Overview of Garbage Collection in Symbolic Computing

Timothy J. McEntee

Texas Instruments

Many readers of this article have heard the term "garbage collection" on occasion without being given much explanation as to what it is. One purpose of this paper, therefore, is to describe what garbage collection is and why it is a subject of concern for computing in general and symbolic computing in particular. The first section briefly describes garbage collection. The next section gives a general description of what has come to be known as the "classical" garbage collection algorithms. A few of the classical algorithms' shortcomings for use in today's systems are described in the next section. This is followed by a section on a few techniques in garbage collection which address the needs of today's and tomorrow's symbolic computing systems.

An Introduction to Garbage Collection

A simplified model of a computer system is illustrated in Figure 1(*a*). There are two parts to the system - a processor and an address space (i.e., memory). Within the processor are several "registers" (labeled R1 thru R4), which could be hardware registers or a scratchpad memory. Within the address space are found several variable-sized *objects* (labeled O1 thru O7). An object is made up of a number of words of memory. Stored in each object word is either data or a pointer to (an address of) another object. In the same way, each register holds either data or a pointer to an object. The term *reference* is often used to mean that a register or an object has a pointer to another object; for example, register R1 references object O1.

Objects are considered "useful" as long as they are accessible to the processor. The only objects which the processor can possibly access are those which are directly referenced by one or more registers (objects O1 and O5), or those which are indirectly referenced by one or more registers (objects O2 and O3). In other words, accessible objects are those objects which are in the *transitive closure* of at least one register. Only an object directly referenced by a register can have its contents read or written to (accessed). An object which is indirectly referenced by a register can become directly referenced by making the object's address available to a register. For instance, object O2 can become directly referenced by reading the value of the first word of object O1 (note that O1 is directly referenced by register R1) and writing that value in register R4.

Object O4 is *not* in the transitive closure of any register; no path of indirection can possibly link O4 with a register. The same is true of objects O6 and O7. Once an object is removed from the transitive closure of all registers, it can never be made accessible again. Such an object is commonly termed "garbage." When an object is first created in the address space, it is "bound to" (referenced by) a register. Therefore, each of the nonaccessible objects (O4, O6, O7) were at one time accessible. A possible scenerio may have been that register R2 at one time referenced object O6. Later, a process requested that the contents of register R3 be copied to register R2. When the contents in register R2 was overwritten, the only pointer which made object O6 accessible was "crushed."

The crushing of pointers is a common occurrence in languages for symbolic computation (e.g., Lisp or Prolog). The frequent crushing of pointers to objects over a period of time results in the accumulation of a large amount of garbage. From the point of view of the processor, this fact is of little concern just as long as it can continue accessing useful objects and creating new ones. However, no system has an address space which is infinitely large. Eventually the entire address space would become completely full of accessible objects and garbage objects and no memory would be available for allocating new objects. Obviously, the solution is *not* to halt the system, nor to write over the accessible objects which the processor expects to remain intact. The best solution would be to reclaim the memory space which contains garbage and make it available for the allocation of new objects. This is the purpose of garbage collection. After the address space has been garbage collected, as shown in Figure 1(*b*), all accessible objects (O1, O2, O3, and O5) have "survived" collection. These objects may, in fact, occupy different memory addresses, but the logical structure of the objects and pointers are unchanged. The garbage blocks (O4, O6, and O7), on the other hand, have been reclaimed.

The whole process of allocation and reclamation of memory is often called *automatic memory management.* A computer system with a language like FORTRAN or COBOL does not require such management since memory can only be used as static storage. The language Pascal does allow dynamic storage, but the management of memory is left to the user of the language; the user is given the responsibility to allocate or destroy the data structures used.

A symbolic computing language like Lisp, however, has rich data structures and an intricate maze of pointers which, if left to the user to manage himself, could easily become an overwhelming burden. Thus, with the onset of the first list processing languages in the late fifties and early sixties, an interest in developing better and better algorithms for garbage collection arose and has continued to this day.

The Classical Algorithms

Mark and Sweep

This algorithm was first introduced in 1960 [McCarthy60] and can be viewed as the precursor to the copying collectors described in a later section. With modifications, it is used in some computing systems to this day; however, the copying collectors are considered more suited to the needs of today's symbolic computing systems.

