Garbage collection

By: Timo Jantunen

JVM and other platforms

In computer science, *garbage collection* (gc) can mean few different things depending on context and definition. In this post, it means "freeing allocated memory which is no longer needed by at least somewhat automated means." Since this text is mostly about Java and JVM (Java Virtual Machine), "object" and "memory area" are used interchangeably.

The Old Way

The original way of handling memory was not to allocate or free it at all - or rather allocate it during compile time and free it when the program exits. This way is still used in use today, usually in smaller realtime and critical systems. While this design can't leak memory it is very limiting and building larger programs with it means that you either need lots of memory or you need to share same memory areas between different functions which can't be run at the same time which can get complex rather quickly.

Manual labor

The other traditional way of allocating and freeing memory is by doing it manually (as opposed to automatic garbage collection). Every object allocation needs to be followed by freeing it once, only once, and only after its last usage. While this can be a very efficient method, it is also error prone, especially if the code is doing optional transformations to objects and/or has complex control flow (eg., using exceptions). To make it somewhat harder to make errors, most recent C++ standards (C++14) have five different types of pointers for managing this (old "dumb" ones and four "smart" ones). If you use the smart pointers correctly they will take care freeing the object once it can no longer be used.

If you forget to free an object, the program leaks memory (ie., it has allocated memory it can no longer use) and if this happens regularly, the program grows over time and will eventually run out of memory and crash.

However, if you continue using an object you have already freed and allocated for a different object, or (in most implementations) free some memory area twice (ie., two different allocations can get the same area), you end up with a memory area which is claimed by two different objects. When two objects share memory, it means that a change in one will affect other as well, which can lead to *interesting* debug sessions. Fortunately, there are some pretty good tools for detecting these kinds of problems (Valgrind is probably the best of those).

The easy way out

The easiest way to free unused objects automatically is to allocate them on stack - once the function returns, all stack-allocated objects are automatically freed.

Of course, this approach has many limitations, the most obvious being that you can't have objects whose lifetime is longer than the function itself. The second limitation is maximum stack size, which depends on operating system and/or threading implementation; while there are ways to extend the default limit on some platforms, on platforms with lots of threads and a 32-bit environment, you might start running out of memory quickly. Finally, while most threading implementations will allocate stack on demand (up to the limit), they usually can't free unused stack memory back to the system before the thread dies.

Bean counting

Maybe the simplest way of performing automatic freeing of objects is reference counting. It means when an object is created and a reference to is given out, the reference count is set as one. Every time the reference is copied somewhere, the counter is incremented by one and every time a reference is deleted or goes out of scope, it is decremented by one. When the counter reaches zero, the object is deleted. This is actually the same thing that the C++ smart pointer `std::shared_ptr` (and partly `std::weak_ptr`) will do. It is also used internally in many "lighter" programming languages like Perl 5 and CPython.

Automatic garbage collection removes the burden of freeing objects from the programmer and can significantly simplify many programs. On the other hand, reference counting can incur quite a large overhead on the program, especially if there are many short-lived objects. This is especially true for multithreaded programs which need to ensure that the counters are used atomically.

Simple reference counting alone can not free objects which (directly or indirectly) refer to themselves, since the reference count will never go to zero, but many implementations provide a way to overcome this in some way. For example, in C++, `std::weak_ptr` is a reference which doesn't increment the reference counter, so if you know you will be making cyclical references, you can still make sure the group of objects will get freed when all outside references to them are gone. There are also some special tricks in Perl and Python which will also try to get rid of self referencing objects, ie. Python will do a tracing garbage collection after certain amount of allocations have happened or when the program explicitly requests it.

Tracing suspected usage

Another automatic way of collecting unused objects is a tracing garbage collector. It will start collecting "root" references from static constants, stacks, and thread registers. Then the objects pointed to by those references are recursively searched for more references until all *reachable* objects have been found. Any object that wasn't visited is an *unreachable* object and since they can't be reached, they can be freed. Java, .NET and many JavaScript engines use this method.

