Garbage Collection:
An Overview of Dynamic Memory Management

Nicholas Picone
Department of Computer Science
Haverford College
Advisor: John Dougherty
Summer 2010

Introduction

Memory in modern systems is divided into a stack and a heap. The stack is organized into frames, and the structure of these frames is determined statically at compile time. In order to assign space for an object in a stack frame, the system needs to know how big the object is at compile time. If it does not know, then the object is assigned space in the heap. The heap is an open area of memory. Objects of variable sizes can be assigned space there, and objects can be assigned space there dynamically.

The space in memory for objects in the stack is reclaimed by the system as a part of the end of procedures so that the space can be reused by other procedures. Objects in the heap, however, remain after the procedures that created them finish. Because of this, memory in the heap has to be managed separately. The heap can be managed explicitly by the programmer, but this may lead to two kinds of errors: memory leaks and dangling references. A memory leak occurs when an object takes up space in the heap but the space for that object is no longer needed. The space could be reused for a new object, but the system is not aware that the space is free. A dangling reference is a live pointer that no longer points to a valid object. The space used by the object has been marked as free, but the program contains a reference to it as if it were still there. A program that uses a dangling reference may read or write bits in memory that are now part of some other object, and this may lead to an incorrect answer. See Figures 1 and 2 for examples. The goal of garbage collection is to prevent these errors by automatically reclaiming space in the heap when it contains an object that the program no longer needs.

In addition to preventing errors, automatic memory management is useful because it gives the programmer increased abstraction. Memory management is a low-level task, and by not

having to think about explicitly deallocating space in memory, the programmer can focus on higher-level tasks. It also promotes modularity between sections of programs. Memory is a shared resource, and if a problem with memory happens as a result of something in one module, it can also cause problems in other modules. Automatic memory management allows the programmer to focus on a single module in order to understand its behavior.

This paper will give an overview of different approaches to garbage collection. It will describe reference counting followed by several approaches to tracing: mark-and-sweep, copying collection, generational collection, and incremental collection. The diagrams in this paper show a before and after look at the system's memory to illustrate these approaches. The space on the left of each half represents the stack, and the space on the right represents the heap. The lines with dots at one end are pointers, and the bare dots are null pointers. The paper concludes with a summary comparison and a look at current research in the field.

