Angelika Langer
www.AngelikaLanger.com

The Art of Garbage Collection Tuning

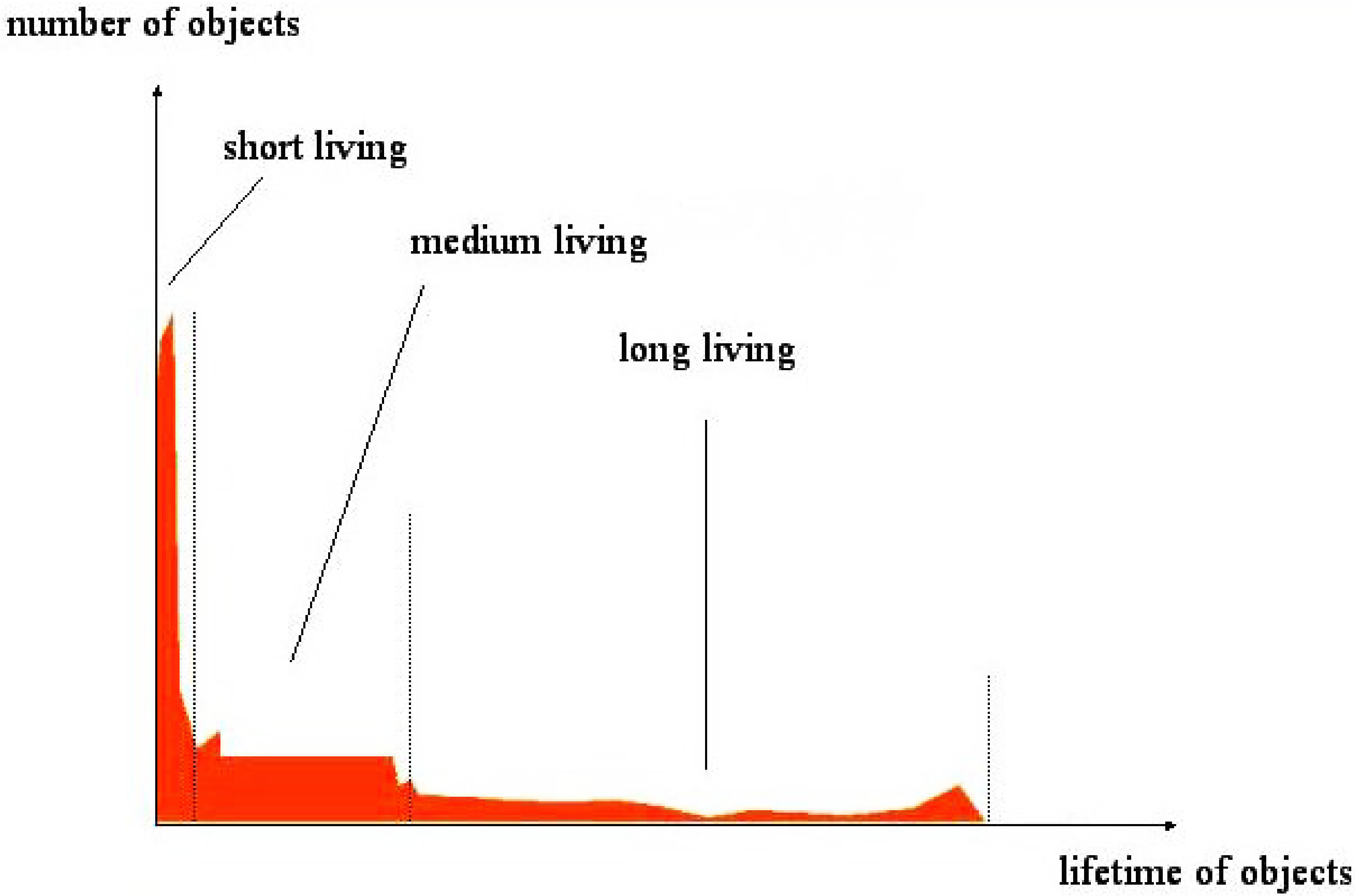
objective

- discuss garbage collection algorithms in Sun/Oracle's JVM
- give brief overview of GC tuning strategies

agenda

- **generational GC**
- parallel GC
- concurrent GC
- "garbage first" (G1)
- GC tuning

