fine-grained incremental techniques especially attractive.

As computer systems become more ubiquitous, networked, and heterogeneous, it is increasingly desirable to integrate modules which run on different computers and may have been developed in extremely different programming languages; interoperability between

[52] Examples of this include the UMass Garbage Collection Toolkit [HMDW91] and Wilson and Johnstone's real-time garbage collector, which has been adapted for use with C++, Eiffel, Scheme, and Dylan [WJ93].

diverse systems is an important goal for the future, and garbage collection strategies must be developed for such systems.

Garbage collection is applicable to many current general-purpose and specialized programming systems, but considerable work remains in adapting it to new, ever-more advanced paradigms.

Acknowledgments

References

[ABC+83] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *Computer Journal*, 26(4):360–365, December 1983.

[AEL88] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent garbage collection on stock multiprocessors. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation* [PLD88], pages 11–20.

[AM92] Antonio Albano and Ron Morrison, editors. *Fifth International Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992. Springer-Verlag.

[AMR92] Saleh E. Abdullahi, Eliot E. Miranda, and Graem A. Ringwood. Distributed garbage collection. In Bekkers and Cohen [BC92], pages 43–81.

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

[AP87] S. Abraham and J. Patel. Parallel garbage collection on a virtual memory system. In E. Chiricozzi and A. D'Amato, editors, *International Conference on Parallel Processing and Applications*, pages 243–246, L'Aquila, Italy, September 1987. Elsevier North-Holland.

[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.

[Bak90] Henry G. Baker, Jr. Unify and conquer: (garbage, updating, aliasing ...) in functional languages. In *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming* [LFP90], pages 218–226.

[Bak91a] Henry G. Baker, Jr. Cache-conscious copying collection. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper.

[Bak91b] Henry G. Baker, Jr. The Treadmill: Real-time garbage collection without motion sickness. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper. Also appears as *SIGPLAN Notices 27*(3):66–70, March 1992.

[Bak93a] Henry G. Baker. Infant mortality and generational garbage collection. *SIGPLAN Notices*, 28(4):55–57, April 1993.

[Bak93b] Henry G. Baker, Jr. Safe and leakproof resource management using Ada83 limited types. Unpublished, 1993.

[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.

[Bar89] Joel F. Bartlett. Mostly-copying garbage collection picks up generations and C++. Technical Note TN-12, Digital Equipment Corporation Western Research Laboratory, October 1989.

[BBDT84] G. Bozman, W. Buco, T. P. Daly, and W. H. Tetzlaff. Analysis of free storage algorithms—revisited. *IBM Systems Journal*, 23(1):44–64, 1984.

[BC79] Daniel G. Bobrow and Douglas W. Clark. Compact encodings of list structure. *ACM Transactions on Programming Languages and Systems*, 1(2):266–286, October 1979.

[BC91] Hans-Juergen Boehm and David Chase. A proposal for garbage-collector-safe compilation. *The Journal of C Language Translation*, 4(2):126–141, December 1991.

[BC92] Yves Bekkers and Jacques Cohen, editors. *International Workshop on Memory Management*

ment, number 637 in Lecture Notes in Computer Science, St. Malo, France, September 1992. Springer-Verlag.

[BDS91] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation* [PLD91], pages 157–164.

[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*, Minneapolis, Minnesota, August 1983. Also available as Technical Report UCB/CSD 83/125, University of California at Berkeley, Computer Science Division (EECS), 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.

[Boe93] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation* [PLD93], pages 197–206.

[Bro84] Rodney A. Brooks. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [LFP84], pages 108–113.

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

[BZ93] David A. Barrett and Bejamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation* [PLD93], pages 187–196.

[CDG+89] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalso, and Greg Nelson. Modula-3 report (revised). Research Report 52, Digital Equipment Corporation Systems Research Center, November 1989.

[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.

[Cha87] David Chase. *Garbage Collection and Other Optimizations*. PhD thesis, Rice University, Houston, Texas, August 1987.

[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.

[CHO88] Will Clinger, Anne Hartheimer, and Erik Ost. Implementation strategies for continuations. In *Conference Record of the 1988 ACM Symposium on LISP and Functional Programming*, pages 124–131, Snowbird, Utah, July 1988. ACM Press.

[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.

[CU89] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented language. In *Proceedings of SIGPLAN '89*, pages 146–160, 1989.