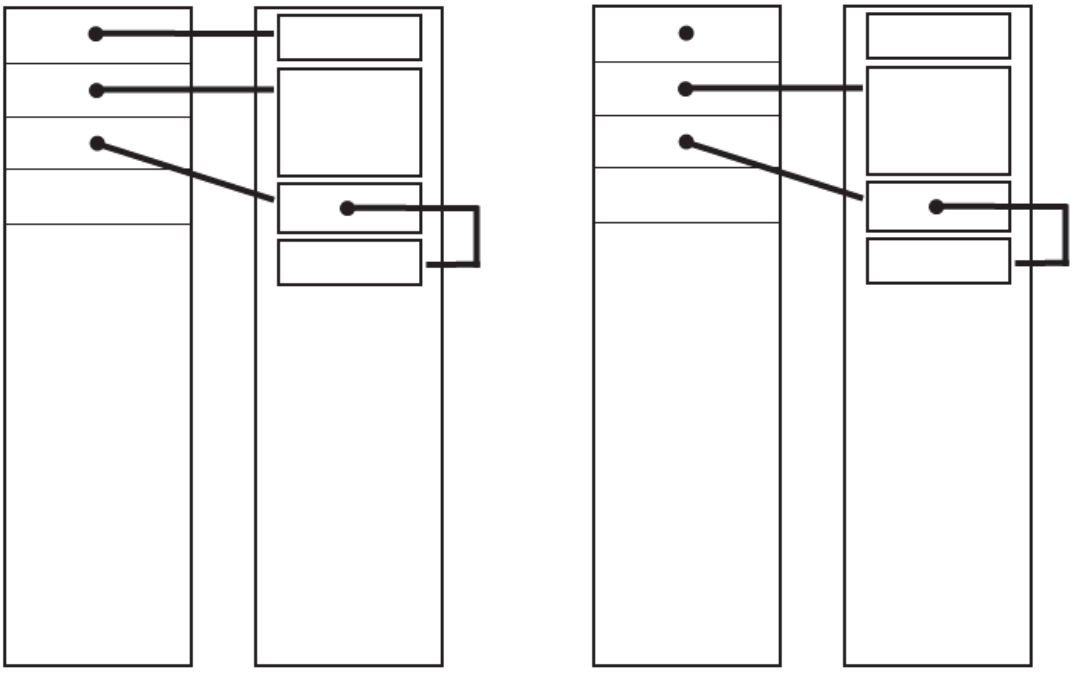

Figure 1 A memory leak. A pointer is set to null, but the space used by the object it points to is not marked as free. The object is no longer reachable, but the space cannot be reused by another object.

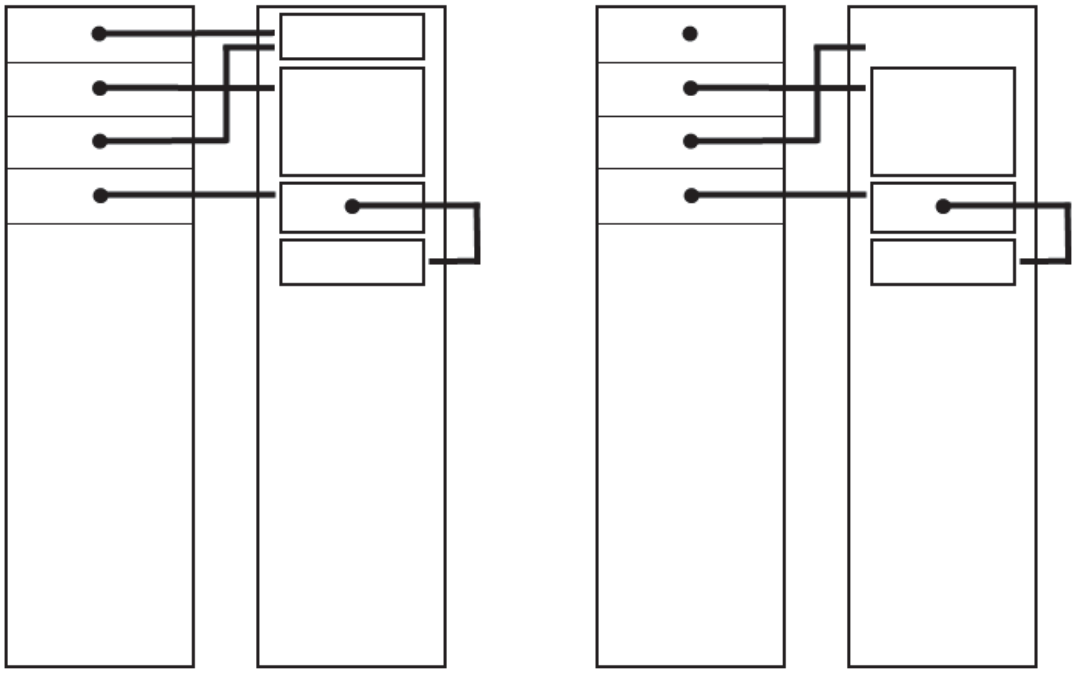

Figure 2 A dangling reference. When the first pointer is set to null, the space for the object it refers to is marks as free. However, the object has another reference to it. The space is free to be used by another object, so the second reference may refer to something unexpected later in the program.

Reference Counting

In the reference counting method, an object is reclaimed when there are no pointers to it. A counter is associated with each object in the heap to keep track of the number of pointers that refer to it. When a new object is created, its reference count is set to one. Creating a reference to an object increases its reference count by one, and deleting a reference to the object decreases its reference count by one. When the reference count of an object reaches zero, the object is reclaimed, as seen in Figure 3. If this object contains any pointers, the system decreases the reference counts of the objects referred to by these pointers and reclaims the objects if necessary.

Reference counting distributes the process of memory management throughout the program. Objects are reclaimed as soon as there are no references to them, and the process is local to each object. Reference counting frees up space in the heap more quickly than tracing

collectors, which wait until the heap is almost full before being invoked.