As the name suggests, there are two phases to the algorithm - the mark phase and the sweep phase. In the mark phase, each object that is accessible to the processor is "marked" (for example, by turning on a bit contained in the object). To do this, each pointer, one at a time, is followed to the object it references (its *referent*), the referent object is marked, and the pointers in the cells of the referent object are then followed. The general algorithm requires a stack to keep track of the cells in already marked objects whose pointers have yet to be followed. Many variations and improvements on the mark phase, such as methods for eliminating a stack, have been made in order to improve performance and lessen overhead (see, for example, [Knuth73], [Shorr67] and [Wegbreit72]). J. Cohen [Cohen81] briefly describes many of the algorithms. In fact, Cohen's article is an excellent resource for getting an overview of reference counting and copying collectors, described later in this article.

After all accessible objects have been marked, the entire address space is "swept." In the simplest implementation of the sweep phase, all "live" objects, objects which have been marked, are passed over except for turning off their mark bit, while the memory occupied by all unmarked objects, the garbage objects, are added to a free list of available memory for use in future allocation.

Reference Counting

Another garbage collection technique called reference counting was also introduced in 1960 [Collins60]. It too has undergone many refinements. The main algorithm for reference counting is to maintain a count of the number of references to each object. As long as an object's reference count remains non-zero, it is potentially accessible. Once the object's reference count reaches zero, indicating no other objects and no registers reference it, it can be guaranteed that the object is not and never will be accessible to the processor (it is garbage). Therefore, that object's memory space can be reclaimed.

The model of a computer system used previously (Figure 1(*a*)), may be of help in understanding the general technique of reference counting. Object O1 is referenced by register R1. It is the only pointer to the object. Therefore, in the reference counting scheme, object O1 would have a reference count of 1. By the same reasoning, object O3 would have a reference count of 3. Imagine that a process requested that the pointer from register R2 to object O5 be crushed. Object O5's reference count, with a value of 2 before the request was made, would be decremented to 1. A second request to crush the pointer contained in register R3 would result in another decrement to object O5's reference count, giving it a count of zero. By definition, then, object O5 would have become garbage and could be reclaimed.

An *immediate* reference counting algorithm adjusts reference counts each time a pointer is written into a register or a word in an object. Also, an object is reclaimed as soon as its count drops to zero. Reference counting takes a considerable amount of time. Each time a pointer is written, the old contents of the register or word of memory must be read so that, if it is a pointer, the count of the referent object can be decremented. The new pointer's referent must then have its reference count increased. When an object's count diminishes to zero, each word in the object must be scanned to decrement the counts of each object they reference. The total overhead for immediate reference counting is about 20% of CPU time [Ungar84a].

An improved algorithm, called the Deutsch-Bobrow *deferred* reference counting algorithm [Deutsch76], reduces the cost of maintaining reference counts. The algorithm lessens the time spent adjusting reference counts by both storing the counts separately from the objects themselves, and postponing any updates to the reference counts for a period of time. Instead, any pending updates are stored in tables. There is statistical evidence [Clark77] that an overwhelming majority of objects (97%) in most Lisp programs will not have more than one reference to it. By postponing the updating of counts as soon as pointers are created, the chances are good that an individual object will be referenced and then its pointer crushed shortly thereafter. The system periodically stops and reconciles the reference counts all at once. Deferred reference counting requires about half as much CPU time as the immediate reference counting algorithm [Ungar84a].

Shortcomings of the Classical Algorithms

Reclaiming cyclic structures: Cyclic structures are not an uncommon data structure in symbolic computing. Looking back to Figure 1, objects O6 and O7 each point to the other. Taken together, they form a circular, or *cyclic* structure. Yet, neither is in the transitive closure of a register. The two objects are garbage. Their address space would be reclaimed after a mark and sweep garbage collection is done. The same is not true if reference counting is used for collection. Because the objects in cyclic structures will never reach zero (both object O6 and O7 will maintain a reference count of one), their address spaces will never be reclaimed. This problem, discounting the facts that extra space is needed for the reference counters and that there is a significant amount of overhead in updating the counters, makes reference counting by itself an inadequate means of garbage collection in today's systems. Combining reference counting with mark and sweep garbage collection (or a copying garbage collection, for that matter) eliminates this inadequacy. The majority of reclamation would be done via reference counting, and mark and sweep would occur as a last resort.

Virtual memory: The classical mark and sweep algorithm was originally designed for systems without virtual memory. Garbage collection for today's systems must work well with virtual memory. As J. Cohen [Cohen81] points out, in virtual memory systems, when the ratio of the size of secondary memory to the size of main memory is small, the pass through memory during the sweep phase is not a serious problem for performance. However, when the ratio of the sizes is large, the classical algorithm is no longer suitable. This is because all addresses in the address space must be swept, including addresses which are not in physical memory at the time. It will not be uncommon to find systems of the future with large virtual memory (an address space of 2^{26} to 2^{32} words of memory) or *very* large virtual memory (an address space greater than 2^{32} words of memory). In these systems, the vast majority of the addresses would be out on disk and would have to be paged in during the sweep of the address space. This would result in poor performance of the garbage collector. A better alternative is to avoid the sweep phase of collection as is done in copying garbage collection, described in the next section.

