Part 1: Introduction to the G1 Garbage Collector

December 6, 2016 | Matt Robson

< Back to all posts

Tags: *Technical Account Managers*

To most people, the Java Garbage Collector is a black box that happily goes about its business. Programmers develop an application, QE validates the functionality and the Operations team deploys it. Within that process, you may do some tweaking of the overall heap, PermGen / Metaspace or thread settings, but beyond that things just seem to work. The question then becomes, what happens when you start pushing the envelope? What happens when those defaults no longer suffice? As a developer, tester, performance engineer or architect, it's an invaluable skill set to understand the basics of how Garbage Collection works, but also how to collect and analyze the corresponding data and translate it into effective tuning practices. In this

ongoing series, we're going to take you on a journey with the G1 Garbage Collector and transform your understanding from beginner to aficionado that places GC at the top of your performance pile.

We're leading off this series with the most fundamental of topics: What is the point of the G1 (Garbage First) Collector and how does it actually work? Without a general understanding of its goals, how it makes decisions and how it's designed, you are setting out to achieve a desired end state with no vehicle or map to get you there.

At its heart, the goal of the G1 collector is to achieve a predictable soft-target pause time, defined through -XX:MaxGCPauseMillis, while also maintaining consistent application throughput. The catch and ultimate goal is to be able to maintain those targets with the present day demands of high-powered, multi-threaded applications with an appetite for continually larger heap sizes. A general rule with G1 is that the higher the pause time target, the achievable throughput, and overall latency become higher. The lower the pause time target, the achievable throughput and overall latency become lower. Your goal with Garbage Collection is to combine an understanding of the runtime requirements of your application, the physical characteristics of your application and your understanding of G1 to tune a set of options and achieve an optimal running state that satisfies your business requirements. It's important to keep in mind that tuning is a constantly evolving process in which you establish a set of baselines and optimal settings through repetitive testing and evaluation. There is no definitive guide or a magic set of options, you are responsible for evaluating performance, making incremental changes and re-evaluating until you reach your goals.

For its part, G1 works to accomplish those goals in a few different ways. First, being true to its name, G1 collects regions with the least amount of live data (Garbage First!) and compacts/evacuates live data into new regions. Secondly, it uses a series of incremental, parallel and multi-phased cycles to achieve its soft pause time target. This allows G1 to do what's necessary, in the time defined, irrespective of the overall heap size.

Above, we made reference to a new concept in G1 called 'regions'. Simply put, a region represents a block of allocated space that can hold objects of any generation without the need to maintain contiguity with other regions of the same generation. In G1, the traditional Young and Tenured generations still

