Mostly Concurrent Garbage Collection

Marco Ammon
marco.ammon@cs.fau.de
Lehrstuhl für Informatik 4
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany

ABSTRACT

Automated *Garbage Collection* (GC) is desired for many applications due to difficulties with manual memory management. Conventional tracing garbage collectors require a complete halt of the program while unreachable objects are detected and reclaimed. However, the resulting increased latency is not always tolerable. This paper gives an introduction into GC fundamentals before thoroughly explaining and discussing the mostly concurrent mark-and-sweep algorithm presented in [8]. The approach is based on the principle of combining a concurrent marking phase with a short stop-the-world pause to process modifications which occurred simultaneously. Additionally, the influence of this concept on modern garbage collectors on the Java Virtual Machine is reviewed.

1 INTRODUCTION

Manual memory management in a large software system often is a significant cause of subtle bugs and security vulnerabilities [29], thus leading to increased development costs. For example, forgotten deallocations of no longer needed heap objects can lead to ever-increasing memory footprint during program execution. While potentially tolerable for short-lived programs, wasteful heap usage is an important problem for long-running server applications and may result in a crash. Contrarily, when freeing an in-use object, its memory location can be reused by a subsequently allocated object, provoking possibly unnoticed data corruption.

Consequently, automatic memory management, which reliefs the programmer from the task of deallocating no longer used objects, is highly demanded. Such *Garbage Collection* (GC) is deeply intertwined with heap management and often integrated into a run-time environment. In the context of this paper, an object is the unit of memory allocation and is represented by a struct, object (as in object-oriented programming), array, or similar construct.

One common flavour of garbage collection is *reference counting*, in which each object includes a counter for all references that are directed to it [25, 29]. When a reference is created or copied, the counter is increased. Conversely, it is decremented on reference deletion. Thus, when the counter amounts zero, the associated object can be safely deallocated because it is no longer reachable. The drawbacks of reference counting include increased work when creating, deleting or copying references as well as the inability to reclaim unreachable cycles of objects.

In contrast, *tracing* garbage collectors do not need individual reference counters per object: When invoked, they try to find all reachable objects by recursively traversing the object graph, starting from a known set of reachable references. After finishing this marking process, all objects not included in the constructed set are considered unreachable and can be reclaimed by the allocator [21]. For performing an accurate reachability analysis, program execution is usually halted during garbage collection, which is called a *stop-the-world* (STW) approach.

Consequently, as this process requires to scan large portions of the heap, application latency can significantly increase due to long pauses. In particular for interactive applications, web services, or real-time constrained programs, this may not be tolerable [8, 13, 27]. Hence, alternatives to the STW method are often desired.

In Section 2, different approaches to tracing garbage collectors are presented. Section 3 explains and discusses a particular algorithm for mostly concurrent GC [8] proposed by Boehm et al., which combines a short stop-the-world phase with longer, concurrently executed marking operations. Given the historic significance, its influence on some contemporary GC algorithms for the popular *Java Virtual Machine* (JVM) is reviewed in Section 4. Finally, Section 5 concludes the paper.

2 FUNDAMENTALS OF TRACING GARBAGE COLLECTION

Generally, the process of garbage collection can be split into two conceptional phases: Garbage detection, during which unreachable objects are enumerated, and garbage reclamation, which actually frees heap space by deallocating the previously identified objects. However, the distinction between both stages can be blurry as they are often interleaved in practice [29].

For detecting garbage efficiently, correct identification of pointers is a key requirement. Otherwise, references cannot reliably be followed. Often, pointer identification is realised through compiler and/or run-time support with approaches such as pointer tagging or "magic numbers" in object headers [9].

Furthermore, a starting set of known reachable references has to be chosen. From this root set, which consists of references found on the stack and in the registers in addition to statically allocated objects [6], pointers to other objects are recursively traced.

Within the subdomain of tracing garbage collectors, a variety of approaches can be found in the literature and in practical use [29]. Nonetheless, some underlying key concepts can be identified, which are addressed in the following.