Another important part of virtual memory systems involves the *working set* of a process. The working set can loosely be defined as the objects in the address space which are "actively" accessed by the processor in order to carry out normal execution of the current process. The virtual address space is divided into pages. A finite number of pages can be in physical memory at one point in time. If an object is accessed which is not in a page of physical memory, it must be paged in from the backing store. This process of paging in adversely affects performance. Therefore, it is best to minimize accesses of objects which are not in physical memory. The best performance would be achieved if the working set of objects were kept in physical memory. As time progresses, the objects which are heavily accessed could become distributed in far more virtual address pages than there are pages in physical memory, resulting in severe *thrashing*. The solution is to *compact* the accessible objects into a minimal number of virtual address pages. This is what is meant when a system is said to "improve the locality of reference."

The original mark and sweep algorithm is generally not used in virtual memory systems since no compaction of objects in the address space occurs. Therefore, many variations to the algorithm exist in which the remaining accessible objects are relocated in the address space (see, for example, [Haddon67], [Lang72], or [Zave75]). Unfortunately, this requires one or more additional "sweeps" through the address space, further impairing performance. This fact makes the copying garbage collector algorithm even more desirable.

Concurrency: The classical algorithms were adequate for time-shared and batch-job systems. However, in the last few years there has been a shift to using interactive, single-user work stations (i.e., Lisp Machines) for symbolic computation. Thus, the garbage collector must be designed in such a way as to insure tolerable user response time in an interactive environment. User perceived pauses should be small. Furthermore, the execution of "real-time" applications is often a requirement in today's symbolic computing systems. Such applications require minimal processor interruption.

The mark and sweep garbage collector is termed a *pausing* collector (i.e.,when garbage collection is in progress, execution of regular processing is suspended). When the free memory available for allocating new objects is exhausted or nearly exhausted, the garbage collector is activated, all other processing pauses, the collector runs to completion, and regular processing then resumes. In a system with a very small address space, a pausing collector may be tolerable for the user, but the larger the memory, the more intolerable the pauses become. Dijkstra [Dijkstra78] and Steele [Steele75] each designed a system in which mark and sweep garbage collection could proceed simultaneously with normal processing. Although neither system was actually implemented, the design called for two separate processors executing in parallel - one processor used exclusively for reclaiming objects and the other for carrying out normal processing. An alternative method of time-sharing a *single* processor, as is done for the copying collectors described in the next section, has the advantage of being less complicated and costly.

Three Modern Algorithms

A type of garbage collector prevelant in today's systems is known as a *copying* collector, first proposed by R. Fenichel and J. Yochelson [Fenichel69] and C. Cheney [Cheney70]. It is suitable for use in a virtual memory since compaction of memory is a natural side-effect. All three techniques for collection described in this section are based on the copying collector.

Baker's Algorithm

Even though H. G. Baker's contribution to the copying collector was to adapt it to real-time collection, the generic copying collector is often referred to as *Baker's algorithm.* The address space is divided into two equal-sized semispaces called *oldspace* (also known as *fromspace*) and *newspace* (also known as *tospace*). During garbage collection, all accessible objects are copied from oldspace to contiguous locations in newspace. A *forwarding address* is left at the oldspace location of the object. Whenever a pointer is followed and the referent object is found to contain a forwarding address, the pointer is updated to point to the copied object in newspace. When *all* accessible objects have been copied to newspace, the entire address space occupied by the oldspace can be reclaimed. The two semispaces are "flipped" (i.e., newspace becomes oldspace and oldspace becomes the newspace for the next garbage collection). Note that, since oldspace can be reclaimed as a whole, there is no need to "sweep" through the address space looking for objects to reclaim. Also note that, since the accessible objects are copied to contiguous locations in newspace, compaction of the address space is accomplished.

The above description was rather a quick synopsis of copying garbage collection. An illustration can be a valuable aid in understanding the general algorithm. Figure 2(a) shows the memory space divided into its two semispaces - oldspace and newspace. The first thing the collector must do is copy all blocks which are referenced by the *root set.* The root set, in the case of the simple copying collector, is the set of registers contained in the processor. The registers form the "root" of all objects to be copied. Register R1 has a pointer to an object in oldspace (object O1). Therefore, object O1 must be copied to newspace. In Figure 2(b), object O1' is the newspace copy which is an exact duplicate of the original object. A forwarding reference is placed in the first word of the original object (O1) as indicated by the dotted pointer. The need for the forward reference is shown in Figure 2(c). The collector moves on to copy the object referenced by the next register (register R2). It is found to have a pointer to object O1. But, on examination of the contents of the first word of object O1, the forward reference is detected. It would be a serious error to create duplicate copies of the same object in newspace. Instead, register R2's contents is replaced by the address of object O1', which happens to be the value of the forward reference. The result is illustrated in Figure 2(d).