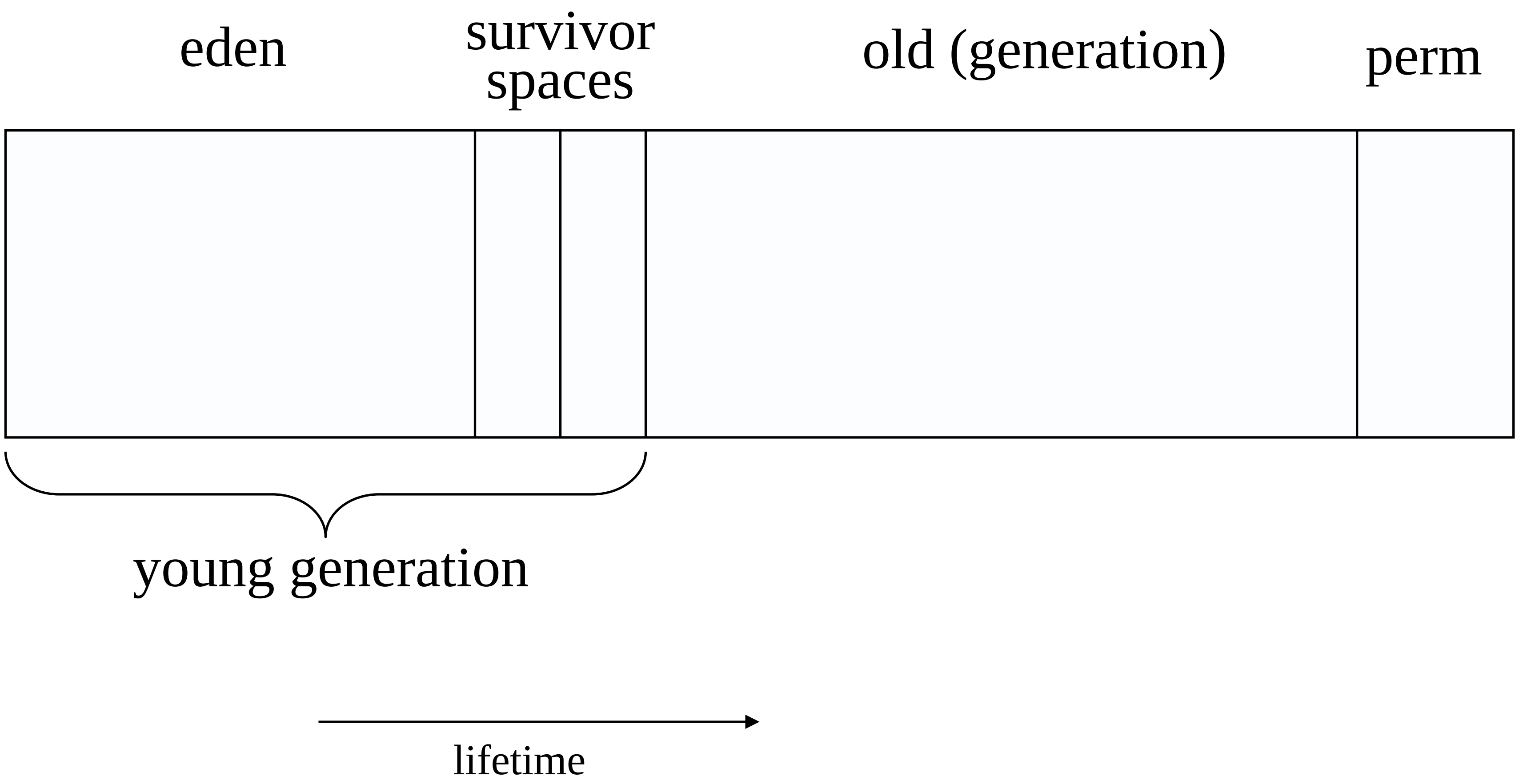
three typical object lifetime areas



generational GC

- key idea:
  - incorporate this typical object lifetime structure into GC architecture
- statically:
  - different heap areas for objects with different lifetime
- dynamically:
  - different GC algorithms for objects with different lifetime

static heap structure



different algorithms

- *mark-and-copy GC on young gen:*
  collect objects with a short and short-to-medium lifetime
  - fast algorithm, but requires more space
  - enables efficient allocation afterwards
  - frequent and short pauses (*minor GC*)

- *mark-and-compact GC on old gen:*
  collect objects which a medium-to-long and long lifetime
  - slow algorithm, but requires little space
  - avoids fragmentation and enables efficient allocation
  - rare and long pauses (*full GC*)

promotion

- aging and promotion
  – live objects are copied between survivor spaces and eventually to old generation

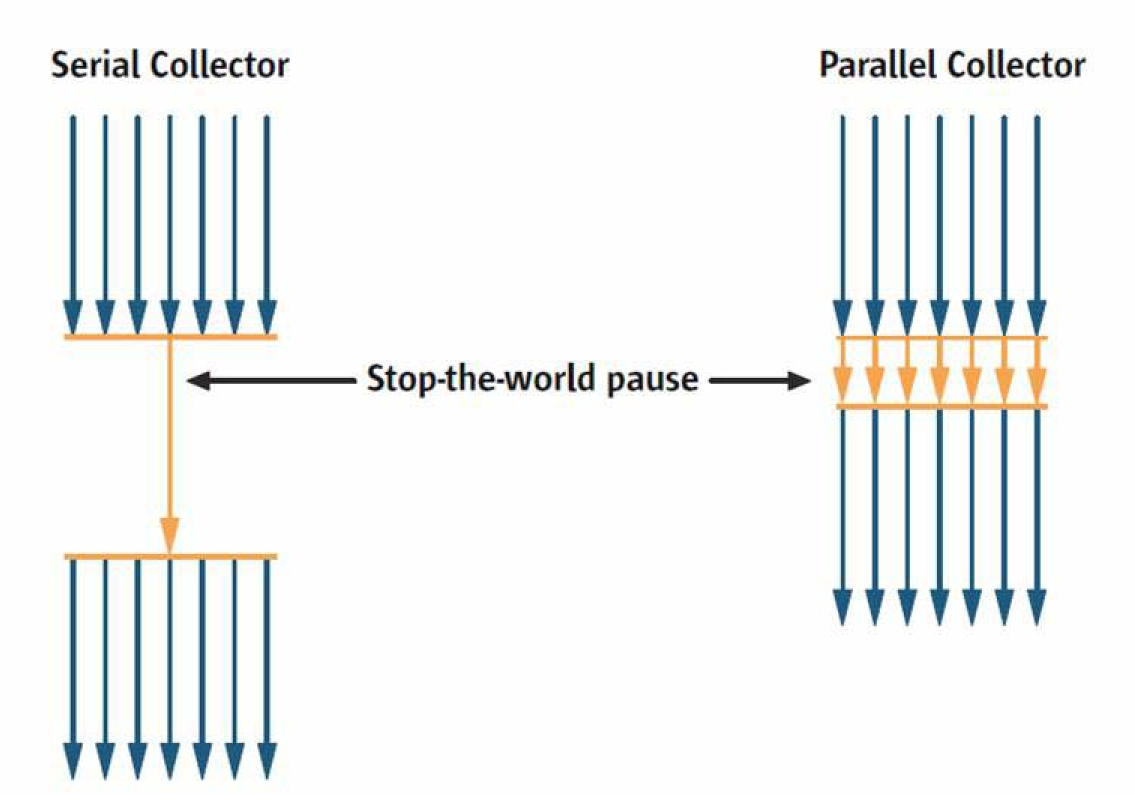
- how often live objects are copied between survivor spaces depends on ...
  – size of survivor space
  – number of live objects in eden and old survivor space
  – age threshold

agenda

- generational GC
- **parallel GC**
- concurrent GC
- "garbage first" (G1)
- GC tuning

multicore & multi-cpu architectures

- **parallel GC means:**
  - several GC threads + "stop-the-world"



parallel GC

- **parallel young GC** (since 1.4.1)
  - mark-sweep-copy
- **parallel old GC** (since 5.0_u6)
  - mark-sweep-compact
  - *mostly* parallel, i.e. has a serial phase

parallel young GC

- *mark phase (parallel)*
  - put all root pointers into a work queue
  - GC threads take tasks (i.e. root pointers) from work queue
  - GC threads put subsequent tasks (branches) into queue
  - *work stealing*: GC threads with empty queue "steal work" from another thread's queue (requires synchronization)
- *copy phase (parallel)*
  - challenge in parallel GC: many threads allocate objects in to-space
  - requires synchronization among GC threads
  - use *thread local allocation buffers* (GCLAB)