While this is nice and elegant (can't leak, doesn't have counting overhead), it is not that easy. Any decently-sized program usually has at least tens of thousands of objects, -some can have millions-and going through each and every one of them whenever the program needs more memory is going to be very slow. Also, for the tracing to work efficiently you need to allocate more memory from the system than is strictly required so that you don't need to do the run before every allocation.

Fortunately, most objects die young: separating old and young objects from each other and into different memory areas (heaps) and trying to clean only young objects first will usually work really well. These kinds of collectors are known as generational collectors. Young objects usually get to the old heap by surviving the new heap gc enough times. For this purpose, the new heap is usually further divided into survivor/tenuring spaces where where the objects that survive the first gc are kept until they are either collected or deemed "old" enough to be moved to the old heap.

Garbage Collecting in Java

"don't"

The most efficient way of handling garbage collection in Java is to avoid doing it at all. For smaller command line programs (like a Maven compilation of a small project) it might be best just to give it enough heap space that it can run without doing a single round of garbage collection - it will be all freed once the program finishes.

Another way of reducing objects which need to be traced is by using stack allocation, which is much more lightweight. You can't control where Java allocates the object directly, but modern JVMs can do escape analysis on objects used in functions; if the analysis can prove that the object can't be used outside that function (escape), it can be allocated on stack. This analysis benefits from function inlining: for example, if a function uses a small builder method in another class, that method (if it is small enough) can be inlined to the calling class and then the created object is a candidate for stack allocation.

Using streams is often another good way to avoid putting potentially large objects in the heap. With streams, you can do several operations to a collection of objects one object at a time instead of doing one operation for whole collection at a time. (Of course, you could have always done that, streams just make this kind of thing much more easier and elegant.) So instead of having entire collections' worth of intermediate results, the JVM will only need to handle single intermediate result at a time. Since most lambda functions in streams are simple (can be inlined), it is also more probable that even those single values will be allocated on stack.

Allocating memory for JVM

`-Xms512m -Xmx2048m` is often enough in modern JVM (with numbers adjusted to application). Minimum memory should be about the steady state memory consumption for the application, though specifying it is no longer as important as it used to be. The maximum

amount of memory should be enough for the highest load with some margin to reduce garbage collection times.

In addition to heap for normal objects, JVM has another memory area called metaspace (older versions had permgen which was mostly the same thing.) Metaspace contains JVM "internal" data like class bytecode and other data which is usually pretty permanent and thus needs to be checked for garbage rarely. By default, metaspace can grow without limit, so it is good to limit it by something like `-XX:MaxMetaspaceSize=128m` switch instead of letting it slowly eat memory in case of leaks. And unfortunately, if you are redeploying J2EE programs, it is both rather common and annoyingly hard to debug: classes can't be garbage collected until the ClassLoader instance which created them is garbage collected and that can't be collected until all external references to it are gone. But meanwhile, there is this utility class in some random dependency which created a timer or background thread which functions as a gc root and you have no knowledge of it's existence… I haven't found any good tools for diagnosing these kind of problems so they remain annoyingly hard to debug.

JVM garbage collectors

Current Java 8 contains several different types of garbage collectors which are suitable for different scenarios. They are all generational collectors with at least old and new spaces and some further divide the new space to survivor/tenuring spaces.

Naming

Unfortunately different Java garbage collectors and tools use different names for same things and basically the same things so for clarity some definitions:

- Memory areas which store several same kind of objects are called spaces, generations, gens or heaps.
- A place where new objects are created can be called Eden, new gen, young generation and they also contain survivor spaces. Tenured objects are usually in survivor spaces.
- A place where older surviving objects are moved can be called old gen or old generation.

Ye olde gc

The oldest gc algorithms still supported are the serial and parallel collectors, which work with the stop-the-world principle (the program is stopped for the duration of the gc). They are defined separately for new/old generation heaps and can grow and shrink the total heap size (allocate memory from and release it back to the operating system.)

