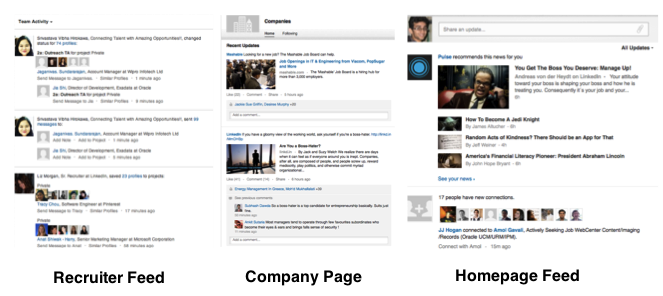
**Garbage Collection Optimization for High-Throughput and Low-Latency Java Applications**

High-performance applications form the backbone of the modern web. At LinkedIn, a number of internal high-throughput services cater to thousands of user requests per second. For optimal user experience, it is very important to serve these requests with low latency.

For example, a product our members use regularly is the Feed - a constantly updating list of professional activities and content. Examples of various feeds across LinkedIn include those in company pages, school pages, and most importantly - the homepage feed. The underlying feed data platform that indexes these updates from various entities in our economic graph (members, companies, groups, etc.) has to serve relevant updates with high throughput and low latency.

*Figure 1: LinkedIn Feeds*

For taking these types of high-throughput, low-latency Java applications to production, developers have to ensure consistent performance at every stage of the application development cycle. Determining optimal Garbage Collection (GC) settings is critical to achieve these metrics.

This blog post will walk through the steps to identify and optimize GC requirements, and is intended for a developer interested in a systematic method to tame GC to obtain high throughput and low latency. Insights in this post were gathered while building the next generation feed data platform at LinkedIn. These insights include, but are not limited to, CPU and memory overheads of [Concurrent Mark Sweep (CMS)](http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf) and [G1](http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/G1GettingStarted/index.html) collectors, avoiding incessant GC cycles due to long-lived objects, performance changes obtained by optimizing task assignment to GC threads, and OS settings that are needed to make GC pauses predictable.

## When is the right time to optimize GC?

GC behavior can vary with code-level optimizations and workloads. So it is important to tune GC on a codebase that is near completion and includes performance optimizations. But it is also necessary to perform preliminary analysis on an end-to-end basic prototype with stub computation code and synthetic workloads representative of production environments. This will help capture realistic bounds on latency and throughput based on the architecture and guide the decision on whether to scale up or scale out.

During the prototyping phase of our next generation feed data platform, we implemented almost all of the end-to-end functionality and replayed query workload being served by the current production infrastructure. This gave us enough diversity in the workload characteristics to measure application performance and GC characteristics over a long enough period of time.

## Steps to optimize GC

Here are some high level steps to optimize GC for high-throughput, low-latency requirements. Also included, are the details of how this was done for the feed data platform prototype. We saw the best GC performance with ParNew/CMS, though we also experimented with the G1 garbage collector.

### 1. Understand the basics of GC

Understanding how GC works is important because of the large number of variables that need to be tuned. Oracle's [whitepaper](http://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf) on Hotspot JVM Memory Management is an excellent starting point to become familiar with GC algorithms in Hotspot JVM. To understand the theoretical aspects of the G1 collector, check out this [paper](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.63.6386&rep=rep1&type=pdf).

### 2. Scope out GC requirements

There are certain characteristics of GC that you should optimize, to reduce its overhead on application performance. Like throughput and latency, these GC characteristics should be observed over a long-running test to ensure that the application can handle variance in traffic while going through multiple GC cycles.

* Stop-the-world collectors pause the application threads to collect garbage. The duration and frequency of these pauses should not adversely impact the application's ability to adhere to the SLA.
* Concurrent GC algorithms contend with the application threads for CPU cycles. This overhead should not affect the application throughput.
* Non-compacting GC algorithms can cause heap fragmentation, which leads to long stop-the-world pauses due to full GC. The heap fragmentation should be kept to a minimum.
* Garbage collection needs memory to work. Certain GC algorithms have a higher memory footprint than others. If the application needs a large heap, make sure the GC's memory overhead is not large.
* A clear understanding of GC logs and commonly used JVM parameters is necessary to easily tune GC behavior should the code complexity grow or workload characteristics change.

