Effective Software

Lecture 13: Memory Management in JVM – Memory Layout, Garbage Collectors

David Šišlák
david.sislak@fel.cvut.cz

[1] Jones, R., Hosking, A., Moss, E.: The Garbage Collection Handbook. CRC Press, USA 2012.
[2] JVM source code - http://openjdk.java.net
[3] Oaks, S.: Java Performance: 2^{nd} Edition. O'Reilly, USA 2020.

Outline

- Memory management
  - Properties of automated memory management
  - Generational concept
  - Identify reachability of objects
  - Design architecture
- Parallel collector
  - Remember set
  - Parallel scavenge
  - Parallel mark compact
- Garbage first collector
  - Properties
  - Minor, mixed, full activities
  - Humongous objects

Automatic Memory Management

» **advantages** over explicit memory management
  - no crashes due to errors – e.g., usage of de-allocated objects
  - no memory leaks

» *components*
  - parts in **application code**
    - allocation
    - read/write references
  - **garbage collector**
    - discovers unreachable objects (not transiently reachable from **roots** – variables and stack operands in frames, static fields, special native references from JNI)
    - reclaims garbage

```
New():
    ref ← allocate()
    if ref = null
        collect()
        ref ← allocate()
        if ref = null
            error "Out of memory"
    return ref
```

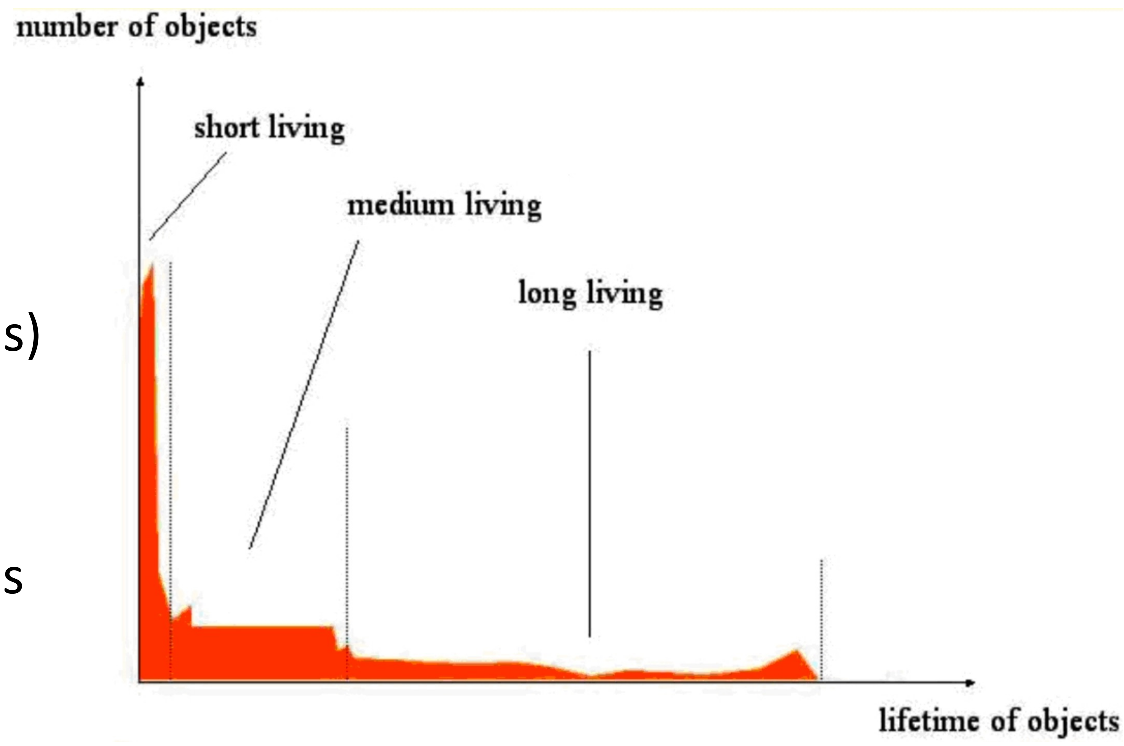
Automatic Memory Management

» <u>*memory management characteristics*</u>

- **safety** – never reclaim space of reachable objects, thread-safe
- **throughput** – application code performance
  - allocation performance – avoid fragmentation
  - *handles* or *direct references*
  - expensive *reference counting* or *cross-region reference tracking*
    - read/write barriers – e.g., added compiled code
  - later reads affected by re-ordering – breaking data locality, false sharing
- **completeness and promptness**
  - eventually all garbage
  - promptness of reclamation – how long garbage occupies memory
- **pause time** – stop the world (global safe point)
- **space overhead**
  - additional cost per capacity/reference
  - double heap for copying
- **scalability and portability -** multicore, large heaps