[CU91] Craig Chambers and David Ungar. Making pure object-oriented languages practical. In Paepcke [Pae91], pages 1–15.

[CWB86] Patrick J. Caudill and Allen Wirfs-Brock. A third-generation Smalltalk-80 implementation. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '86) Proceedings*, pages 119–130. ACM Press, October 1986.

[CWZ90] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the 1990 SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–310, White Plains, New York, June 1990. ACM Press.

[Daw82] Jeffrey L. Dawson. Improved effectiveness from a real-time LISP garbage collector. In *Conference Record of the 1982 ACM Symposium on LISP and Functional Programming*, pages 159–167, Pittsburgh, Pennsylvania, August 1982. ACM Press.

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

[Del92] V. Delacour. Allocation regions and implementation contracts. In Bekkers and Cohen [BC92], pages 426–439.

[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.

[Det91] David L. Detlefs. *Concurrent, Atomic Garbage Collection*. PhD thesis, Dept. of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, November 1991. Technical report CMU-CS-90-177.

[Det92] David L. Detlefs. Garbage collection and runtime typing as a C++ library. In *USENIX C++ Conference* [USE92].

[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 *Proceedings of the 1992 SIGPLAN Conference on Programming Language Design and Implementation*, pages 273–282, San Francisco, California, June 1992. ACM Press.

[Dou93] Fred Douglis. The compression cache: Using on-line compression to extend physical memory. In *Proceedings of 1993 Winter USENIX Conference*, pages 519–529, San Diego, California, January 1993.

[DSZ90] Alan Dearle, Gail M. Shaw, and Stanley B. Zdonik, editors. *Implementing Persistent Object Bases: Principles and Practice (Proceedings of the Fourth International Workshop on Persistent Object Systems)*, Martha's Vineyard, Massachusetts, September 1990. Morgan Kaufman.

[DTM93] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs with intensive heap allocation. Submitted for publication, August 1993.

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

[ED93] John R. Ellis and David L. Detlefs. Safe, efficient garbage collection for C++. Technical Report 102, Digital Equipment Corporation Systems Research Center, 1993.

[Ede92] Daniel Ross Edelson. Smart pointers: They're smart, but they're not pointers. In *USENIX C++ Conference* [USE92], pages 1–19. Technical Report UCSC-CRL-92-27, University of California at Santa Cruz, Baskin Center for Computer Engineering and Information Sciences, June 1992.

[EV91] Steven Engelstad and Jim Vandendorp. Automatic storage management for systems with real time constraints. In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper.

[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.

[GC93] Edward Gehringer and Ellis Chang. Hardware-assisted memory management. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection* [OOP93]. Position paper.

[GG86] Ralph E. Griswold and Madge T. Griswold. *The Implementation of the Icon Programming Language*. Princeton University Press, Princeton, New Jersey, 1986.

[Gol91] Benjamin Goldberg. Tag-free garbage collection for strongly-typed programming languages. In *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation* [PLD91], pages 165–176.

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

[Gud93] David Gudeman. Representing type information in dynamically-typed languages. Technical Report TR93-27, University of Arizona, Department of Computer Science, Tucson, Arizona, 1993.

[GZ93] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. *Software Practice and Experience*, 23(8):851–869, August 1993.

[Hö93] Urs Hölzle. A fast write barrier for generational garbage collectors. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection* [OOP93]. Position paper.

[Han69] Wilfred J. Hansen. Compact list representation: Definition, garbage collection, and system implementation. *Communications of the ACM*, 12(9):499–507, September 1969.

[Hay91] Barry Hayes. Using key object opportunism to collect old objects. In Paepcke [Pae91], pages 33–46.

[Hay92] Barry Hayes. Finalization in the garbage collector interface. In Bekkers and Cohen [BC92], pages 277–298.

[HH93] Antony Hosking and Richard Hudson. Remembered sets can also play cards. In OOPSLAGC [OOP93]. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC93.

[HJ92] Reed Hastings and Bob Joyce. Purify: Fast detection of memory leaks and access errors. In *USENIX Winter 1992 Technical Conference*, pages 125–136. USENIX Association, January 1992.

[HL93] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pages 73–82, San Diego, California, May 1993. ACM Press. Published as *SIGPLAN Notices 28*(7), July 1993.

[HMDW91] Richard L. Hudson, J. Eliot B. Moss, Amer Diwan, and Christopher F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, MA 01003, September 1991.