The purpose of the copying collector is to copy *all* accessible blocks to newspace before reclaiming the oldspace. Object O2, however, is not directly referenced by a register, so it would not be copied if collection stopped after all objects referenced by the registers were copied. Two pointers maintained by the garbage collector are used to make sure all accessible objects are copied. The first pointer points to the next available location in newspace at which an object may be copied. The pointer is initialized to point to the top of newspace. Each time an object is copied to newspace, the pointer is incremented by the number of words in the copied block. Once all objects directly referenced by the root set are copied, *scavenging* of the newspace objects begins. By scavenging the objects it is meant that the words of each newspace object is scanned to find pointers to other objects. The second pointer, also initialized to point to the top of newspace, is used to keep track of where the collector is in its sequential scan of newspace. If a pointer is found, the object it references must also be copied to newspace. In Figure 2(d), the pointer to object O2 found in the word of object O1' would eventually be scanned. The same steps are then followed : if the referent object is found to have a forward reference, the pointer is updated to point to the newspace object; otherwise, the object is copied to newspace and a forward reference is left in the original object. This scavenging of the newspace objects continues until the scavenge pointer reaches the first pointer. When the pointers meet, all accessible objects have been copied to newspace, and the two semispaces can then be flipped.

Baker [Baker78] has proposed a means of avoiding significant processor interruption which occurs during pausing garbage collection. Each time an object is allocated in the address space, a fixed number of objects are copied from oldspace to newspace. In effect, the processor time-shares normal execution with garbage collection rather than pausing normal execution in order to perform the collection in its entirety. This technique is often given the name *incremental* garbage collection. It is possible to use this technique in systems executing real-time applications because the only time normal processing is interrupted is during the "flip" of the semispaces and the fixed amount of time in which the objects referenced by the root set are copied and the root set's pointers are updated.

The two semispaces are simultaneously active. The processor must be able to access both objects which are necessary for normal computation and objects which are being copied during the incremental collection. This means that the working set is increased, which is unfortunate in a virtual memory system. However, since the algorithm gives rise to compaction of the accessible objects in memory, fewer page faults will occur during normal execution.

Generational Garbage Collection

H. Lieberman and C. Hewitt [Lieberman80], the first to propose a generational garbage collection algorithm, make the observation that in systems with dynamically allocated storage, the most common use of memory is as *temporary* storage. Most objects are created, accessed for a while, and then thrown away (i.e., their pointers are crushed). But the classical algorithms and even Baker's algorithm make no distinction between the temporary objects and those which are more permanent. All objects, temporary or not, are garbage collected at the same time in the same way. The strategy behind generational garbage collection is to take advantage of this distinction between objects.

Generational garbage collection is used in both the Berkeley Smalltalk system [Ungar84a] [Ungar84b] and the latest version of the Symbolics-3600 Lisp Machine [Moon84]. It exploits the observation that "young objects die young" [Ungar84a]. A young object is one which has been recently created (allocated). There is a high probability that it is a temporary object which will soon become inaccessible to the processor. If it does remain accessible for an extended period of time, it is quite likely that it will continue to be accessible for an extensive period of time. The basic strategy of the algorithm is to divide the address space into several *generations.* In Figure 3, the address space is divided into four generations. Generation 4 would be garbage collected less often than would generation 3, and much less often than would generation 2 or 1. Ideally, the entire address space of the youngest generation, generation 1, would always be resident in physical memory (i.e., none of its pages would be paged–out to the backing store). An object in a generation that "survives" several garbage collections, indicating its stability, is moved ("promoted") to the next older generation. The number of times an object survives the garbage collector is kept as a counter in the object. The number of survivals that are needed for promotion to an older generation can be adjusted as the work-load of the system changes.

When garbage collection is necessary for a given generation, that generation and all younger generations are collected together. This reduces the number of references into a generation that must be accounted for when collection is done. Two types of references are indicated in Figure 3 : young-to-old references and old-to-young references. In list-processing languages, it is much more common for recently–created objects to reference older objects than for objects which have been around for a while to reference recently–created objects. The old-to-young references are sparse enough that their bookkeeping does not create significant overhead. If the collector were to also keep track of young-to-old pointers, the overhead involved would become a detriment. The root set for garbage collection of a generation thus becomes : 1) the registers (just as in Baker's algorithm), and 2) all old-to-young pointers into the generation. Since garbage collection of all younger generations goes on at the same time, young-to-old pointers need not be included in a given generation's root. This is due to the fact that all younger generation objects will be scanned when they are collected and any pointers to accessible objects in older generations being collected will be discovered at that time.

