Debugging and Resolving Performance Issues in Real World Java Application

By
- Vaibhav Choudhary (JVM Engineer)

https://www.youtube.com/BangaloreJUG

Staying on older JDK major version

- Those team who want to gain major performance gain should always be on the latest JDK. At least, latest LTS version.

- I can show you three simple example here and prove the statement
  - "Hello World" run (how to find the startup time?)
  - "Allocation/Deallocation" run
  - "Computation" run

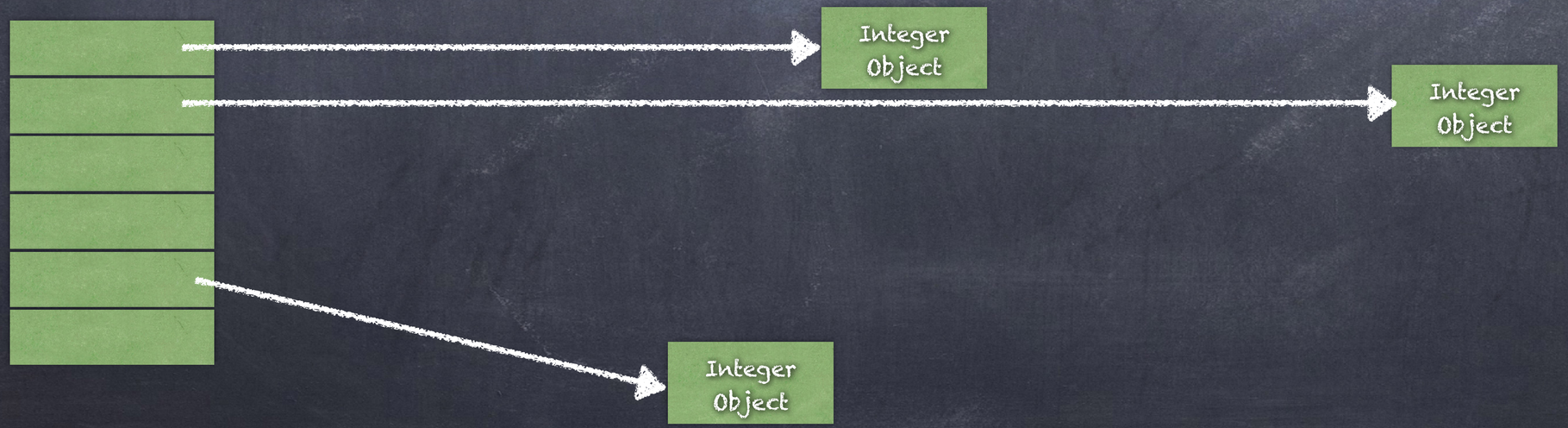
- Why I don't migrate to the latest JDK version ?
  - Myths :
    - It's lesser stable than older JDK
    - It's time consuming.
  - Facts :
    - I am lazy
    - My vision don't go so far

Excessive Garbage Collection

- Excessive garbage creation is uncovered issue in many applications.
- Lets find out what wrong in this piece of code.
- Creating too many humongous objects in algorithm like G1 GC is not a good idea. If unavoidable we must tune it.
- Myth: GC Ergonomics find out best for me.
- Fact: GC Ergonomics can tune as per system config not as per code like humongous allocation here.

Careful Computation

- Locality issue - Select your data structure carefully when running parallelism.
- This simple code can demonstrate that Integer can be a killer.
- Myth: Integer just take extra space.
- Fact: Integer take extra time as well.



Blocking compute time is crime

- World of cloud and cost per computation is like traveling in a meter running auto. We must optimize our computation to the best.
- This simple demo will show issue with logging system.
- Use asynchronous logging libraries like Logback or Log4j2
- Myth: Writing async code is too tough.
- Fact: Not now !

Thread Computation

- A world where we have an undefined blend of Compute and IO jobs, we must carefully handcraft the number of threads to be created.
- Either apply the right threading model or use advance features like virtual threads.
- Myth: Identify IO job or compute job is tough
- Fact: Most of the profiler can do it very easily.

Classic thread model

- N_thread = N_core X Utilization X (1 + Wait Time / Compute Time)

-> Let's say we have 24-core CPU, we aim 80% CPU utilization and we have compute-intensive tasks so our (Wait Time / Compute Time) = 1/10. Then we find our thread pool size as (24 * 0.8 * (1 + 1/10)) = 21.

-> If we have IO heavy tasks and our (Wait Time / Compute Time) = 10/1. Then we find our thread pool size as (24 * 0.8 * (1 + 10/1)) = 211.

-> Theory is just a start point. Seeing is believing.

Thank you

vaibhav.kumar@gmail.com

See you in JavaFest '25