parallel old GC

- *marking phase (parallel)*
  - divide generation into fixed-sized regions => one GC thread per region
  - marks initial set of directly reachable live objects
  - keep information about size and location of live objects per region
- *summary phase (serial)*
  - determine *dense prefix*
    - point between densely and loosely populated part of generation
  - no objects are moved in dense prefix
  - loosely populated region is compacted
- *compaction phase (parallel)*
  - identify empty regions via summary data
  - parallel GC threads copy data into empty regions

agenda

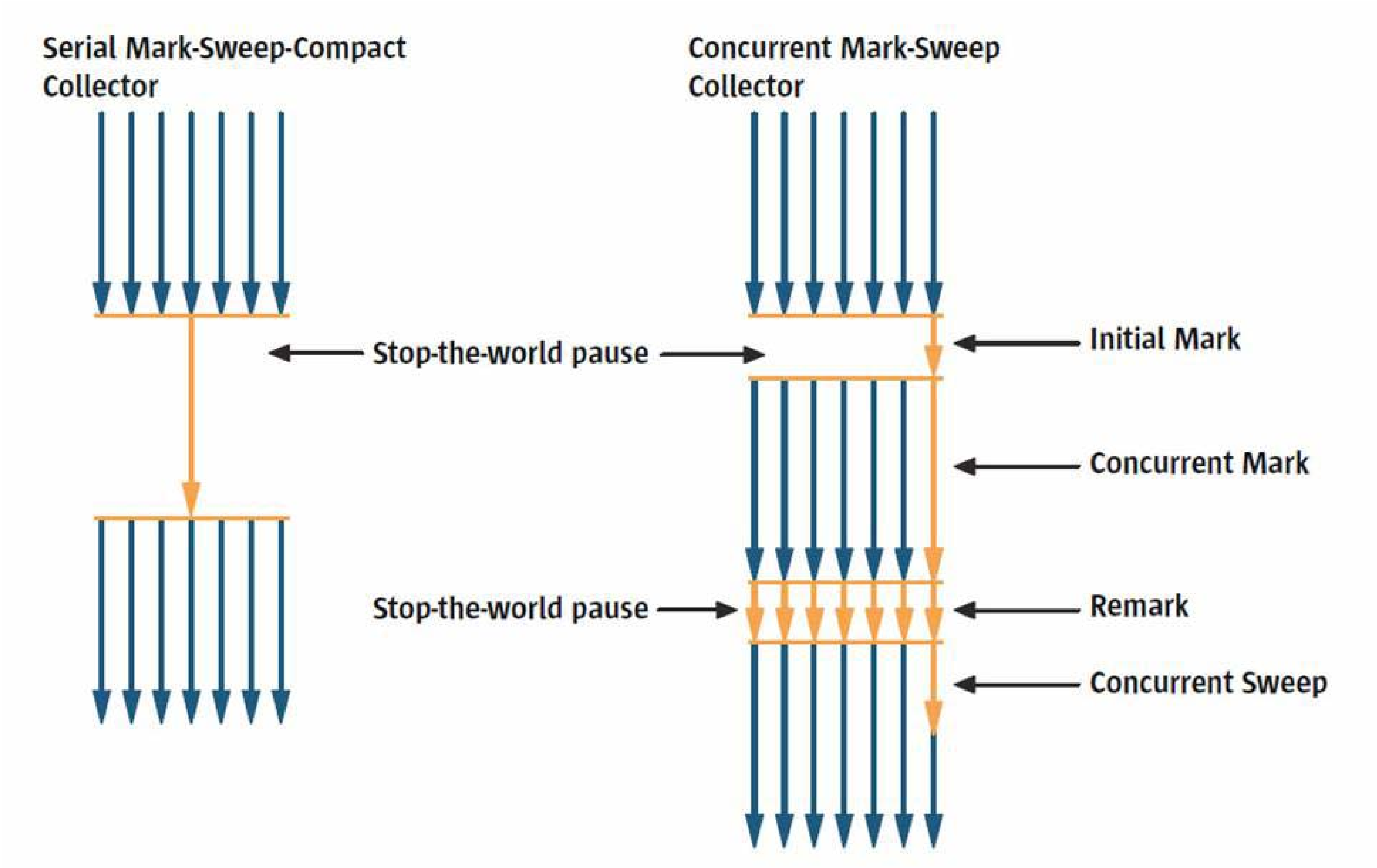
- generational GC
- parallel GC
- **concurrent GC**
- "garbage first" (G1)
- GC tuning

concurrent old generation GC

- **concurrent GC means:**
  - no "stop-the-world"
  - one or several GC threads run concurrently with application threads

- **concurrent old GC** (since 1.4.1)
  - **c**oncurrent **m**ark-and-**s**weep GC algorithm (*CMS*)
  - goal: shorter pauses
  - runs *mark-and-sweep* => no compaction
  - works *mostly concurrent*, i.e. has stop-the-world phases

serial vs. concurrent old gc



concurrent old GC - details

- several phases
  - initial marking phase (serial)
  - marking phase (concurrent)
  - preclean phase (concurrent)
  - remarking phase (parallel)
  - sweep phase (concurrent)

concurrent old GC - details

- *initial marking (serial)*
  - identifies initial set of live objects

- *marking phase (concurrent)*
  - scans live objects
  - application modifies reference graph during marking
    => *not all live objects are guaranteed to be marked*
  - record changes for remark phase via *write barriers*
  - multiple parallel GC threads (since 6.0)

concurrent old GC - details

- *preclean (concurrent)*
  - concurrently performs part of remarking work

- *remarking (serial)*
  - finalizes marking by revisiting objects modified during marking
  - some dead objects may be marked as alive
    => collected in next round (*floating garbage*)
  - multiple parallel GC threads (since 5.0)

- *sweep (concurrent)*
  - reclaim all dead objects
  - mark as alive all objects newly allocated by application
    - prevents them from getting swept out

CMS trace

- use `-verbose:gc` and `-XX:+PrintGCDetails` for details

non-concurrent mark	`[GC [1 CMS-initial-mark: 49149K(49152K)] 52595K(63936K), 0.0002292 secs]`
concurrent mark	`[CMS-concurrent-mark: 0.004/0.004 secs]`
concurrent preclean	`[CMS-concurrent-preclean: 0.004/0.004 secs]`
	`[CMS-concurrent-abortable-preclean: 0.000/0.000 secs]`
	`[GC[YG occupancy: 3445 K (14784 K)]`
	`[Rescan (parallel) , 0.0001846 secs]`
non-concurrent re-mark	`[weak refs processing, 0.0000026 secs]`
	`[1 CMS-remark: 49149K(49152K)] 52595K(63936K), 0.0071677 secs]`
concurrent sweep	`[CMS-concurrent-sweep: 0.002/0.002 secs]`
concurrent reset	`[CMS-concurrent-reset: 0.000/0.000 secs]`

no mark and compact ...