Each generation can be garbage collected, using Baker's algorithm, without disturbing the older, more stable generations. Younger generations with higher garbage production rates can be garbage collected much more frequently than older generations. The overall effect is an improved system performance – valuable processor time is not wasted uselessly garbage collecting older, stable objects.

Garbage Collection Using Areas

Garbage collection is, in general, proportional to the size of the address space being collected. In a large virtual address space, the collection overhead of an unmodified Baker's algorithm would be highly prohibitive. P. Bishop's [Bishop77] strategy for reducing this overhead was to introduce the concept of *areas* in which small pieces of the virtual address can be separately garbage collected. Figure 4 shows a computer system in which the address space is divided into three areas. The number of areas can conceivably be far more than three. Garbage collection is then performed on an area-by-area basis. The frequency of garbage collection within an area can be tuned to the expected rate of garbage accumulation in that area.

When garbage collection is invoked on an area, such as Area Z in the illustration, Baker's algorithm is used. The major addition to the algorithm is that the root set of the garbage collection must be expanded to include not only the processor's registers, but also any "inter-area" pointers referencing objects in the area. If only the registers which point to objects in the area were used to determine which objects are useful, objects O6, O7, O8, and O9 would be copied to newspace. Examination of the illustration indicates that all objects in Area Z except for object O3 are accessible to the processor. Yet, the whole strategy of using areas is to avoid collecting all the areas at the same time. The solution is to keep track of all inter-area pointers into the area and copy all objects which are directly or indirectly referenced by them in addition to all objects referenced by the registers. Objects O1, O3, and O7 would thus be copied to newspace and the scavenging process would pick up the rest of the accessible objects in the area.

If objects were placed in random areas when they are created, the advantage of tuning the frequency of garbage collection on an area-by-area basis would be defeated *and* the number of inter-area pointers would become large, increasing the overhead of the system. Therefore, each area is expected to exhibit locality of reference corresponding to some logical division of the processes being executed in the computer system. Locality of reference implies that there are far fewer pointers between areas than within an area. This makes it feasible to implement a bookkeeping system which keeps track of each area's inter-area pointers.

Garbage collection using areas seems desirable for improving the performance of a computer system with a large virtual address space. For a *very* large virtual address space ($> 2^{32}$), one could argue that it becomes more of a necessity.

Garbage Collection for the Future

Garbage collection techniques have come a long way in the last 25 years. Since the first "classical" algorithms were introduced as an effective way to reclaim memory in a list processing environment, the needs of symbolic computation have grown. The requirements of tomorrow's symbolic computing systems call for garbage collectors which are responsible for the memory reclamation and compaction of virtual address spaces ranging in size from small to *very* large. The systems of tomorrow also must accomodate the interactive user and support real-time applications. Today's state-of-the-art garbage collection techniques are quite adequate for today's systems and much can be gained from understanding them. An approach taken in many research and development laboratories today is to incorporate the salient features of today's collection techniques and develop a garbage collection scheme which can meet the needs of tomorrow.

References

[Baker78] H. G. Baker, "List processing in real time on a serial computer," *CACM*, vol. 21, no. 4, pp. 280–294, April 1978.

[Bishop77] P. B. Bishop, "Computer systems with a very large address space and garbage collection," Technical Report TR-178, Laboratory for Computer Science, Cambridge, MA, May 1977.

[Cheney70] J. Cheney, "Nonrecursive list compacting algorithm," *CACM*, vol. 13, no. 11, pp. 677–678, November 1970.

[Clark77] D. Clark and C. Green, "An empirical study of list structure in Lisp," *CACM*, vol. 20, no. 2, pp. 78–86, February 1977.

[Cohen81] J. Cohen, "Garbage collection of linked data structures," *ACM Computing Surveys*, vol. 13, no. 3, pp. 341–367, September 1981.

[Collins60] G. E. Collins, "A method for overlapping and erasure of lists," *CACM*, vol. 3, no. 12, pp. 655–657, December 1960.

[Deutsch76] L. Deutsch and D. Bobrow, "An efficient, incremental, automatic garbage collector," *CACM*, vol. 19, no. 9, pp. 522–526, September 1976.

[Dijkstra78] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens, "On-the-fly garbage collection: an exercise in cooperation," *CACM*, vol. 21, no. 11, pp. 966–975, November 1978.

