OREGON PROGRAMMING LANGUAGES SUMMER SCHOOL

System Design and Innovation: A Garbage Collection Case Study — Kathryn S McKinley

Lecture 1 - June 26, 2025

1 A Brief Introduction

With the current advancements in the field of AI, it seems as if we are currently at the cusp of the 4th revolution of computer science and innovation. This revolution will lead to the development of new applications, programming languages, and will change the landscape of computer science as a whole. It is quite similar to how garbage collection revolutionized the world of computer science decades ago. But what are garbage collectors and why should we care about them today?

2 A Brief Reasoning

In short, garbage collection frees the programmer from manually managing memory, thereby avoiding some errors, as well as providing memory safety. As such, more and more languages, even new ones such as the Go language and the Rust language, use garbage collection due to it making the development of programs easier, as well as lowering the chance of memory leaks in general.

We will analyze garbage collection in both a technical and a nontechnical sense, wherein we will be framing the details around decades of research and how garbage collection has evolved throughout these decades.

3 A Brief History

The historical foundations of garbage collection go back to the 1960s in the McCarthy era, wherein John McCarthy invented Garbage collection around 1960 with the purpose of simplifying memory management.

The 1960s was a decade of innovation in the field of garbage collection where new algorithms would constantly innovate upon one another. The main garbage collection algorithms that we will be considering in this paper are:

- Mark-Sweep (MacCarthy 1960)
- Mark-Compact (Styger 1967)
- Semi-Space (Cheney 1969)
- Mark-Region (Blackburn and McKinley 2008)

We will attempt to cover these algorithms in the latter half of this set of notes.

4 Garbage Collection Fundamentals

The three fundamentals of a garbage collector are:

1. Allocators
2. Identifiers
3. Reclamation

Let us give a very high level view of each:

Allocation is when the programmer says `new`. It signals that we have to put an object somewhere in the memory.

We *identify* when we decide it is time to collect, and we want to run the garbage collector, e.g., because we ran out of heap space.

During *reclaiming* we know where the alive and dead memory is, and we want to reorganize it in a way that allows us to allocate into it again.

4.1 Allocation

There are a number of ways in which allocation is treated, some of which include:

- Free List
  - Groups objects by their size
  - Blocks are identified by free size – for every size, we hold a list of free blocks with a given size
  - Each object is allocated to the block that can accommodate it and has the smallest amount of free memory – we remove its block from the free list
- Bump Allocation
  - Puts objects in a sequential buffer
  - New objects are appended to the end

4.2 Identification

How do we go about identifying what is and what isn't dead memory?

- Tracing (Implicit)
  - Flexible
  - Low overhead
  - Tracing works by scanning roots (= pointers to the heap from caches and stack). At the time, of identification, we examine the graph of objects and references between them. All objects that are reachable from the roots are marked as alive.
- Reference Counting (Explicit)
  - Predictable
  - Simple
  - High overhead (we need to remember all the numbers of references)
  - Poor locality (locality is a measure of cache performance. It measures whether or not you got a cache hit)
  - We count the number of references pointing to every object. Once the number reaches zero, the object is identified as dead.
  - It is not really used as it is incomplete (we do not get rid of cycles).
  - If we want to restore completeness, we usually do Tracing

4.3 Reclamation

How do we reclaim (free memory)?

- Sweep-to-free
  - We go through the whole memory on heap allocated to our program and all the blocks unmarked in identification are add back to the free memory.
  - If we are using free lists for allocation (as is the case for Mark-sweep later), this is easy as we just add the blocks to the free lists.
- Compact
  - Similar to sweep-to-free – the difference is in moving all the alive objects to a contiguous space in memory
  - Takes more time, but saves more space
  - Makes less sense together with free list
- Evacuate
  - We keep two pieces of memory of the same size, one empty (prepared for the evacuation), the other is used for our data
  - Once we are freeing, we copy all the alive objects to the empty piece, erase the one that was used and swap their roles
- Sweep-to-region
  - Memory is divided into fixed sized regions, each of which is either free or unavailable.
  - Objects cannot span multiple regions.
  - The collector marks any region containing a live object as unavailable and all others as free.
  - Similarly to Evacuate, we keep some free memory for copying, but it does not have to that much.

Surprisingly! or rather unsurprisingly, it is possible to combine any version of the allocation, identificaton and reclamation together.

5 A Brief Return to The Algorithms

We will briefly cover the 4 algorithms that have been previously mentioned in a bit more detail in this section. We will also discuss some of their key issues and strengths below:

5.1 Mark-Sweep

- Allocation = Free list
- Identification = Tracing
- Reclamation = Sweep to free list

One of the key issues of Mark-sweep is that it tends to have poor locality which leads to more cache misses, but it has low computational overhead.

5.2 Mark-Compact

- Allocation = Bump pointer
- Identification = Tracing
- Reclamation = Compact

Mark-compact tends to have good locality and super fast allocation due to the fact that you are bumping a pointer. Furthermore, it has no fragmentation and tracing tends to be fast. However, one key issue of the algorithm is that execution tends to be the slowest one of all 4 algorthims.