2.1 Mark-and-Sweep GC

One of the fundamental schemes are *mark-and-sweep* collectors, as devised in the original proposition of garbage collection [21]:

Starting from the root set, all identified pointers are followed to their referenced objects, which are then marked accordingly, as shown in Figure 1. Examples of such marking include toggling one bit in a field of the object itself or in a global table. Likewise, pointers from these objects are followed recursively.

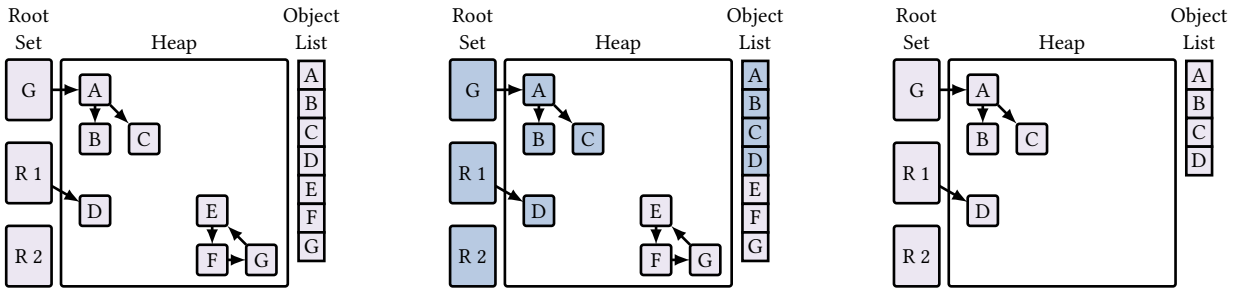


Figure 1: During program execution, the memory allocator of the mark-and-sweep GC tracks allocated objects in a list. In the mark phase, the GC recursively follows all pointers from the root set (statically allocated object G, registers R 1 and R 2) and marks visited objects. After a full graph traversal, all reachable objects are marked. Consequently, unmarked objects are no longer needed and can be reclaimed for subsequent memory allocations, while all mark bits are cleared again.

Once this graph traversal is finished, all objects are scanned for mark bits, e.g. by iterating over an object list provided by the memory allocator. The unmarked objects are deemed unreachable and can be safely deallocated. Finally, all mark bits are reset and the program execution continues.

Such a garbage collector could be invoked periodically, based on the allocation rate, or only when the available heap space is insufficient to allocate new objects.

While mark-and-sweep garbage collectors are able to reclaim memory, two major problems remain [29]: Firstly, after multiple cycles of object allocation and deallocation, the heap can be highly fragmented. While a sufficient amount of free memory exists for allocating a large object in total, the largest contiguous area may be smaller and allocation fails. Secondly, the spatial locality between surviving, still-needed objects is low. Hence, accesses to those can lead to considerable paging overhead or suboptimal cache usage.

2.2 Mark-and-Compact GC

Nonetheless, a different class of GC algorithms can resolve both issues through heap compaction: During memory reclamation, surviving objects are moved to the front of the heap [3]. Consequently, all live objects are located on fewer pages and the remaining heap space is a large contiguous region. Furthermore, memory-allocation complexity is reduced to simply adjusting a pointer.

For these *mark-and-compact* garbage collectors, exact pointer identification is strictly required because all references to live objects have to be corrected after compaction. Otherwise, there are two potential problems: Firstly, the target of unidentified, dangling pointers may be overwritten by subsequent allocations, leading to corruption. Secondly, data that is incorrectly identified as a pointer, is altered by a flawed pointer update.

In general, pauses potentially can be long because multiple passes over the managed objects are required [29]. Thus, pause durations are one of the most severe problems for mark-and-compact GCs.

Contrary to its name, no actual garbage is collected in mark-and-compact GC: Instead, the non-garbage is collected and moved; the remaining heap is implicitly considered garbage and, therefore, discarded. This insight forms the foundation of the more general notion of *copying* garbage collectors.

2.3 Copying GC

Instead of moving live objects in the same conceptional heap, a common implementation of copying garbage collectors partitions the heap into two distinct regions named *from space* and *to space* [15]. Only one region is used at a time in such a semi-space collector.