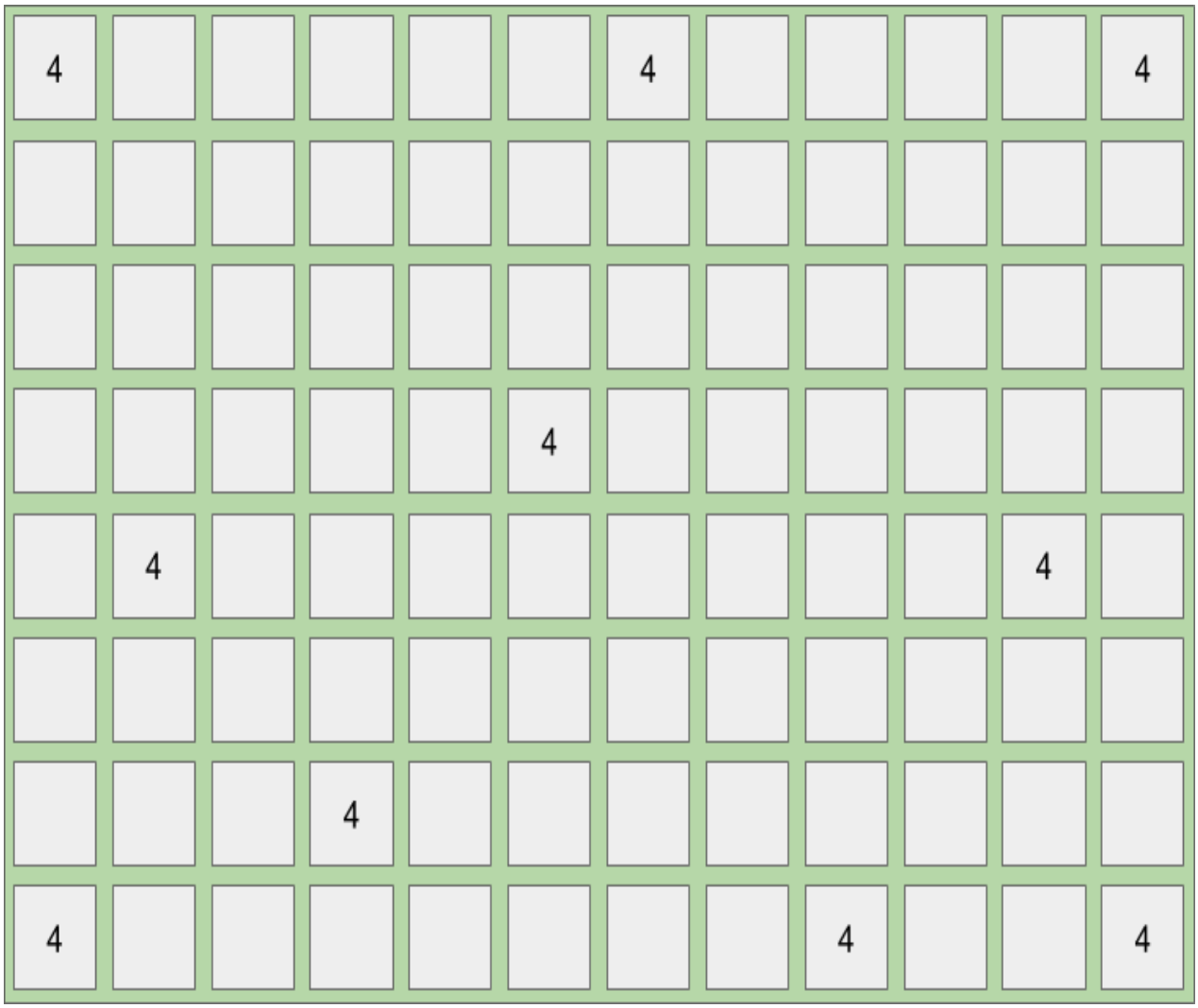
objects start and Survivor space, where live eden objects are copied to during a collection. Objects remain in the Survivor space until they are either collected or old enough for promotion, defined by the XX:MaxTenuringThreshold (defaults to 15). The Tenured generation consists of the Old space, where objects are promoted from the Survivor space when they reach the XX:MaxTenuringThreshold. There is, of course, an exception to this and we'll cover that towards the end of the article. The region size is calculated and defined when the JVM starts. It is based on the principle of having as close to 2048 regions as possible where each region is sized as a power of 2 between 1 and 64 MB. More simply put, for a 12 GB heap:

