# Lecture Notes on Advanced Garbage Collection

15-411: Compiler Design
André Platzer

Lecture 21
November 4, 2010

## 1 Introduction

More information on garbage collection can be found in [App98, Ch 13.5-13.7] and [Wil94, Section 1-2] at http://www.cs.cmu.edu/~fp/courses/15411-f08/misc/wilson94-gc.pdfweb.

## 2 Generation Scavenging

The basic observation behind generation scavenging garbage collectors is that old objects seldom die. New objects either die quickly or become old objects. The principle is to partition memory into regions $R_1, \ldots, R_n$ and allocate all new memory in $R_1$. Whenever a memory region $R_i$ is full, we run garbage collection with copying into $R_{i+1}$. The exception is the last memory region $R_n$, where garbage collection algorithms other than mark-and-copy could be used. The effect of generation scavenging is that old objects successively walk towards $R_n$ and will not have to be touched during most garbage collection runs. In order to prevent explicit traversal of the old memory regions $R_n$, lists of all references or pages referenced are sometimes used.

## 3 Garbage Collection Complexity

Let $R$ be the number of words of reachable heap objects and $H$ be the size of the heap. Strictly speaking, we only have to visit pointer data in the

reachable heap objects, but we ignore this for simplicity. The mark phase takes time $O(R)$ for depth-first search, because each reachable object needs to be visited once. The cleanup phase (e.g., sweep) takes time $O(H)$. That is the total cost of a mark and free or mark and sweep garbage collector run is of the form $cR + dH$ for some constants $c, d$, typically $c > d$. That is a notable complexity. But the point is that we only need to do it occasionally. The beneficial outcome of running garbage collection is that it frees $R - H$ words of memory. Thus the cost of garbage collection per word that it frees is

$$\frac{cR + dH}{H - R}$$

After all, we only need to run garbage collection again after we ran out of memory again, which now contains $H - R$ free words more than before. This is the *amortized cost*. From this amortized cost, we can directly read off that garbage collection has a high cost if we run it all the time for reclaiming every single word of unused memory. Then the denominator $H - R$ is small and the total amortized cost high. Yet if we wait long enough to reclaim more memory ($H - R$ is large), then the benefit is large and the amortized cost low. If $H$ is much larger than $R$, then the amortized cost per word is approximately $d$.

If the garbage collector determines after a run that the ratio $R/H$ of used to total memory is large, e.g., $R/H > 0.5$, then its benefit is limited and it can ask the operating system for more memory.

## 4 Schorr-Waite Pointer-Reversal Marking Algorithm

The Schorr-Waite algorithm for garbage collection works without a stack. It works by pointer reversal instead. Before visiting a child node, the Schorr-Waite algorithm will change the child pointer to point back to the parent node instead. When the algorithm returns back to that node, it restores the old pointer value. The algorithm does not need a stack, because it uses the graph structure to store the search information instead. But it needs a counter at each edge to store which child to visit next. This counter can be small because it only needs to be as large as the maximum number of pointers in any data structure. See Figure 1.

