Garbage Collection
Fast Dynamic Languages Seminar Paper

Stefan Fehrenbach

Philipps Universität Marburg
fehrenbach@mathematik.uni-marburg.de

Abstract. New garbage collection techniques are needed to cope with higher demands by applications and to keep up with increasing computing power. This work introduces how garbage collection works today and then continues to describe what ideas exist to solve the problems of the not that distant future by comparing three algorithms devised in the last decade.

1 Introduction

Garbage collection faces two main problems these days.

Firstly there is the ever growing popularity of memory managed languages. While this is nice in general, it means there is an increasing amount of applications demanding stricter real time guarantees.

Secondly there is *Moore's Law* constantly increasing main memory size. Seeing that single CPU cores cannot keep up, there is also a trend to more and more parallel processing in a single machine.

This prompts the question of how garbage collectors can satisfy the higher demand for short collection pauses when collection time should actually increase because of larger heaps.

Fortunately research has not halted either. In this paper I will briefly introduce how garbage collection worked some years ago, up to the time of writing this. We will then have enough background to understand how algorithms found in recent years solve the emerging problems.

2 Terminology

Some terms are not uniformly used in research literature discussing garbage collection. Terms and their meaning in this paper are briefly described below.

A *tricolour* collector regards objects as in one of three states. *White* objects of unknown liveliness, *grey* for live objects and *black* for live objects with their children already marked. Note that these states are oftentimes only used conceptually and need not be represented with actual mark bits.

Threads doing the work the application is actually written to do are called *mutators* because they change references, the sole reason for garbage collection.

A *concurrent* garbage collector does at least some amount of work while mutators are running too. A garbage collector is called *parallel* if it does its own work in parallel on more than one CPU core at a time.

Running in a *cooperative* environment means that a garbage collector's implementer may modify mutator code e.g. to emit *write barriers*, that is code that gets executed on every write of a pointer field. It also implies that it is possible to gather all references from heap, registers and stacks and distinguishing them from primitive data. While it is possible to write garbage collectors for uncooperative environments they are usually *imprecise* meaning they do not free all unused space. All algorithms described in this paper target the JVM, a cooperative environment. See e.g. [4, 3] for more information on garbage collection in uncooperative settings.

3 Generational Garbage Collection

Generational garbage collection exploits the fact that most objects are short lived.[5] Thus collection efficiency (memory freed per time spent) can be increased considerably by only examining young objects.

A straightforward implementation of this idea is splitting the heap in two regions, called nursery and mature space. All allocation happens in the nursery. When there is no space left to allocate from only the nursery gets collected. Objects surviving a small amount of collections get promoted to mature space. Eventually there will be no more space to promote objects to. Only then the mature space needs to be collected.

The semispace copying algorithm (see Wilson's survey[3] for a description) is ideally suited for the typically small nursery. Linear allocation is very efficient and few live objects makes copying and forwarding pointers very cheap.

The algorithm used for mature space should be more space efficient and handle large amounts of live objects gracefully. Allocation speed is not that big of a concern since typically only few objects get promoted each young collection.

Additionally pointers from mature space to objects within the nursery must be tracked. Otherwise the entire old generation would have to be scanned, contradicting the very idea of generational collection. Fortunately there are usually few such references. One way to store them is described below.

4 Generational Mostly-concurrent Garbage Collector

The *Generational Mostly-concurrent Garbage Collector* (GMG) is based on Boehm's *et al. mostly parallel GC*[6] and is described in detail in [2].

It serves as default mature space collection algorithm in Sun's HotSpot JVM using a semispace collector for the nursery.

Its main design goal is to reduce worst-case pause times compared to alternative algorithms implemented in the JVM at that time. It solves the problem that mature space collection is basically a full heap collection, by doing most of

the marking and all of the sweeping work concurrently with the mutators, only briefly stopping mutators to record their state and doing cleanup work.

The generational mostly-concurrent garbage collector works in four phases detailed below.

4.1 Initial marking

All mutators need to be suspended to record the set of *roots*, that is objects that are definitely live, like objects pointed to by references in registers or on any thread's stack.

All encountered objects get marked live, then the mutator threads are resumed. While this pause is proportional to the number of threads it is rather brief because objects only get marked not recursively scanned immediately.