[HMS92] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanović. A comparative performance evaluation of write barrier implementations. In Andreas Paepcke, editor, *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '92)*, pages 92–109, Vancouver, British Columbia, October 1992. ACM Press. Published as *SIGPLAN Notices 27*(10), October 1992.

[HPR89] Susan Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the SIGPLAN '89 SIGPLAN Symposium on Compiler Construction*, June 1989. Published as *SIGPLAN Notices 24*(7).

[Hud86] Paul Hudak. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pages 351–363, Cambridge, Massachusetts, August 1986. ACM Press.

[JJ92] Niels Christian Juul and Eric Jul. Comprehensive and robust garbage collection in a distributed system. In Bekkers and Cohen [BC92], pages 103–115.

[JM81] Neil D. Jones and Steven S. Muchnick. Flow analysis and optimization of LISP-like structures. In Steven S. Muchnik and Neil D. Jones, editors, *Program Flow Analysis*, pages 102–131. Prentice-Hall, 1981.

[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.

[KdRB91] Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[Kel93] Richard Kelsey. Tail recursive stack disciplines for an interpreter. Available via anonymous FTP from nexus.yorku.ca in /pub/scheme/txt/stack-gc.ps. Slightly enhanced version of Technical Report NU-CCS93-03, College of Computer Science, Northeastern University, 1992., 1993.

[KKR+86] David A. Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN Symposium on Compiler Construction*, pages 219–233, Palo Alto, California, June 1986. Published as *ACM SIGPLAN Notices 21*(7), July 1986.

[KLS92] Phillip J. Koopman, Jr., Peter Lee, and Daniel P. Siewiorek. Cache performance of combinator graph reduction. *ACM Transactions on Programming Languages and Systems*, 14(2):265–297, April 1992.

[Knu69] Donald E. Knuth. *The Art of Computer Programming*, volume 1: Fundamental Algorithms. Addison-Wesley, Reading, Massachusetts, 1969.

[Kol90] Elliot Kolodner. Atomic incremental garbage collection and recovery for large stable heap. In Dearle et al. [DSZ90], pages 185–198.

[Kra88] David A. Kranz. *ORBIT: An Optimizing Compiler For Scheme*. PhD thesis, Yale University, New Haven, Connecticut, February 1988.

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

[LD87] Bernard Lang and Francis Dupont. Incremental incrementally compacting garbage collection. In *SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques*, pages 253–263, Saint Paul, Minnesota, June 1987. ACM Press. Published as *SIGPLAN Notices 22*(7), July 1987.

[Lee88] Elgin Hoe-Sing Lee. Object storage and inheritance for SELF, a prototype-based object-oriented programming language. Engineer's thesis, Stanford University, Palo Alto, California, December 1988.

[LFP84] *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, Texas, August 1984. ACM Press.

[LFP90] *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming*, Nice, France, June 1990. ACM Press.

[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.

[LH86] Kai Li and Paul Hudak. A new list compaction method. *Software Practice and Experience*, 16(2), February 1986.

[LH88] James R. Larus and Paul N. Hilfinger. Detecting conflicts between record accesses. In *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation* [PLD88], pages 21–34.

[Lla91] Rene Llames. *Performance Analysis of Garbage Collection and Dynamic Reordering in a Lisp System*. PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Champaign-Urbana, Illinois, 1991.

[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.

[Mey88] Norman Meyrowitz, editor. *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '88) Proceedings*, San Diego, California, September 1988. ACM Press. Published as *SIGPLAN Notices 23*(11), November 1988.

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

[MN88] Pattie
Maes and Daniele Nardi, editors. *Meta-Level Architectures and Reflection*. North-Holland, Amsterdam, 1988.

[Moo84] David Moon. Garbage collection in a large Lisp system. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [LFP84], pages 235–246.

[Mos89] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Second International Workshop on Database Programming Languages*, pages 269–285, Glenedon Beach, Oregon, June 1989. Also available as Technical Report 89-68, University of Massachusetts Dept. of Computer and Information Science, Amherst, Massachusetts, 1989.

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

[NOPH92] Scott Nettles, James O'Toole, David Pierce, and Nicholas Haines. Replication-based incremental copying collection. In Bekkers and Cohen [BC92], pages 357–364.

[NR87] S. C. North and J. H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *ACM Conference on Functional Programming Languages and Computer Architecture*, number 274 in Lecture Notes in Computer Science, pages 113–133. Springer-Verlag, September 1987.