New objects collection is defined by using one of the following switches: `-XX:+UseSerialGC` (serial), `-XX:+UseParNewGC` (parallel) or `-XX:+UseParallelGC` (newer parallel collector). The serial one is the most efficient, but the parallel ones will work more quickly in a multicore machine. Out of the two parallel collectors, the newer one should be used specially in the case of larger heaps (gigabyte sized.) It will also benefit from `-XX:+UseAdaptiveSizePolicy` (enabled by default) switch which tries to tune young/old

heap size ratio according to the program. Old object collection is defined by -XX:+UseParallelOldGC switch (use -XX:-* to turn on the serial one.)

Some combination of the above switches is still the default for Java 8. Exactly which combination depends on the -client/-server switch, operating system, and CPU architecture and as a whole is very confusing - if you want to use these, it's better to use explicit switches.

CMS (Concurrent Mark Sweep)

When smallest pause times are critical, the CMS gc (-XX:+UseConcMarkSweepGC) is the most suitable one. It is a parallel collector which does most of the work while the program is actually running - there is a small pause when the collector starts (which marks all objects as "not reached") and one near the end (to re-check any objects changed during the tracing phase.) At the end, all tracing and cleaning of objects happens while the program is actually running and the typical pause times are usually less than 10 milliseconds, with peaks in the low tens of milliseconds. CMS only does old gen, while new gen is still handled by the parallel collectors (but new gen collections should be very fast regardless).

There are some limitations to CMS, though. The most severe is that unlike the other collectors, CMS can't do heap compaction (move objects to remove empty spaces between objects) and it is possible for it to run out of memory even when there plenty of space available - just not a big enough spot for the currently allocating object. It also means that the collector can't release heap memory back to the operating system.

Another limitation is that since the program is running at the same time, it is possible that it is allocating memory faster than the collector has time to free it. At least this can be somewhat addressed by increasing the maximum size of the heap so that the gc has more time to finish its collection, and increasing the new heap size to make sure that the short-lived objects don't escape to the old heap (where cleaning is much slower).

In both failure cases, the CMS will switch to emergency mode, which is basically the old serial collector. Since that is stop-the-world and can compact heap, it will solve both of the problems though the pause time can be (and usually is) several orders of magnitude larger (for larger heaps it is not rare to see it exceeding a second). It is usually possible to tune CMS parameters and/or the program to avoid these if it is really critical.

G1GC

G1 (garbage first) gc is the newest collector in the standard Java implementations. As its name says it tries to collect garbage first. It does this by splitting the heap into several smaller parts (by default to 2048) which can be individually assigned to being part of the old or new generation. "Large enough" objects will get their own part. G1 is partially stop-the-world but since it is collecting those small parts, its pause times are relatively quick. It is also multithreaded and utilizes some advanced techniques like reducing heap lock contention with thread local allocation buffers (TLAB), so it will scale well to multi-gigabyte heaps

When G1 is collecting garbage, it tries to start with heap spaces which had the most garbage and move surviving objects to other spaces - in the ideal case, the whole space is garbage and can be deallocated as a whole.

It is possible to request G1 to keep pause times under certain maximum limits (`-XX:MaxGCPauseMillis=XXX`) and it seems to perform well at least down to 100ms (again, much depends on your application and the machine you are running it on). G1 does contain plenty of knobs to tune its behavior, but it usually does good work on adapting itself to most loads.

Which GC to use

Which of the three(ish) gcs you should be using? The default answer to that is G1GC, which seems to perform well in most situations. Even though it is not optimal in all situations, it is still usually pretty close to it.

For calculating CPU-bound batch programs, you should be using serial/parallel collectors (serial if you only have a single core, parallel otherwise). If you have multiple cores, but your calculation can't fill them all, CMS or G1GC might perform better; they will use more CPU but do most of the work in separate threads so the calculating threads won't be stopped for that long of a time period.

If the gc pause times below 100ms are important, CMS is the best option. While it depends on the application and the machine running it, getting consistent pause times below 10ms is possible with CMS.

