Avoiding Stalls in Garbage Collected Systems

Anton Golov (3809277)
jesyspa@gmail.com

April 21, 2014

Abstract

When implementing a general-purpose programming language, the usage of a mark-sweep or copying algorithm for garbage collection can lead to significant stalls: periods of time during which the program must be suspended while garbage is collected. In this paper, we show how such stalls can be avoided using incremental and generational collection techniques at the cost of execution speed.

1 Introduction

The implementation of a general-purpose garbage-collected programming language can be seen as a cooperative process between a mutator and a collector. The mutator is responsible for executing the given program, for which it must to allocate memory. The collector is responsible for locating memory that the mutator can no longer make use of and deallocate it, making it available for later reuse.

The mark-sweep [McC60] and copying [FY69] algorithms lie at the basis of garbage collection approaches [Wil92]. However, in their unmodified form, these algorithms are 'stop-the-world' approaches: the mutator is allowed to run until memory is exhausted, at which point the mutator is stopped and the collector is invoked. Only once collection is complete is the mutator allowed to continue. In programs which use a large amount of memory, this collection process can take a significant time, leading to the program appearing to stall. In interactive programs, such stalls can hamper user experience, and should thus be avoided if possible.

We provide an overview of existing work on reducing stalls, first presenting the mark-sweep and copying algorithms in more detail, and then showing how incremental and generational techniques can be incorporated in each, allowing for stalls to be reduced in frequency and maximum duration. We have

chosen not to consider approaches based on reference counting; while these are naturally less susceptible to pauses, they cannot reclaim cyclic structures and so are not suitable for a general-purpose programming language implementation. Finally, we do not have the room to cover concurrent approaches.

2 Garbage Collection

Given an object in a program we can talk about its lifetime: the time between its allocation and deallocation. In many cases, it is not possible to determine when an object can be deallocated at compile-time. In this case, we have two options: the writer of the program must specify when this deallocation happens, or the runtime system of the language must determine this automatically. The latter option is collectively known as garbage collection, following the analogy that objects that can be safely deallocated are of no further use ('garbage') and should be collected. We concern ourselves with how this can be implemented when program response time must be kept low; in other words, how to prevent apparent 'stalls' in program execution while garbage is collected.

For the purpose of talking about garbage collection it is convenient to see memory as a directed graph, with the objects as objects and the pointers between them as edges. Note that this graph will contain objects whose lifetime need not be managed by us. Such objects are collectively known as the root set. The objects we must manage, those whose lifetime is determined at runtime; are dynamically allocated, and the region of memory used for these objects is called the heap.

We refer to an object that can safely be deallocated as dead, and to any other object as alive. Any object in the root set should not be deallocated by us, and is thus by definition alive. Furthermore, any object which is pointed to by a living object is also alive. All other objects are dead. If, by the above, an object can be either dead or alive, it is dead. Assuming that the mutator can access objects in the root set and follow pointers, but has no other way of finding objects, this guarantees that any object we consider dead will never be accessed by the mutator again. Its memory can thus be recycled.

Apart from liveness, we ascribe objects some further properties. If object X has a pointer to object Y, we say Y is a child of X, and talk about the set of all such Y as the children of X. Furthermore, we give each object a colour to indicate the information the collector has about it [DLM+78]. A white object is an object the collector has yet to see, a grey object is an object the collector has seen, but has not yet inspected the internals of, and a black object is one that the collector is fully done with.

3 Basic Algorithms

The two basic algorithms we consider are the mark-sweep [McC60] and copying [FY69] collectors. Both algorithms occasionally stop the mutator, usually when a request for allocation cannot be fulfilled, perform a graph search through the living objects, and then deallocate all dead objects. The differences between the two lie in how the graph search is performed and how living objects are preserved.

3.1 Mark-Sweep

We explain a generalisation of the mark-sweep algorithm presented by McCarthy et al. [McC60]. The algorithm consists of a marking phase followed by a sweeping phase. During marking, we start from the root set and mark all objects reachable through a graph search as black. During sweeping, we check the colour of every object and deallocate any that are white. Note that the recursive nature of a graph search closely parallels our definition of liveness.