4.2 Concurrent marking

Recall that initially marked objects do not get scanned recursively in the initial marking pause, instead their live bits are set in an external array of mark bits.

In this phase the algorithm recursively marks live objects reachable from the roots.

To mark all live objects GMG proceeds as follows: Linearly scan the mark bit vector for live objects. Every live object *cur* found gets pushed to a *to-be-scanned stack*. While the stack is not empty recursive scanning happens. Every referenced object *ref* is dealt with by

- simply marking it in case it points ahead of the linear scan pointer *cur*. It will be scanned recursively when *cur* advances.
- In case *ref* points behind *cur*, *ref* is both marked and placed on the to-be-scanned stack. That is it will be scanned recursively before linear scanning advances.

In other words: every time the stack is empty marked objects behind *cur* are considered *black* and marked objects ahead of *cur* are *grey*.

This procedure is a compromise between the size of the to-be-scanned set and the length of the initial marking pause. The extreme approaches would be to place every root in the to-be-scanned set, making its size proportional to the number of threads, which is impossible, considering we do garbage collection when memory is low. Alternatively it would be possible to do the full recursive scan while pausing, rendering the to-be-scanned set unnecessary but sacrificing concurrency.

Marking as described above is neither precise (objects marked live can actually be dead) nor complete (live objects might not get marked) on its own.

The imprecision is simply ignored as garbage missed gets collected next time.

Ignoring is not an option in the second case. Modified pointers need to be recorded to make marking complete. See the following section for how this is done.

The Card Table. Printezis *et al.* use a combination of data structures called *card table* and *mod union table* to track pointer updates. A *card* is a small region (512 bytes in their implementation) containing more than one object in general. The card table is needed in a generational setting to track references from old to young space to allow fast young space collection.

Every time a reference in old space is changed to point to an object in young space, its card is marked dirty. This is done by a two instruction write barrier.

When collecting young space only dirty cards need to be scanned for pointers into young space. Cards that contain no pointers into young space are cleaned when collecting. They need not be considered in the next collection cycle unless they are changed in which case they get marked dirty again.

The mostly-concurrent collector however also needs to keep track of updates not to young space but to mature space. This is done by introducing the mod union table - a bit vector of one bit for every card. Every time a young collection cleans a dirty page the page's bit in the mod union table is set.

This means every changed reference is recorded in the card table or the mod union table or both. This information is used by the next phase to complete marking.

4.3 Final marking

Concurrent marking might have missed new or changed references. Updated references on the heap are recorded in the card table or the mod union table, others are in registers and on the mutator's stacks.

To rectify these misses all mutators are suspended again and thread local roots get recursively marked. Updated references in modified heap regions identified by marked cards are considered additional roots. Note that this second pause is brief because most live objects have already been marked in the concurrent marking phase.

4.4 Concurrent sweeping

With mutators restarted sweeping begins. The heap is scanned and unmarked objects get deallocated. Free memory gets organised into double linked free lists coalescing consecutive blocks. This is done to prevent average block size from shrinking continuously.

Allocation happening in this phase is done with live bits set to prevent the sweeping process from deallocating new objects on sight.

5 The Next Generation of Garbage Collectors

While GMG outperformed its contenders new challenges emerge and call for adapted garbage collectors. Following I will describe three algorithms proposed lately and their *raison d'être*.

All challenges are grounded in larger and larger heaps and more CPU cores. The propositions of how to approach them however differ by quite a bit.

All of them have one thing in common, they divide the heap into smaller regions. These are obviously faster to collect and it is possible to select regions with high amounts of garbage first for collection.

5.1 Immix: A Mark-Region Garbage Collector

Immix' authors Stephen M. Blackburn and Kathryn S. McKinley describe a new family of garbage collectors, exhibiting a common pattern. I will concentrate on their reference implementation called Immix.

A *bon mot* in garbage collection is: "Space efficiency, fast collection, and mutator performance. Pick any two." Immix' goal is to beat existing collectors in all three aspects. This is achieved through a combination of efficient allocation, fine grained reclamation and clever defragmentation.

Immix has very low meta data overhead and fine grained space reclamation making it very space efficient. Its linear allocation scheme makes allocation very cheap. No write barrier overhead and good locality due to its compacting pass lead to little mutator slowdown. Parallel marking, little copying (only few objects per evacuation candidate) and coarse grained sweeping enable fast collection.