During garbage collection, the object graph in *from space* is traversed and reached objects are instantly copied to the *to space*. In the original memory location, a forwarding pointer, which contains the new address of the copy, is placed. It can be used to avoid copying the same object multiple times if it is reached through different paths [11]. After a complete graph walk, *to space* contains only live objects and the contents of *from space* can be discarded. Then, the remaining references are updated to point to *to space*, the regions' roles are swapped, and the program continues.

In comparison to mark-and-compact GC, less passes are required because live objects can be enumerated and copied at the same time. A second pass is only necessary for updating references. As a consequence of the described semi-space approach, only a part of the heap size is actually available to an application.

2.4 Generational GC

A key insight of memory management is that many objects have a relatively short lifetime, and can be reclaimed soon after allocation [19]. The objects remaining after a few collections are likely

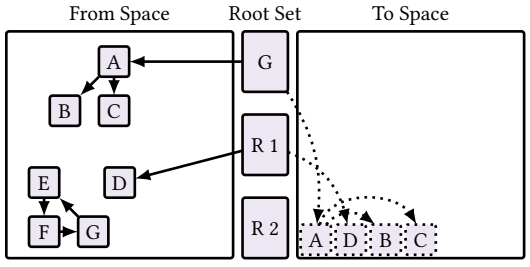


Figure 2: In a copying semi-space collector, only one half of the heap is used at a time. During collection, all reachable objects are copied and compacted breadth-first from *from space* to *to space* to reclaim memory and reduce fragmentation.

to be reachable for a much longer time. However, copying GCs still move them in every collection, with little chance of reclamation.

Consequently, the total set of objects can be split into multiple *generations*, separating young from old objects [19, 27]. For each generation, a different collection strategy can be applied. In particular for the old generation, a significantly lower collection frequency can be chosen to avoid excessive copying [19, 27, 29].

Conceptually, a counter is associated with each object, tracking the number of survived collections. When a certain threshold is exceeded, the object is promoted to an older generation.

Ideally, a young generation is collectible independently of other, older generations. However, pointers from the old generation into the younger complicate this process, as these pointers would have to be used as additional tracing roots. Simply adding the old generation to the root set would nullify the advantages of generational collection. Consequently, more advanced approaches are required. One solution are write barriers, e.g. compiler-inserted instructions or dirty-page-based tracking, for each write of a pointer. If the pointer is inter-generational, its target is added to a table, whose entries are used as roots for a young-generation collection.

While generational GC is often used in copying collectors, it can be used in non-copying algorithms, too. As an example, the algorithm examined in Section 3 is both generational and in place.

2.5 Concurrent/Parallel GC

In an orthogonal fashion, GC-induced pauses of program execution can be reduced by running concurrently with the application instead of lowering the total collection work. While some schemes run on the same processor in serial and are interleaved with normal program execution [3], others can run fully parallel on a dedicated processor [1, 14, 25]. In both cases, the program is usually named *mutator* because—from the GC's point of view—it changes the memory state. Often, explicit cooperation from the mutator is required to keep all references consistent and valid [13, 16].

3 "MOSTLY-PARALLEL GARBAGE COLLECTION" BY BOEHM ET AL.

Motivated by demands of interactive environments, Boehm et al. proposed a generational mark-and-sweep GC algorithm designed for short GC pauses [8]. It splits the marking process into a concurrently running tracing section and a short STW correction phase, which is necessary to incorporate potential writes by the mutator. For detecting these modifications, the *Memory Management Unit* (MMU) is used. In particular, they suggest that most tracing STW GCs can be adapted to their *mostly concurrent* scheme. Notably, the algorithm is even usable without any explicit compiler or run-time cooperation, thus tolerating unreliable pointer identification.

In the following, the overall idea behind this algorithm is explained and several design choices are examined. Furthermore, adapting the scheme to copying GCs is reviewed and, finally, their technique is compared with other approaches.

3.1 Allocation and Sweeping

As reclaiming unreachable objects is not required to happen in an STW pause, Boehm et al. integrate the sweeping process into the memory allocator. Thus, the heap is incrementally swept whenever the mutator requests new heap elements.

Overall, the allocator splits the heap into distinct regions with each containing same-sized objects. In particular, every region is at least as large as a virtual memory page. For each page, a bitmap of mark bits for all object slots is created. Additionally, the allocator manages two lists for each object size: One for sweepable pages, and one for free slots on any page. At the end of the STW marking phase, processed pages are queued in the sweepable list.

