Compilers: Compiler Design

Lecture Notes on Garbage Collection

André Platzer

Carnegie Mellon University || Karlsruhe Institute of Technology

Lecture 20

1 Introduction

In the previous lectures we have considered a programming language C0 with pointers and memory and array allocation. Until now, C0 had no way of getting rid of allocated memory again when it is not needed anymore. There are basically two solutions. One is to use manual memory management by adding a **free** operation to the language. The other is to add automatic garbage collection instead, which will take care of freeing memory automatically when it is not used anymore. Requiring **free** seems like it would be convenient for the compiler designer, because it places all the burden on the programmer to insert appropriate frees. But that is not quite accurate.

The problem is that, in the presence of explicit free operations, the programmer may also mess things up by freeing memory that, in fact, is still used elsewhere. If, after freeing memory at address a, the reference to a is still around somewhere else, say in pointer p, then dereferencing p by $*p$ will lead to fairly unpredictable behavior. Unlike dereferencing pointers that had never been allocated before, which would immediately raise a SEGFAULT due to a null pointer dereference, dereferencing pointers that point to no longer allocated memory can have a range of strange effects.

In the "best" case, the whole memory page where a resides has become inaccessible. Then the runtime system would signal a memory access violation and nothing worse than that can happen. But the memory page may still belong to the process, and then the pointer dereference of a previously properly allocated pointer will now yield some random memory content. Either the old contents or some arbitrary new contents if the memory location has been reused for a different memory allocation meanwhile. For performance reasons, memory allocation is implemented in a way that allocates a whole block of memory from the operating system and then subdivides the memory block into pieces according to subsequent calls to **alloc**. The set of all free memory regions that have not been given back to the operating system is managed in free lists.

In either case, C0 is not safe when relying on proper placement of **free** operations. For instance, the following program is unsafe

```
int* x;
int* y;
x = alloc(int);
*x = 5;
y = x;
free(x);      // free memory of x
return *y;    // bad access
```

More information on garbage collection can be found in [App98, Ch 13.1-13.3] and [Wil94, Section 1-2] at http://www.cs.cmu.edu/~fp/courses/15411-f08/misc/wilson94-gc.pdfweb.

2 Garbage Collection

The way out of that dilemma is to rely solely on garbage collection to free memory resources when not used anymore. A regular claim is that this is a lot easier than having to do explicit **free** operations. While that is true, garbage collection still does not solve all memory problems. Obviously, it can only work if all references to unused objects have actually been deleted by the programmer. That is not much more trivial than placing explicit **free** operations. The two aspects that still make it a lot easier is that, unlike for **free**, the order of erasing pointers is less critical. Also improper placement of reference erase operations like $x = \textbf{NULL}$ at least does not corrupt memory access.

```
int* x;
int* y;
x = alloc(int);
*x = 5;
y = x;
x = NULL;     // x is happy to have memory freed
return *y;    // good access
```

The biggest problem for programmers using garbage collection thus is to make sure all unnecessary memory can be freed by erasing references. But if they miss one object, the worst that will happen is that the memory cannot be reclaimed, but the program won't crash or have strange effects (except for running out of memory).

The burden on implementing garbage collection and managing all required data structures to track pointer usage is on the compiler designer. That is a non-negligible computational effort. It can be more than 10% of the overall runtime for programs with frequent allocation and discard. Typical uses of the `java.lang.String` class to compose strings in earlier JVMs are typical examples of what can go wrong when not being careful, which is why the use of the thread-safe `java.lang.StringBuffer` has been encouraged instead. In JVM 1.5, the non-thread-safe `java.lang.StringBuilder` has been introduced for performance reasons and is even used automatically if possible.

Extremely critical parts of garbage collection implementations is that they need to be highly optimized to avoid lengthy interruptions of the program. The worst impact on garbage collection performance is usually caused by cache misses. And the worst of those is when garbage collection starts when the main memory is almost completely used up, for then swapping in and out of memory will dominate the computational cost.

The basic principle of garbage collectors is to mark all used heap objects by following spanning trees from all references buried on the stack or in registers. The garbage collector either maintains free lists of the free fragments found in the heap space. Or it compactifies the used heap blocks in the heap space in order to reduce fragmentation. Or it just copies all used objects into a new heap. For the last two choices, the garbage collector needs to be able to change *all* pointers pointing to the objects. In particular, it would be fatal if a pointer would be ignored during the change or a non-pointer (e.g., floating point data) mistaken for a pointer and changed.