JVM gc tuning

Tuning garbage collection used to be difficult but important in many applications, since it could have significant performance effects. Today there are much better tools for examining gc and the algorithms are much better at tuning themselves, so in most cases you don't need to do more than specify reasonable maximum heap size.

VisualVM

A nice way of having quick look at heap is the VisualVM with Visual GC plugin (it can be installed inside the application from the Tools / Plugins menu). It will give you a nice overview of what's happening in different heaps. The display will change somewhat according to whatever garbage collection algorithm is in use, but you should see at least new (eden) space, old gen and metaspace / perm gen.

VisualVM can be downloaded from https://visualvm.java.net/download.html and it needs a graphical UI. If you need to run it remotely the best way is probably use Xvnc and ssh pipes, just make sure your vnc client doesn't get confused by localhost and try to use raw mode (no compression).

Logs

`-Xloggc:file -XX:+PrintGCDetails -XX:+PrintGCTimeStamps` should be a good "default" set of gc logging switches (there are dozens of further options as well). Those

options will log most important information and can be left on production servers. If your servers are expected to run for weeks, you'll probably want to add log rotation switches.

General rules

When tuning gc, you should know following facts:

1. Collecting new generation (eden) is fast - with most gc algorithms, discarding an object from eden is *free*. The surviving objects are copied to survivor or old spaces.
2. Collecting old generation is slow. There are usually many objects with many references to other objects and going through all of those will take time.
3. Most objects die young.

Tuning should be done separately for each different situation the program might be in. Below are three basic phases most server type programs have. Of course not all phases might be significant in practice - for example, startup phase should have no significance on a program which is expected to be running for weeks.

Phase: startup

The minimum heap size (set by `-Xms`*`NNN`*`m` switch) should be at least the normal steady state heap size since it will prevent the gc from doing unnecessarily old gen collections while trying to avoid allocating too much memory from the system. This used to have a large impact especially on programs which used to allocate many smaller long-lived objects during the startup phase, but it is no longer that significant.

Phase: steady state

When the program reaches a steady state, there should be no more old gen collections. While most objects should be short lived, there might be a class of objects which have a slightly longer lifetime but are still not "permanent" data (like some LRU caches). In these cases, the eden size and tenuring thresholds should be tuned so that these "middle aged" objets won't be able to escape to old gen.

For reducing steady state memory consumption, `-XX:+UseStringDeduplication` switch might be useful. It will deduplicate char arrays inside strings during idle times and especially in J2EE servers, it might reduce old gen size significantly.