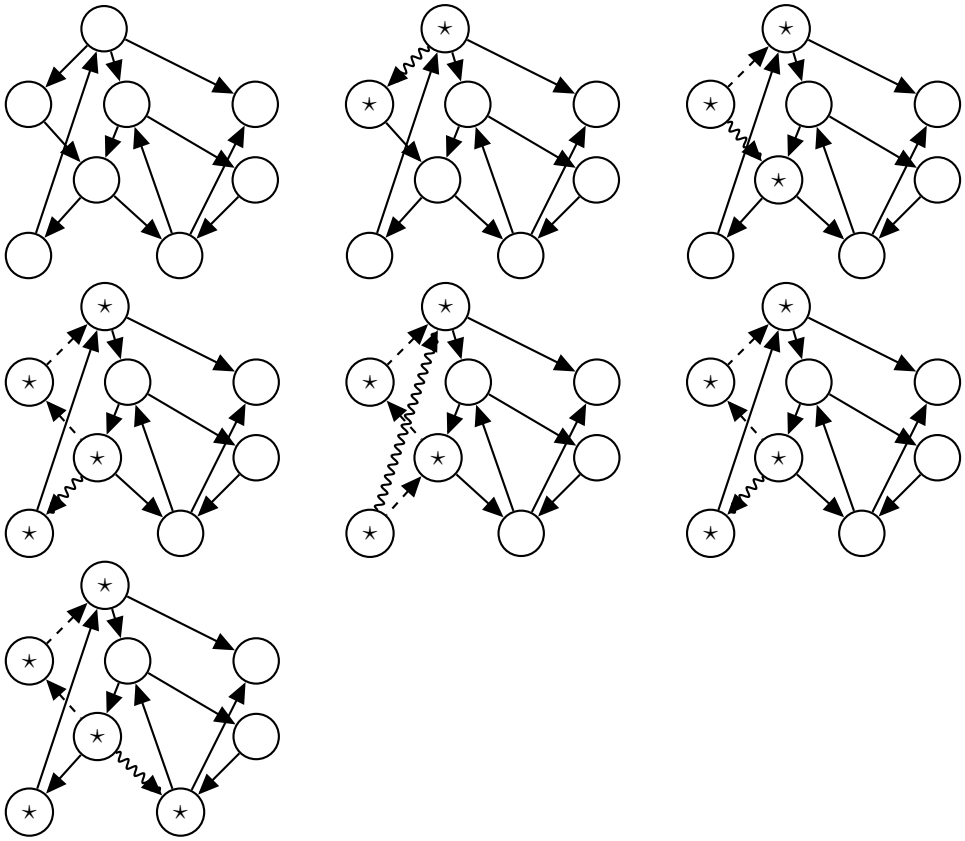

Figure 1: Start of Schorr-Waite run on an object graph

## 5 Incremental Garbage Collection on the Fly

The major problem with the previous garbage collection algorithms is that the program needs to be suspended during the garbage collector run. If the program keeps running concurrently with the garbage collector, it can modify arbitrary pointers. Thus, without suspending the program, the simple mark algorithms cannot ensure that an object that had never been referenced during the garbage collector's marking phase is still unreferenced at the time of freeing the memory. The program might have just deleted a reference from object $a$ to $b$ before the garbage collector visits $a$. If the program then inserts the pointer to $b$ into a different object $c$ later, the garbage collector will never know if it had visited $c$ already.

In principle, the converse problem also exists. The algorithm cannot

ensure that it found all unused heap objects because some might have gone unreferenced since the last check. But this converse problem is not critical at all, because the worst that happens is that some unused memory will remain allocated until the next garbage collector run.

The practical problems of garbage collectors that have to suspend the program run are obvious. The performance becomes unpredictable, because there is no good way of knowing when the garbage collector will interrupt the program and do its job. This makes this form of garbage collection useless for problems with real-time requirements.

Incremental garbage collection due to Dijkstra et al.'s *garbage collection on the fly* algorithm works by tricoloring objects. We distinguish between objects that have not been visited yet (marked *white*), objects where depth-first search is in progress (marked *grey*) and objects that have been visited in full, including all children (marked *black*). When the garbage collector is finished, it will free all objects that have been marked *white*. In order for this to work, the program needs to inform the incremental garbage collector about the pointers it changed.

More precisely, every pointer assignment $b = a$ in the program will change the marking of the target object at address $a$ from *white* to *grey* (if it is *white*, otherwise the marking will stay unchanged). This is one example of a *write barrier*, i.e., an algorithm where something needs to be done at every write. This principle ensures the following invariants, which the program cannot spoil (as long as it sticks to the above pointer assignment tagging principle):

1. *black* objects never point to *white* objects, because the garbage collector would only mark an object to be *black* after all children have been, and the program never changes the marks of *black* objects.

2. Every *grey* node is still on the list of objects that the garbage collector will look at before freeing it (possibly in next garbage collector run).

After the garbage collector is done with a full pass through memory, *white* objects are indeed unused and can be freed.