On allocation requests, the first element in the free list of the corresponding size is selected. If the free list is empty, the allocator sweeps pages from the list of sweepable pages: Slots with a cleared mark bit are garbage and, thus, added to the free list. For objects larger than the page size, multiple pages are swept at the same time.

Thus, the allocation delay depends on how many pages need to be swept until a free slot is found. Usually, this time period is negligible and significant application pauses can only occur due to the marking process, which is addressed in detail in the following.

3.2 Tracking Concurrent Mutator Writes

To detect objects concurrently changed by the mutator, Boehm et al. employ dirty-page bits: When a page of virtual memory is modified, a bit is set for that page. Thus, modified pages are quickly identified.

While modern processors with a MMU provide such a facility in hardware, it has to be accessible by ordinary user programs [6]. Often, access to memory-management hardware is restricted to the *Operating System* (OS). Then, the feature can be emulated by write-protecting all pages: When a write occurs, the OS invokes a GC-provided handler function, in which the access is registered. Due to high costs for these handlers, many GC algorithms could benefit from first-class dirty-bit support by the OS [1, 2, 6, 12].

3.3 Marking

While Boehm et al. propose a rather general scheme with several variations and implementations, their generational GC algorithm described in the following is the most instructive.

When reusing the mark bit for tracking object age, all marked objects are considered part of the old generation, thus only reclaimable in a full collection. Unmarked ones, on the other hand, are considered young and may be recovered in a partial collection.

Firstly, a full collection discards all age information and is visualised in Figure 3 for a devised example:

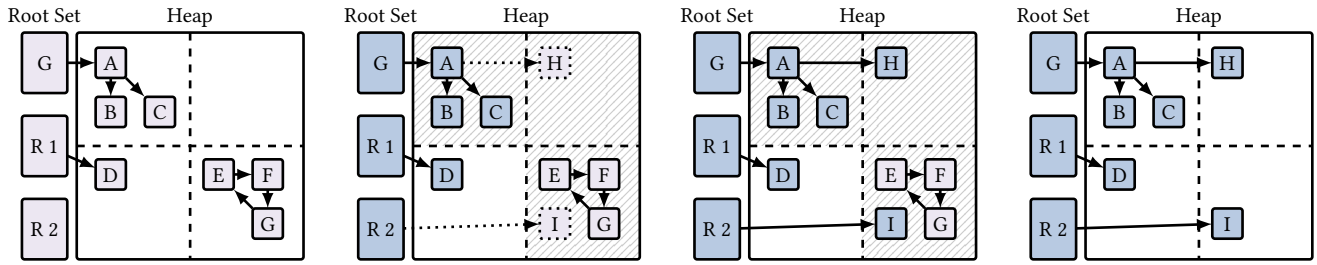
(1) Clear all mark and dirty bits.
(2) Mark all root-set objects and recursively trace from them.
(3) Stop the world, i.e. the application.
(4) Trace from registers and all marked objects on dirty pages.
(5) Clear all dirty bits and restart the world.

Consequently, the stop-the-world duration depends on how many pages were modified by the mutator during concurrent marking.

It has to be noted that some garbage may not be collected in this cycle when the last pointer to an already marked object is changed. However, these objects are correctly identified as unreachable in the next full collection and subsequently reclaimed.

Secondly, a partial collection retains all marking information and can only reclaim young objects allocated since the last collection:

(1) Atomically retrieve and clear dirty bits of all pages.



(a) Initial heap at full collection (b) Heap after concurrent marking (c) Heap after STW marking (d) Heap after sweeping

Figure 3: Before the full collection is started in Boehm et al.'s algorithm, all dirty and mark bits are cleared. While the GC traverses the object graph concurrently, the mutator can allocate new objects (H and I) and change pointers. In the STW correction phase, marking consequently is resumed from the registers (R1, R2) and the marked objects on dirty pages (A, B, C) to guarantee that no required object remains unmarked. Finally, all dirty bits are reset, and heap space can be reclaimed.

(2) Trace from all marked objects on the retrieved dirty pages.
(3) Stop the world.
(4) Trace from registers and all marked objects on dirty pages.
(5) Clear all dirty bits and restart the world.