A disadvantage of reference counting is that the reference count is adjusted every time a pointer is updated or copied. In a program that contains a lot of pointer manipulation, this can lead to a significant amount of extra work. However, multiple increments and decrements of the same variables in a block of code can be eliminated using dataflow analysis [Appel and Ginsburg, 1998].

The major drawback of reference counting is the inability to reclaim objects that contain cyclic references. An object may have no references to it from outside the heap, but there may be a cycle of references between objects in the heap. This keeps the reference count of those objects above zero, and the objects remain in memory even though the program cannot reach them. This problem can be handled using a hybrid system which uses reference counting as the default method and periodically calls a tracing collector to collect cyclic structures.

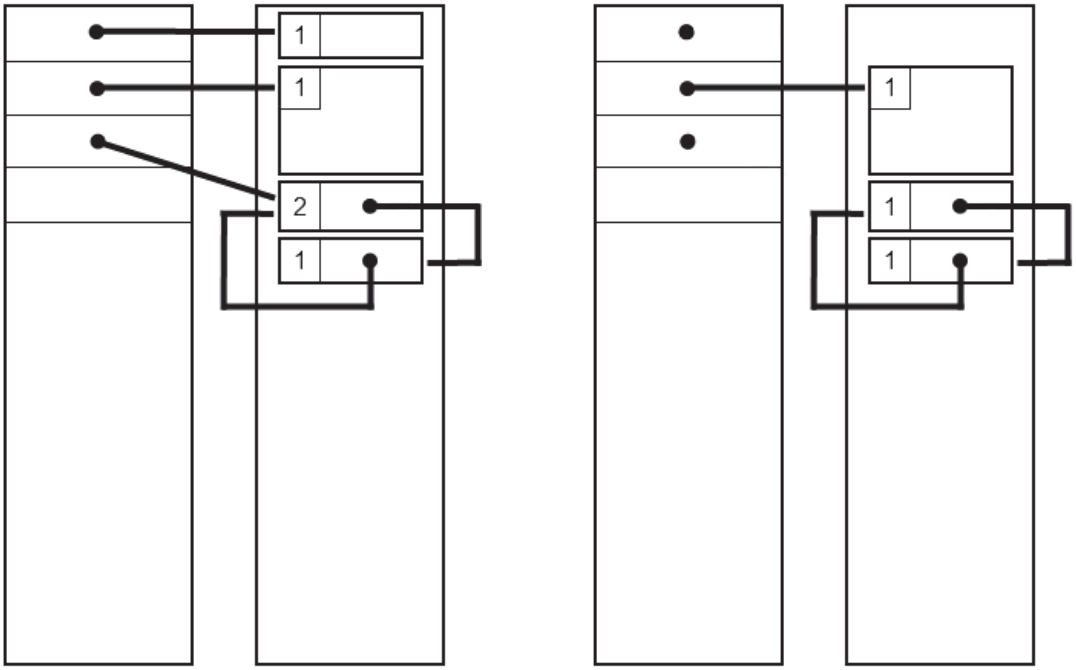

Figure 3 Reference counting. The object at the top is reclaimed when the pointer that refers to it is set to null. The two objects at the bottom remain because of a cycle of references between them that keeps their reference counts above zero.

```
allocate() =
     newcell = free_list
     free_list = next(free_list)
     return newcell

New() =
     if free_list == nil
          abort
     newcell = allocate()
     RC(newcell) = 1
     return newcell

free(N) =
     next(N) = free_list
     free_list = N
```

```
delete(T) =
     RC(T) = RC(T) - 1
     if RC(T) == 0
          for U in Children(T)
               delete(*U)
          free(T)

Update(R, S) =
     delete(*R)
     RC(S) = RC(S) + 1
     *R = S
```

Algorithms for reference counting [Jones and Lins 1996]

Mark-and-Sweep

The other category of algorithms for garbage collection is tracing. In tracing, an object is reclaimed when it cannot be reached by following a chain of pointers starting from outside the heap. Tracing involves a traversal of the entire heap or subsection of the heap in order to determine which objects are reachable and which can be reclaimed.