12288 MB / 2048 Regions = 6 MB - this is not a power of 2

12288 MB / 8 MB = 1536 regions - generally too low

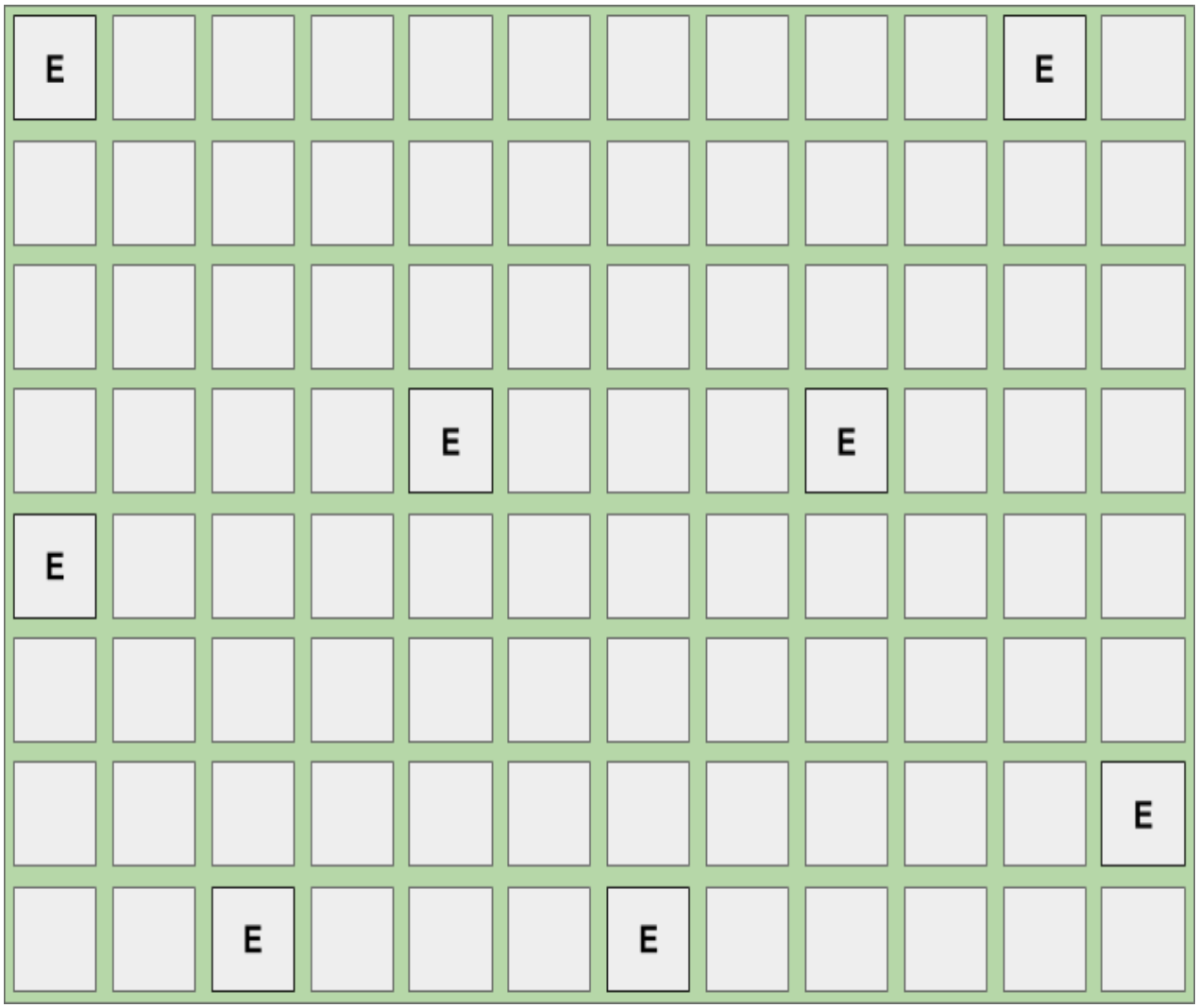
12288 MB / 4 MB = 3072 regions - acceptable

Based on the above calculation, the JVM would, by default, allocate 3072 regions, each capable of holding 4 MB, as illustrated in the diagram, below. You also have the option of explicitly specifying the region size through -XX:G1HeapRegionSize. When setting the region size, it's important to understand the number of regions your heap-to-size ratio will create because the fewer the regions, the less flexibility G1 has and the longer it takes to scan, mark and collect each of them. In all cases, empty regions are added to an unordered linked list also known as the "free list".



