Breaking Java Badly - Programmatic Destructive Testing of Java Virtual Machines (Part Two)

By
Mark Raley

In part one we employed a simple memory stress test (Producer-Consumer Test One) to demonstrate how alarming instabilities can be introduced into the JVM by naively sizing an internal queue. In part two, let's see what happens when we run the same test suite again in the same fashion and on the same hardware, but with G1 replacing the Parallel Old garbage collector.

Comparative Analysis - Test One

Figure one is a boxplot of the G1 and Parallel Old data sets interleaved by columns. All outliers have been removed and some data columns are omitted for clarity. At the top of each column, the difference of the respective medians as a percentage of the Parallel Old median is displayed and may be said to represent the upfront cost of switching garbage collection algorithms for this specific throughput load. While this load is not intended for a latency-oriented garbage collector, mixed loads are very common in many production environments.

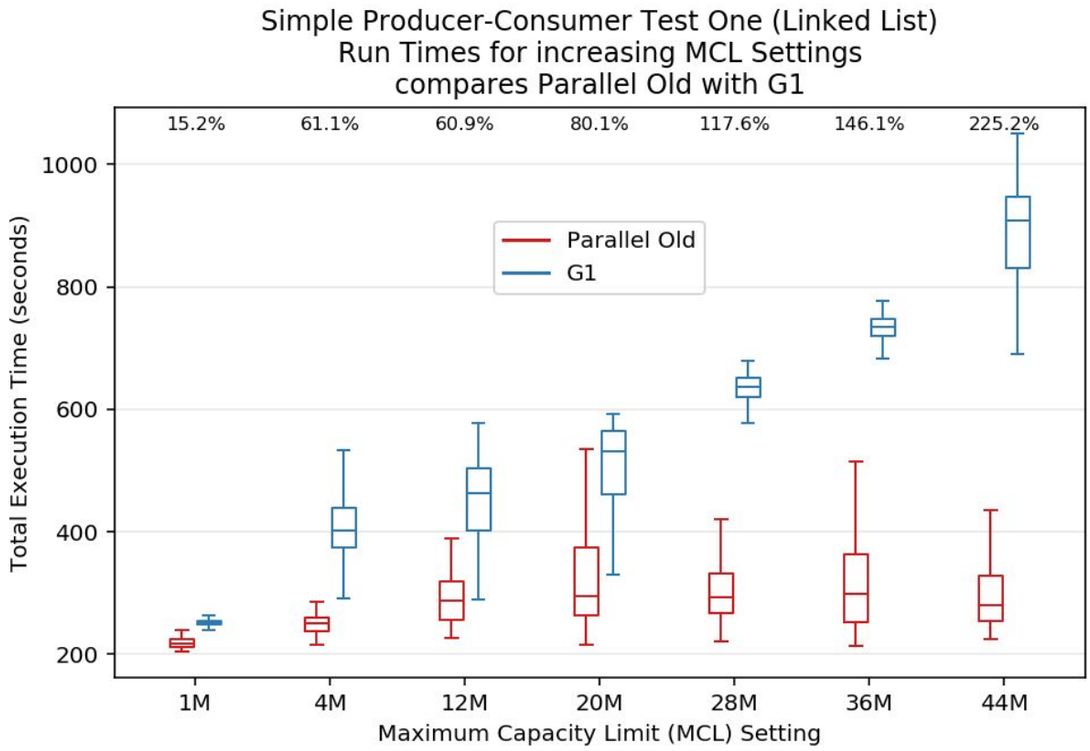

Figure One

In the "1M" column (far left), we see a modest performance regression of 15% which is not unexpected. G1 is designed to favor latency over throughput by spending additional CPU cycles to maximize overall responsiveness - this is principally achieved by reducing or eliminating full garbage collection pauses, a classic example of trading resources for performance. Full garbage collection cycles (FGCC) are especially damaging to latency because all JVM application threads are suspended during the operation for anywhere from a few hundred milliseconds to upwards of a minute. The latter can occur in overloaded situations where the JVM is fighting to stay alive.