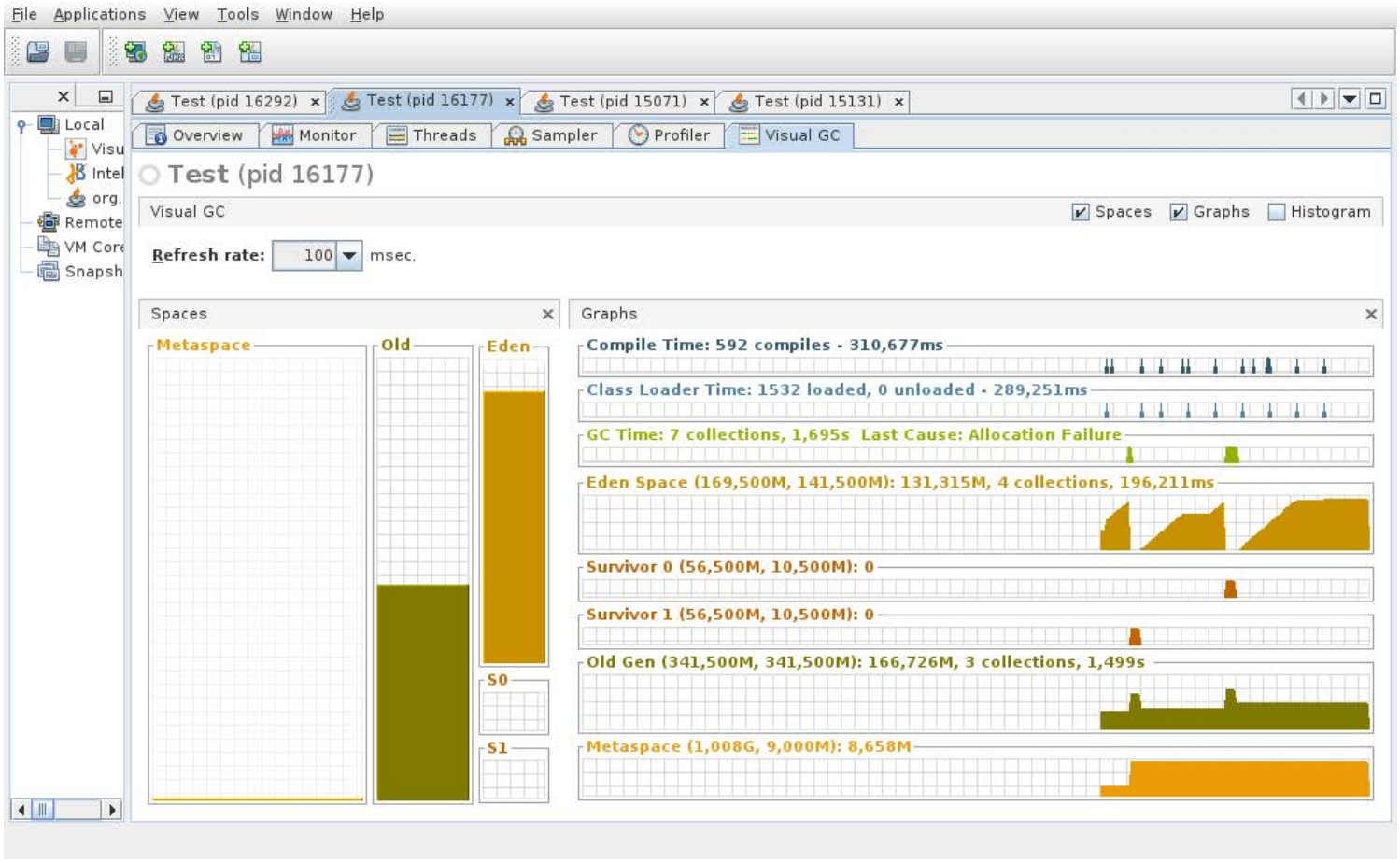
Phase: high load / special actions

When the program is under high load (like running some batch import or doing some daily calculation operation), most objects still shouldn't be able to escape from eden, so as in the steady state, the eden size and tenuring thresholds should be large enough. If the operation involves recalculating some semi-permanent cache or loading a new data set to memory, the old objects will obviously need to be collected (which requires old gen collection) but old gen collections should still be rare. From a garbage collection perspective, it is better to drop all old

data (like cleaning a whole cache) before starting to reload the new data as opposed to dropping and reloading the data one item at a time.

Pretty Pictures

Here are a few examples of examining gc state using some of the tools mentioned earlier, and a very simple test program which just allocates memory while deallocating only parts of previous allocations.