The main complexity of this approach is the need to have a quick operation for conditional marking that turns *white* markings into *grey* markings at pointer assignments. One source of trouble is concurrency of the program. A reliable implementation of the write barrier requires expensive synchronization of program access to the marking and garbage collector marking operations.

Dijkstra et al.'s garbage collector can be refined to a write barrier that updates only some *white* pointers to *grey*, not all of them. If the *white* pointer $a$ is stored into a *black* object $b$, then $a$ is marked *grey*. While there are some variations of the write barrier and also read barrier algorithms, they all share this problem.

A different implementation of the write barrier is due to Boehm, Demers and Shenker that uses the virtual memory management unit of the CPU to implement markings. The way this works is that a memory page in which all objects have been marked *black* is changed to read-only. Then if the program (without any need for synchronization) accesses this page with any write operation to any data field, a page fault is signalled and the operating system guarantees that the page fault handler will be executed before the program. This page fault handler will then mark all objects on the possibly invalidated page to *grey* and change the page to read-write again. This results in higher granularity but reduced synchronization cost.

## 6 Interface to Compiler

What the compiler has to provide to an informed garbage collector is runtime information about type layout either statically or dynamically with self-identifiers (for languages with polymorphic types). For instance, the first entry of every data record could point to a type descriptor declaring the size of the object and the offsets of each pointer field.

Problems arise for pointers that are only stored in registers, not on the stack. These are either treated conservatively (impossible for copy algorithms and compactify algorithms), or by providing sufficient register type information similar to debugging information, or by ensuring that register references are always somewhere on the stack (which can be surprisingly subtle).

The garbage collector and compiler first need to coordinate at which sites a function can be interrupted. Especially, whether the program has to be interruptible by the garbage collector at every point in the control flow, which requires exhaustive pointer identification information. The alternative is to agree on periodically occurring points of the program at which garbage collection is safe. For instance, at all procedure calls and returns and at all backward edges (from loops) in the control flow graph. Another convention is to enforce a global partition of registers into pointer-only registers and nonpointer-only registers. Then the site information reduces to function site information.

The most common choice is for the compiler to identify pointer types by runtime information. The tricky part is that this can change at each program location. For each possible site (especially function sites) we need a pointer map describing the location of all live pointers in the register or on the stack. Unfortunately, callee-save registers need special attention, because the callee does not know if the values in the registers are pointers, since that depends on who called. Consequently, the caller $g$ needs to identify which of the register that its callee $h$ will have to save are pointers, and which of the registers may be a pointer, because it didn't overwrite them, depending on the callee-save registers of its own caller $f$ (here $f$ calls $g$ calls $h$). The best way to identify function sites themselves is indexed by return address, because these are stored in the stack frame and can be found out when walking the stack top down.

In single-threading programs, the problem simplifies a little bit, because garbage collection will only occur when **alloc** is called, or when another function is called (which could call **alloc**).

## 7 Derived Pointers

Garbage collection may interfere with code optimization. For example, if a loop accesses $a[i - 100]$ for all $i$, then an optimizing compiler may compute this address as

$$\begin{aligned} b &\leftarrow a - 100 \\ m &\leftarrow b + i \\ v &\leftarrow M[m] \end{aligned}$$

The compiler may even move the constant expression $b \leftarrow a - 100$ out of the loop to avoid having to recalculate it every time. Yet, what do we tell garbage collection about what $b$ represents? Is it a pointer? Or is it not? We can't ignore it altogether if it's the only location where we store the address. We can't just pretend it is a pointer either, because $b$ itself is not the base address of the object and will just point to a meaningless address. Consequently, the pointer map will have to identify $b$ as a derived pointer and will need to name the displacement constant $100$ so that the garbage collector can readjust $b$ to the appropriate base pointer $b + a' - a$ when it relocates $a$ to a different place $a'$ in memory during copy or compactify garbage collectors. Otherwise, the derived pointer will be broken after a relocating garbage collector run. In addition, we need to make sure that

the real base address $a$ is still live even when only the derived pointer $b$ is used, because, otherwise, the garbage collector may be entirely confused about where the object really starts.

## References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Submitted to ACM Computing Surveys, 1994.