CMS does not compact

- compacting cannot be done concurrently

downsides:

- fragmentation
  - requires larger heap sizes

- expensive memory allocation
  - no contiguous free space to allocate from
  - must maintain *free lists* = links to unallocated memory regions of a certain size
  - adverse affect on young GC (allocation in old gen happens during promotion)

fall back to serial GC

- CMS might not be efficient enough
  - to prevent low-memory situations

- CMS falls back to serial mark-sweep-compact
  - causes unpredictable long stop-the-world pauses

fall back to serial GC

concurrent GC

serial GC

out of memory

```
[GC [1 CMS-initial-mark: 49149K(49152K)] 63642K(63936K), 0.0007233 secs]
    [CMS-concurrent-mark: 0.004/0.004 secs]
    [CMS-concurrent-preclean: 0.004/0.004 secs]
    [CMS-concurrent-abortable-preclean: 0.000/0.000 secs]
 [GC[YG occupancy: 14585 K (14784 K)]
    [Rescan (parallel) , 0.0050833 secs]
    [weak refs processing, 0.0000038 secs]
    [1 CMS-remark: 49149K(49152K)] 63735K(63936K), 0.0051317 secs]
    [CMS-concurrent-sweep: 0.002/0.002 secs]
    [CMS-concurrent-reset: 0.000/0.000 secs]

[Full GC [CMS: 49149K->49149K(49152K), 0.0093272 secs] 63932K->63932K(63936K),
         [CMS Perm : 1829K->1829K(12288K)], 0.0093618 secs]

[GC [1 CMS-initial-mark: 49149K(49152K)] 63933K(63936K), 0.0007206 secs]
java.lang.OutOfMemoryError
Heap
     par new generation   total 14784K, used 14784K
     eden space 13184K, 100% used
     from space 1600K, 100% used
     to   space 1600K,   0% used
     concurrent mark-sweep generation total 49152K, used 49150K
     concurrent-mark-sweep perm gen total 12288K, used 1834K
```

concurrent mark-and-sweep

- decreases old generation pauses

- at the expense of
  - slightly longer young generation pauses
  - some reduction in throughput
  - extra heap size requirements

agenda

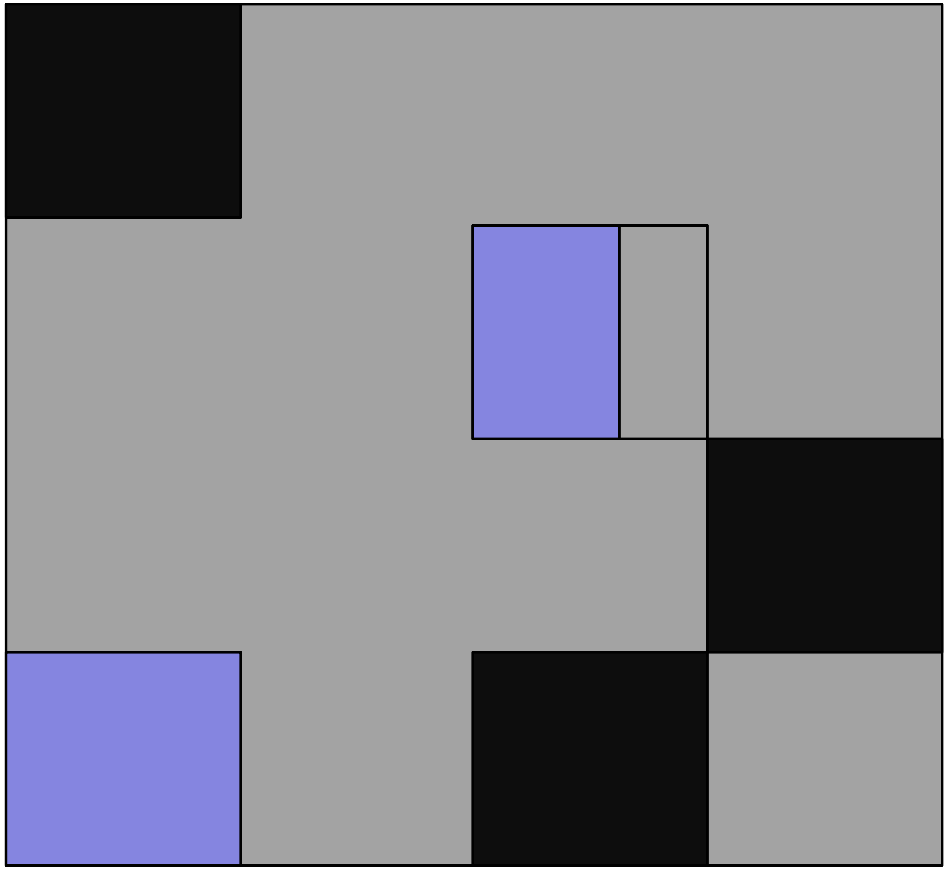
- generational GC
- parallel GC
- concurrent GC
- **"garbage first" (G1)**
- GC tuning

garbage-first (G1) garbage collector

- available since Java 6 update 14 (experimental)
- features:
  - compacting
    - no fragmentation
  - more predictable pause times
    - no fall back to serial GC
  - ease-of-use regarding tuning
    - self adjustment; barely any options