3 Uninformed Boehm Garbage Collector

Without information from the compiler, garbage collection can still be implemented in what is called uninformed garbage collection. In Boehm's garbage collector, for instance, that is based on conservative overapproximations of checking all bit patterns for if they could possibly represent references. The Boehm garbage collector checks register, stack, and heap contents and conservatively decides whether they could represent references. If a bit pattern could be a pointer, then pretend it would be and represent a reference. Information used to determine if something could be a reference include the question whether that page table has actually been allocated, whether it belongs to the heap space that the garbage collector manages, and whether the lower bit information matches alignment rules (only possible in languages without arbitrary pointer arithmetic and when optimizers are not using it). Obviously, floating point information or strings could be mistaken to represent pointers. With this approach, objects cannot be relocated in memory, which can lead to issues of fragmented memory.

4 Uninformed Reference Counting Garbage Collector

Another uninformed option to perform garbage collection is based on explicit reference counting where each object contains a a size information and a reference counter memorizing how many references point to it. Upon reference copying or removal, this reference counter is incremented or decremented. When it hits zero, the object can safely be removed. Reference counting libraries are popular in C++.

The downside is that there is a non-negligible overhead for each pointer assignment. In addition, memory leaks can result from cyclic pointer chains that are not referenced from the program anymore, but still reference each other. Then their reference counts will stay nonzero and the objects persist. So the programmer has to take care to erase pointers recursively in all (cyclic) data structures.

5 Informed Garbage Collectors

Informed garbage collectors, instead, use information obtained from the compiler about the positions of all pointers in memory and the sizes of all objects. They need to know about the locations of all pointers and the allocated object sizes. This information can either be obtained by runtime type information, e.g., in object-oriented programming languages or in the presence of polymorphism. Or it can be obtained from static typing information, which can be stored as information associated appropriately with the pointers for use at runtime. In C0 implicit typing information is perfectly sufficient because there is no polymorphism and we have static typing information for all variables, including pointers. We also know the size of each type statically, because there are no dynamically allocated sizes, except for arrays, which need a length information. Yet we have already needed this length information for arrays for compiling array access safely.

We discuss where this information comes from in more detail in the next lecture after the garbage collection algorithms have been discussed.

6 Mark and Sweep / Mark and Free

In order to find out which heap objects are still used, garbage collectors visit each reachable object by either a depth-first search or a breadth-first search. It marks every reachable heap object. After marking is finished, the garbage collector walks each heap object again and frees all objects that have not been marked. This phase also clears the marking in order to prepare for the next garbage collector run. For this we need one bit of marking storage per object. The marking bit also helps us not to revisit objects twice, because we do not have to revisit a marked object. Note that the garbage collector still needs enough memory of its own to work. In order to implement the search procedures, we also need space to manage to search data structures. For depth-first search, we either need to use the stack and take care not to cause a stack-overflow or use separate stack data structures.

1. mark each object reachable from roots $\rightsquigarrow O(R)$
   Use mark bit to prevent double exploration
2. for all objects i: if mark(i) then unmark(i) else free(i) $\rightsquigarrow O(H)$

The time complexity in terms of the number R of reachable words and the size H of the heap is as indicated on the right. The space complexity is $O(H)$ plus the memory for the garbage collector and its search implementation and its free list memory management data structures.

After marking, we can choose how to free memory. When working with *free lists*, it makes sense to manage a list of all allocated objects that is maintained during **alloc** and **alloc_array** calls. During the free phase, we can directly accumulate the next free list. The major downside of free lists is that they can lead to unnecessary external fragmentation, because we cannot even easily join adjacent free memory blocks, since the list is not sorted by memory addresses. Another downside of free lists is the extra data storage of doubly linked lists that is needed for fast insertion and deletion at alloc and free respectively. The advantage is that we never need to change memory addresses.

7 Mark and Compact

When working with compactification in space, the algorithm is called *mark and compact*. It needs three passes through memory. A first pass during the mark phase to find the free parts of memory. A second pass to change all addresses by a depth-first search. And a third pass to relocate the objects to a consecutive block in memory, which will finally erase the marking again.