Generational Concept

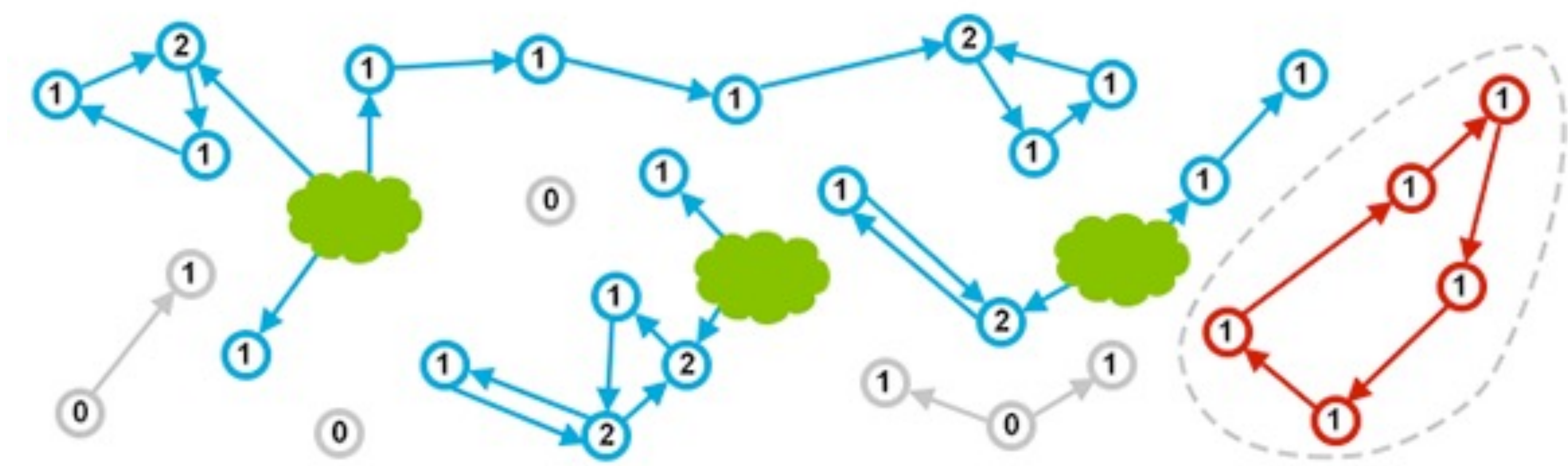
- **generational hypothesis**
  - **weak** – most objects die young
    - there exist few references from older to younger objects
  - **strong** – younger object dies earlier then older
- segregate objects by age into **generations** (JAVA use 2 generations) to **minimize pause time**
  - **young**
    - small size
    - frequent fast **minor** collections (milliseconds)
  - **old** (tenured)
    - large size
    - rare slow **full** collections (seconds)
- promotion of objects during minor collections



Identify Reachable Objects – Option 1

» **reference counting**

- additional counter for every object – number of references to the object
- a lot of atomic operations to have it thread-safe
  - slow down application code
- doesn't support cyclic references
- pollute cache a lot with additional memory operations
- can immediately remove objects when counter is 0 with further decreasing counts on reference objects