[Fenichel69] R. Fenichel and J. Yochelson, "A Lisp garbage collector for virtual memory computer systems," *CACM*, vol. 12, no. 11, pp. 611–612, November 1969.

[Haddon67] B. K. Haddon and W. M. Waite, "A compaction procedure for variable length storage cells," *Computer Journal*, vol. 10, pp. 162–165, August 1967.

[Knuth73] D. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, Addison-Wesley, Reading, MA, 1973.

[Lang72] B. Lang and B. Weigbreit, "Fast compactification," Technical Report 25-72, Harvard University, November 1972.

[Lieberman80] H. Lieberman and C. Hewitt, "A real time garbage collector that can recover temporary storage quickly," Technical Report 569, Artificial Intelligence Laboratory, MIT, April 1980.

[McCarthy60] J. McCarthy, "Recursive functions of symbolic expressions and their computation by machine i," *CACM*, vol. 3, pp. 184–195, 1960.

[Moon84] D. A. Moon, "Garbage collection in a large Lisp system," In *Proc. 1984 ACM Symp. Lisp and Functional Programming*, pp. 235–246, August 1984.

[Shorr67] H. Shorr and W. Waite, "An efficient nachine-independent procedure for garbage collection in various list structures," *CACM*, vol. 10, no. 8, pp. 501–506, August 1967.

[Steele75] G. L. Steele, Jr., "Multiprocessing compactifying garbage collection," *CACM*, vol. 18, no. 9, pp. 495–508, September 1975.

[Ungar84a] D. Ungar, "Generation Scavenging: a non-disruptive high performance storage reclamation algorithm," In *Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments*, pp. 157-167, Pittsburgh, Pennsylvania, April 1984.

[Ungar84b] D. Ungar, R. Blau, P. Foley, D. Samples, and D. Patterson, "Architecture of SOAR: Smalltalk on a RISC," In *Proc. 11th Annu. Int. Symp. Computer Architecture*, Ann Arbor, MI, June 1984.

[Wegbreit72] B. Wegbreit, "A space efficient list structure tracing algorithm," *IEEE TRansaction on Computers*, vol. C21, pp. 1009-1010, September 1972.

[Zave75] D. A. Zave, "A fast compacting garbage collector," *Info. Process. Letters*, vol. 3, no. 6, pp. 167-169, July 1975.