One form of tracing is known as mark-and-sweep. It involves two phases: the marking phase, in which the collector identifies and marks reachable objects, and the sweep phase, in which the collector reclaims unmarked objects. The collector starts at pointers outside the heap and explores all linked data, marking each newly discovered object. It then visits every block in the heap and moves the unmarked blocks to the free list. Finally, the collector unmarks the marked objects in the heap so that they can be reclaimed later if they become unreachable. See Figure 4 below.

The collector explores linked data recursively. It could do this using a stack, but the

problem with this is that garbage collection occurs when the heap is almost full, and there may not be enough room left in the heap to hold the stack. An alternate approach, called pointer reversal, involves using the pointer fields of objects in heap to keep track of the path the collector has taken. The collector reverses pointers as they are explored to point back to the previous block and marks the pointer as visited. Marked pointers are restored when the block is visited again.

The collector needs to be able to locate the pointers in an object in order to perform the marking step. This can be accomplished by giving the system access to the type information of an object, such as including a pointer to the type information in each instance of the object. Another way is a variation on mark-and-sweep called conservative collection. In conservative collection, the collector assumes that everything that looks like a pointer in a heap block is one and marks each block that appears to have a pointer to it. This reduces the amount of overhead required to use the collector, but it may result in some useless blocks being reclaimed.

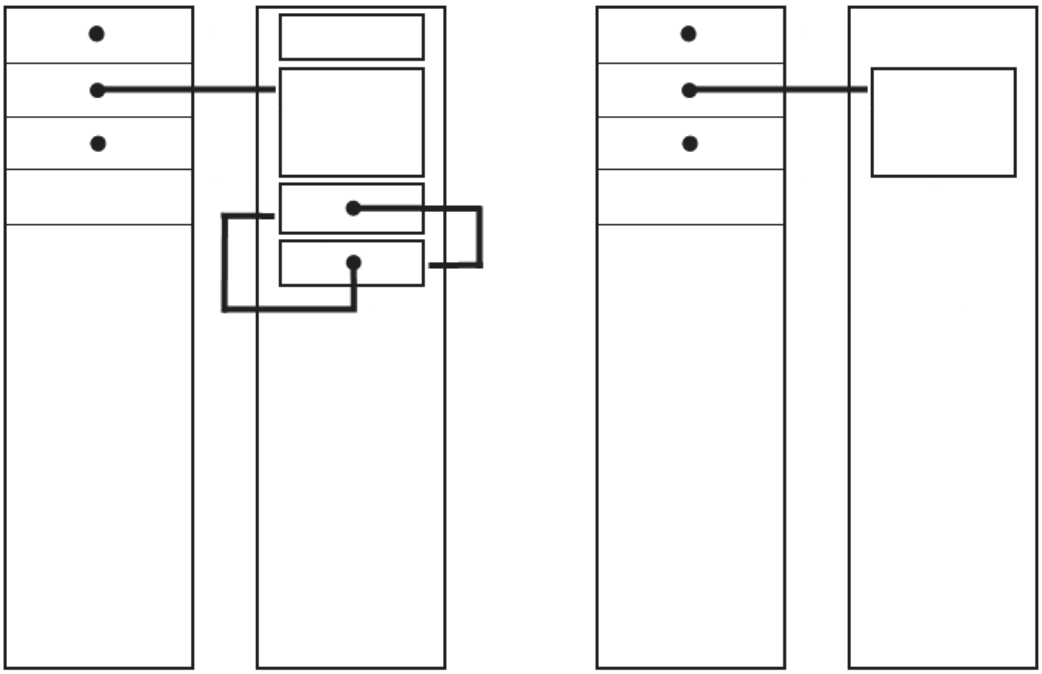

Figure 4 Mark-and-sweep. The second object in the heap is marked during the marking phase, saved during the first pass through the heap, and unmarked during the second pass through the heap.

```
New() =
     if free_pool is empty
          mark_sweep()
     newcell = allocate()
     return newcell

mark_sweep() =
     for R in Roots
          mark(R)
     sweep()
     if free_pool is empty
          abort
```

```
mark(N) =
     if mark_bit(N) == unmarked
          mark_bit(N) = marked
          for M in Children(N)
               mark(*M)

sweep() =
     N = Heap_bottom
     while N < Heap_top
          if mark_bit(N) == unmarked
               free(N)
          else mark_bit(N) =
unmarked
```