More formally, the algorithm is as follows. Initially, we mark all objects in the root set as grey and all other objects as white. This starts the marking phase. While there are still grey objects, we choose any grey object X and colour any white children it has grey. We then colour X itself black. As objects never become lighter, and every grey object eventually becomes black, we can be sure that this process terminates. Once this happens, the marking phase is complete, and we proceed to sweeping. For every object, we check whether it is white, and deallocate it if that is the case. Having done this for every object we know that any object still left is alive.

This approach requires setting aside extra space for the list of grey objects, but otherwise has little memory overhead per object, and requires no modifications to the mutator. The time cost of sweeping is linear with the size of the heap, making it the dominant term. However, on modern hardware, the linear memory access sweeping has leads to marking taking up most of the time [JHM11, Section 2.5].

3.2 Copying

The copying algorithm we explain is due to Cheney [Che70] and is most suitable when the program only requires a small part of the memory available. Initially, the heap is split in two, and all allocation happens in one half. When this half is full, we copy all live objects into the other half, updating pointers as necessary. Any objects in the first half can then be deallocated, and the

roles of the halves are reversed. Note that as the root set is not part of the heap, it should not be copied; there is simply no need to, as its lifetime is not managed by the collector.

More formally, the algorithm is as follows. At program start-up, the heap is split in two, with one half marked the to-space and the other the from-space. All allocation happens in to-space, and every object is allocated with an extra `newest` pointer that points to itself. When memory is exhausted, the mutator is stopped and the from-space and to-space labels are swapped. We create two pointers, `seen` and `scanned` that point to the bottom of to-space.

We can then proceed with the copying. All objects not in the root set that are children of a root set object are copied to to-space. Whenever such a copy occurs, we update the `newest` pointer of the old instance to point to the new instance. This means that if an object is a child of two root set objects, it will not be copied twice. With every copy we update `seen` to point just above the copied object. All copies thus end up between the `scanned` and `seen` pointers. We can see all objects that are below the `scanned` pointer as black, any objects that are between the `scanned` and `seen` pointers as grey, and any objects that have not yet been copied as white.

While the `scanned` and `seen` pointers are not equal, we update all pointers of the object pointed to by `scanned` that still point to from-space, copying objects as above when necessary. Once this is complete, we advance the `scanned` pointer. Note the similarity with the mark-sweep algorithm: copying objects can be seen as marking grey objects white, while advancing the `scanned` pointer can be seen as marking the last object black. Termination thus follows for the same reasons. Once this copying is complete, any objects in from-space are either dead or copied, and can thus be deallocated. This can be done in constant time, making it more efficient than sweeping.

The compact state of the heap after each collection, together with all deallocation happening after a copy, means that allocation can be done simply by advancing a pointer. This makes copying collection efficient; the throughput depends largely on how often copies have to be performed, which decreases as memory grows. This leads to copying collection becoming faster than both marking collection and stack allocation given sufficient extra memory [App87]. The downside is the increased memory usage, and the cost of copying an object is usually higher than the cost of only marking it.

4 Incremental Techniques

A possible solution to the stalling problem is to split garbage collection work that must be done into smaller portions and perform these occasionally as the

mutator is running. Possibilities have been found to perform small amounts of marking [DLM+78], sweeping [JHM11], and copying [Bak78] during certain mutator actions, usually on an allocation request. These solutions only changes the distribution of collector work over time; there is just as much work to be done by the collector, and sometimes the mutator must do more in order to keep the data structures consistent.

4.1 Incremental Sweeping

We start by considering incremental sweeping [JHM11, Section 2.5], as it is the simplest option. In a system with a mark-sweep collector, we know that the mutator cannot directly access the colour of an object. Moreover, we know that once an object has become garbage, it cannot be seen by the mutator again until it has been deallocated and then reallocated. We can thus delay deallocating these objects for as long as we like, providing that we can still satisfy allocation requests.

The algorithm is thus a simple modification of mark-sweep as described above. Instead of performing both marking and sweeping before we allow the mutator to run again, we only perform the marking phase. After that, sweeping is done at every allocation request, deallocating some (small) number of objects.

This change reduces the duration of the stall to be linear with the number of live objects on the heap, making the algorithm comparable in asymptotic complexity to a copying collector. It also does not require any additional work of the mutator, making it simple to implement.