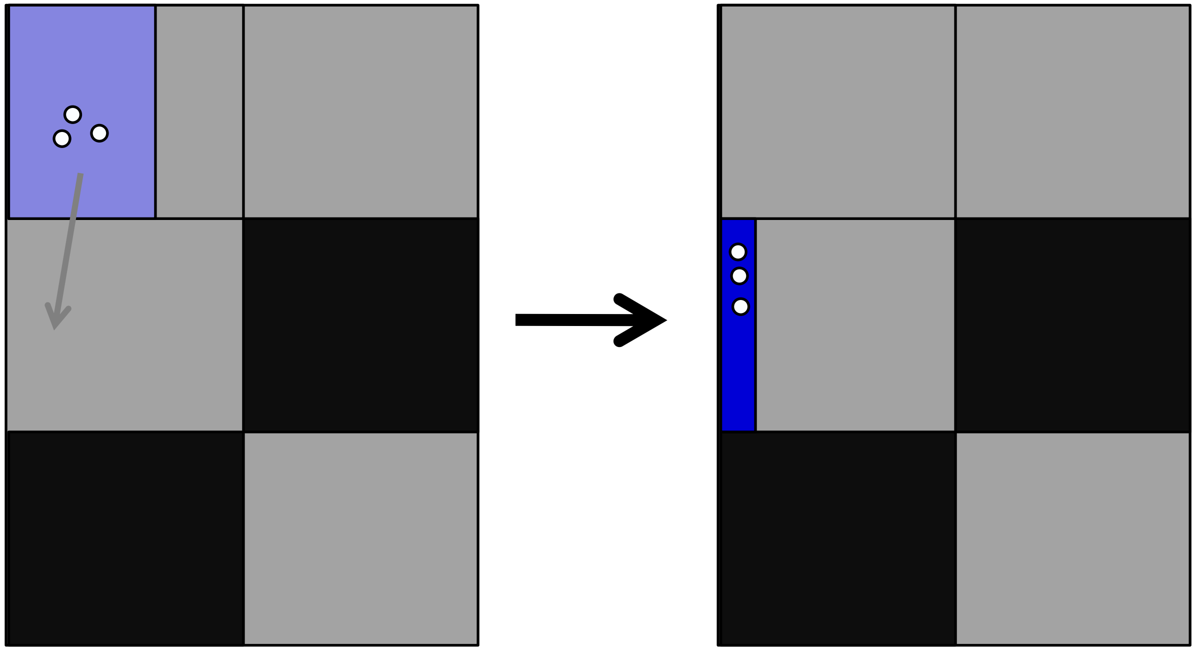
general approach

- heap split into regions (+ perm)
  - 1 MByte each
- young region
  + old region
  - dynamically arranged
  - non-contiguous



young regions: collection

- copy live objects from young regions to survivor region(s)



young collection (details)

- coping of live objects = *evacuation pause*
  - stop-the-world, i.e. no concurrent execution of application
    - no good !!!

- but: evacuation is parallel
  - performed by multiple GC threads
    - good !!!

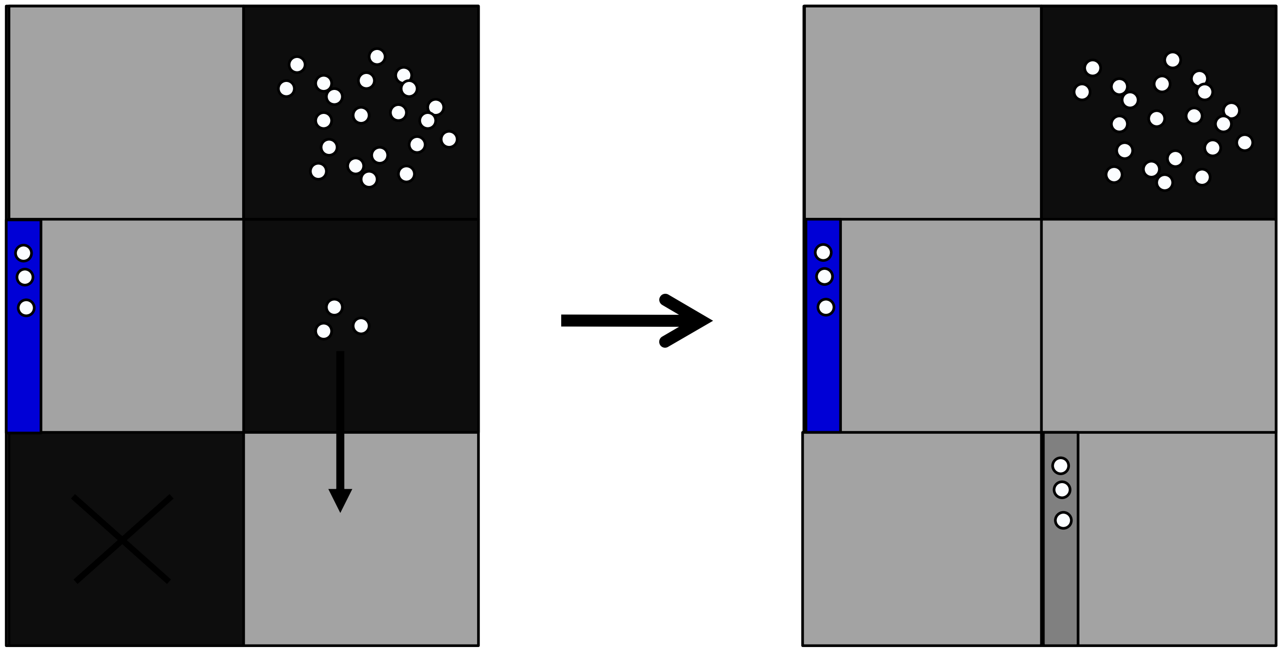
- parallel GC threads
  - GC operation is broken into independent tasks (work stealing):
    - determine live objects (marking stack)
    - copy live objects via GCLAB (similar to TLAB during allocation)

old regions: collection

- idea: collect regions with most garbage first
  - hence the name: "garbage-first"

- approach:
  - some regions may contain no live objects
    - very easy to collect, no coping at all
  - some regions may contain few live objects
    - live objects are copied (similar to young collection)
  - some regions may contain many live objects
    - regions not touched by GC

old regions: collection (cont.)



regions considered for GC evacuation

- *collection set* = regions considered for evacuation

- generational approach
  - sub-modes:
    - *fully young*: only young regions
    - *partially young*: young regions + old regions as pause time allows
    - GC switches mode dynamically

- which regions are put into the collection set ?
  - dynamically determined during program execution
  - based on a global marking that reflects a snapshot of the heap

note

- young and old regions have more similarities than before
- but still differences, i.e. it is generational GC
  - young regions:
    - where `new` objects are allocated
    - always evacuated
    - certain optimizations (e.g. no write barriers for remembered set update)
  - old regions:
    - only evacuated if time allows
    - only evacuated if full of garbage ("garbage-first")

benefits

- highly concurrent
  - most phases run concurrently with the application
    - some write barriers
    - some non-concurrent marking phases (similar to CMS)
  - even GC phases run concurrently
    - evacuation runs while global snapshot is marked
- highly parallel
  - multiple threads in almost all phases

benefits (cont.)