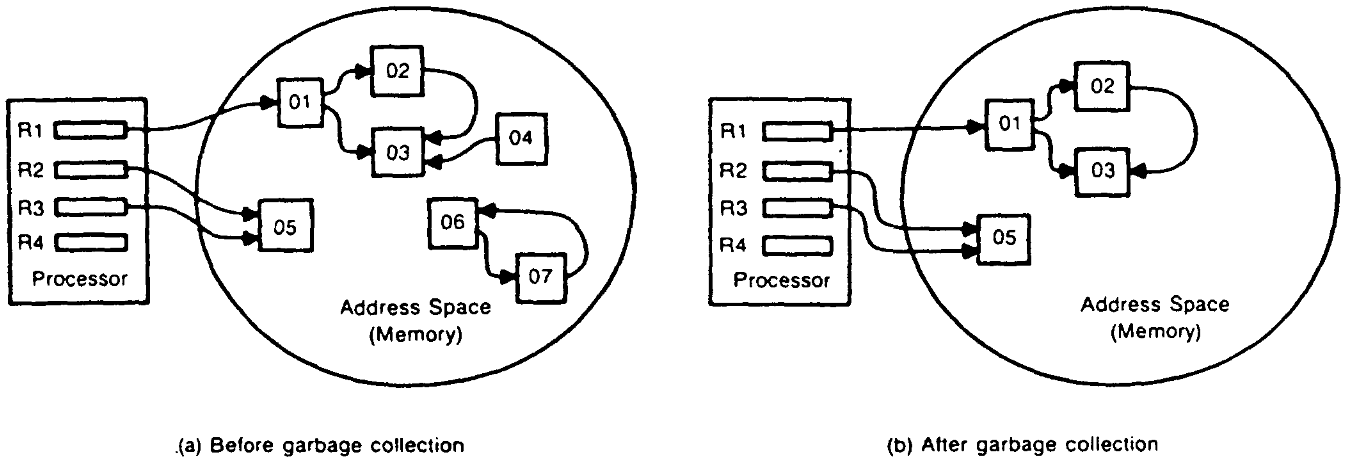

Figure 1. Before and After Garbage Collection

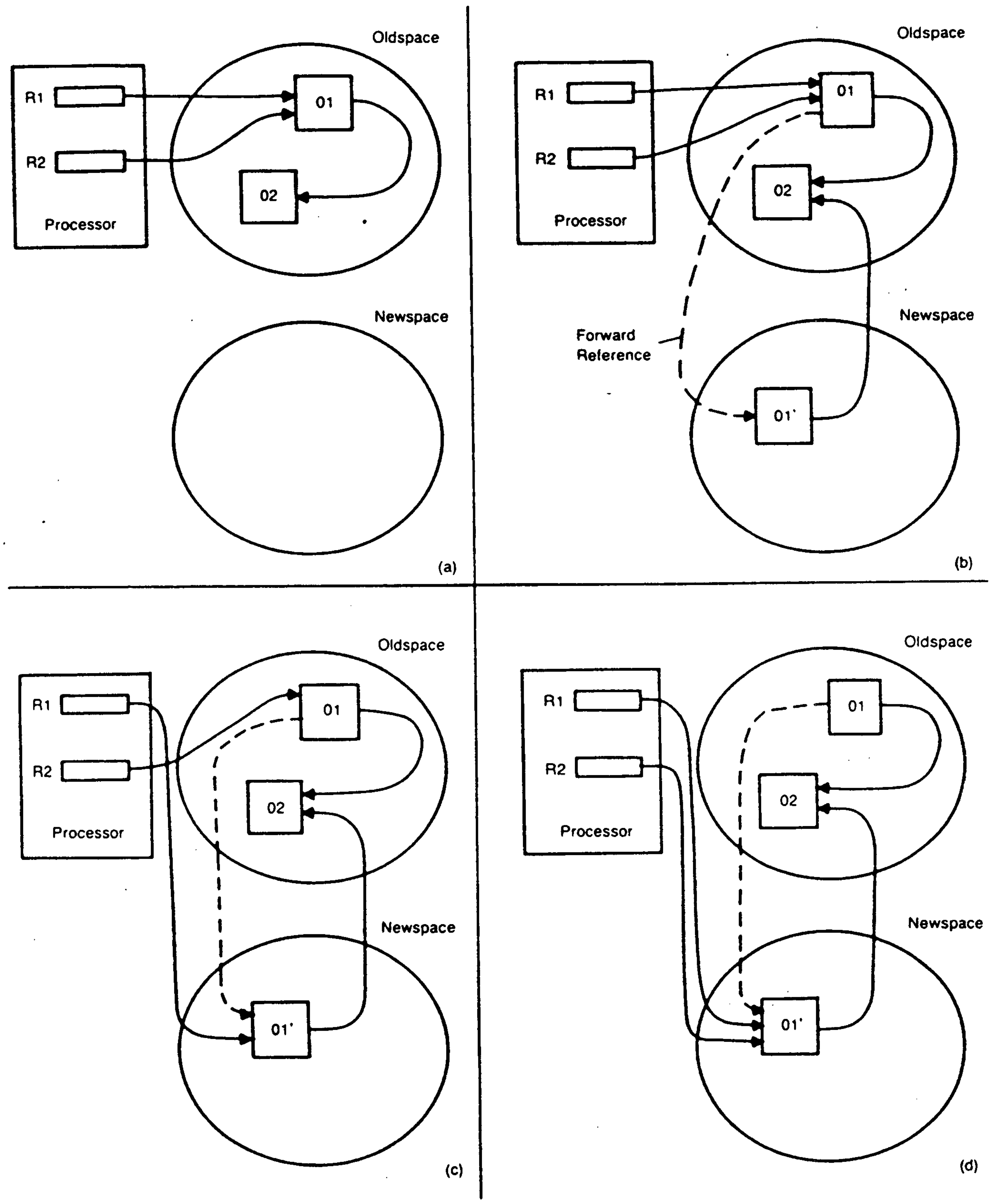

Figure 2. Copying Garbage Collection