We used Hotspot [Java7u51](http://www.oracle.com/technetwork/java/javase/7u51-relnotes-2085002.html) on Linux OS and started the experiment with 32 GB heap, 6 GB young generation, and a -XX:CMSInitiatingOccupancyFraction value of 70 (percentage of the old generation that is filled when old GC is triggered). The large heap is required to maintain an object cache of long-lived objects. Once this cache is populated, the rate of object promotions to the old generation drops significantly.

With the initial GC configuration, we incurred a young GC pause of 80 ms once every three seconds and 99.9th percentile application latency of 100 ms. The GC behavior that we started with is likely adequate for many applications with less stringent latency SLAs. However, our goal was to decrease the 99.9th percentile application latency as much as possible. GC optimization was essential to achieve this goal.

### 3. Understand GC metrics

Measurement is always a prerequisite to optimization. Understanding the [verbose details of GC logs](http://engineering.linkedin.com/26/tuning-java-garbage-collection-web-services)(with these options: -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime) helps with the overall understanding of the application's GC characteristics.

At LinkedIn, our internal monitoring and reporting systems, [inGraphs](https://engineering.linkedin.com/32/eric-intern-origin-ingraphs) and [Naarad](https://github.com/linkedin/naarad), generate useful visualizations for metrics such as GC stall time percentiles, max duration of a stall, and GC frequency over long durations. In addition to Naarad, there are a number of open source tools like [gclogviewer](https://code.google.com/p/gclogviewer/) that can create visualizations from GC logs.

At this stage, it is possible to determine whether GC frequency and pause duration are impacting the application's ability to meet latency requirements.

### 4. Reduce GC frequency

In generational GC algorithms, collection frequency for a generation can be decreased by (i) reducing the object allocation/promotion rate and (ii) increasing the size of the generation.

In Hotspot JVM, the duration of young GC pause depends on the number of objects that survive a collection and not on the young generation size itself. The impact of increasing the young generation size on the application performance has to be carefully assessed:

* Increasing the young generation size may lead to longer young GC pauses if more data survives and gets copied in the survivor spaces, or if more data gets promoted to the old generation in each young collection. Longer GC pauses may lead to an increased application latency and/or reduced throughput.
* On the other hand, the pause duration may not increase if the number of objects that survive each collection does not increase substantially. In this case, the reduction in GC frequency may lead to a reduction in overall application latency and/or an increase in the throughput.

For applications that mostly create short-lived objects, you will only need to control the aforementioned parameters. For applications that create long-lived objects, there is a caveat; the promoted objects may not be collected in an old generation GC cycle for a long time. If the threshold at which old generation GC is triggered (expressed as a percentage of the old generation that is filled) is low, the application can get stuck in incessant GC cycles. You can avoid this by triggering GC at a higher threshold.

As our application maintains a large cache of long-lived objects in the heap, we increased the threshold of triggering old GC by setting: -XX:CMSInitiatingOccupancyFraction=92 -XX:+UseCMSInitiatingOccupancyOnly. We also tried to increase the young generation size to reduce the young collection frequency, but reverted this change as it increased the 99.9th percentile application latency.

### 5. Reduce GC pause duration

The young GC pause duration can be reduced by decreasing the young generation size as it may lead to less data being copied in survivor spaces or promoted per collection. However, as previously mentioned, we have to observe the impact of reduced young generation size and the resulting increase in GC frequency on the overall application throughput and latency. The young GC pause duration also depends on tenuring thresholds and the old generation size (as shown in step 6).

With CMS, try to minimize the heap fragmentation and the associated full GC pauses in the old generation collection. You can get this by controlling the object promotion rate and by reducing the -XX:CMSInitiatingOccupancyFraction value to trigger the old GC at a lower threshold. For a detailed understanding of all options that can be tweaked and their associated tradeoffs, check out [Tuning Java Garbage Collection for Web Services](http://engineering.linkedin.com/26/tuning-java-garbage-collection-web-services) and [Java Garbage Collection Distilled](http://mechanical-sympathy.blogspot.com/2013/07/java-garbage-collection-distilled.html).