Moving towards the right a curious rising stair-step pattern of steadily degrading performance is seen leading up to as much as a staggering 225% performance degradation of the median ("44M" column far right). This comparison then demonstrates how the nature of the contents of the heap, the inter-relationships, access patterns, and the choice of garbage collector itself can

have a dramatic effect on performance and stability. It suggests that by understanding the design choices of the past we become much better prepared to move into the future.

Full Analysis - G1 - Test One

Figure two shows the complete boxplot of G1 garbage collection data for test one. The median value (in seconds) is marked by a gold line as well as numerically at the top of the column. Lower is better.

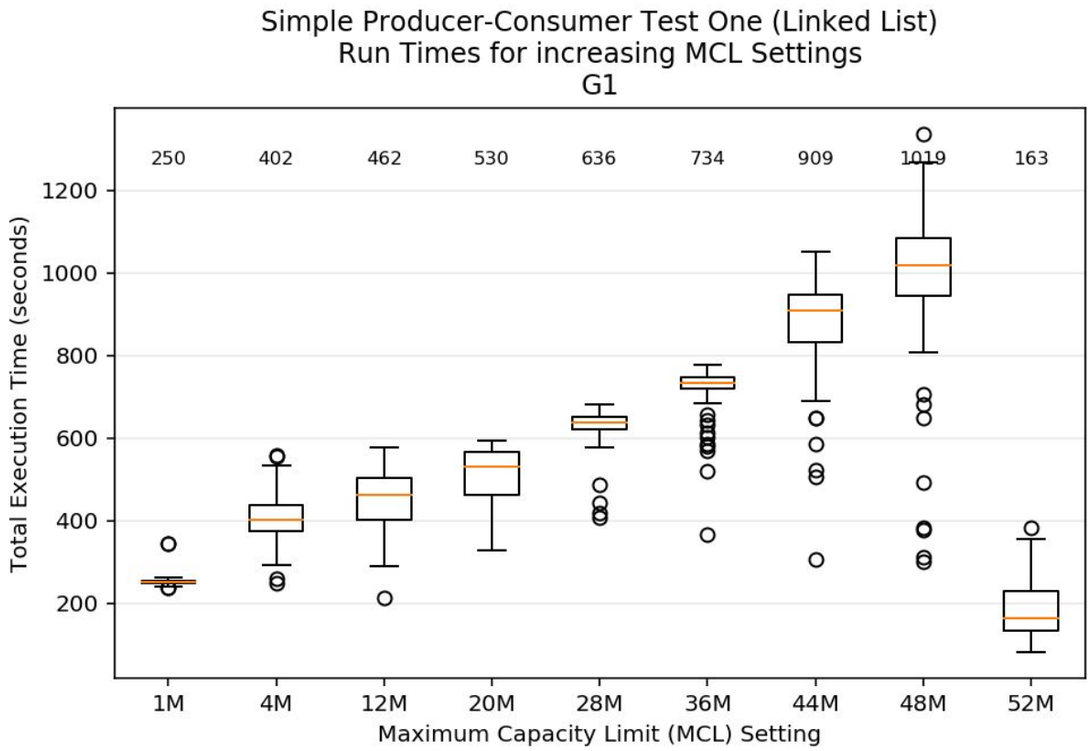

Figure Two

Some observations -

- At 52 million MCL (far right column) **all** runs terminate with an out-of-memory error from the JVM. Parallel Old, in contrast, begins to fail intermittently around 40 million MCL and above (see part one). This suggests that the G1 garbage collector is operating 15 to 25% more efficiently than Parallel Old in terms of tenured heap utilization.
- Almost all outliers are on the low side of execution times, as opposed to Parallel Old where the opposite is true - outliers are almost all on the high side of execution times. Is the G1 collector perhaps faster at memory allocation than retirement? One important consequence is that in cases where a load is unsuitable for this collector performance will degrade in an obvious manner over a small number of runs.
- The stair-step pattern referenced above is confirmed.

Even with Producer-Consumer Test One (which is a memory throughput stress test by design), the G1 collector demonstrates significant advantages. It is more memory efficient performing the same work with a smaller memory footprint. It also fails consistently when overburdened - both desirable features.

