# Performance and Software Engineering Aspects of Automatic Memory Management

Godmar Back

Virginia Tech

April 8, 2020

# Performance Aspects

- Real-world Garbage Collectors vary widely in the trade-offs they make. Decades have been spent engineering them.
  - E.g. Java 10 ships with four different collectors; ZGC is a 5th collector added in 2018.
- They differ in many characteristics: program throughput, memory overhead, GC throughput, scalability, etc.
- Having a good understanding of your workload is a must to properly tune the garbage collector's policies and understand its performance impact

VIRGINIA TECH.

# Modeling the Cost of GC

- **Compacting Collectors:** Garbage collectors have near perfect knowledge of the object graph and in particular the locations where pointers to other objects are stored. Thus, they can move objects (and update any pointers to them), allowing for *compaction*. Live objects are "evacuated" or "scavenged" from a region of the heap, leaving only unreachable objects behind, eventually allowing the region to be reclaimed in one fell swoop.
- Cost of GC thus depends on
  1. Cost of marking + evacuation - proportional to the size of the live objects in the area that is marked
  2. Cost of sweeping - in theory, constant. In practice, allocator will need to zero out memory for reuse in most languages, thus proportional to the amount of garbage produced

VIRGINIA TECH

# Memory Allocation Time Profile

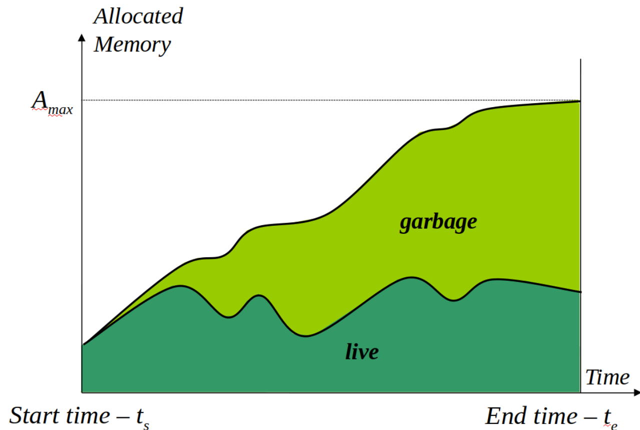


Figure 1: Memory Allocation vs Time in the absence of GC. $A_{\text{max}}$ denotes the heap limit. Garbage increases monotonically.

VIRGINIA TECH

# Simplified Memory Allocation Time Profile

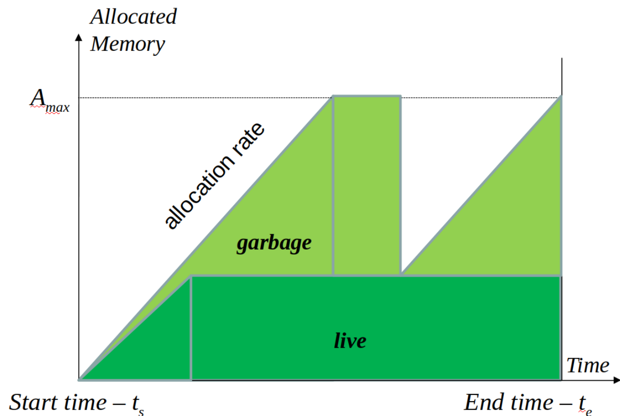


Figure 2: Simplified memory allocation profile, showing one GC

# Synthetic Example

## LargeLiveHeap.java

```
public class LargeLiveHeap
{
  public static void main(String []av) {
    int numLive
        = Integer.parseInt(av[0]);
    int numAllocations
        = Integer.parseInt(av[1]);

    byte[][] l = new byte[numLive][];
    for (int i = 0; i < numAllocations; i++)
      l[i % numLive] = new byte[100000];
  }
}
```

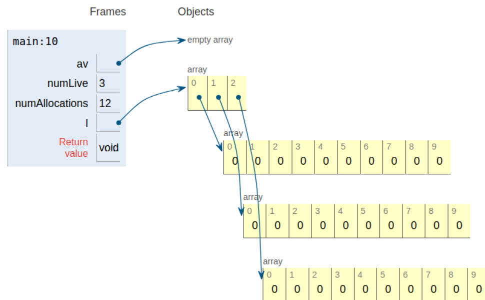


Figure 3: numLive = 3, numAllocations = 12

How does GC time change with $A_{max}$?

VIRGINIA TECH.