Identify Reachable Objects – Option 2

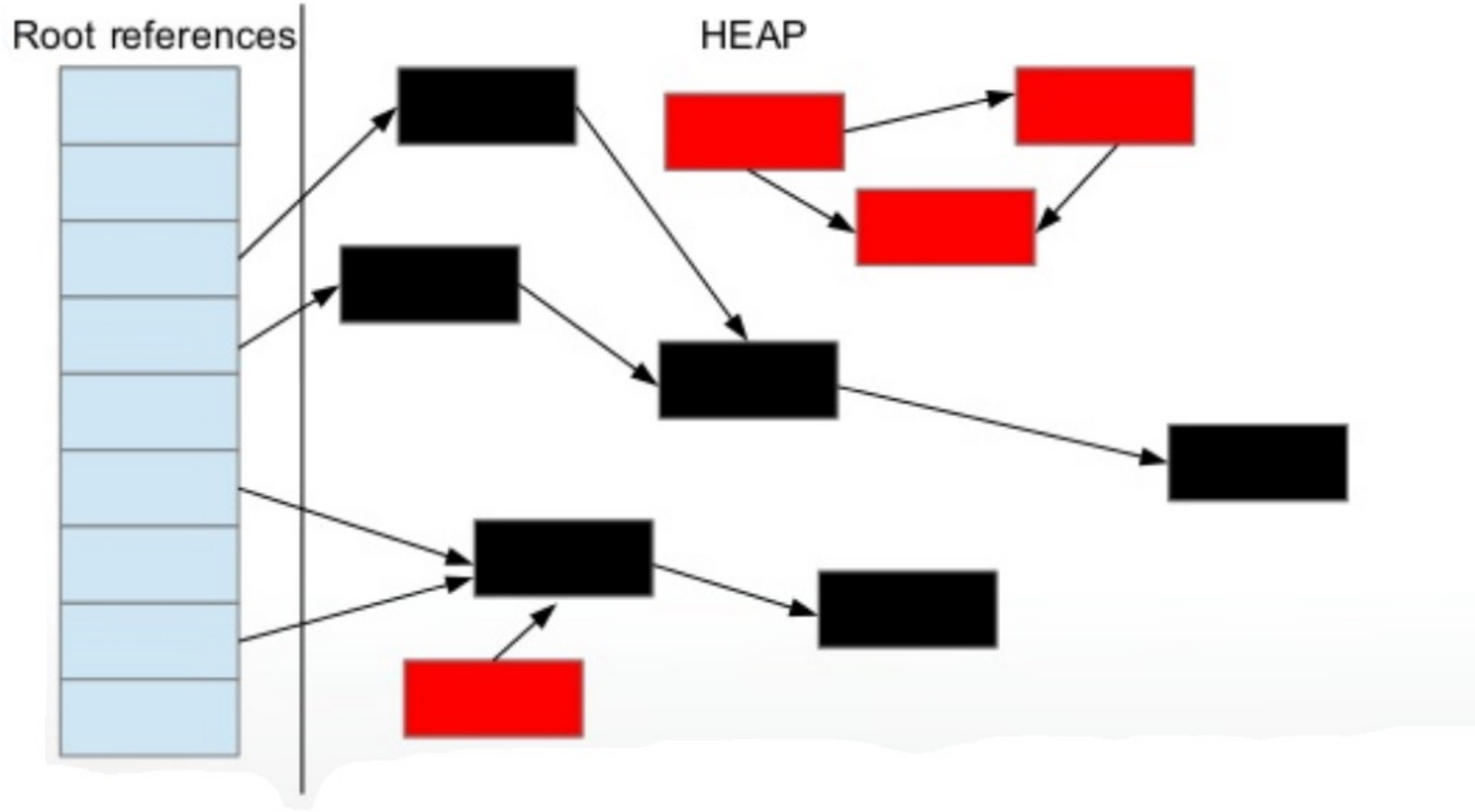
» **reference tracing approach**

- no direct slow down of application code
- **find** references
  - root in frames (stack, variables incl. parameters) using **OopMaps**
    - compiled maps for every possible *safepoint* entry

```
OopMap{rsi=Oop [48]=Oop rdx=Oop [72]=Oop off=1734}
```

  - in a different object using object type
    - reference positions in internal Klass VM structure
- **marking phase** traverse all objects from **roots**
  - depth-first search, breath-first search
  - dominates collection time due to random access to memory
    - cache prefetching to reduce cost
- use **mark flags** to avoid cycles
  - in object header – standard writes with possible partial re-traversal
  - side bitmaps (1 bit for 64 bits) – improving cache operations, atomics