Notably, the stop-the-world pause can be shortened through repeated executions of steps (1) and (2) before stopping the mutator.

3.4 Pointer Identification in the Absence of Compiler Assistance

For non-copying GCs, unreliable pointer identification is tolerable as long as an over-approximation is found [9, 12]. If arbitrary data, e.g. integers, can be falsely identified as pointers, not all unreachable objects may be reclaimed. However, no actual pointer is miscategorised and not followed. Thus, all actually reachable objects are also reachable for the GC; they are not mistakenly deallocated. Consequently, provided the GC does not move objects, data and pointer integrity stays intact after collection.

The most naive approach is to interpret all pointer-sized bit patterns as pointers. However, improving this very rudimentary estimation can reduce both tracing work and floating garbage. Some typical characteristics of pointers can be exploited to distinguish arbitrary data more reliably: Firstly, if only aligned pointers are allowed, all non-aligned values can be safely dismissed [6]. Secondly, valid pointer values can only be found in the interval between the lowest and the highest heap address [9].

Usually, for every reachable object, a pointer to its beginning exists. However, when using optimising compilers, this invariant does not always hold [6, 7, 9]: Pointers could be calculated by using a base address and some offset. Thus, the actual reference is not known to the GC. Similarly, an application could keep pointers only to the middle of an object [9, 12].

Overall, sufficiently accurate pointer identification is possible with a non-assisting but non-adversary compiler and some caution by the programmer [6, 7, 9, 12].

3.5 Adaption to Copying Garbage Collectors

The notion of mostly concurrent garbage collection is not restricted to mark-and-sweep algorithms, but is also adaptable to copying GCs. In this case, an auxiliary forward-pointer field is associated with each object in addition to dirty-page information.

Initially, all dirty bits and forward fields are cleared. During concurrent tracing, reachable objects are copied from *from space* to *to space* in an exhaustive graph traversal [11]. After copying an object, the address of the duplicate is stored in the forward field. However, the mutator always sees objects in *from space*.

Then, in a stop-the-world phase, potential mutator changes have to be incorporated: Only objects on dirty pages require correction, as those on clean pages could not have been modified by the mutator. Similarly to the non-copying case, the root set is always dirty.

Thus, for each reachable object on a dirty page in *from space*, not-yet-copied referenced objects are recursively copied to *to space*. Also, pointers to *from space* are corrected with the respective values from the forward field. Additionally, the object has to be copied from *from state* to *to state* again due to possible changes in non-reference fields of the object. Finally, all forward fields and dirty-page bits are reset, and the mutator can resume execution.

3.6 Discussion

One deficiency of mostly parallel GCs is their inadequacy for hard real-time constraints because the STW pause in both generational and full collections cannot be bounded by a constant. Nonetheless, pauses can be sufficiently short in many use cases [8].

Due to its autonomy of compiler and run-time assistance, the proposed algorithm can be utilised for whole-system garbage collection across multiple programs. Notably, the same GC can be used for various programming languages and run-time environments [28].

However, such independence comes at the cost of worse reclamation rates due to pointer misidentification. Furthermore, locality-improving measures, such as compaction, are not possible.

Additionally, determining the root set of an application is sometimes challenging: Especially in large, sparsely populated heaps with relocatable libraries, finding all valid references is not trivial without dedicated OS support [6].

If more compiler assistance than precise pointer identification is available, cooperation with the mutator enables other parallel GC approaches. In particular, compiler-inserted checks can be used to detect or prevent mutator changes to the traversed object graph.

Based on write barriers, Dijkstra et al. propose a parallel mark-and-sweep algorithm [14]: After the GC starts tracing from the root set, all pointer stores have to be examined. If a pointer is stored

in an already marked object, the target has to be traced as well; otherwise no special handling is required. A similar scheme [25] supports compaction as well.

Another approach is Baker's algorithm for fully parallel copying GC [3]: To prevent the mutator from accessing any object in *from space*, the semi-space collector intercepts all pointer dereferences with read barriers. If the target is not yet in *to space*, it is copied there and then used by the mutator. Similarly, a non-copying version was proposed [4], both can satisfy real-time constraints.