# Heap Size vs GC Frequency

- Large Live Heap Sizes tend to increase frequency of garbage collections when JVM approaches heap limit (in Java `-Xmx` switch)
- Policy question: should JVM ask OS for more memory and if so, how much?
- General trade-off between amount of memory JVM is willing to use and achievable throughput
- Traditional rule of thumb: budget $1.5\times - 2.5\times$ the size of the live heap to stay within acceptable performance overhead compared to explicit allocation; but appears to be too optimistic
- Hertz 2005 [2]:
  - GC outperforms malloc with $5\times$
  - GC needs $3\times$ for 17% degradation
  - With only $2\times$ may be up to 70% slower
- Performance degradation ("gc thrashing") as live heap size approaches maximum heap size

# The Generational Hypothesis

- Likelihood of objects to stay live increases over time, aka "most objects die young"
- Allocate objects in special area called "nursery," or "Eden" space which is collected more frequently in minor collections
- Evacuate surviving objects into older generation(s) which is less frequently collected in major collections
- This requires coordination between mutator (user program) and collector through write barrier:
  - Mutator must inform collector of pointers into Eden Space: `old.field = young` must add a root for `young`

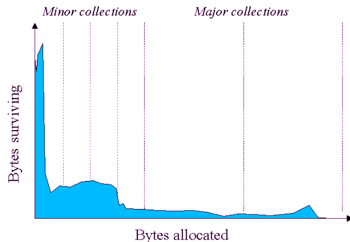


Figure 4: Generational Hypothesis

VIRGINIA TECH

# Garbage Collection vs Mutators

**What if the reachability graph changes while GC takes place?**

Must avoid inadvertently missing the last pointer to an object that keeps it alive

- "Stop-the-World" approach. Stop all mutators while collecting. Leads to "GC Pauses"
- Incremental collection. Do small chunks of GC work while allocating objects
- Concurrent/Parallel collection. GC runs in a separate thread that synchronizes with mutator threads in some way - typically, the mutator informs the collector of new edges created, and/or the collector informing the mutator when pointers have changed

VIRGINIA TECH

# G1GC and ZGC

- Current default collector: G1 GC see Beckwith 2013 and G1 Tutorial
  - Designed for multicore systems where idle cores are available to assist in GC
- ZGC (2018), see ZGC Video (Listen to 7:51 on tuning)
  - Single generation collector with low latency (stops mutator only during root scanning)
  - Use load barriers to be able to move objects without stopping mutator
- For a balanced perspective, recommend these blog posts:
  - Modern garbage collection: Hearn 2016
  - part 2: Hearn 2019

VIRGINIA TECH.

# Programmer's Perspective

1. Your program runs out of memory (in Java, `OutOfMemoryError` is thrown). This happens when your live heap size exceeds the memory limit, which typically happens for one of the following 3 reasons:
   - Heap Size Limit is too low
   - Leaks
   - Bloat
2. Your program's performance degrades because a lot of time is spent in GC
   - Churn
   - Lack of headroom

VIRGINIA TECH

# Leak vs Bloat vs Churn

- Leaks: the live heap contains reachable objects that the program will not access in the future (though it could), e.g.
  - items placed in hash maps that will not be looked up
  - event handlers and/or callbacks registered and not unregistered
- Weak references allow garbage collector to free objects that can be recreated, e.g. when caching
- Bloat: your heap contains only objects your program will access, but these objects take up too much space, e.g. [1]
  - Boxed integer in Java
  - Items stored in HashMaps
- Memory Analysis tools help with leaks and bloat
- Churn: your program allocates many short-lived objects (high allocation rate); particularly important on resource-constrained devices such as Android [URL]

# Conservative Garbage Collection

- *Precise* garbage collectors have precise information regarding the structure of all heap objects and stack frames, thus precise knowledge of all locations that store pointers
- This is a viable assumption in managed languages: Java, $C^{\#}$, JavaScript, Python, Go, but it's (generally) not viable in C/C++
- *Conservative* collectors scan the heap and assume anything that could be a pointer is one
- May keep some objects alive that are not reachable
- Boehm's GC is a well-known collector for C/C++
- GC Safety is the property of a compiler to avoid producing code that could mislead a garbage collector into missing pointers (available in C++11)
- valgrind's leak detection uses the same approach [URL]

# References

[1] Adriana E. Chis, Nick Mitchell, Edith Schonberg, Gary Sevitsky, Patrick O'Sullivan, Trevor Parsons, and John Murphy.
Patterns of memory inefficiency.
In *Proceedings of the 25th European Conference on Object-Oriented Programming*, ECOOP'11, page 383–407, Berlin, Heidelberg, 2011. Springer-Verlag.

[2] Matthew Hertz and Emery D. Berger.
Quantifying the performance of garbage collection vs. explicit memory management.
In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, page 313–326, 2005.

VIRGINIA TECH.