NG2C: N-Generational GC for Big Data Memory Management

Rodrigo Bruno, Paulo Ferreira
INESC-ID / Instituto Superior Técnico, University of Lisbon
{rodrigo.bruno,paulo.ferreira}@inesc-id.pt

The widely accepted empirical rule that states that most objects die young is currently used for designing Garbage Collection (GC) algorithms which run on most platforms. However, this rule is not suited for a wide class of Big Data systems. First, systems designed for storage (eg. Cassandra and Tachyon) maintain frequently accessed data in memory (caches), and the objects representing such data tend to be reachable (live) for a long time (from the GC point of view). Second, systems that allocate memory very fast will trigger minor collections within a short interval, making recently allocated objects (that live for a short period of time) to be promoted to the old generation (instead of being collected in a minor collection). Examples of such systems are distributed processing systems (eg. Spark, Hadoop) that can have dozens or even hundreds of parallel tasks being processed in a single process.

This mismatch between the objects' real life-cycle timings and the GC assumptions has serious consequences for memory management in Big Data applications since such applications work with large amounts of data in memory: i) minor collection pauses increase as the amount of objects to promote increase, and ii) as more objects get promoted into the old generation, the risk of incurring into a full collection increases as well. This results in increased application overhead, including significant application pause times, which can be harmful for Service Level Agreements (SLAs).

With NG2C we want to be able to control how the GC handles the data inside a Big Data application in order to: i) minimize the number of objects that get promoted into the old generation (thus reducing the pause times associated to minor collections) by avoiding doing that for objects that will effectively die later, and ii) selectively collect objects in the old generation (to avoid reaching limit scenarios when a full collection is necessary) given that they were erroneously promoted to old according to their true life-cycle timings.

This goal cannot be attained easily by tweaking the heap or GC parameters; it needs a more fundamental approach. For example, simply increasing the size of the young generation might result in a bigger time interval between minor collections and thus result in less live objects to promote; however, increasing the young generation size is dangerous because it can easily lead to high pause times if the number of live objects to promote is also high (for example due to a workload shift). This gets even worse if we increase the number of parallel working tasks in the local process since memory gets allocated faster.

There are several solutions for dealing with high volumes of data while avoiding the GC overhead. These typically employ off-heap memory (i.e. allocate memory for the application outside the GC-managed heap). While this is an interesting approach to allocate and keep data out of the range of the GC, it has several important drawbacks: i) off-heap data needs to be serialized to be saved in off-heap memory and deserialized before being used by the application (this obviously has some performance overhead); ii) off-heap memory must be explicitly collected by the programmer (which is error prone and completely ignores the advantages of running inside a memory managed environment).

Our solution is based on the idea that developers know and understand the objects' life-cycle much better than the GC. This comes from the fact that, in most Big Data applications, there are clearly defined stages which limit the reachability of objects. For example, a memory cache flush or the end of a computation task clearly limits the life time of all the objects created for these specific stages. By taking advantage of this knowledge, it is possible to improve the cooperation between the developer and the GC, resulting in better GC decisions, leading to less performance overhead.

Therefore, in NG2C, we propose to extend the current GC design of two generations (young and old) to an arbitrary number of generations, each holding objects with similar life times. The programmer is able to create new generations and to allocate objects directly in each generation. When most of the objects inside a generation are expected to be dead, the programmer asks the GC to collect a specific generation. With this solution, we avoid costly minor collections since objects are allocated directly in a specific generation (according to theirs expected life-cycle) and also avoid full collections since each generation can be collected separately.

Following the previous example, when using a cache, the programmer would allocate all objects associated to a specific cache in a particular generation and only when the cache is flushed, the programmer asks the GC to collect that specific generation (which will contain mostly dead objects). Similarly, for the other example (processing tasks), all objects related to a task or group of tasks would be allocated in a specific generation which would be collected when the task is finished.

NG2C is being implemented in the OpenJDK 8 Hotspot Java Virtual Machine. We also extended the JDK to include methods for creating and deleting memory generations. Preliminary results are encouraging as we are able to achieve both goals: i) avoid (erroneous) object promotion, and ii) avoid (lengthy) full collections. The next step is to modify existing Big Data applications to take advantage of NG2C and measure the obtained performance.