Visual GC with parallel garbage collector

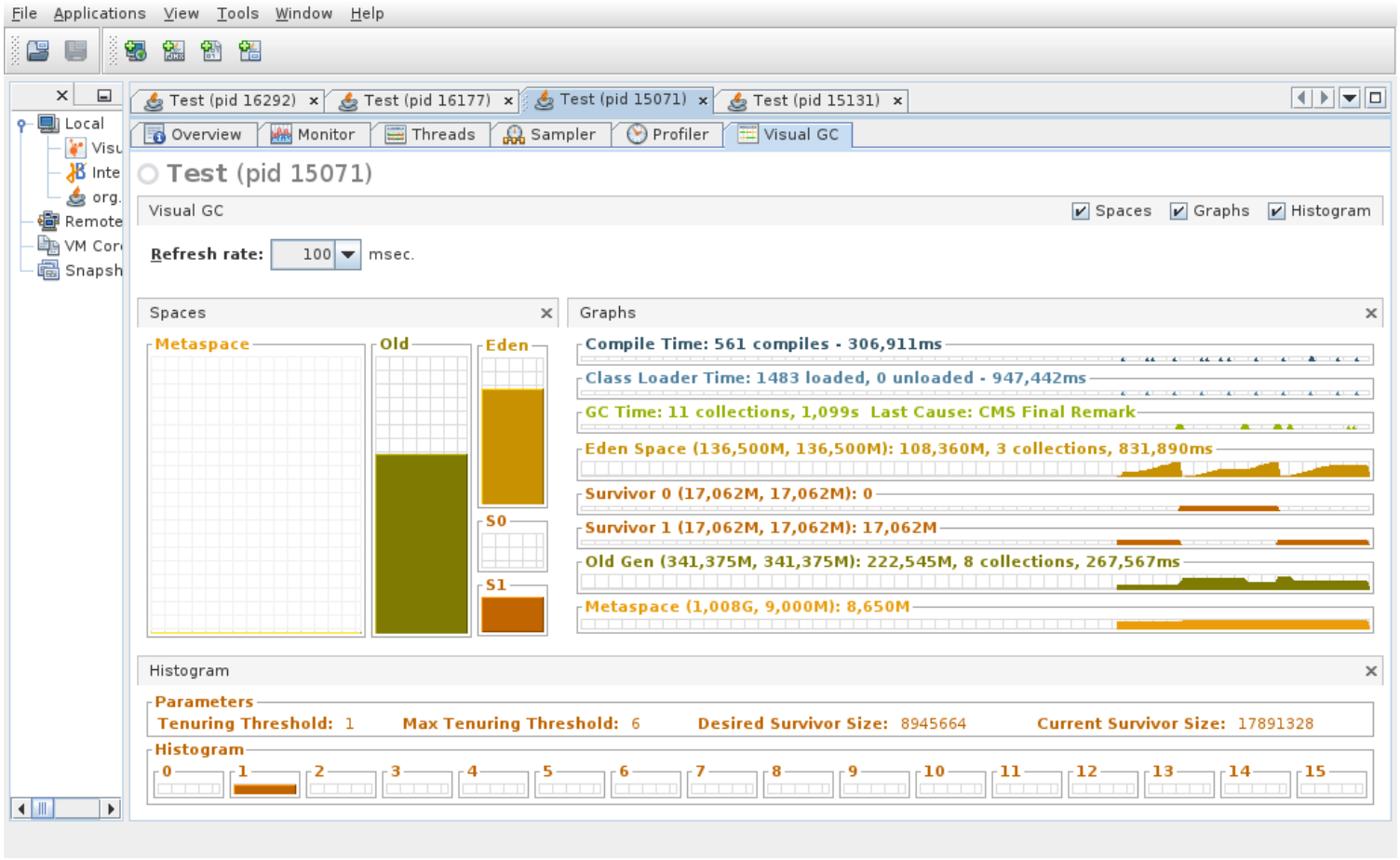
On the left side, we have bars representing current state of heap spaces and on the right side, historical graphs of the same (and some other data). From the graphs we can see some gc rounds (not all rounds, since some happened before VisualVM was attached to the process). The memory sizes in parentheses are maximum size and currently allocated (from system) size.

Here, we can see that after the eden gc, the surviving objects go to both survivor space and old gen. Then, old gen collection is triggered which both removes the survivor gen (probably by moving objects to old gen) and collects some of the old gen objects. All old gen

changes seem sharp not because it happens instantaneously, but because the collection is stop-the-world so the program can't even answer to VisualVM during while it is running.

From the gc times, you can also see why you should try to keep short-lived objects in the young generations: four eden space collections took 0.2 seconds while three old gen collections took 1.5 seconds, so old gen collections took *ten times* more time each. The used times are CPU times and since a parallel collector will use several threads, it doesn't mean this time is the same as the gc pause time (it would be for a serial collector, though).