Algorithms for mark-and-sweep [Jones and Lins 1996]

Copying Collection

In copying collection, the heap is divided into two regions of equal size. Objects are allocated space in one of the halves, and the other half is left free. The half currently in use is called from-space, and the free half is called to-space. When from-space fills up, the collector explores it and copies reachable objects to to-space. Once this is complete, the collector switches its notion of which half is from-space and which half is to-space and continues, allocating space for new objects in the new from-space. All of the useful objects have been copied to the new from-space, so the objects in the old from-space can be safely overwritten during the next round of garbage collection. See Figure 5 below.

When copying objects from from-space to to-space, the objects in from-space need to be marked as copied in case they are visited again by another reference. This can be achieved by leaving a forwarding address pointer in the old cell that points to the copied cell. When visiting a cell, the collector first checks if it has a forwarding address. If it does not, then the collector copies the cell, sets the reference it followed to get there to point to the new cell, and explores

the cell. If it does, then the collector simply sets the reference it followed to point to the copied cell. It does not explore the cell because this was already done when the cell was copied.

Copying collection both gets rid of garbage and compacts data. This simplifies the process of allocating space in the heap because free space is always contiguous; other collectors may result in fragmentation. Another advantage of copying collectors is that they only visit useful blocks in the heap. Mark-and-sweep, on the other hand, visits useful blocks during the marking phase and all the blocks in the heap during the sweeping phase.

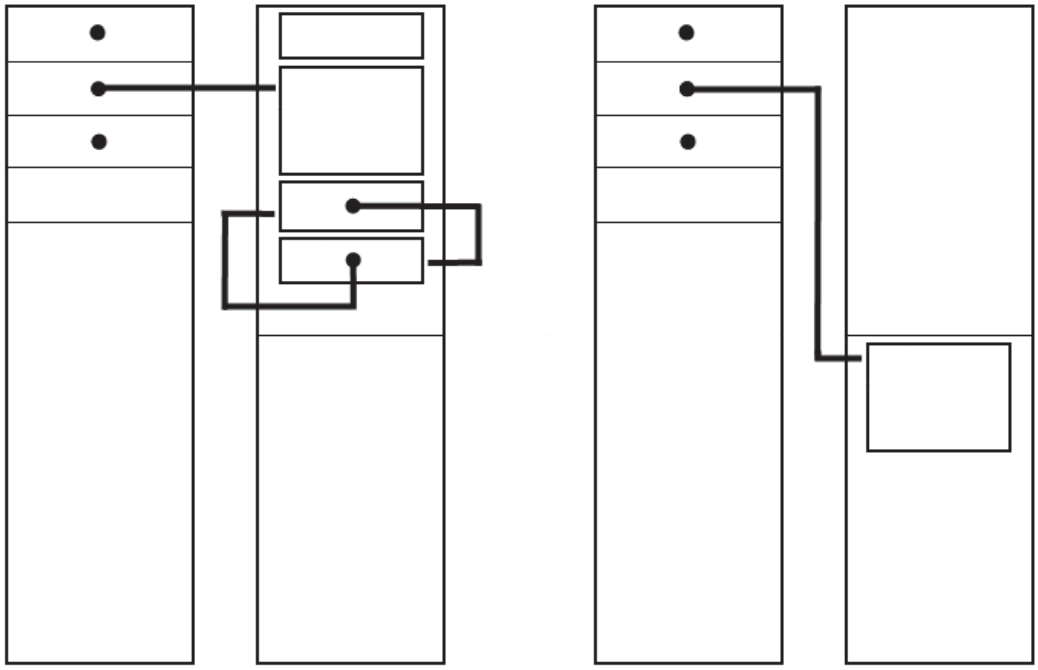

Figure 5 Copying collection. The object found during the trace is moved, and the space in the first half of the heap is free to be overwritten during the copying phase of the next round of collection.