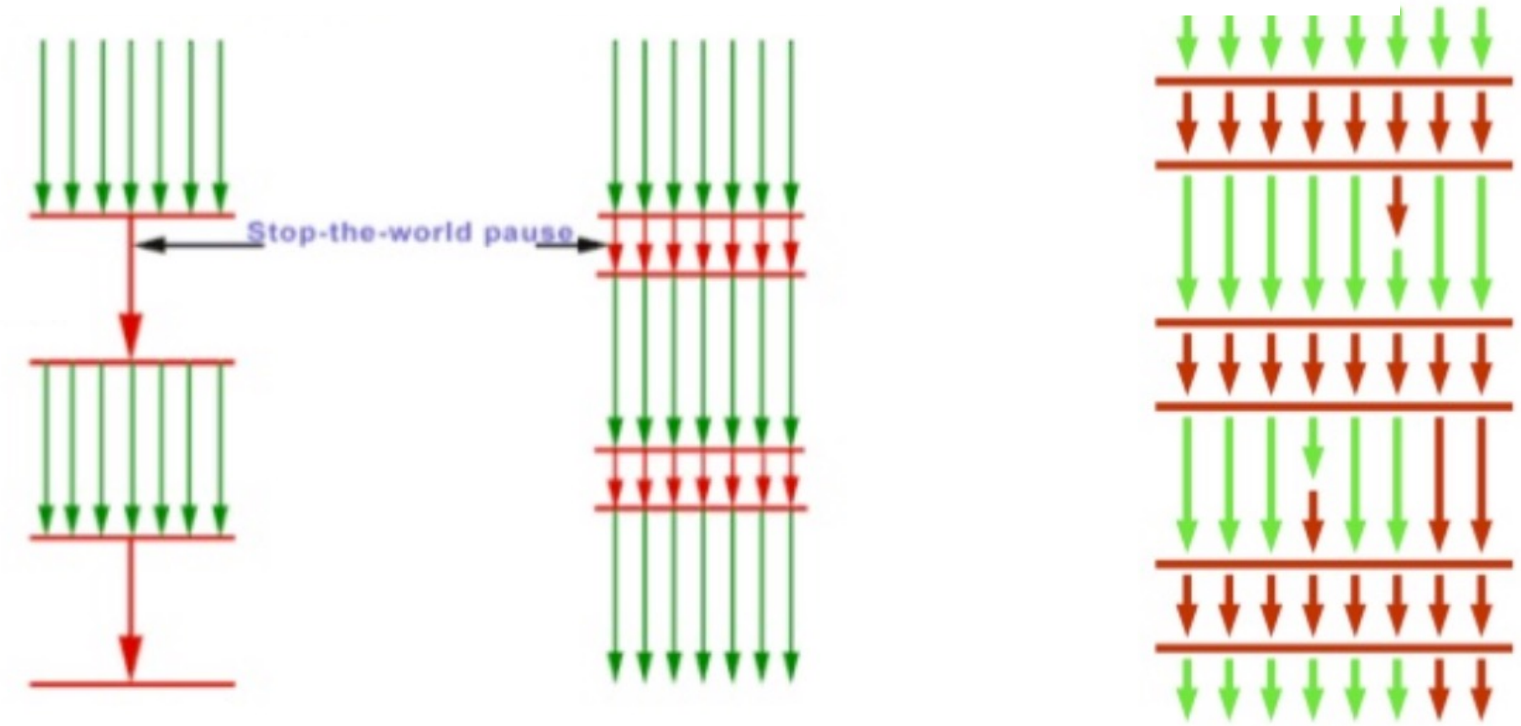
Identify Reachable Objects – Reference Tracking



