

Wilson - Uniprocessor Garbage Collection Techniques

Comments on:

Wilson, Paul R. (1992). Uniprocessor Garbage Collection Techniques. Preprint. Final paper appeared in *Proceedings of the 1992 International Workshop on Memory Management* (St. Malo, France, September 1992). Preprint available on the Web at <ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps> (Undated; Accessed 27 April 2007).

This question is about the history of garbage collection. In the reading, *Revenge of the Nerds*, garbage collection made it as one of the nine points that made LISP different. So I am assuming that John McCarthy was the first person to use garbage collection. Does this mean that all functional languages like LISP have to use garbage collection since they do not use a stack based environment? If so, can we assume that garbage collection was invented as a result of functional programming languages?

Almost every functional programming language uses a garbage collector. However, it seems that garbage collection as a research area really took off when object-oriented programming started to take advantage of it. Note that it is a bit of a misinterpretation to say that LISP is not stack based - procedure calls often involve a call stack. However, it is often the case that there are multiple stacks.

I am also curious as to how designers test the garbage collection methods that they have created. The reading states that one of the problems with explicit deallocation of memory is that bugs fail to show up repeatedly. I am not quite sure how garbage collection avoids this erratic behavior of results during testing.

Garbage collectors are written with care to be logically correct. Abstracting the problem of garbage collection away from the rest of the program also helps ensure that it is correct.

Is garbage collection commonly used in small computer devices and embedded systems that have limited resources, or is it uncommon because these are real time devices.

Do statically-typed languages make garbage collection easier?

I think the reading answers this question.

In the footnote on page 9, the author says that read/write operations on files are more expensive on modern machines. Why is this so?

The author says that compared to memory operations, read/write operations on files are much more expensive. The obvious reason - memory is fast, disks are comparatively slow.

Some data collection techniques such as Copying Garbage Collection require that the program stops for the reclamation phase. Programmers writing code for critical systems such as life support machines must therefore take this into consideration and either implement application-specific garbage collection or must resort to the more inefficient reference counting technique. Are there any other garbage collection

techniques that do not require the program to stop?

The reading explains a number of techniques by which the garbage collector can run in parallel with the main program.

What techniques for Garbage Collection are used by C, Scheme, and Java, and why were those techniques chosen?

Standard C does not do garbage collection. C was designed so that programmers explicitly allocated and deallocated memory. The core design feature of C is that it's high-level assembly, so such direct manipulation makes sense.

Each implementation of Scheme is free to choose its own garbage collection strategy. It looks to me like Chez Scheme uses a generational garbage collection strategy (<http://www.scheme.com/csug7/smgmt.html>). As the Wilson article suggests, generational collectors tend to balance the competing demands put on collectors (efficiency, locality, overhead, time per collection, etc.)

An article in Java World (<http://www.javaworld.com/javaworld/jw-03-2003/jw-0307-j2segc.html>) suggests that Java has used a variety of garbage collectors and currently uses a variety. It sounds like the first version was mark and sweep. My guess is that they found it easiest to implement (correctly). There are now a lot of versions so that programmers can, if necessary, choose the one that best suits their purposes.

The author on page 16 in reference to copying collectors states, "all garbage collection strategies involve similar space-time tradeoffs". Is this statement true for any garbage collection strategy including those that may be invented at a later time or for all currently known strategies?

I cannot envision a strategy that would not require less garbage collection if there were more memory, but I'm not sure there is a formal proof of that claim available.

In the start of the paper Wilson states the advantages of both the Incremental and Generational schemes for garbage collection, without indicating a preference for one over another. However, on page 5 he states that one advantage of reference counting method, which is used to distinguish live objects from garbage, is the incremental nature of most of its operations. Here he seems to prefer an incremental scheme. Is one clearly better to use than another or is better to use one in some cases and another in some, which is the idea I had got in the start.

However, Wilson also states some significant disadvantages of reference counting. The key issue is that different applications have different expectations of the garbage collector. Most systems want the system to be efficient, others need to support real-time computation. I think the reading of Wilson as preferring the incremental scheme is a significant misreading.

Since sweep based collection strategies can be arbitrarily efficient based on the frequency with which they are run, the amount of garbage that collects is also (usually) going to increase as the frequency decreases. Is there a good way to detect the amount of deallocation that is happening at runtime and adjust the frequency to that level?

Since there is no explicit deallocation, the only way to figure out how much stuff is garbage (or, more conservatively, not reachable) is to apply some sort of garbage collection technique. If the algorithm runs in time proportional to the reachable values, rather than the total space (or amount of garbage), it's okay if a lot of garbage gathers.

The article states on page 2 that "explicit allocation and reclamation lead to program errors..." and then goes on to explain some of the problems with explicit allocation and reclamation. Despite these problems which we have seen in abundance in the C programming language its very popular, whereas a language like scheme which has automatic garbage collection isn't very popular. How important is automatic garbage collection and what explains C's popularity when it does not have this feature?

C is much less popular today than it was a decade ago. You'll also note that C++ seems much less popular than Java. Why? One reason is that Java does garbage collection. I do believe that automatic garbage collection is included in so many modern languages (Java, Perl, Python, Ruby, etc.) because most programmers have acknowledged that it is useful except in a few situations in which the programmer needs to get the best possible performance from the system, and chooses to rely on more explicit low-level control.