The key is that while G1 is a generational collector, the allocation and consumption of space is both non-contiguous and free to evolve as it gains a better understanding of the most efficient young to old ratio. When object production begins, a region is allocated from the free list as a thread-local allocation buffer (TLAB) using a compare and swap methodology to achieve synchronization. Objects can then be allocated within those thread-local buffers without the need for additional synchronization. When the region has been exhausted of space, a new region is selected, allocated and filled. This continues until the cumulative Eden region space has been filled, triggering an evacuation pause (also known as a young collection / young gc / young pause or mixed collection / mixed gc / mixed pause). The cumulative amount of Eden space represents the number of regions we believe can be collected within the defined soft pause time target. The percentage of total heap allocated for Eden regions can range from 5% to 60% and gets dynamically adjusted after each young collection based on the performance of the previous young collection.

Here is an example of what it looks like with objects being allocated into non-contiguous Eden regions;

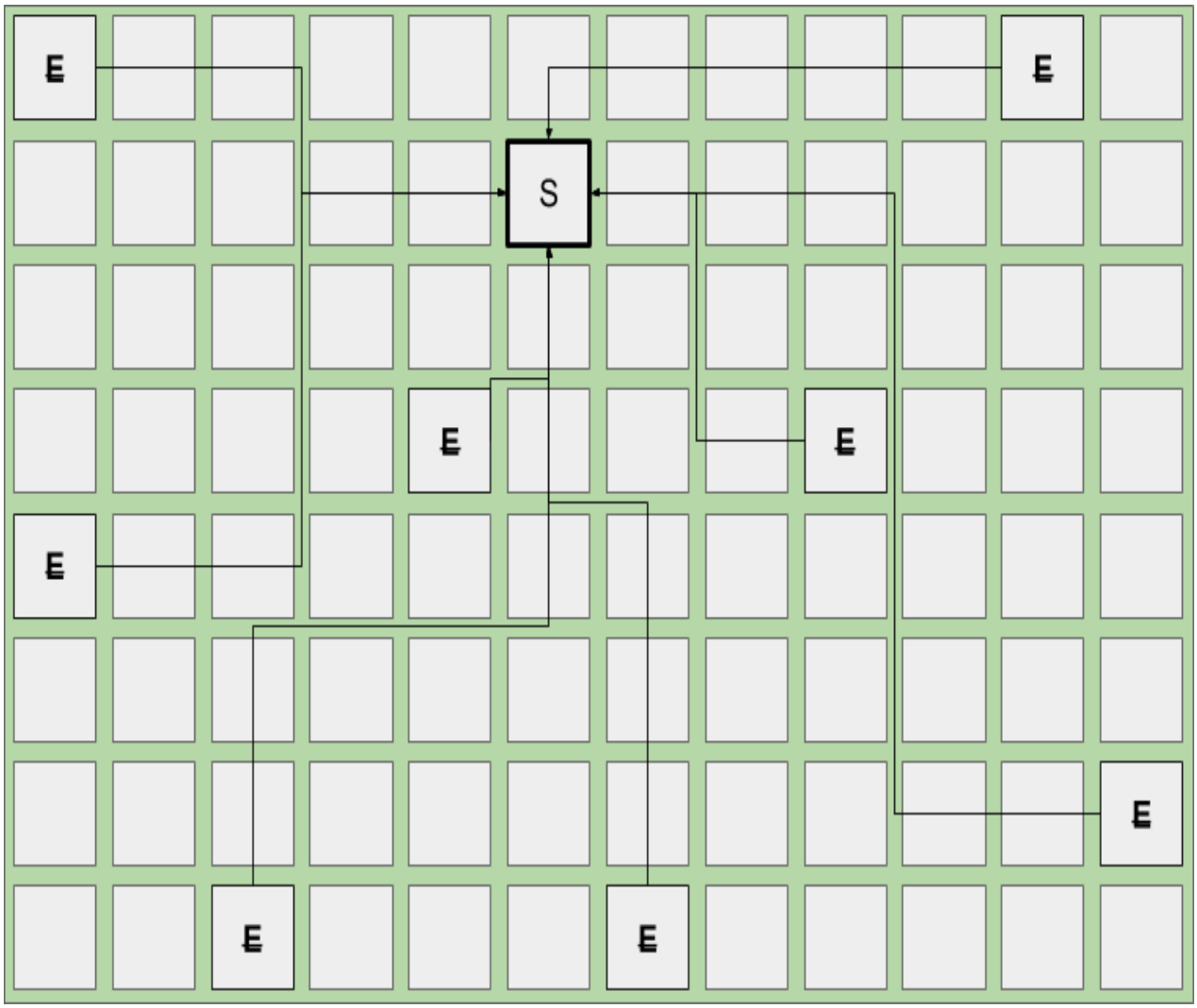



```
GC pause (young); #1
          [Eden: 612.0M(612.0M)->0.0B(532.0M) Survivors: 0.0B->8
GC pause (young); #2
          [Eden: 532.0M(532.0M)->0.0B(532.0M) Survivors: 80.0M->
```