In contrast to GMG, Immix is not concurrent. This implies long stop-the-world pauses, parallel marking and freeing however mitigate this deficiency. This is especially true, considering that higher mutation rate common on multi processor machines also increases final marking pause times for GMG. Unfortunately, Immix' authors did not publish pause time measurements.

One of Immix' central ideas is to divide mature space into *blocks* and to further divide these blocks of 32KB into *lines* of 128B. This is a common characteristic of all described "Next Generation" collection algorithms.

A very brief insight into its inner working follows. For details please refer to the original publication.[7]

Marking. As does every other tracing garbage collector, Immix finds and marks live objects by recursively scanning references starting with the roots.

Objects may span lines but not blocks (handling of larger objects is not described due to space constraints). Every line containing at least (part of) one live objects also gets marked. Small objects (up to one line) get treated somewhat conservatively to avoid size checking overhead. They are assumed by the allocator to span one additional line.

Additionally the marking process collects some basic usage statistics. In the next collection cycle this data will be used to determine evacuation candidates.

Evacuation happens in the same pass as marking. Its goal is fighting fragmentation by evacuating objects from sparsely populated blocks into continuous regions. Determining good heuristics to identify evacuation candidates remains one area of future work.

Objects in candidate pages get evacuated (unless they are *pinned* e.g. for FFI calls) and marked. A forward pointer gets installed in the old object header. Every reference encountered, that points to a forwarded object, gets updated.

Immix' evacuation candidate selection is in part based on approximate space availability predictions. Should it run out of space prematurely, defragmentation is simply stopped.

Reclamation. After marking is complete, reclamation is very simple. The smallest reclaimable unit is one line. Since the marking pass updated line marks, reclamation just needs to linearly scan this map. Completely free blocks get collected in a global pool. Partially free blocks get *recycled* for the allocator.

Allocation generally happens in *recycled* blocks. Every mutator thread allocates into its own thread local allocation block. The allocator searches linearly for big enough "holes" to accommodate the request. To avoid skipping large amounts of wee holes, medium sized (greater than a line) objects get allocated into empty blocks.

5.2 Garbage First Garbage Collection

Garbage First[8] is the algorithm destined to be the default garbage collector in future server versions of the JVM. Designed in part by the same people who wrote GMG, it addresses one of GMG's weakest points: pause time when mutation rate is high, as it typically is in large server applications.

The goal is pause times well below fractions of a second and at least some hundred milliseconds apart. This is achieved by allowing for incremental collection, that is not scanning the whole heap but instead collecting only parts at a time.

While true *real-time* collectors exist (e.g. [9]) their throughput is usually rather low. Garbage First compromises on the real-time guarantees to achieve higher performance. The (user specified) *soft real-time* goal is only accomplished with high probability.

In turn higher throughput is possible, as is needed in large scale server applications to keep up with allocation requests.

Heap regions. Similar to Immix, Garbage First compartmentalises the heap into regions (of 1M by default.) Unlike Immix allocation happens only into initially empty regions. Again every mutator thread allocates its objects in a thread local manner. This has two distinct advantages: no contention between concurrently running threads and increased data locality.

Remembered Sets. Similar to the young space in the simple generational setting above, every region needs to record incoming references to allow for arbitrary combinations of regions to be collected without scanning all others.

Special care is taken to ensure updates are cheap, both in time and space. The actual implementation exceeds the scope of this paper.

It is sufficient to know that all incoming pointers get recorded.

Remembered sets can grow quite large. Experiments showed that often there are a smaller amount of *popular* objects referenced by many others. One way to reduce space overhead is to handle these objects differently. The interested reader may find a description of this in the original paper.

Generational mode. There are benefits easily exploitable by handling young objects separately from old ones. Garbage First can run in a generational mode to benefit from the typical patterns exploited by generational collectors. In generational mode it is not needed to keep track of remembered sets for young regions, all bookkeeping is being done as part of the usual post evacuation scanning.

Generational mode can lead to greatly decreased space overhead but it does not mix well with strict real-time requirements since all young regions need to be chosen for the next collection set.

Marking is done concurrently. Except for the very beginning when roots need to be recorded.