Especially if the cost of acquiring a list of dirty memory pages is high, mutator cooperation can significantly reduce GC overhead [17]. Besides, non-pointer writes do not unnecessarily taint dirty-bit information with fine-grained write detection.

While the algorithm itself proposed in [8] may not be competitive with more advanced, contemporary copying GCs, the overall scheme of parallel tracing and short stop-the-world correction is highly desired, especially on many- and multi-core CPUs [13, 16].

4 COMPARISON WITH GC ALGORITHMS ON THE JAVA VIRTUAL MACHINE

The domain of garbage collection remains a highly relevant and active field of research [13, 16] over 60 years after its inception [21]. In particular, increases in both computing capacity and demands have provided shifting challenges and possibilities: Garbage collection and its associated pause times are essential to keep latency in managed programming languages, such as Java, control- and tolerable even with increasing heap sizes.

As GC demands often depend on the executed application, the reference Java implementation OpenJDK's JVM provides a variety of GC algorithms for different needs [18, 26]. In the following, the *Concurrent Mark-Sweep* (CMS) and *Garbage-First GC* (G1GC) algorithms are reviewed and compared with Boehm et al.'s concept of mostly concurrent garbage collection.

4.1 Concurrent Mark-Sweep GC

Like most GCs on the JVM, the CMS GC [23] is using a generational approach [26]. It explicitly builds on the mostly concurrent algorithm introduced by Boehm et al. for collecting the old generation.

As mutator cooperation is already required for tracking pointers from the old into the young generation, the existing compiler-inserted write barrier is adapted for recording pointers changed during the concurrent marking phase. For full collections including the old generation, two short serialising phases are needed: One for determining the root set, and the other one for fixing mutations incurred during the concurrent tracing phase. In their implementation, only one thread is used for marking and sweeping. If the garbage collector falls behind too far, and the old generation fills up before a concurrent marking phase completes, the mutator is paused completely until the collection finishes [22]. Essentially, the algorithm then degrades to a serial scheme. Potential improvements include parallelising the GC work on multiple threads.

In the young generation, a copying semi-space collector is combined with a dedicated *eden* memory space for freshly allocated objects [26]. While GC is performed on several threads, the whole task must be carried out in a stop-the-world fashion.

CMS significantly reduces average and maximum pause times compared to serial GC [23], but multi-second pauses are still possible for large real-world applications [10, 30]. Especially for server programs, the client experience can be seriously impacted. Consequently, CMS is deprecated as of Java 9 [20] and was subsequently removed in Java 14 [24].

4.2 Garbage-First GC

Among several other optimisations, the intended successor G1GC tries to perform much tracing work in parallel on multiple threads [13]. It is a generational, copying GC intended to balance high throughput and low pause times to meet soft real-time goals [5].

Contrarily to CMS, the heap is split into a large number of evenly sized regions. Regions are categorised as either empty, *eden*, *survivor*, and *old*. In general, the eden regions contain freshly allocated objects, while survivor regions are comprised of young, retained objects. Together, both form the young generation. Sufficiently old live survivor objects are in turn evacuated to old regions.

A major design goal in G1GC is collecting a region on its own, without any dependencies on others. Therefore, *remembered sets* are associated with each region for tracking the origin of inter-regional pointers, and filled by pointer-write barriers.

In G1GC, there are three different kinds of collections: Firstly, in a young collection, only eden and survivor regions are considered for GC. In a stop-the-world fashion, the root set is calculated and traced from. Notably, the remembered sets are used to account for young objects only referenced from old regions. Hence, they are not mistakenly identified as garbage. Live objects are copied into a fresh survivor or—depending on object age—old region to improve locality and reduce heap fragmentation.

Secondly, concurrent marking of the complete heap is performed when the number of old and survivor regions exceeds a given threshold. In this case, the root set is re-used from a young-collection cycle. For tracking mutator changes, a *snapshot-at-the-beginning* (SATB) algorithm [31] is employed: During concurrent heap traversal, the old values of all pointer modifications are recorded with a write barrier. Afterwards, these values are used as additional roots for marking. This guarantees that all objects reachable at the start of the marking phase are eventually traversed, even if overwritten by the mutator in the meantime. Hence, freshly allocated objects are considered reachable. Furthermore, liveness information is calculated on-the-fly per region.