[NS90] Kelvin Nilsen and William J. Schmidt. A high-level overview of hardware assisted real-time garbage collection. Technical Report

TR 90-18a, Dept. of Computer Science, Iowa State University, Ames, Iowa, 1990.

[NS92] Kelvin Nilsen and William J. Schmidt. Cost-effective object space management for hardware-assisted real-time garbage collection. *ACM Letters on Programming Languages and Systems*, 1(4):338–355, December 1992.

[OA85] R. R. Oldehoeft and S. J. Allan. Adaptive exact-fit storage management. *Communications of the ACM*, 28(5):506–511, May 1985.

[ONG93] James O'Toole, Scott Nettles, and David Gifford. Concurrent compacting garbage collection of a persistent heap. In *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, Asheville, North Carolina, December 1993. ACM Press. Published as *Operating Systems Review 27*(5).

[OOP91] *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems*, October 1991. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC91.

[OOP93] *OOPSLA '93 Workshop on Memory Management and Garbage Collection*, October 1993. Available for anonymous FTP from cs.utexas.edu in /pub/garbage/GC93.

[Pae91] Andreas Paepcke, editor. *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '91)*, Phoenix, Arizona, October 1991. ACM Press. Published as *SIGPLAN Notices 26*(11), November 1991.

[PLD88] *Proceedings of the 1988 SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988. ACM Press.

[PLD91] *Proceedings of the 1991 SIGPLAN Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 1991. ACM Press. Published as *SIGPLAN Notices 26*(6), June 1992.

[PLD93] *Proceedings of the 1993 SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 1993. ACM Press.

[PN77] J. L. Peterson and T. A. Norman. Buddy systems. *Communications of the ACM*, 20(6):421–431, June 1977.

[PS89] C.-J. Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Computer Sciences Dept. University of Wisconsin, Madison, Wisconsin, July 1989.

[RM88] Christina Ruggieri and Thomas P. Murtagh. Lifetime analysis of dynamically allocated objects. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 285–293, San Diego, California, January 1988. ACM Press.

[Ros88] John R. Rose. Fast dispatch mechanisms for stock hardware. In Meyrowitz [Mey88], pages 27–35.

[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.

[Sch75a] Jacob T. Schwartz. Optimization of very high level languages—I. Value transmission and its corollaries. *Journal of Computer Languages*, 1:161–194, 1975.

[Sch75b] Jacob T. Schwartz. Optimization of very high level languages—II. Deducing relationships of inclusion and membership. *Journal of Computer Languages*, 1:197–218, 1975.

[SCN84] W. R. Stoye, T. J. W. Clarke, and A. C. Norman. Some practical methods for rapid combinator reduction. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming* [LFP84], pages 159–166.

[SH87] Peter Steenkiste and John Hennessy. Tags and type checking in Lisp. In *Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS II)*, pages 50–59, Palo Alto, California, October 1987.

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

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

[Sta80] Thomas Standish. *Data Structure Techniques.* Addison-Wesley, Reading, Massachusetts, 1980.

[Sta82] James William Stamos. A large object-oriented virtual memory: Grouping strategies, measurements, and performance. Technical Report SCG-82-2, Xerox Palo Alto Research Center, Palo Alto, California, May 1982.

[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.

[Ste83] C. J. Stephenson. Fast fits: New methods for dynamic storage allocation. In *Proceedings of the Ninth Symposium on Operating Systems Principles*, pages 30–32, Bretton Woods, New Hampshire, October 1983. ACM Press. Published as *Operating Systems Review 17*(5), October 1983.

[Ste87] Peter Steenkiste. *Lisp on a Reduced-Instruction-Set Processor: Characterization and Optimization*. PhD thesis, Stanford University, Palo Alto, California, March 1987. Technical Report CSL-TR-87-324, Stanford University Computer System Laboratory.

[Str87] Bjarne Stroustrup. The evolution of C++, 1985 to 1987. In *USENIX C++ Workshop*, pages 1–22. USENIX Association, 1987.

[Sub91] Indira Subramanian. Managing discardable pages with an external pager. In *USENIX Mach Symposium*, pages 77–85, Monterey, California, November 1991.

[TD93] David Tarditi and Amer Diwan. The full cost of a generational copying garbage collection implementation. Unpublished, September 1993.

[UJ88] David Ungar and Frank Jackson. Tenuring policies for generation-based storage reclamation. In Meyrowitz [Mey88], pages 1–17.