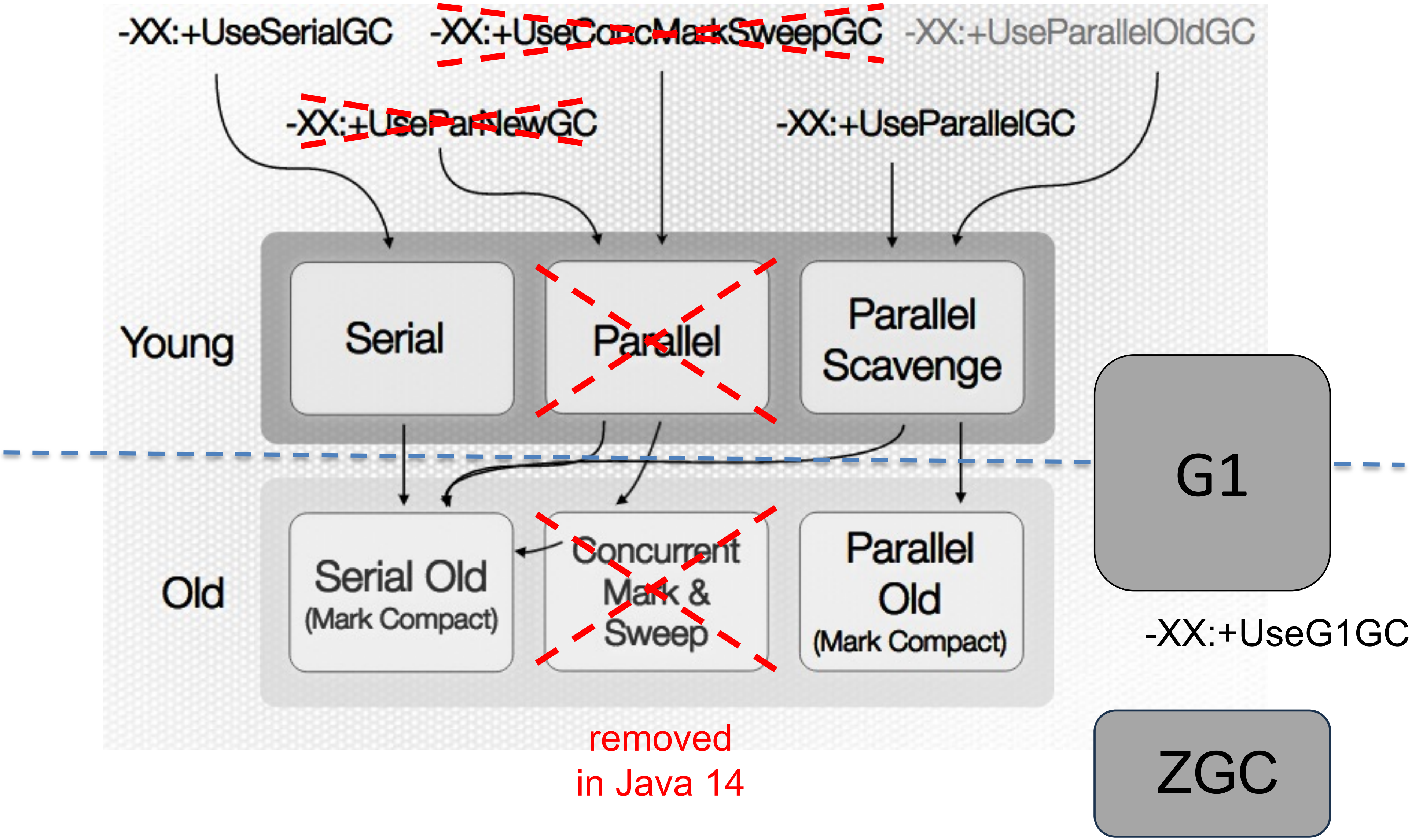
Garbage Collector Design Architecture

- *serial* vs. *parallel*
- *concurrent* vs. *stop the worlds*
- *compacting/sliding* vs. *non-compacting* vs. *copying*

Serial Parallel Garbage-First

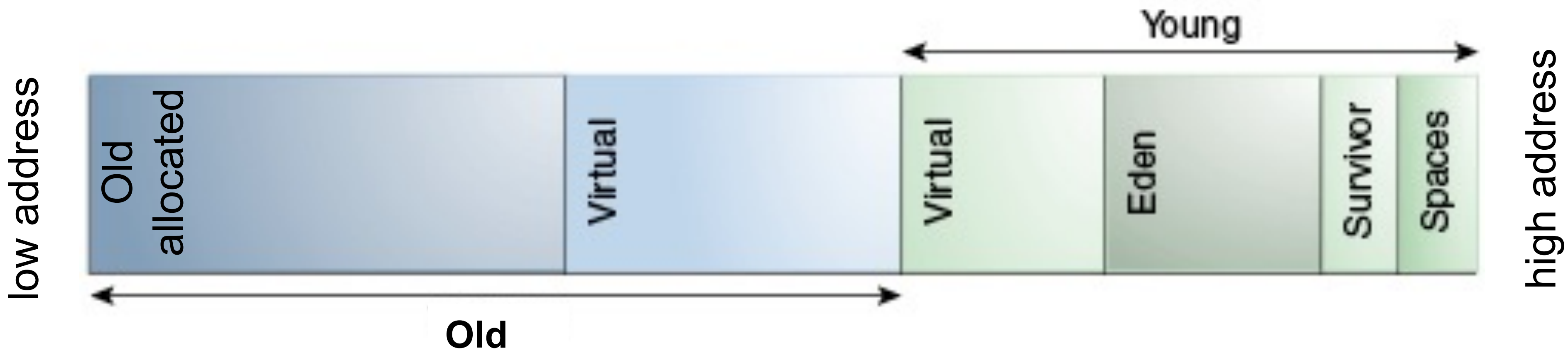


Garbage Collector Design Architecture



Parallel Collector

» **JVM heap layout** supporting adaptive resizing (*virtual* has no physical pages)