5.3 Semi-Space

- Allocation = Bump pointer
- Identification = Tracing
- Reclamation = Evacuate

Semi-space trades memory to get good locality. Additionally, it has no fragmentation but spends double the memory space; however, the collector itself is quite fast.

5.4 Mark-Region

- Allocation = Bump pointer
- Identification = Tracing
- Reclamation = Sweep to region

Mark-Region doesn't use one continuous space, rather it divides the region into blocks. While it has low fragmentation, there will be a need for copying to get rid of fragmentation, thus raising the total cost.

5.5 Opportunistic Defragmentation

Opportunistic defragmentation was introduced to minimize the heap size. Its heuristic is that there is an abundance of free memory, but most of which was not deallocated or in other terms, it compares the ratio of usable memory to that of the free memory if it is too big, we start the defragmentation. Moreover, the garbage collector is using the same allocator that was used for the application to determine where to place the object which tends to make it lightweight.

5.6 Mark-Region Immix: Lines and Blocks

Immix is a mark-region garbage collector with space efficiency and fast collection. It combines mark-region and opportunistic defragmentation. It is important to take the regions into account because if the regions were increased in size and bordering on too big, then there would be more contiguous memory allocation but with the drawback of having more fragmentation. Now, if the regions are too small, then there will also be fragmentation, as well as more metadata overhead which constrains the buffer.

For a reader interested in comparison of the algorithms introduced above, we include the following graphs from [BM08].

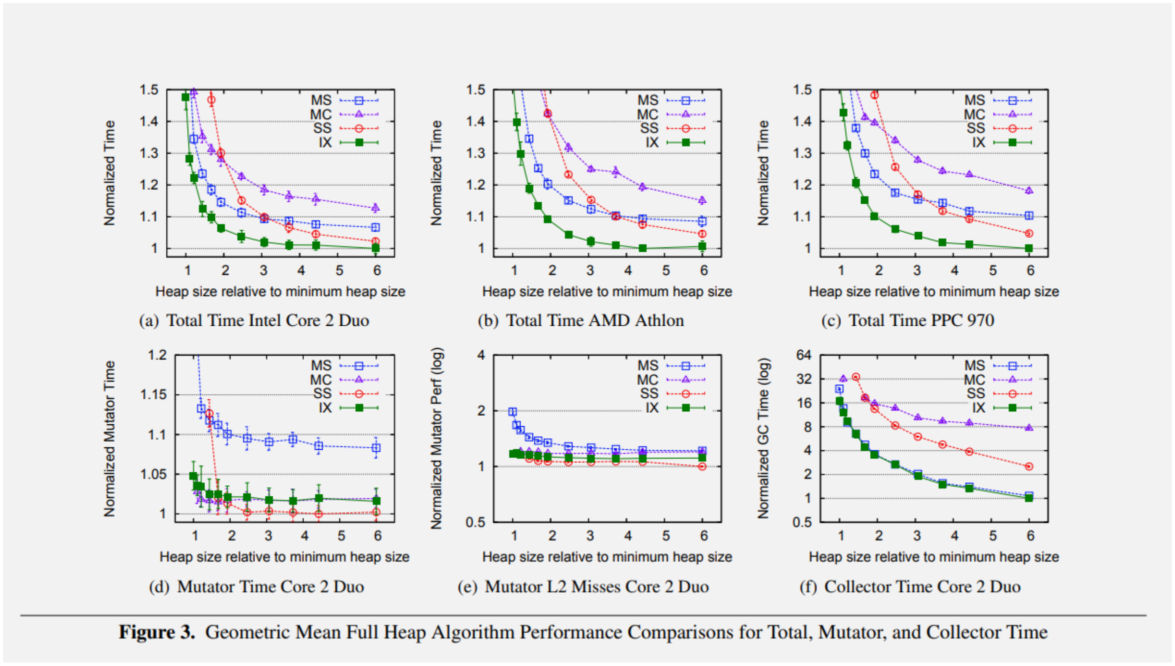

Figure 1: Comparison of performances of the algorithms from [BM08]

6 Generational Garbage Collectors and Parallelism

Generational garbage collectors use bumper pointer nursery and any of the previously mentioned heap organizations for the old space. They tend to be stop the world collectors, and the most popular one is the Garbage-First (G1) garbage collector.

Parallel collectors are generational collectors as well. These collectors have achieved their goal of getting smaller pauses, and while they might seem promising, they aren't perfect. Running them at 100 % utilization will cause performance to suffer, so it is better to run them at 70 % utilization. One drawback of parallel collectors is that they have periodic length pauses which add latency and queuing delays, which is why they have not achieved their goal of getting lower latency. Another drawback is that they have read and write barriers which tend to affect performance. Write barriers can be optimized to be fast, but read barriers, which cause a performance hit of 5 to 10 %, cannot be optimized to be faster.

References

[BCM04] Stephen Blackburn, Perry Cheng, and Kathryn Mckinley. "Myths and Realities: The Performance Impact of Garbage Collection". In: *Performance Evaluation Review* 32 (May 2004). DOI: 10.1145/1012888.1005693.

[BM08] Stephen Blackburn and Kathryn McKinley. "Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance". In: vol. 43. June 2008, pp. 22–32. DOI: 10.1145/1379022.1375586.