The downside to copying collection is that only half of the heap can be used at a time. This means there is less room for objects in the heap, and as a result the collector gets run more frequently than other collectors.

```
init() =
     Tospace = Heap_bottom
     space_size = Heap_size / 2
     top_of_space = Tospace + space_size
     Fromspace = top_of_space + 1
     free = Tospace

New(n) =
     if free + n > top_of_space
          flip()
     if free + n > top_of_space
          abort
     newcell = free
     free = free + n
     return newcell

flip() =
     Fromspace, Tospace = Tospace, Fromspace
     top_of_space = Tospace + space_size
     free = Tospace
     for R in Roots
          R = copy(R)

copy(P) =
     if atomic(P) or P == nil
          return P
     if not forwarded(P)
          n = size(P)
          P' = free
          free = free + n
          temp = P[0]
          forwarding_address(P) = P'
          P'[0] = copy(temp)
          for i = 1 to n-1
               P'[i] = copy(P[i])
     return forwarding_address(P)
```

Algorithms for copying collection [Jones and Lins 1996]

Generational Collection

Generational collection is based on the observation that most dynamically allocated objects are short-lived. Generational collectors work by concentrating on recently allocated data,

where there is a higher proportion of garbage. The heap is divided into two or more regions, and new objects are allocated space in the youngest region. When space runs low, the collector explores the youngest region and moves any objects that survive collection to the next older region. Over time garbage may also accumulate in this region, and when necessary the collector explores the region and moves objects to the next older region, and so on.

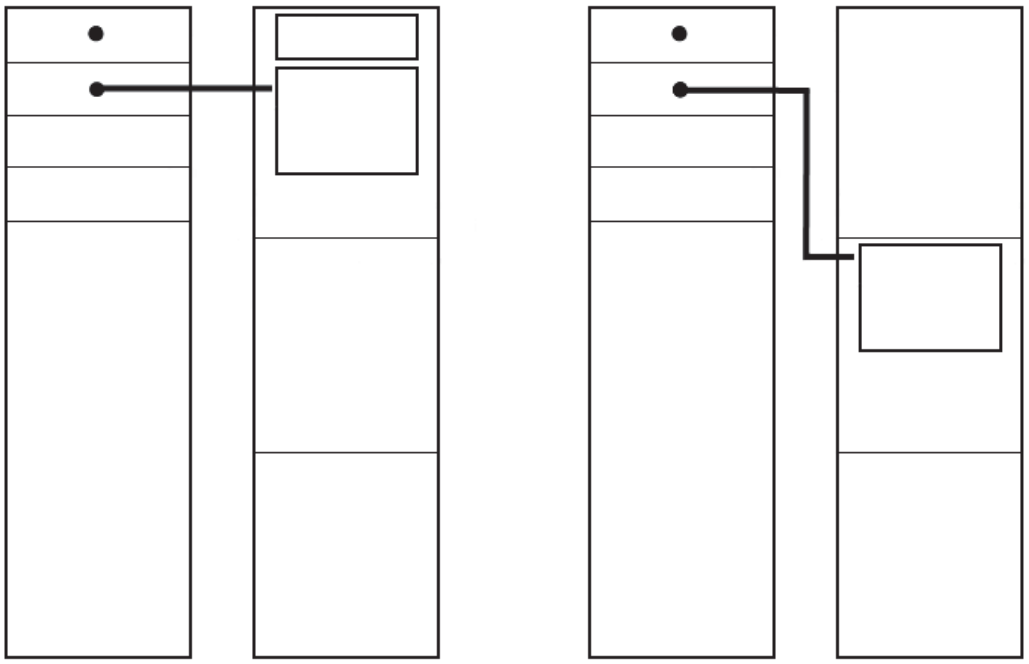

Figure 6 Generational collection. The object found during the trace is moved, and the space in the first section of the heap is free to be used for allocation.

There may be pointers between objects in different regions of the heap. Pointers from older objects to newer objects are noteworthy for two reasons. First, these pointers are starting points for exploring data in the youngest region along with pointers from outside the heap. Second, if the object one of these pointers refers to moves from a younger region to an older one, the pointers need to be updated. For this reason, these old-to-new pointers need to be noted, such as by checking for this at assignment and adding them to a list if that is the case.