1. mark each object reachable from roots $\rightsquigarrow O(R)$
   Use mark bit to prevent double exploration
2. for all objects i: if mark(i) then change all addresses using i $\rightsquigarrow O(H)$
3. for all objects i: if mark(i) then relocate $\rightsquigarrow O(H)$

A common variation of mark and compact is due to Wegbreit [Weg72]. It changes the addresses before relocating objects based on the difference $a - s$ where a is the address to change and s is the current sum of the sizes of free memory blocks. Overall, simple mark algorithms are fairly easy to implement yet with complicated address arithmetic. But the program needs to be suspended during the garbage collector run and the memory needs to be visited twice (once during mark, once during compactify). The major downside is the effort needed to calculate all new addresses.

8 Mark and Copy

When working with copies into a new memory location, the algorithm is called *mark and copy*. It allocates a new memory region and copies objects that are still in use from the old memory region over to the new memory region, leaving a forwarding address in the old memory region.

1. mark each object reachable from roots $\rightsquigarrow O(R)$
   Use mark bit to prevent double exploration
2. for all objects i: if mark(i) then copy(i) with forward address $\rightsquigarrow O(H)$

Especially when working with breadth-first search, this algorithm can increase locality in memory access, because objects that belong together will be allocated close to one another in the new memory. The algorithm also removes fragmentation, which simplifies object allocation. When allocating an object, we do not have to search along a free list to look for a memory fragment that is large enough to hold the object. This is especially crucial for functional languages that allocate new objects on a regular basis to prevent object mutation. The major downside is that the virtual memory consumption doubles and the procedure even needs to copy heap objects that do not contain any pointers (which are in some languages are 30% of all objects). The mark and copy garbage collector or copying collector is really easy to implement.

9 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 $R_1, \dots, R_n$ and allocate all new memory in R_1. Whenever a memory region R_i is full, we run garbage collection, e.g., with mark and copy 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.

10 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 amortized 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., if $R/H > 0.5$, then its benefit is limited and it can ask the operating system for more memory.

11 Schorr-Waite Pointer-Reversal Marking Algorithm

The Schorr-Waite algorithm for pointer traversal 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.

The Schorr-Waite algorithm is a bit tricky. But there even is an implementation of Schorr-Waite that has been formally verified to be correct [Bub07] by the theorem prover KeY.

12 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 (or longer if the problem repeats).

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 *blue*), objects where depth-first search is currently in progress