Thirdly, the next young collection additionally sweeps some of the previously marked old regions: During such a mixed collection, live objects in old regions are compacted, finally reclaiming the space of dead old objects. In particular, G1GC prioritises regions with low liveness, which contain a large number of dead objects, to merge as many regions as possible in a single collection cycle.

For all these phases, G1GC tracks the elapsed collection times. Based on these records, a prediction of how many regions can be collected within a user-defined pause interval is derived. As a result, G1GC can meet soft real-time deadlines with a high probability. However, no actual guarantees can be determined.

The old-generation collection in G1GC thus significantly differs from the mostly concurrent algorithm of Boehm et al., especially due to its usage of a SATB scheme. Furthermore, this approach

is much more involved and requires a large degree of mutator cooperation.

In practice, G1GC's design goal of providing high throughput with short, predictable pauses cannot always be met. User-specified pause times are sometimes exceeded significantly [10]. Furthermore, pauses caused by full GC in big-data applications are reported to be longer than those of CMS in some circumstances [30].

5 CONCLUSION

Due to the importance of garbage collection, various basic and advanced approaches have been developed in the last sixty years. In particular, the domain of tracing garbage collectors offers a diversity of different algorithms. As a result of large heaps and required short pause durations, plain stop-the-world algorithms offer no adequate solutions for interactive programs and latency-sensitive server applications. Thus, collecting garbage at least partially in parallel with the application is a must.

One possible solution—in combination with a more general principle for adapting STW collectors—was proposed in [8]. It combines concurrent marking with a short full mutator pause for correcting potentially changed objects. Additionally, the generational algorithm supports partial collections of recently allocated objects, which likely have a short lifespan. Along with other sophisticated optimisations, that idea can still be witnessed in varying degrees in contemporary GC algorithms on the JVM.

However, many modern GCs employ widely differing strategies for each generation. Especially in programming languages and run-time environments where significant cooperation between the application and the GC is possible, many other optimisations can be used. In particular, compiler-inserted barriers can reduce tracing work during STW pauses, and heap compaction improves heap fragmentation and can increase locality.

Overall, even with multi-core CPUs available in commodity hardware, popular GCs are not fully concurrent and still require short stop-the-world pauses. Hence, the general idea behind the proposal in [8] has prevailed and is still highly relevant today, despite different implementations being used in practice.

REFERENCES

[1] Andrew W. Appel, John R. Ellis, and Kai Li. 1988. Real-Time Concurrent Collection on Stock Multiprocessors. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. 11–20. https://doi.org/10.1145/53990.53992

[2] Andrew W. Appel and Kai Li. 1991. Virtual Memory Primitives for User Programs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, California, USA) *(ASPLOS IV)*. 96–107. https://doi.org/10.1145/106972.106984

[3] Henry G. Baker. 1978. List Processing in Real Time on a Serial Computer. *Commun. ACM* 21, 4 (April 1978), 280–294. https://doi.org/10.1145/359460.359470

[4] Henry G. Baker. 1992. The Treadmill: Real-Time Garbage Collection without Motion Sickness. *ACM SIGPLAN Notices* 27, 3 (March 1992), 66–70. https://doi.org/10.1145/130854.130862

[5] Monica Beckwith. 2013. *Garbage First Garbage Collector Tuning*. https://www.oracle.com/technical-resources/articles/java/g1gc.html

[6] Hans-J. Boehm. 1991. Hardware and operating system support for conservative garbage collection. In *Proceedings of the International Workshop on Object Orientation in Operating Systems*. 61–67. https://doi.org/10.1109/IWOOOS.1991.183023

[7] Hans-J. Boehm and David Chase. 1992. A Proposal for Garbage-Collector-Safe C Compilation. *The Journal of C Language Translation* 4, 2 (1992), 126–141.

[8] Hans-J. Boehm, Alan J. Demers, and Scott Shenker. 1991. Mostly parallel garbage collection. *ACM SIGPLAN Notices* 26, 6 (1991), 157–164.