We observed that much of our Eden space was evacuated in the young collection and very few objects died in the survivor spaces over the ages three to eight. So we reduced the tenuring threshold from 8 to 2 (with option: -XX:MaxTenuringThreshold=2), to reduce the amount of time spent in data copying in young generation collection.

We also noticed that the young collection pause duration increased as the old generation was filling up; this indicated that the object promotion was taking more time due to backpressure from the old generation. We addressed this problem by increasing the total heap size to 40 GB and reducing -XX:CMSInitiatingOccupancyFraction value to 80 to kick off old collection sooner. Though the -XX:CMSInitiatingOccupancyFraction value was reduced, increasing the heap size ensured that we avoided the incessant old GCs. At this stage, we observed a young collection pause of 70 ms and 99.9th percentile latency of 80 ms.

### 6. Optimize task assignment to GC worker threads

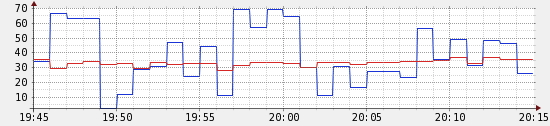
To reduce the young generation pause duration even further, we decided to look into options that optimized task binding with GC threads.

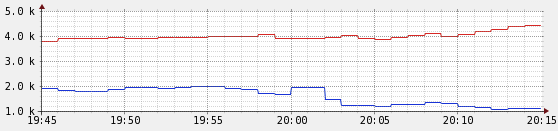
The -XX:ParGCCardsPerStrideChunk option controls the granularity of tasks given to GC worker threads and helps get the best performance out of a [patch](http://bugs.java.com/bugdatabase/view_bug.do?bug_id=7068625) written to [optimize card table scan time](http://blog.ragozin.info/2012/03/secret-hotspot-option-improving-gc.html)spent in young generation collection. The interesting learning here was that the young GC time increases with the increase in the size of old generation. Setting this option to a value of 32768 brought our young collection pause down to an average of 50 ms. We also achieved a 99.9th percentile application latency of 60 ms.

There are also a couple of other interesting options that deal with mapping tasks to GC threads. -XX:+BindGCTaskThreadsToCPUs option binds GC threads to individual CPU cores if the OS permits it.-XX:+UseGCTaskAffinity allocates tasks to GC worker threads using an affinity parameter. However, our application did not see any benefit from these options. In fact, some investigation revealed that these options are NOPs on Linux systems [1, 2].

### 7. Identify CPU and memory overhead of GC

Concurrent GC typically increases CPU usage. While we observed that the default settings of CMS behaved well, the increased CPU usage due to concurrent GC work with the G1 collector degraded our application's throughput and latency significantly. G1 also tends to impose a larger memory overhead on the application as compared to CMS. For low-throughput applications that are not CPU-bound, high CPU usage by GC may not be a pressing concern.

*Figure 2: CPU usage in % with ParNew/CMS and G1: the node with relatively variable CPU usage used G1 with the option-XX:G1RSetUpdatingPauseTimePercent=20*

*Figure 3: Requests served per second with ParNew/CMS and G1: the node with lower throughput used G1 with the option-XX:G1RSetUpdatingPauseTimePercent=20*

### 8. Optimize system memory and I/O management for GC

Occasionally, it is possible to see GC pauses with (i) low user time, high system time and high real (wallclock) time or (ii) low user time, low system time and high real time. This indicates problems with the underlying process/OS settings. Case (i) might imply that the JVM pages are being stolen by Linux and case (ii) might imply that the GC threads were recruited by Linux for disk flushes and were stuck in the kernel waiting for I/O. Refer to this [slide deck](http://www.slideshare.net/cuonghuutran/gc-andpagescanattacksbylinux) to check which settings might help in such cases.

We used the JVM option -XX:+AlwaysPreTouch to touch and zero out a page at application startup to avoid the performance penalty at runtime. We also set vm.swappiness to zero so that the OS does not swap pages unless it is absolutely necessary.