However, were a JVM with this kind of load to be switched from Parallel Old to G1 there would be the possibility of as much as a 225% performance degradation or a series of outright failures not long after server start. These results suggest memory management is inherently complex and design choices are best made with an eye towards how objects are interrelated at the data

structure level and for what scale of operation. Which collector is chosen can dramatically change JVM behavior in non-obvious ways.

Comparative Analysis - Array Deque - Test Two

Lastly let's replace the Linked List with an Array Deque, a data structure that is far more appropriate for this load. It does not have to maintain forward and backward references between elements yet can still use the same Queue interface for buffering integers. Internally, an array implementation is generally expected to use less memory overall and use it more efficiently over a linked list because it does not have to maintain constant insertion time over the whole sequence, only at the ends (one or both, in this case, both). Therefore, in theory, at least, there are much fewer internal memory references and the load on the collector selected may be much reduced.

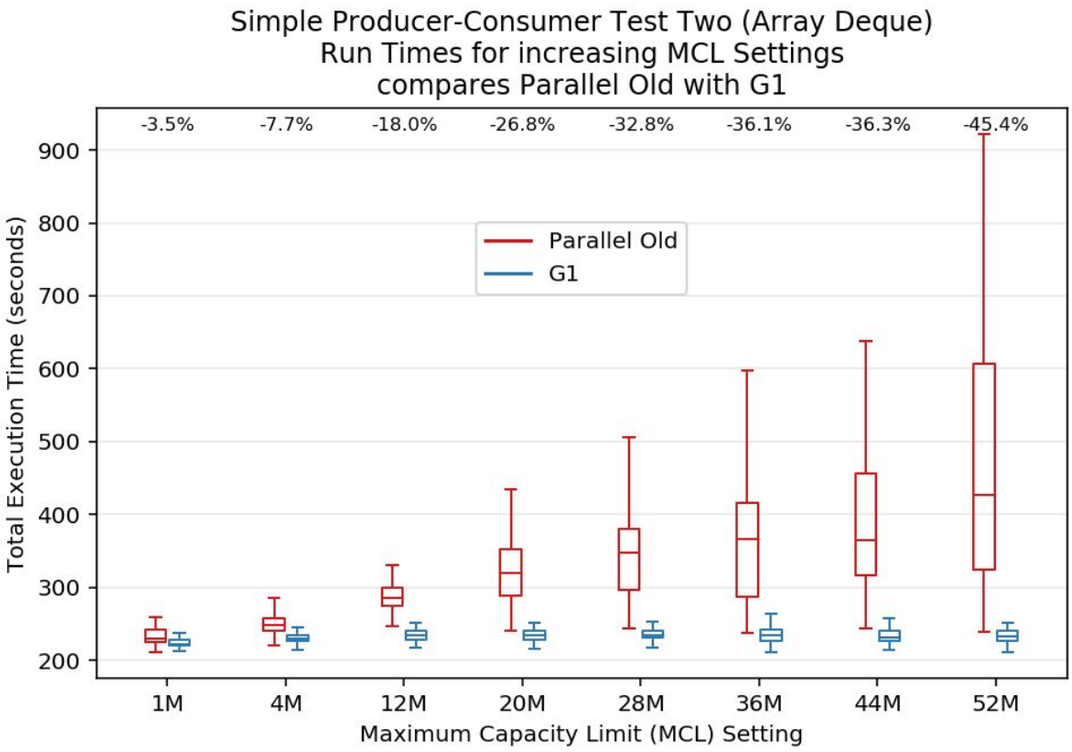

Figure Three

In figure three Parallel Old fares much better than with the Linked List approach, but still displays significant volatility in run times. Switching to G1 significantly improves overall performance in exemplary fashion, remaining stable over this range of MCL settings. This serves as a demonstration of how G1's approach to memory collection doesn't waste effort on inactive sections of the heap, and how the G1 collector may be of service even in throughput test load. By restricting heap changes to either the head or tail of the array, released memory will be clustered which is exactly what the G1 collector is designed to leverage. If we had to use Parallel Old to store long sequences of integers, we would do well to perhaps consider different approaches.

During test execution, we've also collected complete JVM logs with full GC details. In part three, we're going to drill down deep to get a better idea of what is consuming so much time in some of these test runs.