4.2 Incremental Marking

When introducing incremental marking [DLM+78], we need to ensure that the actions of the mutator do not lead to us missing any live objects. As stated, our algorithm will only mark objects that are reachable through white objects from a grey object. If the mutator creates a pointer from a black object to a white object X, and then destroys all white paths from grey objects to X, X will never be marked by the collector and will be deallocated in the next sweeping phase. The pointer that was created by the mutator is now dangling, and if another object is allocated at the same location, modifications to one will affect both.

In order to prevent this we must ensure that the mutator either cannot create a pointer from a black object to a white object, or cannot destroy all white paths from a grey object to any white object. Taking the latter approach means that any object alive at the beginning of a marking phase

will certainly get marked, which is often undesirable: if an object becomes garbage, we would rather reclaim it sooner than later. The former is thus more often employed.

The algorithm must be modified as follows: instead of marking when memory is exhausted, a marking phase starts as soon as a sweeping phase is complete, and a small number of objects are marked periodically. Additionally, whenever the mutator creates a pointer from a black object to a white object it must make either of the objects grey. Both choices guarantee correctness, but have a drawback; greying the white object can lead to reclaiming less garbage than possible, while greying the black object can prolong the marking phase.

Due to the mutator cooperation necessary to maintain consistency, this modification comes at a cost of mutator throughput; while the work of the collector will be better spread out over time, the work of the mutator will be strictly greater than what it had been. Furthermore, care must be taken that marking occurs fast enough to never run out of memory; if an allocation request cannot be satisfied, the remainder of the marking phase must be performed immediately, leading to a stall. This problem can, fortunately, be resolved by keeping track of how much memory is free at the beginning of a marking phase and adjusting the rate of marking accordingly.

4.3 Incremental Copying

An incrementally copying collector [Bak78] must avoid all the pitfalls of an incrementally marking collector, as described above, but must also ensure that any reads are performed on the newest copy of an object. If no precautions are taken, an object may be copied and then written to by an outdated pointer. As a consequence, writes may silently be ignored, while the same situation for reads can cause access to an old value.

We can avoid this by ensuring that whenever the mutator accesses a pointer, it is updated to point into to-space. The algorithm is thus as follows: when garbage collection starts, the roles of to-space and from-space are reversed. On every allocation, some number of objects are copied, as in a non-incremental copying collector. In addition to that, whenever the mutators reads a pointer value, it checks whether this pointer is to from-space. If so, the object is copied if necessary, and the pointer is updated. This way, we ensure that the mutator never sees old copies of objects.

This approach is effective, but rather inefficient without hardware support. Every read now involves a conditional branch, which can degrade performance significantly. Furthermore, a read can also involve a copy, which makes performance less predictable.

A more intricate but often better-performing scheme is to only copy objects when a pointer to a from-space object is written to an already scanned object. In order to prevent reading and writing incorrect data, an extra indirection is always performed through the `newest` pointer. This can only reduce the amount of copying work that must be done, as objects may become unreachable after being read from; however, it is still at the cost of a pointer indirection per read.

5 Generational Techniques

Experience shows that most programs written in garbage-collected languages have a large number of objects that live briefly, and significantly fewer objects that persist for a long time [LH83]. This suggests that garbage-collecting recently allocated objects is more fruitful than collecting older objects, and should thus be done more often.

In order to make use of this we split the heap into several 'generations', numbered from zero onwards. All allocation happens in generation zero. When garbage collection must be performed, instead of always collecting the whole heap, we only collect all generations up to some given one. This ensures that generation zero, which will contain most of the garbage, will be collected most often. Each object tracks how many collections it has survived; once this number passes a threshold it is moved to the next generation. Put together, this ensures that long-living objects are still collected, but less often.

Collecting only a few generations takes significantly less time than collecting the entire heap, reducing the average pause time. Given the expected memory usage pattern, older objects will have to be touched less often, reducing the overall work of the collector, and increasing program throughput. However, when the oldest generations are scanned, the full heap will still be traversed and so the maximum pause time is unaffected.

Generational collection techniques have two pitfalls. Extra care must be made to handle so-called down-pointers: pointers from an older generation to a younger one. Additionally, generational collection is an optimization for the common case of memory usage. In applications where usage patterns are vastly different, these changes will do more good than harm.