Objects allocated while concurrent marking is in progress get allocated live. They need not however be recursively scanned. The distinction is simple because allocation is linear in each region. It is only needed to keep track of where the highest allocated object was when the last collection cycle occur ed. All objects above this mark are necessarily considered live.

Marking itself is done very much like concurrent marking in GMG with a stack for some of the grey objects. This could be done for more than one region in parallel, the description in [8] however does not remark on that.

In Garbage First there are actually two mark bit arrays that switch roles. One for the currently running collection and one for the run before that. This is needed because marking and evacuation can happen concurrently. As the details of these interactions are quite subtle please refer to the original description.

Garbage First implements a *snapshot-at-the-beginning* marking algorithm. Readers unfamiliar with this terminology please refer to [3]. In short it means changed references need to be recorded for this algrithm to work. This is done by a write barrier logging pointer writes for later marking.

A brief stop-the-world pause is needed to ensure all logged reference updates and changed roots get processed to complete marking.

Evacuation is performed with mutators suspended but possibly concurrent with a marking run. This leads to a fair amount of interaction subtleties, e.g. references to evacuated objects need to be updated properly, not only in the heap and thread's call stacks but also in the marking phase's grey stack. Other interactions are omitted in the original description.

For the purposes of this overview it is sufficient to know that no dead object ever gets evacuated and the process is transparent to the mutators.

The real-time goal. Achieving the user-supplied soft real-time goal is the primary constraint. This consists of two things: firstly keeping each pause below the allowed duration, and secondly scheduling pauses to not exceed the allowed amount of total garbage collection time in any one time slice.

To estimate the time spent evacuating each region, usage statistics gathered in the concurrent marking phase are used. There are several variables that factor into this: amount of live bytes in the considered region, remembered set size, as well as cards that need to be processed to bring the remembered set up to date. The actual calculation is quite simple and is done at runtime. Several heuristic refinements could be made at runtime or devised offline for common allocation patterns.

Scheduling pauses is the second part of meeting the real-time goal. As long as there is enough free space all pauses can be delayed. Although all young generation regions must be collected at once in generational mode it is possible to delay even young collections by simply treating further allocated regions like non generational or old regions.

The goal of all these heuristics is to pause for short amounts of time as seldom as possible by freeing as much memory as possible in each pause. This strongly suggests collecting regions with a high proportion of garbage first[1].

5.3 The Pauseless GC Algorithm

Azul Systems build custom hardware to power huge business applications, using up to (in 2005) 256 gigabytes of memory, running on up to 384 CPU cores, and having tight pause time constraints.

Memory management is a big concern in this setting. The collector needs to be massively concurrent as well as parallel to keep up with the mutators demands. Due to pause time constraints it is not feasible to do whole heap collections and stop-the-world pauses. Their implementation actually pauses briefly at some points but only for engineering reasons, the algorithm is designed completely pauseless.

The Pauseless GC Algorithm[10] has the luxury of being implemented on custom hardware, designed with garbage collection in mind. While it is not necessarily needed to implement the algorithm it is crucial for performance. In the following description I will point out the places where hardware support is used.

Collection is divided into three main phases. Unlike GMG these phases run concurrently with each other, not only with the mutators. Recall that in both Garbage First and GMG changes to pointers on the heap need to be recorded, at least while concurrent marking is in progress. Pauseless, in no phase, places such a burden on the mutators. This combined with the point above means that in no phase there is a need to rush to return to a normal mode of operation.

[1] hence the name

One other important property Cliff Click *et al.* call *self-healing* is that mutator threads may witness inconsistent state. However they are enabled to fix it themselves on demand, actually reducing the garbage collection thread's work.

Marking. The marking starts by initialising internal data structures. Similar to Garbage First, the Pauseless algorithm uses two bitmaps to record liveliness information, one for the current and one for the previous collection cycle.

Running mutator threads mark their own root set. Blocked threads get marked in parallel by auxiliary garbage collection threads. Note that this is a *checkpoint*, not a stop-the-world pause. Marking cannot proceed until all threads crossed the checkpoint, the mutators however can resume their non garbage related work immediately after marking their own roots.