» **max heap** size (virtual space allocated) –Xmx

- default ¼ RAM up to 25 GB if there is >=100 GB RAM

» **initial heap** size (really allocated) –Xms

- default 1/64 RAM up to 1 GB if there is >=128 GB RAM

» **Young** vs. **Old** ratio –XX:NewRatio=<n>

- default 2 – thus *Old* is 2x larger than *Young*

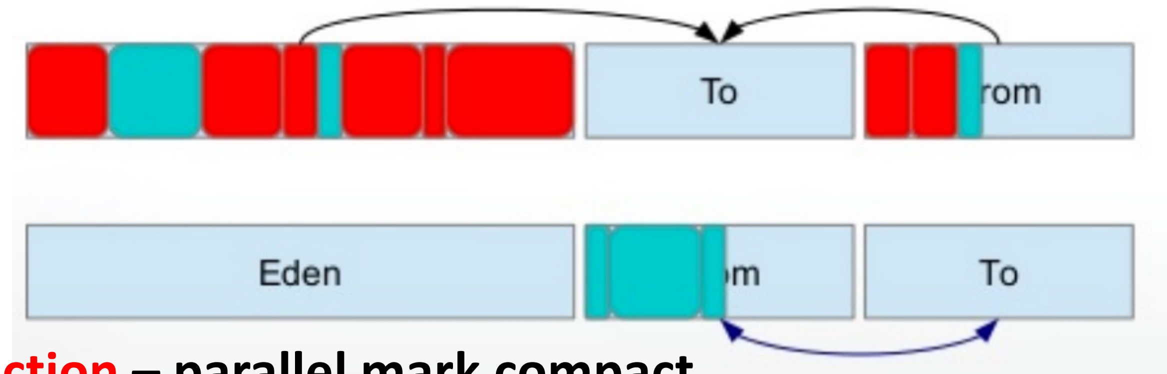
» **Survivor** spaces vs. **Eden** ratio –XX:SurvivorRatio=<n>

- default 8 – thus *Eden* is 8x larger than one *Survivor* space

Parallel Collector

» **object allocations**
- in TLAB inside *Eden* – if no space in TLAB left, new TLAB is allocated
- in *Eden* directly for objects larger than TLAB
- in *Old* directly for objects larger than *Eden*

» **minor collection – parallel scavenge**
- triggered when there is no space for new TLAB/object in *Eden*
- collection in the *Young* generation only, promotes to *Survivor* or *Old*
- results into clean *Eden*, **swap** of active *Survivor* space (one empty)



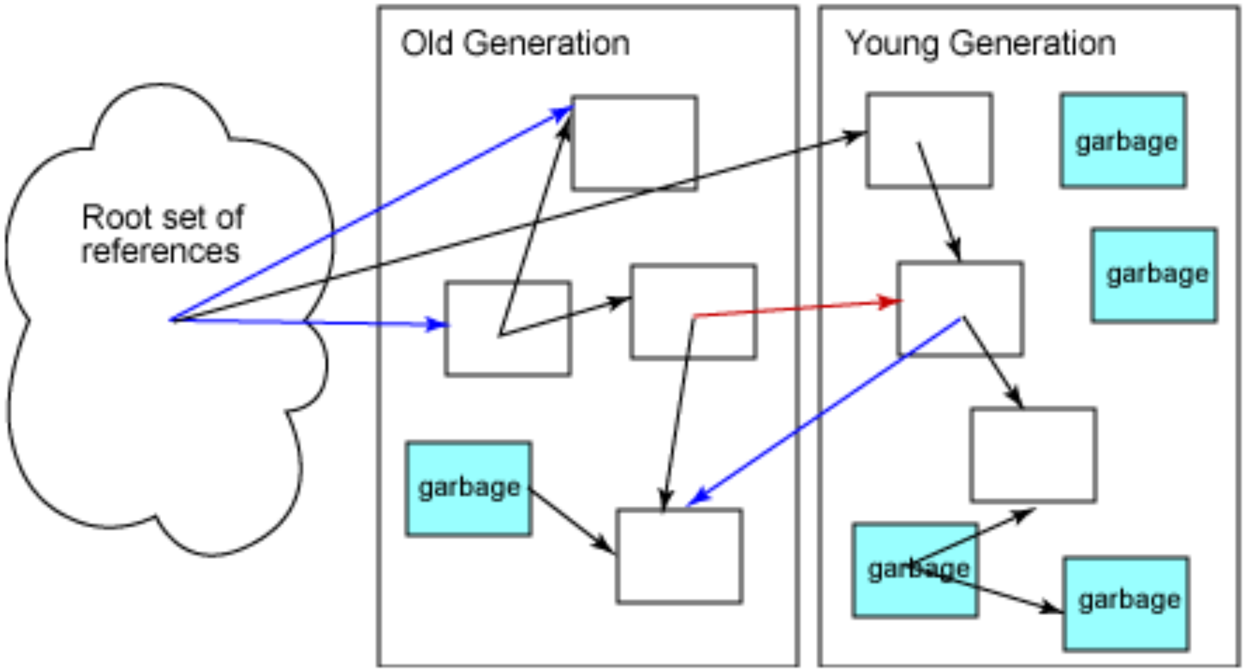
» **full collection – parallel mark compact**
- triggered when there is no space for promotion or a new object in *Old*
- collection in *Young* and *Old* generations
- results into completely clean *Young* (*Eden*, both *Survivor* spaces)