- fully self-adapting
  - just specify: max pause interval + max pause time
  - collection set is chosen to meet the goals
    - based on figures from various book keepings
    - e.g. previous evacuations, snapshot marking, write barriers

agenda

- generational GC
- parallel GC
- concurrent GC
- "garbage first" (G1)
- **GC tuning**

know your goals

- different applications require different GC behavior
  - no one-size-fits-all solution regarding GC and performance

- user aspects:
  - throughput
  - pauses

- engineering aspects:
  - footprint
  - scalability
  - promptness

profiling before you tune

- purpose
  - determine status quo
  - gather data for subsequent verification of successful tuning

- two sources
  - GC trace from JVM
  - profiling and monitoring tools

JVM options

`-verbose:GC`

`-XX:+PrintGCDetails`

- switch on GC trace
- details variry with different collectors

`-XX:+PrintGCApplicationConcurrentTime`

`-XX:+PrintGCApplicationStoppedTime`

- measure the amount of time the applications runs between collection pauses and the length of the collection pauses

```
Application time: 0.5291524 seconds
[GC [DefNew: 3968K->64K(4032K), 0.0460948 secs] 7451K->
6186K(32704K), 0.0462350 secs]
Total time for which application threads were stopped:
0.0468229 seconds
```

JVM options

`-XX:+PrintGCTimeStamps`

– enables calculation of total time, throughput, etc.

`–Xloggc:<filename>`

– redirect GC trace to output file

`-XX:+PrintTenuringDistribution`

– how often objects are copied between survivor spaces

`-XX:+PrintHeapAtGC`

– prints description of heap before and after GC

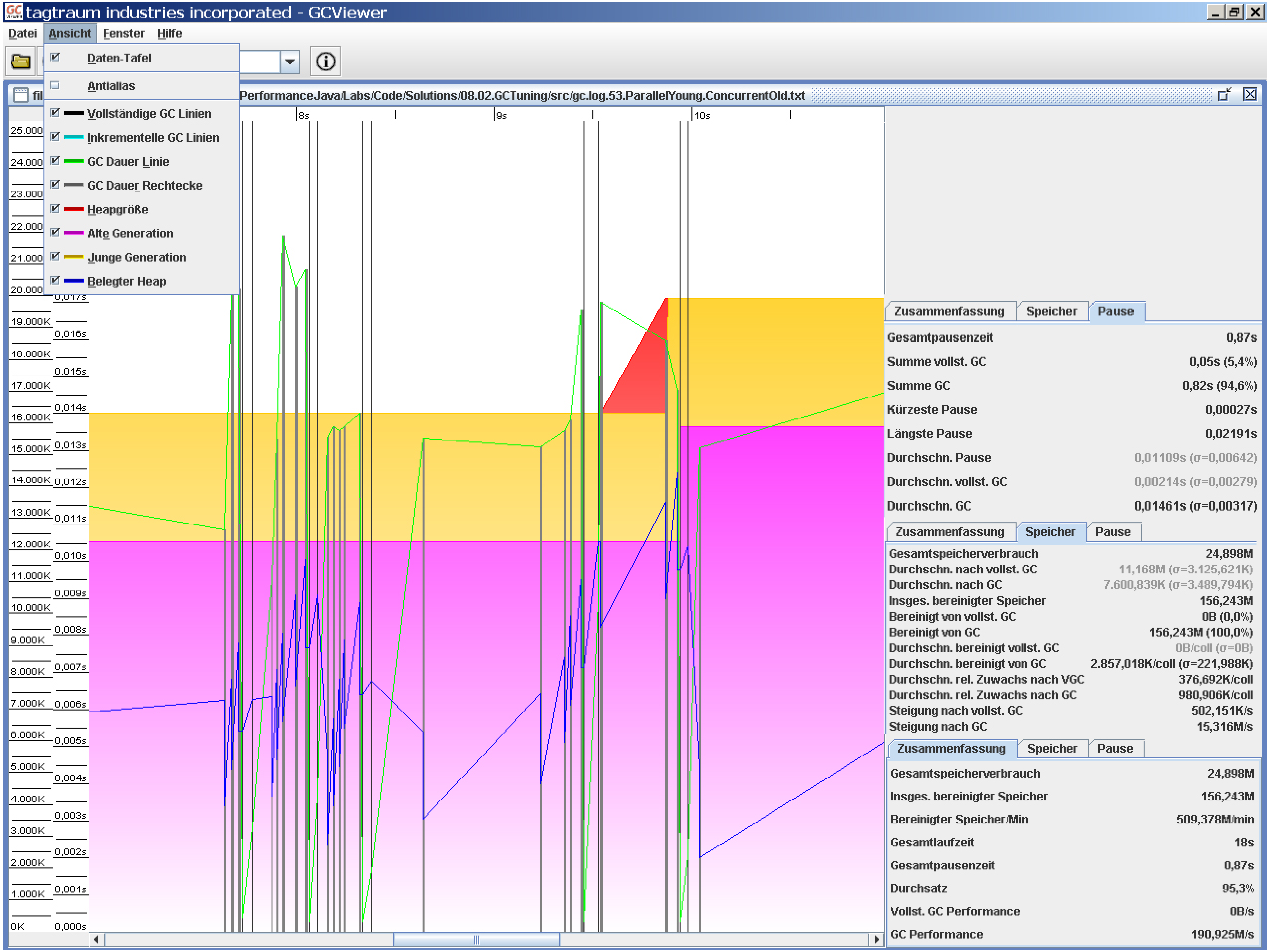
– produces massive amounts of output

heap snapshots

```
{Heap before GC invocations=1:
Heap
 def new generation   total 576K, used 561K [0x02ad0000, 0x02b70000, 0x02fb0000)
  eden space 512K,  97% used [0x02ad0000, 0x02b4c7e8, 0x02b50000)
  from space 64K, 100% used [0x02b60000, 0x02b70000, 0x02b70000)
  to   space 64K,   0% used [0x02b50000, 0x02b50000, 0x02b60000)
 tenured generation   total 1408K, used 172K [0x02fb0000, 0x03110000, 0x06ad0000)
   the space 1408K,  12% used [0x02fb0000, 0x02fdb370, 0x02fdb400, 0x03110000)
 compacting perm gen  total 8192K, used 2433K [0x06ad0000, 0x072d0000, 0x0aad0000)
   the space 8192K,  29% used [0x06ad0000, 0x06d305e8, 0x06d30600, 0x072d0000)
No shared spaces configured.
 Heap after GC invocations=2:
Heap
 def new generation   total 576K, used 20K [0x02ad0000, 0x02b70000, 0x02fb0000)
  eden space 512K,   0% used [0x02ad0000, 0x02ad0000, 0x02b50000)
  from space 64K,  31% used [0x02b50000, 0x02b55020, 0x02b60000)
  to   space 64K,   0% used [0x02b60000, 0x02b60000, 0x02b70000)
 tenured generation   total 1408K, used 236K [0x02fb0000, 0x03110000, 0x06ad0000)
   the space 1408K,  16% used [0x02fb0000, 0x02feb1b8, 0x02feb200, 0x03110000)
 compacting perm gen  total 8192K, used 2433K [0x06ad0000, 0x072d0000, 0x0aad0000)
   the space 8192K,  29% used [0x06ad0000, 0x06d305e8, 0x06d30600, 0x072d0000)
No shared spaces configured.
}
```