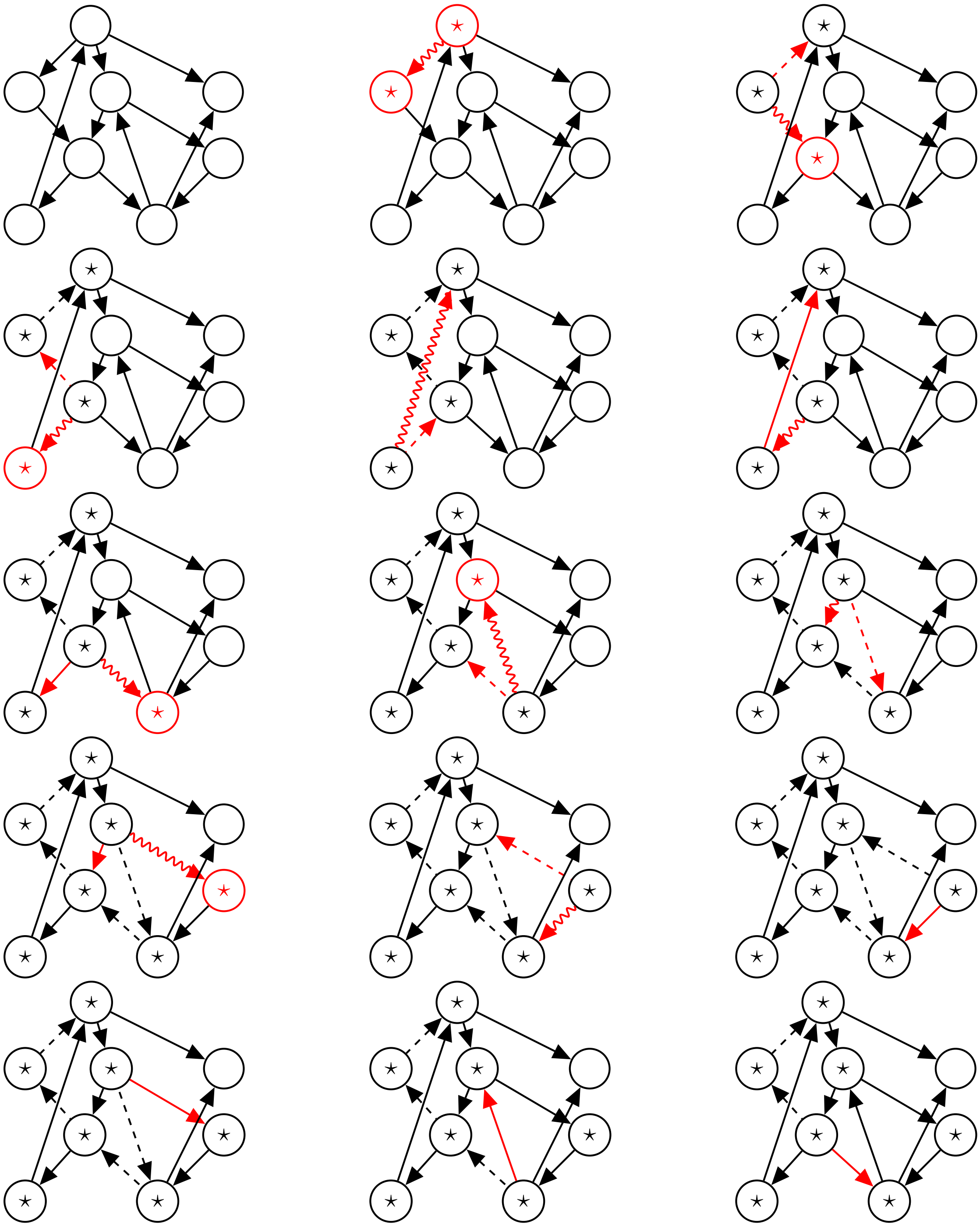

Figure 1: Start of Schorr-Waite run on an object graph (mark = ⋆, visiting = snake, reversed = dashed)

(marked *red*) 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 *blue*. 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 *blue* to *red* (if it is *blue*, 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 *blue* 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 *red* 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, *blue* 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 *blue* markings into *red* markings at pointer assignments. One source of trouble is concurrency of the overall 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 *blue* pointers to *red*, not all of them. If the *blue* pointer a is stored into a *black* object b, e.g., by `b->x = a` then a is marked *red*. While there are some variations of the write barrier and also read barrier algorithms, they all share the issue of fast conditional marking.

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 *red* and change the page to read-write again. This results in higher granularity but quite significantly reduced synchronization cost.

13 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/sweep 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 call 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. That way, the respective flow of the callee-save registers can also be found out by walking the stack backwards and checking on each function's convention for callee-save registers. In particular for relocating garbage collectors, only saved positions in the stack need to be modified and the current set of registers, because the stack saved registers will be loaded back into the registers on return.

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**).

14 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

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

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 either the displacement constant 100 or the fact that it derives from a 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.

For mark and free garbage collectors, the situation can be simplified, because addresses don't change there. But this still requires that the base pointer is made live as long as any derived pointer is around.

Quiz

1. Can you solve the issues with pointers in callee-save registers by saving all registers on the stack in alloc calls or in gc calls?

References

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

[Bub07] Richard Bubel. The Schorr-Waite-algorithm. In Bernhard Beckert, Reiner Hähnle, and Peter Schmitt, editors, *Verification of Object-Oriented Software. The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*, pages 569–587. Springer, 2007. DOI 10.1007/978-3-540-69061-0_15. URL: `http://dx.doi.org/10.1007/978-3-540-69061-0_15`.

[Weg72] Ben Wegbreit. A generalised compactifying garbage collector. *Comput. J.*, 15(3):204–208, 1972.

[Wil94] Paul R. Wilson. Uniprocessor garbage collection techniques. Submitted to ACM Computing Surveys. Available at http://www.cs.cmu.edu/~fp/courses/15411-f08/misc/wilson94-gc.pdf, 1994.