5.1 Handling Down-Pointers

An unfortunate consequence of only collecting a part of the heap is that the rest of the heap becomes part of the root set. While the algorithms presented above are still correct for a root set of such size, the time necessary to scan it

is too great for practical use. One of the main challenges in the development of generational garbage collectors is how to reduce the number of objects that need to be scanned.

There are three key approaches to this problem: indirection tables, store lists, and marking. The first paper on generational collection [LH83] suggested that all references from older generations to newer ones could go through so-called indirection tables, which would be significantly smaller than the generations themselves, and could thus be scanned in reasonable time. Whenever a generation was collected, its tables would also be cleaned of any unused entries.

However, due to the need to check at each access whether a pointer is direct or indirect, this scheme is not efficient enough for practical use unless hardware support is available. Both alternative schemes allow for pointers between generations to be direct, storing the data elsewhere. The difference lies in what is tracked: store list schemes such as Ungar's Generational Scavenging [Ung84] keep track of objects that are pointed to by older generations while marking schemes such as page marking [Moo84] keep track of areas of memory where pointers to such objects occur. The choice of approach is non-trivial; store lists are easier to combine with an incremental collector, while marking can benefit from hardware support.

5.2 Memory Usage Patterns

Generational garbage collection aims to reduce copying of old objects while reclaiming young objects sooner. As seen above, this provides a significant benefit if the number of short-lived objects greatly exceeds the number of long-lived objects. In the presence of a different memory usage pattern, however, the usage of multiple generations can instead increase collector work. For example, if the lifetime of most objects is roughly equal, objects that have been promoted are likely to die sooner than those in the young generation. Work done to collect the youngest generation will mostly be wasted, while objects in older generations will be deallocated much later than possible.

Another important requirement to make generational garbage collection efficient is that there are relatively few pointers from older generations to newer ones. If this is violated, the cost of finding all such pointers, or of tracking all pointees of such pointers, becomes higher than the gain from collecting only a part of the heap.

Knowledge about the memory use patterns of a program can allow tuning the garbage collector to reduce these issues. Until compiler technology reaches the level where such knowledge can be distilled from the program, it is beneficial to allow the user to tweak parameters such as the relative sizes

of the heaps, the rate of advancement to older generations, and the frequency at which older generations are collected.

6 Conclusions

We have seen that when necessary, pauses can be reduced to acceptable levels using incremental and generational techniques. The two can be combined provided the requirements on the mutator are compatible, which is usually the case. Generational garbage collection can be used to improve overall program efficiency and reduce the average pause time, while incremental collection can be used to decrease both average and maximum pause time. Both have mutator overhead, but it is sufficiently low for the approaches to be useful in practice.

Further measures can be taken, particularly if more is known about the way memory is used. If the problem domain is restricted so that structures are exclusively acyclic, a reference counting algorithm could be used instead [Col60]. Alternatively, restricting the mutator to immutable operations allows us to make use of the algorithm presented by Armstrong and Virding [AV95]. Research has also been done on allowing the mutator and collector to be run concurrently [DLM+78, NR87].

References

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Inf. Process. Lett.*, 25(4):275–279, June 1987.

[AV95] Joe L. Armstrong and Robert Virding. One pass real-time generational mark-sweep garbage collection. In *Proceedings of the International Workshop on Memory Management*, IWMM '95, pages 313–322, London, UK, UK, 1995. Springer-Verlag.

[Bak78] Henry G. Baker, Jr. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, April 1978.

[Che70] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, November 1970.

[Col60] George E. Collins. A method for overlapping and erasure of lists. *Commun. ACM*, 3(12):655–657, December 1960.

[DLM+78] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21(11):966–975, November 1978.

[FY69] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, November 1969.

[JHM11] Richard Jones, Antony Hosking, and Eliot Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Commun. ACM*, 26(6):419–429, June 1983.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, April 1960.

[Moo84] David A. Moon. Garbage collection in a large lisp system. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP '84, pages 235–246, New York, NY, USA, 1984. ACM.

[NR87] Stephen C. North and John H. Reppy. Concurrent garbage collection on stock hardware. In *Proc. Of a Conference on Functional Programming Languages and Computer Architecture*, pages 113–133, London, UK, UK, 1987. Springer-Verlag.

[Ung84] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. *SIGSOFT Softw. Eng. Notes*, 9(3):157–167, April 1984.

[Wil92] Paul R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, pages 1–42, London, UK, UK, 1992. Springer-Verlag.