GC trace analyzer - GCViewer

- GCViewer
  - freeware GC trace analyzer
  - until 2008 by Hendrik Schreiber at `http://www.tagtraum.com/gcviewer.html`
  - until 2008 by Jörg Wüthrich at `https://github.com/chewiebug/GCViewer`
- reads JVM's GC log file
  - post-mortem or periodically
- produces diagrams and metrics
  - throughput
  - pauses
  - footprint

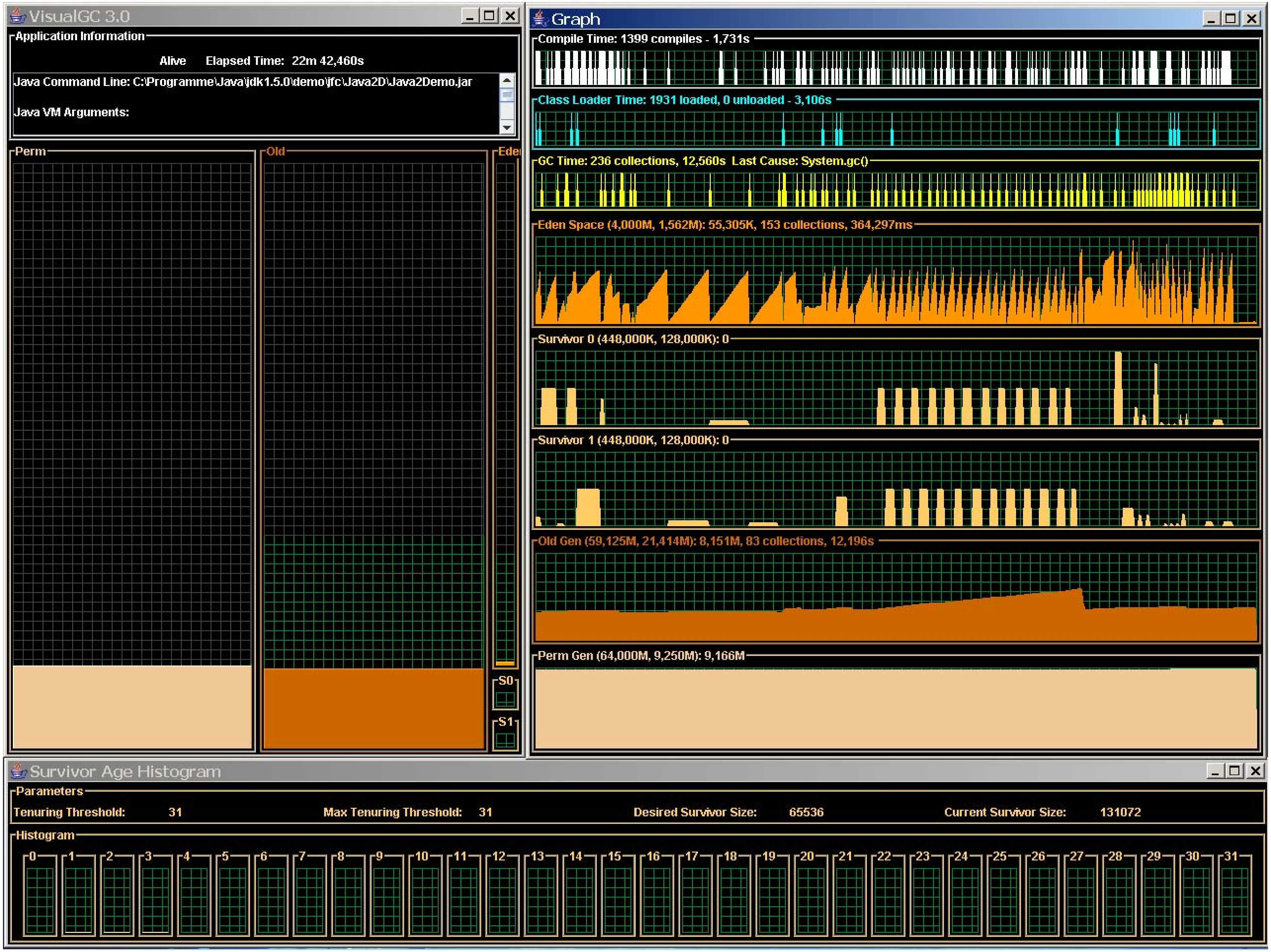


tagtraum industries incorporated - GCViewer
Datei
Ansicht
Fenster
Hilfe
Daten-Tafel
Antialias
Vollständige GC Linien
Inkrementelle GC Linien
GC Dauer Linie
GC Dauer Rechtecke
Heapgröße
Alte Generation
Junge Generation
Belegter Heap
PerformanceJava/Labs/Code/Solutions/08.02.GCTuning/src/gc.log.53.ParallelYoung.ConcurrentOld.txt
Zusammenfassung
Speicher
Pause
Gesamtpausenzeit 0,87s
Summe vollst. GC 0,05s (5,4%)
Summe GC 0,82s (94,6%)
Kürzeste Pause 0,00027s
Längste Pause 0,02191s
Durchschn. Pause 0,01109s (σ=0,00642)
Durchschn. vollst. GC 0,00214s (σ=0,00279)
Durchschn. GC 0,01461s (σ=0,00317)
Gesamtspeicherverbrauch 24,898M
Durchschn. nach vollst. GC 11,168M (σ=3.125,621K)
Durchschn. nach GC 7.600,839K (σ=3.489,794K)
Insges. bereinigter Speicher 156,243M
Bereinigt von vollst. GC 0B (0,0%)
Bereinigt von GC 156,243M (100,0%)
Durchschn. bereinigt vollst. GC 0B/coll (σ=0B)
Durchschn. bereinigt von GC 2.857,018K/coll (σ=221,988K)
Durchschn. rel. Zuwachs nach VGC 376,692K/coll
Durchschn. rel. Zuwachs nach GC 980,906K/coll
Steigung nach vollst. GC 502,151K/s
Steigung nach GC 15,316M/s
Gesamtspeicherverbrauch 24,898M
Insges. bereinigter Speicher 156,243M
Bereinigter Speicher/Min 509,378M/min
Gesamtlaufzeit 18s
Gesamtpausenzeit 0,87s
Durchsatz 95,3%
Vollst. GC Performance 0B/s
GC Performance 190,925M/s