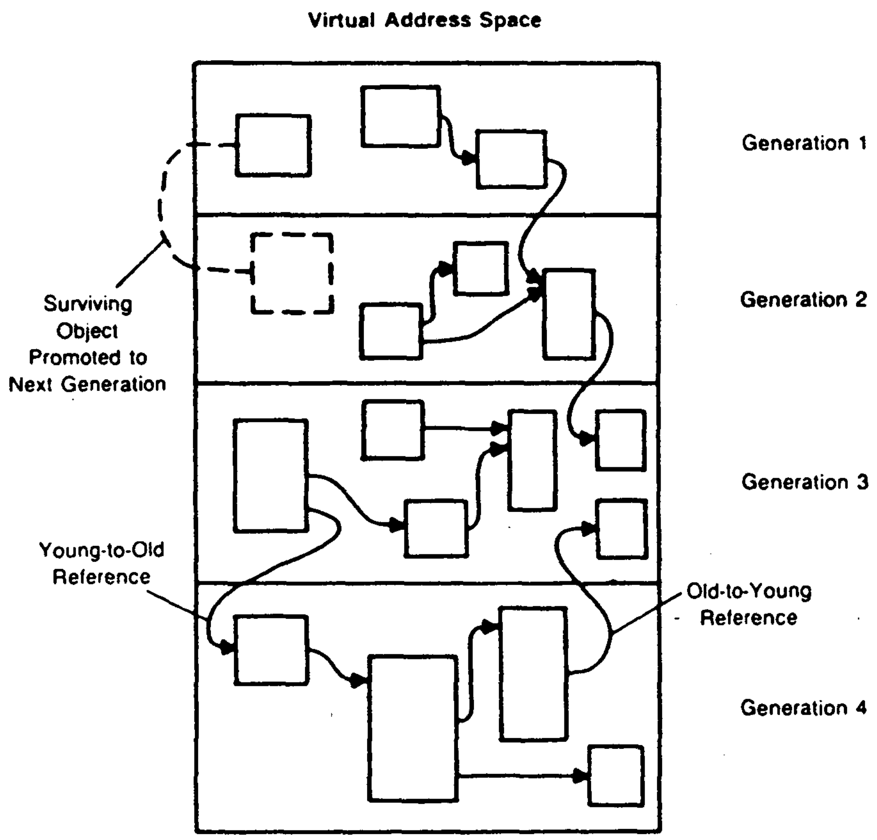

Figure 3. Generational Garbage Collection

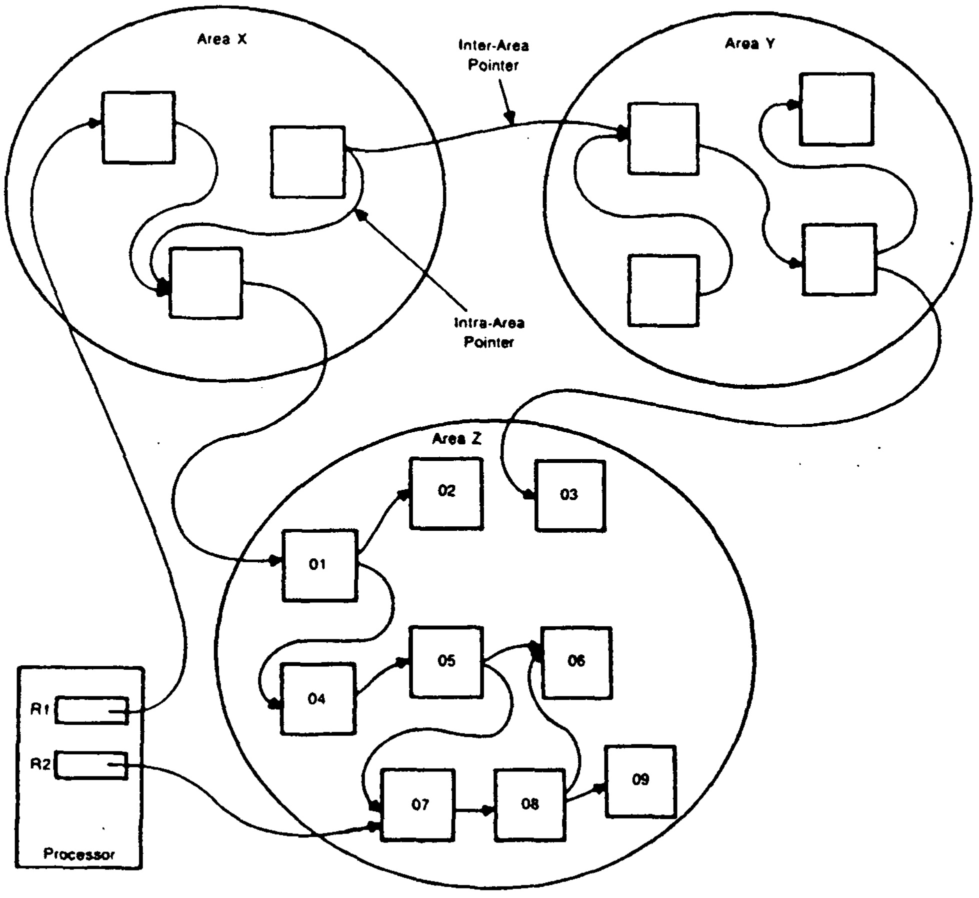

Figure 4. Garbage Collection Using Areas