[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. ACM Press, April 1984. Published as *ACM SIGPLAN Notices 19*(5), May, 1987.

[USE92] USENIX Association. *USENIX C++ Conference*, Portland, Oregon, August 1992.

[Wan89] Thomas Wang. MM garbage collector for C++. Master's thesis, California Polytechnic State University, San Luis Obispo, California, October 1989.

[WB94] Paul R. Wilson and V. B. Balayoghan. Compressed paging. In preparation, 1994.

[WDH89] Mark Weiser, Alan Demers, and Carl Hauser. The portable common runtime approach to interoperability. In *Proceedings of the Twelfth Symposium on Operating Systems Principles*, December 1989.

[WF77] David S. Wise and Daniel P. Friedman. The one-bit reference count. *BIT*, 17(3):351–359, September 1977.

[WH91] Paul R. Wilson and Barry Hayes. The 1991 OOPSLA Workshop on Garbage Collection in Object Oriented Systems (organizers' report). In Jerry L. Archibald, editor, *OOPSLA '91 Addendum to the Proceedings*, pages 63–71, Phoenix, Arizona, October 1991. ACM Press. Published as *OOPS Messenger 3*(4), October 1992.

[Whi80] Jon L. White. Address/memory management for a gigantic Lisp environment, or, GC considered harmful. In *LISP Conference*, pages 119–127, Redwood Estates, California, August 1980.

[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*, October 1990. Also appears 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*, pages 80–86, Palo Alto, California, October 1991. IEEE Press. Revised version to appear in *Computing Systems*.

[Wis85] David S. Wise. Design for a multiprocessing heap with on-board reference counting. In *Functional Programming Languages and Computer Architecture*, pages 289–304. Springer-Verlag, September 1985. Lecture Notes in Computer Science series, no. 201.

[Wit91] P. T. Withington. How real is "real time" garbage collection? In *OOPSLA '91 Workshop on Garbage Collection in Object-Oriented Systems* [OOP91]. Position paper.

[WJ93] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In *OOPSLA '93 Workshop on Memory Management and Garbage Collection* [OOP93]. Expanded version of workshop position paper submitted for publication.

[WLM91] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Effective static-graph reorganization to improve locality in garbage-collected systems. In *Proceedings of the 1991*

SIGPLAN Conference on Programming Language Design and Implementation [PLD91], pages 177–191. Published as *SIGPLAN Notices 26*(6), June 1992.

[WLM92] Paul R. Wilson, Michael S. Lam, and Thomas G. Moher. Caching considerations for generational garbage collection. In *Conference Record of the 1992 ACM Symposium on LISP and Functional Programming*, pages 32–42, San Francisco, California, June 1992. ACM Press.

[WM89] Paul R. Wilson and Thomas G. Moher. Design of the Opportunistic Garbage Collector. In *Conference on Object Oriented Programming Systems, Languages and Applications (OOPSLA '89) Proceedings*, pages 23–35, New Orleans, Louisiana, 1989. ACM Press.

[WW88] Charles B. Weinstock and William A. Wulf. Quickfit: an efficient algorithm for heap storage allocation. *ACM SIGPLAN Notices*, 23(10):141–144, October 1988.

[WWH87] Ifor W. Williams, Mario I. Wolczko, and Trevor P. Hopkins. Dynamic grouping in an object-oriented virtual memory hierarchy. In *European Conference on Object Oriented Programming*, pages 87–96, Paris, France, June 1987. Springer-Verlag.

[YS92] Akinori Yonezawa and Brian C. Smith, editors. *Reflection and Meta-Level Architecture: Proceedings of the International Workshop on New Models for Software Architecture '92*, Tokyo, Japan, November 1992. Research Institute of Software Engineering (RISE) and Information-Technology Promotion Agency, Japan (IPA), in cooperation with ACM SIGPLAN, JSSST, IPSJ.

[Yua90a] Taichi Yuasa. The design and implementation of Kyoto Common Lisp. *Journal of Information Processing*, 13(3), 1990.

[Yua90b] 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, EECS Department, December 1989. Technical Report UCB/CSD 89/544.

[Zor90] Benjamin Zorn. Comparing mark-and-sweep and stop-and-copy garbage collection. In *Conference Record of the 1990 ACM Symposium on LISP and Functional Programming* [LFP90], pages 87–98.

[Zor93] Benjamin Zorn. The measured cost of conservative garbage collection. *Software—Practice and Experience*, 23(7):733–756, 1993.