Remembered Set

» **track old-to-young references**

» **speeds-up** frequent identification of reachable objects **for minor collection**

- marking phase starts from **roots** and **references old-to-young**
- do not traverse objects out of *Young* generation
  - fast bit operations using generation size 2^n



red – old-to-young, **blue** – to old (*don’t need to trace during minor collection*)

Card Table Compressed Remembered Set

- whole heap divided to **512 Bytes chunks** (8 cache lines of 64 Bytes)
  - each chunk has one card table slot
- thread-safe **card table** is Byte based
  - avoids expensive atomic read-update-write for bit operations
  - standard byte writes
    - **dirty** (0) – possibly contain reference to *Young* (has false positive)
    - **clean** – cannot contain reference to *Young* (no false negatives)
  - 100 GB heap => 200 MB card table (<0.2%)
    - one cache line holds cards for 32kB of heap
- *write reference to object* implies **assembly code write barrier**
  - no tracking for null writes or reference writes in newly allocated objects
  - tracks **standard object start address** CARD_TABLE[object address >> 9] = 0;
  - tracks **real element address for native reference arrays**

    CARD_TABLE[array slot address >> 9] = 0;
  - **imprecise but very fast** without any condition
    - tracks *Young* generation references only, all reference writes

Card Table Compressed Remembered Set – Write Barriers

write non-null reference in RAX to **standard object** at R11, standard OOP, 64-bit:

```
mov    %rax,0x10(%r11)
mov    %r11,%r8
shr    $0x9,%r8
movabs $0x215153000,%r9
movb   $0x0,(%r9,%r8,1)
```

- `mov %rax,0x10(%r11)` — store reference in RAX to the first field in the object
- `mov %r11,%r8` / `shr $0x9,%r8` — compute card offset from obj. start (R11) directly
- `movabs $0x215153000,%r9` — **card table** start address to R9
- `movb $0x0,(%r9,%r8,1)` — store **dirty** flag to the card table

write non-null reference in RAX to **array** at R10 index EBP, standard OOP, 64-bit:

```
movslq %ebp,%r11
shl    $0x3,%r11
lea    0x18(%r10,%r11,1),%r11
mov    %rax,(%r11)
shr    $0x9,%r11
movabs $0x215153000,%r8
movb   $0x0,(%r8,%r11,1)
```

- `movslq %ebp,%r11` / `shl $0x3,%r11` / `lea 0x18(%r10,%r11,1),%r11` — count address of the slot in the array to R11
- `mov %rax,(%r11)` — store reference in RAX to the array slot
- `shr $0x9,%r11` — compute card offset from slot address (R11)
- `movabs $0x215153000,%r8` — **card table** start address to R9
- `movb $0x0,(%r8,%r11,1)` — store **dirty** flag to the card table

Native Object array structure standard OOP, 64-bit:

Offset		
0x00:	mark word	
	Klass ref.	
0x10:	array length	empty padding
	object reference on index 0	
0x20:	object reference on index 1	
	⋮	

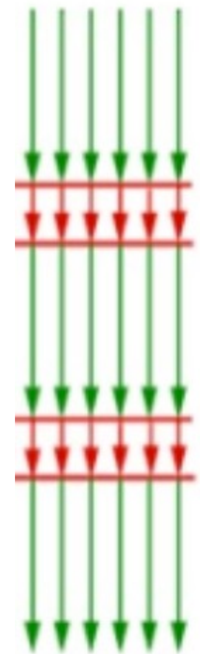
Card Table Compressed Remembered Set – Write Barriers

- **no optimization** for multi reference writes to the same object (which is fast due to already cached part of the card table)
  - object can overlap over 512 Bytes chunk boundary
- **false sharing** in contended multi-thread writes (even worse on multi-CPU)
  - 64B cache line implies sharing of cards for 32kB of Heap (64*512)
  - speed-up with **conditional card table updates** (–XX:+UseCondCardMark)

```
if (CARD_TABLE [address >> 9] != 0) CARD_TABLE [address >> 9] = 0;
```