There is one bit flipped for every thread, called the expected *not-marked-through* (NMT) bit. Also each reference includes one NMT bit masked off for memory operations, slightly decreasing the 64bit address space. Every time a reference is followed a hardware *read barrier* checks whether the threads NMT bit matches the references NMT bit. If they do not match the thread encountered a possibly unmarked object and informs the marking threads. The references NMT bit gets set to the expected value, to not cause this interruption again.

This solves the same problem as the write barriers in Garbage First and GMG, but instead of an overhead of several instructions on every write, the read barrier condition is checked in hardware (in one cycle in the common case of not trapping) and traps at most once per reference.

Some care is required to properly handle multiple threads updating the same reference and to ensure all references will be discovered. These details exceed the scope of this paper, the enquiring reader may refer to the original publication.

Relocating. Unlike Immix and much like Garbage First only whole heap regions of 1M get reclaimed. Therefore it is necessary to evacuate sparsely filled regions.

Reclamation happens continuously to in turn allow mutators to continuously allocate objects. Here it is that the second set of marking bits comes into play. Relocation uses the current set of mark bits, the next marking phase will use the other set and keep the current mark bits untouched.

With basic live data counts made while marking, the relocation phase chooses mostly empty regions to evacuate.

After initialising an external forwarding table the page to be evacuated gets *protected*.

The forwarding pointer needs to be stored externally since a page's *physical* memory will get freed once fully evacuated. *Virtual* memory is not returned until there are no more stale references.

Following a stale reference, one into a protected page, is detected by a read barrier. The trap handler looks up the new location in the forwarding table and replaces the reference in memory, ensuring this will happen at most once. Is the object not already copied the mutator does so itself, again reducing the garbage collector's work.

Remapping. This phase's responsibility is to update every stale reference. It actually happens in one go with marking seeing as every reference on the heap needs to be checked, just like marking. When remapping completes no more stale references exist. Then virtual memory belonging to formerly protected pages can be reclaimed and their forwarding information is freed.

6 Conclusions and Future Work

Even though garbage collection has been around for at least 40 years it is still an area of active research. In this paper I highlighted some stepping stones from the basic idea of generational collection, shortening *most* collection pauses and delaying full heap collections, over parallel and concurrent approaches, up to the use of custom hardware to eliminate pauses altogether.

Azul Systems show that control over hardware, operating system and most of the lower software stack can lead to interesting solutions. I see at least one other area where this close control is common: smartphones. Hopefully vendors like Apple and Google use this opportunity.

Even without customised hardware there are still ways to improve on the current situation. One example would be refined heuristics to e.g. handle certain allocation patterns better. Runtime profiling, as used today to optimise code generation, could provide benefits for garbage collection as well.

There are vast amounts of work to be done in garbage collection; interesting times lie ahead of us.

Acknowledgements

I would like to thank my advisor Paolo Giarrusso for reviewing this paper and helping me very much to improve it, and Martin Salzer for reading and commenting on an early draft.

References

1. Marvin Minsky: A LISP garbage collector algorithm using serial secondary storage. *AI Memo 58, Project MAC, MIT* (1963)
2. Tony Printezis and David Detlefs: A generational mostly-concurrent garbage collector. *ACM SIGPLAN Not. Volume 36 Issue 1* (2000)
3. Paul R. Wilson: Uniprocessor Garbage Collection Techniques. *IWMM 1-42* (1992)
4. Hans-J. Boehm and Mark Weiser: Garbage Collection in an Uncooperative Environment. *Software Practice and Experience 18* (1988)
5. Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM 26* (1983)
6. H. Boehm, A. J. Demers, and S. Shenker: Mostly Parallel Garbage Collection. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (1991)
7. Stephen M. Blackburn and Kathryn S. McKinley: Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance. *ACM SIGPLAN Conference on Programming Language Design and Implementation* (2008)
8. David Detlefs, Christine Flood, Steve Heller, and Tony Printezis: Garbage-first garbage collection. *Proceedings of the 4th International Symposium on Memory Management* (2004)
9. David F. Bacon, Perry Cheng and V. T. Rajan: A real-time garbage collector with low overhead and consistent utilization. *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2003)
10. Cliff Click, Gil Tene and Michael Wolf: The Pauseless GC Algorithm *ACM/USENIX International Conference on Virtual Execution Environments* (2005)