Based on the above 'GC pause (young)' logs, you can see that in pause #1, evacuation was triggered because Eden reached **612.0M** out of a total of **612.0M** (153 regions). The current Eden space was fully evacuated, **0.0B** and, given the time taken, it also decided to reduce the total Eden allocation to **532.0M** or 133 regions. In pause #2, you can see the evacuation is triggered when we reach the new limit of **532.0M**. Because we achieved an optimal pause time, Eden was kept at **532.0M**.

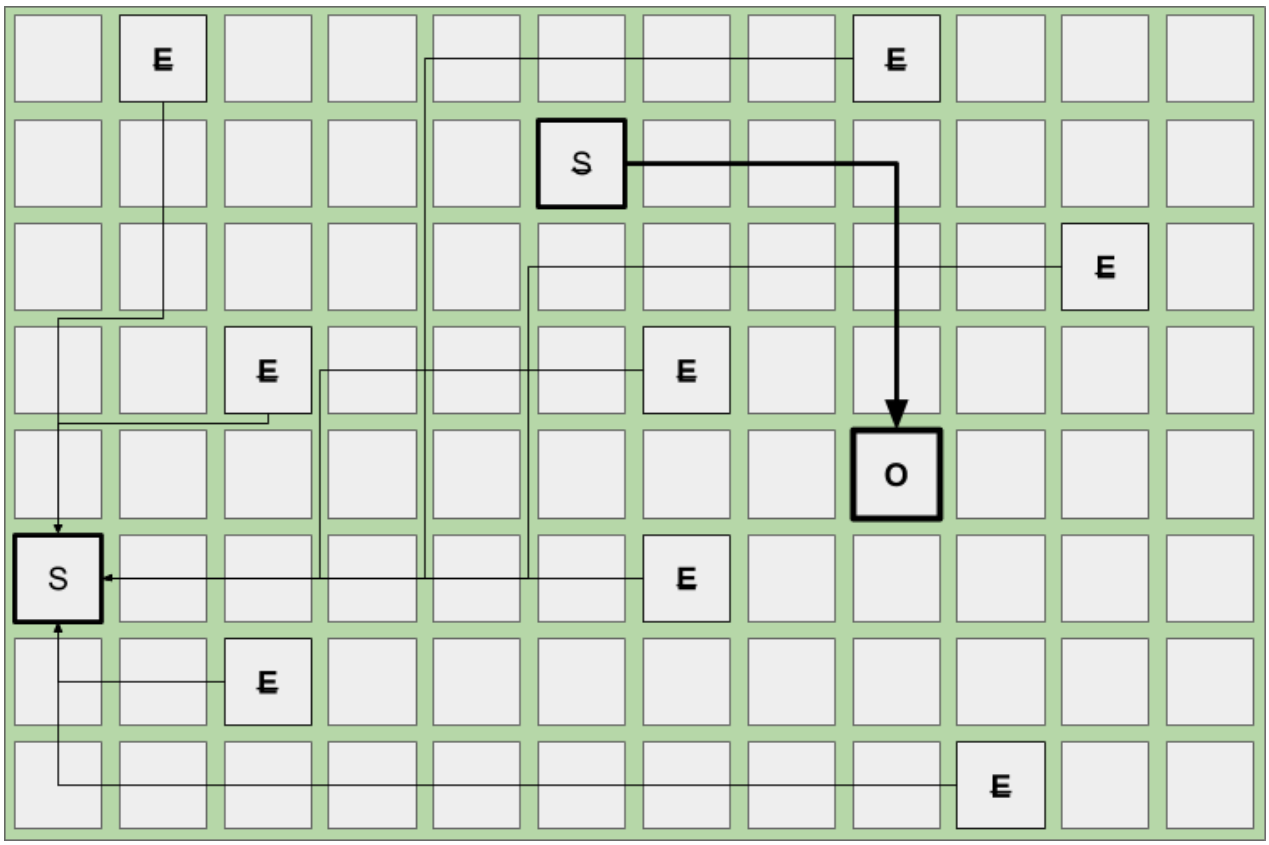
When the aforementioned young collection takes place, dead objects are collected and any remaining live objects are evacuated and compacted into the Survivor space. G1 has an explicit hard-margin, defined by the G1ReservePercent (default 10%), that results in a percentage of the heap always being available for the Survivor space during evacuation. Without this available space, the heap could fill to a point in which there are no available regions for evacuation. There is no guarantee this will not still happen, but that's what tuning is for! This principle ensures that after every successful evacuation, all previously allocated Eden regions are returned to the free list and any evacuated live objects end up in Survivor space.