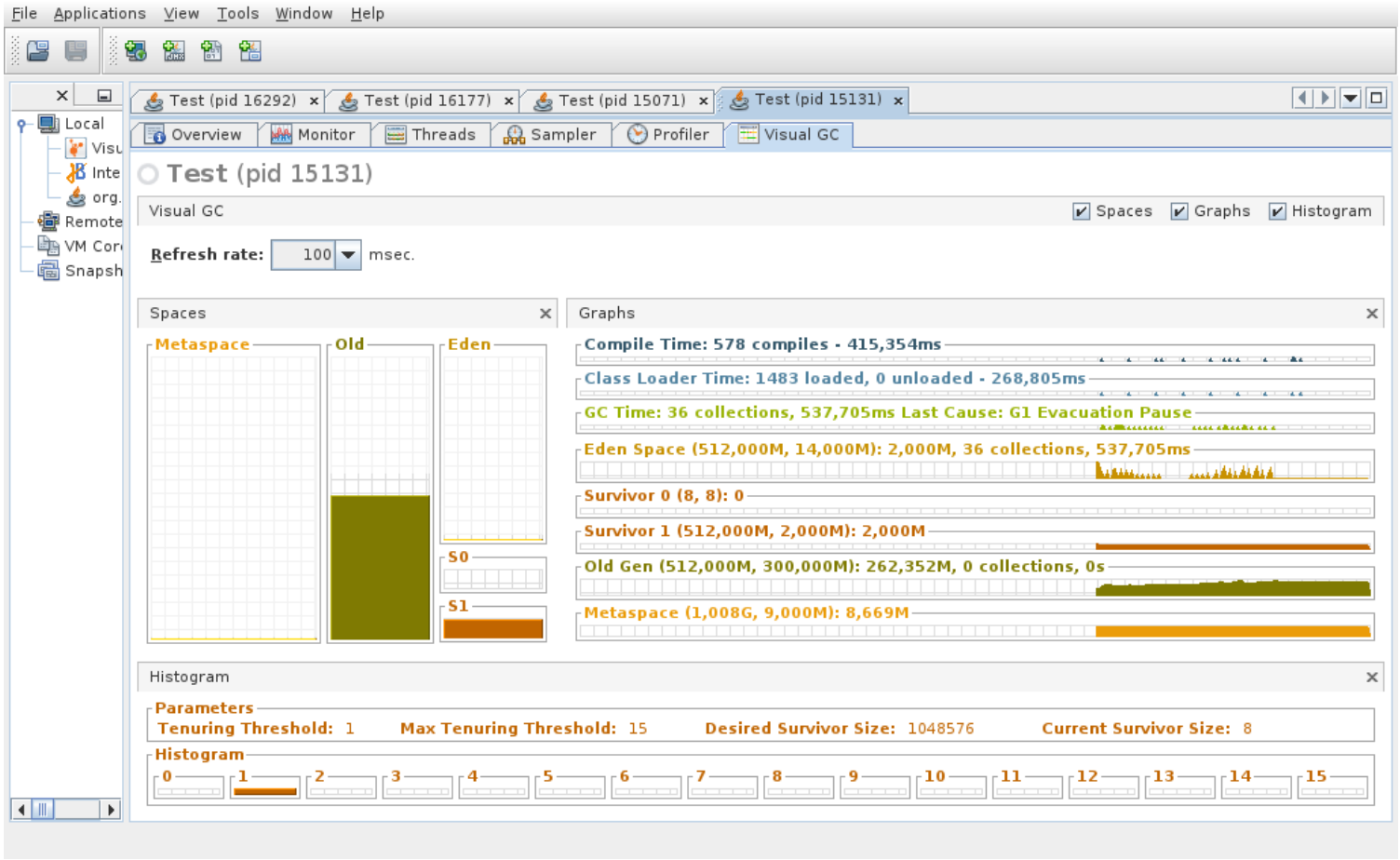
Visual GC with CMS garbage collector



The CMS collector includes tenuring objects which keeps survivor space objects there until they are either collected or survived up to the tenuring threshold and get moved to the old gen (the tenured objects live in survivor spaces and get one more count every time the eden is collected).