Incremental Collection

A disadvantage of tracing is the amount of time it takes. Tracing algorithms deal with whole sections of the heap at once, and this may lead to long pauses in the program. Incremental collectors interleave garbage collection with the program execution. This avoids long delays but adds overhead to the program to maintain invariants that the collectors expect to be true when they resume.

One approach to incremental collection uses tricolor marking of heap objects. Objects in the heap may be white, gray, or black. White objects have not been visited by the tracer. Gray objects have been visited and marked or copied, but their children have not yet been examined. Black objects have been marked or copied, and their children have also been marked or copied. Using this system, the collector can keep track of its progress through the heap as it pauses and resumes collection.

All objects in the heap start out white. When an object is visited, the collector colors it black and colors its children gray. The collector moves through the heap, turning gray objects black and white objects gray. In order to resume and know where to continue, the collector needs to keep track of the current gray objects. When there are no more gray objects, all the remaining white objects are considered garbage.

The invariant of this approach is that no black object points to a white object. Since the main program is also running, it needs to take steps to preserve this invariant. For example, whenever the program stores a pointer to a white object into a black object, it colors the white object gray and adds it to the set of gray objects.

Another approach to incremental collection is Baker's algorithm [Appel and Ginsburg,

1998]. Baker's algorithm is an incremental approach to copying collection. At the beginning of the collection process, the objects pointed to from outside the heap are copied and the main program resumes. When the main program allocates space for a new object, the collector also scans a few pointers and copies objects from from-space.

The invariant of this approach is that the main program only handles pointers to to-space. That way objects in from-space are free to be moved by the collector without having to find and change references to the object. To preserve this, the program checks if a pointer refers to an object in from-space whenever the pointer is used by the program. If it is, then the object referred to by the pointer is immediately copied to to-space.

The following chart summarizes advantages and disadvantages of each method:

	Reference counting	Mark-and-sweep	Copying	Generational	Incremental
Fragmentation	Yes	Yes	No	May occur in oldest region	Tricolor marking - yes, Baker's algorithm - no
Handles cyclic references	No	Yes	Yes	Yes	Yes
Able to use entire heap for allocation	Yes	Yes	No	No	Tricolor marking - yes, Baker's algorithm - no
Adds overhead to program operations	Yes - maintaining reference counts	No	No	Yes - checking for old-to-new pointers	Yes - maintaining invariants
Distributes collection	Yes	No	No	No	Yes

Current Research

Garbage collection is an active research area. Pizlo et al [2010] have developed a system that allows for fragmentation but still provides predictable time and space performance for allocation and heap access. It does this by taking separate approaches to collecting the variable-sized and fixed-sized parts of objects. The elements of arrays and other objects are allocated space in one region of the heap, and array metadata is allocated space in another region. The space in the first section is composed of fixed-sized regions and is collected using mark-and-sweep. The array metadata in the second section varies in size, so this section is collected using copying collection.

Recent work has also looked at ways to perform parallel garbage collection to take advantage of systems with multiple processors. Barabash and Petrank [2010] have written about the parallel processing of tracing collection. Systems that use this approach need to be scalable in order to take advantage of multiple processors. They propose ways to modify the object graphs of linked data to allow for this.

An approach to parallel garbage collection was recently proposed by Anderson [2010]. His system uses a form of generational collection in which the first generation is private and thread-local. These generations are collected independently of other threads, and objects which survive collection are moved to a public heap. The aim of this approach is to try to maximize the cache hit rate by having threads reuse the space for their individual, private generations. Making the first generation thread-local also allows the system to use the same processor and cache space that was used for the main process to be used for garbage collection.

Bibliography

Todd A. Anderson. Optimizations in a private nursery-based garbage collector. *Proceedings of the 2010 International Symposium on Memory Management*, pages 21-30, Toronto, Canada, June 2010.

Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 1998.

Katherine Barabash and Erez Petrank. Tracing garbage collection on highly parallel platforms. *Proceedings of the 2010 International Symposium on Memory Management*, pages 1-10, Toronto, Canada, June 2010.

Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley, 1996.

Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. *Proceedings of the 2010 International Symposium on Memory Management*, pages 146-159, Toronto, Canada, 2010.

Michael L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2009.