Uniprocessor Garbage Collection Techniques

Paul R. Wilson

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

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

1 Automatic Storage Reclamation

Garbage collection is the automatic reclamation of computer storage [Knu69, Coh81, App91]. While in many systems programmers must explicitly reclaim heap memory at some point in the program, by using a "free" or "dispose" statement, garbage collected systems free the programmer from this burden. The garbage collector's function is to find data objects[1] that are no longer in use and make their space available for reuse by the the running program. An object is considered *garbage* (and subject to reclamation) if it is not reachable by the running program via any path of pointer traversals. *Live* (potentially reachable) objects are preserved by the collector, ensuring that the program can never traverse a "dangling pointer" into a deallocated object.

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

1.1 Motivation

Garbage collection is necessary for fully modular programming, to avoid introducing unnecessary inter-module dependencies. A routine operating on a data structure should not have to know what other routines may be operating on the same structure, unless there is some good reason to coordinate their activities. If objects must be deallocated explicitly, some module must be responsible for knowing when *other* modules are not interested in a particular object.

[1] 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.

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

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

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

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

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

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

In the rest of this paper, we focus on garbage collectors that are built into a language implementation. The usual arrangement is that the allocation routines of