You could potentially use mlock to pin the JVM pages to memory so that the OS does not swap them out. However, if the system exhausts all its memory and the swap space, the OS will kill a process to reclaim memory. Typically, Linux kernel picks the process that has a high resident memory footprint but has not been running for long [(the workflow of killing a process in case of OOM)](https://www.kernel.org/doc/gorman/html/understand/understand016.html). In our case, this process will most likely be our application. Graceful degradation is one of the nicer properties of a service and the possibility of the service's sudden demise does not augur well for operability — so we don't use mlock and rely on vm.swappiness to avoid the swap penalty as long as it is possible.

## GC optimization for the feed data platform at LinkedIn

For the prototype feed data platform system, we tried to optimize garbage collection using two algorithms with the Hotspot JVM:

* ParNew for the young generation collection, CMS for the old generation collection.
* G1 for the young and old generations. G1 tries to solve the problem of having stable and predictable pause time below 0.5 seconds for heap sizes of 6GB or larger. In the course of our experiments with G1, despite tuning various parameters, we could not get the same GC performance or pause predictability as with ParNew/CMS. We also observed a bug [3] related to memory leak while using G1, but we are not sure of the root cause.

With ParNew/CMS, we saw 40-60 ms young generation pauses once every three seconds and a CMS cycle once every hour. We used the following JVM options:

|  |  |
| --- | --- |
| 12345678 | // JVM sizing options  -server -Xms40g -Xmx40g -XX:MaxDirectMemorySize=4096m -XX:PermSize=256m -XX:MaxPermSize=256m  // Young generation options  -XX:NewSize=6g -XX:MaxNewSize=6g -XX:+UseParNewGC -XX:MaxTenuringThreshold=2 -XX:SurvivorRatio=8 -XX:+UnlockDiagnosticVMOptions -XX:ParGCCardsPerStrideChunk=32768  // Old generation options  -XX:+UseConcMarkSweepGC -XX:CMSParallelRemarkEnabled -XX:+ParallelRefProcEnabled -XX:+CMSClassUnloadingEnabled -XX:CMSInitiatingOccupancyFraction=80 -XX:+UseCMSInitiatingOccupancyOnly  // Other options  -XX:+AlwaysPreTouch -XX:+PrintGCDetails -XX:+PrintGCTimeStamps -XX:+PrintGCDateStamps -XX:+PrintTenuringDistribution -XX:+PrintGCApplicationStoppedTime -XX:-OmitStackTraceInFastThrow |

[**view raw**](https://gist.github.com/sghike/9924734/raw/gc-opts)[**gc-opts**](https://gist.github.com/sghike/9924734#file-gc-opts) hosted with ❤ by **[GitHub](https://github.com/)**

With these options, our application's 99.9th percentile latency reduced to 60 ms while providing a throughput of thousands of read requests.

## Acknowledgements

Many hands contributed to the development of the prototype application: [Ankit Gupta](http://www.linkedin.com/in/guptaankit87" \t "_blank), [Elizabeth Bennett](http://www.linkedin.com/pub/elizabeth-bennett/23/335/812), [Raghu Hiremagalur](http://www.linkedin.com/pub/raghu-hiremagalur/2/166/2b5" \t "_blank), [Roshan Sumbaly](http://www.linkedin.com/in/rsumbaly" \t "_blank), [Swapnil Ghike](http://www.linkedin.com/in/swapnilghike/" \t "_blank), [Tom Chiang](http://www.linkedin.com/in/tomchiang), and [Vivek Nelamangala](http://www.linkedin.com/in/viveknelamangala" \t "_blank). Many thanks to [Cuong Tran](http://www.linkedin.com/pub/cuong-tran/0/66/306" \t "_blank), [David Hoa](http://www.linkedin.com/in/davidhoa), and [Steven Ihde](http://www.linkedin.com/in/stevenihde) for their help with system optimizations.

## Footnotes

[1] -XX:+BindGCTaskThreadsToCPUs seems to be a NOP on Linux systems because thedistribute\_processes method in hotspot/src/os/linux/vm/os\_linux.cpp is not implemented in JDK7 or JDK8.

[2] The -XX:+UseGCTaskAffinity option seems to be a NOP on all platforms in JDK7 and JDK8 as the task affinity appears to be always set to sentinel\_worker = (uint) -1. The source code is inhotspot/src/share/vm/gc\_implementation/parallelScavenge/{gcTaskManager.cpp, gcTaskThread.cpp, gcTaskManager.hpp}.