Below is an example of what a standard young collection would look like:



Continuing with this pattern, objects are again allocated into newly requested Eden regions. When Eden space fills up, another young collection occurs and, depending on the age (how many young collections the various objects have survived) of existing live objects, you will see promotion to Old regions. Given the Survivor space is part of the young generation, dead objects are collected or promoted during these young pauses.

Below is an example of what a young collection looks like when live objects from the Survivor space are evacuated and promoted to a new region in the Old space while live objects from Eden are evacuated into a new Survivor space region. Evacuated regions, denoted by the strikethrough, are now empty and returned to the free list.



G1 will continue with this pattern until one of three things happens:

1. It reaches a configurable soft-margin known as the InitiatingHeapOccupancyPercent (IHOP).
2. It reaches its configurable hard-margin (G1ReservePercent)
3. It encounters a humongous allocation (this is the exception I referred to earlier, more on this at the end).

Focusing on the primary trigger, the IHOP represents a point in time, as calculated during a young collection, where the number of objects in the old regions account for greater than 45% (default) of the total heap. This liveness ratio is constantly being calculated and evaluated as a component of each young collection. When one of these triggers are hit, a request is made to start a concurrent marking cycle.

```
8801.974: [G1Ergonomics (Concurrent Cycles) request concurrent c
8804.670: [G1Ergonomics (Concurrent Cycles) initiate concurrent
8805.612: [GC concurrent-mark-start]
8820.483: [GC concurrent-mark-end, 14.8711620 secs]
```

In G1, concurrent marking is based on the principle of snapshot-at-the-beginning (SATB). This means, for efficiency purposes, it can only identify objects as being garbage if they existed when the initial snapshot was taken. Any newly allocated objects that appear during the concurrent marking cycle are considered to be live irrespective of their true state. This is important because the longer it takes for concurrent marking to complete, the higher the ratio will be of what is collectible versus what is considered to be implicitly live. If you allocate more objects during concurrent marking than you end up collecting, you will eventually exhaust your heap. During the concurrent marking cycle, you will see young collections continue as it is not a stop-the-world event.

Below is an example of what a heap may look like after a young collection when the IHOP threshold is reached, triggering a concurrent mark.