[9] Hans-J. Boehm and Mark Weiser. 1988. Garbage collection in an uncooperative environment. *Software: Practice and Experience* 18, 9 (1988), 807–820.

[10] Maria Carpen-Amarie, Patrick Marlier, Pascal Felber, and Gaël Thomas. 2015. A Performance Study of Java Garbage Collectors on Multicore Architectures. In *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores* (San Francisco, California) *(PMAM '15)*. 20–29. https://doi.org/10.1145/2712386.2712404

[11] C. J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (Nov. 1970), 677–678. https://doi.org/10.1145/362790.362798

[12] Alan Demers, Mark Weiser, Barry Hayes, Hans-J. Boehm, Daniel Bobrow, and Scott Shenker. 1989. Combining generational and conservative garbage collection: Framework and implementations. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 261–269.

[13] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-First Garbage Collection. In *Proceedings of the 4th International Symposium on Memory Management* (Vancouver, BC, Canada) *(ISMM '04)*. 37–48. https://doi.org/10.1145/1029873.1029879

[14] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. 1978. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Commun. ACM* 21, 11 (Nov. 1978), 966–975. https://doi.org/10.1145/359642.359655

[15] Robert R. Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (nov 1969), 611–612. https://doi.org/10.1145/363269.363280

[16] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools* (Lugano, Switzerland) *(PPPJ '16)*. Article 13, 9 pages. https://doi.org/10.1145/2972206.2972210

[17] Antony L. Hosking and J. Eliot B. Moss. 1993. Protection Traps and Alternatives for Memory Management of an Object-Oriented Language. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. 106–119. https://doi.org/10.1145/168619.168628

[18] Nicole Jagelid. 2020. *Performance evaluation of Java garbage collectors for large heap transaction based applications*. Master's thesis. KTH Royal Institute of Technology - School of Electrical Engineering and Computer Sciences.

[19] Henry Lieberman and Carl Hewitt. 1983. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Commun. ACM* 26, 6 (June 1983), 419–429. https://doi.org/10.1145/358141.358147

[20] Jon Masamitsu. 2015. *JEP 291: Deprecate the Concurrent Mark Sweep (CMS) Garbage Collector*. https://openjdk.java.net/jeps/291

[21] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (April 1960), 184–195. https://doi.org/10.1145/367177.367199

[22] Oracle Corporation. 2014. *Java Platform, Standard Edition 8 HotSpot Virtual Machine Garbage Collection Tuning Guide - Concurrent Mark Sweep (CMS) Collector*. https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/cms.html

[23] Tony Printezis and David Detlefs. 2000. A Generational Mostly-Concurrent Garbage Collector. In *Proceedings of the 2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA) *(ISMM '00)*. 143–154. https://doi.org/10.1145/362422.362480

[24] Thomas Schatzl. 2019. *JEP 363: Remove the Concurrent Mark Sweep (CMS) Garbage Collector*. https://openjdk.java.net/jeps/363

[25] Guy L. Steele. 1975. Multiprocessing Compactifying Garbage Collection. *Commun. ACM* 18, 9 (Sept. 1975), 495–508. https://doi.org/10.1145/361002.361005

[26] Sun Microsystems. 2006. Memory Management in the JavaHotSpot™ Virtual Machine. https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf

[27] David Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE 1)*. 157–167. https://doi.org/10.1145/800020.808261

[28] M. Weiser, A. Demers, and C. Hauser. 1989. The Portable Common Runtime Approach to Interoperability. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. 114–122. https://doi.org/10.1145/74850.74862

[29] Paul R. Wilson. 1992. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management 1992* (St. Malo, France) *(Lecture Notes in Computer Science, Vol. 637)*, Yves Bekkers and Jacques Cohen (Eds.). 1–42. https://doi.org/10.1007/BFb0017182

[30] Lijie Xu, Tian Guo, Wensheng Dou, Wei Wang, and Jun Wei. 2019. An Experimental Evaluation of Garbage Collectors on Big Data Applications. *Proc. VLDB Endow.* 12, 5 (Jan. 2019), 570–583. https://doi.org/10.14778/3303753.3303762

[31] Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software* 11, 3 (1990), 181 – 198. https://doi.org/10.1016/0164-1212(90)90084-Y