JVM monitor - VisualGC

- VisualGC
  - experimental utility (since JDK 1.4)
  - dowload from `java.sun.com/performance/jvmstat/visualgc.html`
- integrated into VisualVM
  - download the VisualGC plugin (since JDK 6_u7)

- dynamically tracks and displays the heap
  - dynamic diagrams of all heap areas
  - no metrics at all


VisualGC 3.0
Application Information
Alive Elapsed Time: 22m 42,460s
Java Command Line: C:\Programme\Java\jdk1.5.0\demo\jfc\Java2D\Java2Demo.jar
Java VM Arguments:
Perm
Old
Ede
S0
S1
Graph
Compile Time: 1399 compiles - 1,731s
Class Loader Time: 1931 loaded, 0 unloaded - 3,106s
GC Time: 236 collections, 12,560s Last Cause: System.gc()
Eden Space (4,000M, 1,562M): 55,305K, 153 collections, 364,297ms
Survivor 0 (448,000K, 128,000K): 0
Survivor 1 (448,000K, 128,000K): 0
Old Gen (59,125M, 21,414M): 8,151M, 83 collections, 12,196s
Perm Gen (64,000M, 9,250M): 9,166M
Survivor Age Histogram
Parameters
Tenuring Threshold: 31
Max Tenuring Threshold: 31
Desired Survivor Size: 65536
Current Survivor Size: 131072
Histogram
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

tuning for maximum throughput

- strategy #1: increase heap size
  - reduced overall need for GC

- strategy #2: let objects die in young generation
  - GC in old generation is more expensive than in young generation
  - prevent promotion of medium lifetime objects into old generation

let objects die in young generation

- increase young generation size
  - only limited by need for old generation size
- keep objects in survivor space
  - increase survivors space
  - raise occupancy threshold
  - raise age threshold
  - pro: prevents promotion of medium lifetime objects
  - con: needlessly copies around long lifetime objects
- use parallel young GC
  - increases throughput, if >>2 CPUs available

tuning for minimal pause time

- use parallel GC (parallel young and parallel compact)
  - reduces pause time, if >>2 CPUs available

- use concurrent GC (CMS)
  - pro: mostly concurrent
  - con: fragmentation + more expensive young GC

- try out "G1"
  - designed to limit pause time and frequency

tuning CMS

- strategy: avoid stop-the-world pauses
  - reduce duration of full GC
  - avoid full GC altogether

prevent fallback to stop-the-world GC

- increase heap size
  - defers the problems ("night time GC")

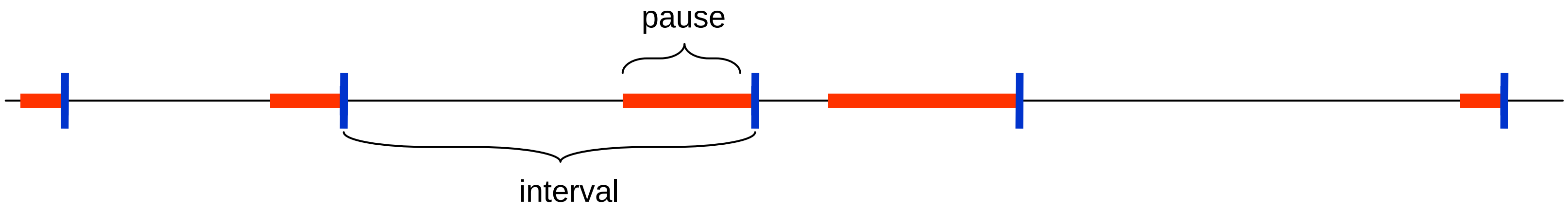
- start CMS early, i.e. lower occupancy threshold
  - reduces throughput because GC runs practically all the time

- increase young generation size
  - avoids fragmentation in the first place

tuning G1

- tuning G1 is different from classic GCs
  - generation sizes irrelevant
    - dynamically determined by G1 algorithms
  - still relevant: absolute memory size
    - grant as much memory as you can

- only 2 tuning parameters:
  - max pause + min interval



G1 tuning options

- `MaxGCPauseMillis`
  - upper limit for length of pause
  - what you demand from the GC
- `GCPauseIntervalMillis`
  - lower limit for length of interval in which GC pauses occur
  - how much GC activity you allow
  - short interval => many pauses in rapid succession

- defaults (might be too relaxed, for smaller apps)
  - `GCPauseIntervalMillis` = 500 ms
  - `MaxGCPauseMillis` = 200 ms

G1 tuning

- G1 "feels sluggish"
  - tuning goals are usually NOT met
- high variance compared to classic GCs
  - results differ even with identical tuning parameters
- G1 does not like overtuning
  - relaxed goal yields better results than ambitious goal

observations

- ambition is no good
  - raise pause time goal, i.e. demand shorter pause
  - (e.g. only 50 ms pause within 500 ms interval = 90% throughput)
  - result: G1 tries harder
    - make more pauses
    - often fails to reach the goal (pause time exceeds limit)

- relaxing is good
  - relax interval goal, i.e. allow more pauses
  - (e.g. 100 ms pause within 200 ms interval = only 50% throughput)
  - result: gives G1 more latitude and more flexibility
    - even pause times might decrease (without loss of throughput)
    - also avoids full GCs

wrap-up

- generational GC
  - split heap into generations
  - use different algorithms for each region

- young generation
  - mark-and copy (either serial or parallel)
    - many short stop-the-world pauses
    - needs survivor spaces

wrap-up

- old generation
  - mark-and-compact (either serial or parallel)
    - few gigantic stop-the-world pauses
    - no fragmentation
  - concurrent mark-and-sweep (CMS)
    - runs concurrently with the application
    - few short stop-the-world pauses (either serial or parallel)
    - falls back to mark-and-compact if needed
- "garbage first" (G1)
    - highly dynamic + very complex + hard to tune

wrap-up

- main tuning goals
  - throughput and pause times
- maximize throughput
  - let objects die in young generation
- minimize pauses times
  - avoid stop-the-world pauses

authors

Angelika Langer

Training & Consulting

Klaus Kreft

Performance Consultant, Germany

http://www.AngelikaLanger.com

Q & A