With CMS, we can see old gen is collected more asynchronously from eden and since it is mostly done in separate threads, the program will keep running while it is happening. Unlike with serial/parallel collectors, you can actually see slopes on old gen graphs since the objects are moved or freed while the program keeps running (at least, assuming you don't hit the emergency stop-the-world collection).

In this case, the gc times mean only the used CPU time and unfortunately there is no way to display the gc pause time in Visual GC.



Visual GC with G1 garbage collector

G1 largely resembles CMS, but you can see much more frequent collections and heap size adjustments (which come from handling the multiple smaller heaps separately). Also, if you compare G1 to two previous examples, you can see that it using the least resources even when all of them are running exactly the same program (least gc CPU used and system memory allocated).

Verbose GC logs

Not technically a picture, but:

```
OpenJDK 64-Bit Server VM (25.65-b01) for linux-amd64 JRE (1.8.0_65-b17), built on Nov
27 2015 13:48:00 by "mockbuild" with gcc 5.1.1 20150618 (Red Hat 5.1.1-4)
Memory: 4k page, physical 16116760k(8051284k free), swap 8069116k(8069116k free)
CommandLine flags: -XX:InitialHeapSize=268435456 -XX:MaxHeapSize=536870912 -
XX:MaxNewSize=178958336 -XX:MaxTenuringThreshold=6 -XX:NewSize=178958336 -
```

```
XX:OldPLABSize=16 -XX:OldSize=357912576 -XX:+PrintGC -XX:+PrintGCDetails -
XX:+PrintGCTimeStamps -XX:+UseCompressedClassPointers -XX:+UseCompressedOops -
XX:+UseConcMarkSweepGC -XX:+UseParNewGC
2.214: [GC (Allocation Failure) 2.214: [ParNew: 139776K->17472K(157248K), 0.4671903
secs] 139776K->145534K(285568K), 0.4672556 secs] [Times: user=2.32 sys=0.21, real=0.46
secs]
2.681: [GC (CMS Initial Mark) [1 CMS-initial-mark: 128062K(128320K)] 149644K(285568K),
0.0088577 secs] [Times: user=0.04 sys=0.00, real=0.01 secs]
2.690: [CMS-concurrent-mark-start]
2.816: [CMS-concurrent-mark: 0.126/0.126 secs] [Times: user=0.38 sys=0.00, real=0.13
secs]
2.816: [CMS-concurrent-preclean-start]
2.816: [CMS-concurrent-preclean: 0.000/0.000 secs] [Times: user=0.00 sys=0.00,
real=0.00 secs]
2.816: [CMS-concurrent-abortable-preclean-start]
5.049: [CMS-concurrent-abortable-preclean: 0.932/2.233 secs] [Times: user=3.19
sys=0.00, real=2.23 secs]
5.050: [GC (CMS Final Remark) [YG occupancy: 104126 K (157248 K)]5.050: [Rescan
(parallel) , 0.0212647 secs]5.071: [weak refs processing, 0.0000100 secs]5.071: [class
unloading, 0.0002042 secs]5.071: [scrub symbol table, 0.0006137 secs]5.072: [scrub
string table, 0.0002026 secs][1 CMS-remark: 128062K(128320K)] 232188K(285568K),
0.0223865 secs] [Times: user=0.15 sys=0.00, real=0.02 secs]
```

Not all “CommandLine flags” were specified by hand, the verbose log just prints related values even when they are the defaults. This example uses the CMS gc with 256MB initial and 512MB maximum heap size but otherwise default parameters.

Here we can see one new gen collection (starting at 2.214 seconds). The new gen heap goes from 137MB to 17MB. Those ???K->???K(???K) numbers you see are the initial, final and total sizes of the space in question. First is young gen, second is total heap.

CMS old gen scan starts from 2.671 and ends in 5.050. Only the initial mark (0.0088577 secs) and final remark (0.0212647 secs) phases are stop-the-world; everything else is happening while the program is still running.

tl;dr

Use `-XX:+UseG1GC` with big enough `-Xmx???m`.