- for highly contended reference writes up to 7 times faster

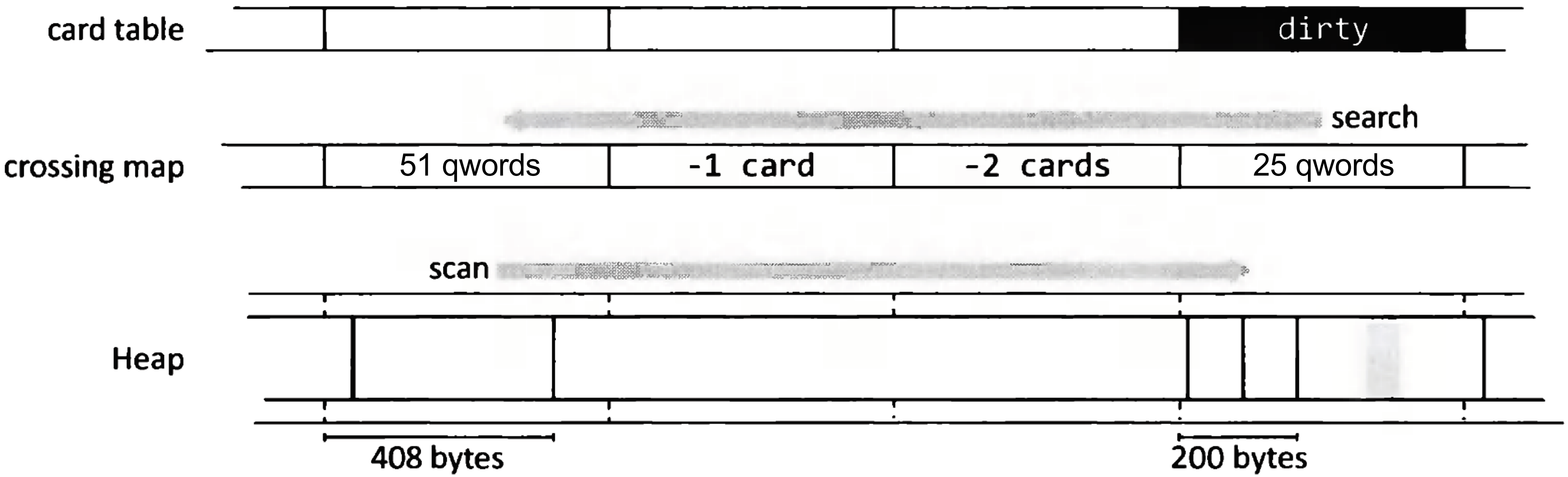
Minor Collector – Parallel Scavenge



- » known also as **throughput garbage collector**
- » default for Oracle JVM 8
- » utilize more cores/CPUs (-XX:ParallelGCThreads=<N>)
  - default #HW threads for <= 8
  - 3+5/8 of #HW threads otherwise (e.g., 13 for 16 threads)
- » stop-the-world manner
- » **copying** with *Survivor* spaces ("from" and "to" are swapped)
  - relocate reachable objects in *Young* generation to "to" *Survivor*
    - if no space, relocate them to *Old* (or trigger full collection)
  - *Eden* and "from" *Survivor* space is empty after the minor collection

- » **parallel processing** of **task queue** initially filled with
  - stripes of cards for searching for old-to-young references (only allocated)
  - JNI handles and VM internal references
  - frames from stacks of all threads
  - static references

Minor Collector – Scan Old for References to Young



» **crossing map** - Byte per 512 Bytes chunk like card table, for **old only**

- updated during allocation/promotion of object and full collection
- **speed-up search for object start**

  N>0 object start offset in align positions of the last object in the card

  N<0 object start offset start –N cards back or the there is the next –N

» **clean cards** before **DFS** queuing of **processing of addresses of old-to-young refs**

- already **forwarded objects** are updated immediately without queuing
- -XX:PrefetchScanIntervalInBytes=576 (9 cache lines)

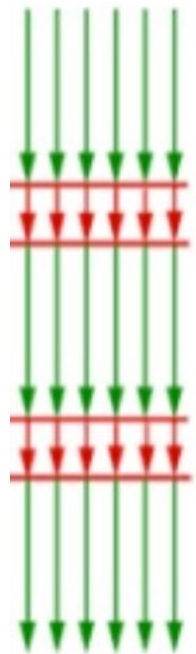
Minor Collector – Process Address of –to-Young Reference

- target is already **marked/forwarded** – mark word (forwarding address | 0b11)
  - **update reference** to forwarding address
- target **not marked** yet
  - current **age** < tenuring threshold
    - copy object to “to” *Survivor* using 32k **PLAB** (-XX:YoungPLABSize=4096)
  - older or no space in *Young*
    - copy object to *Old* using 8k **PLAB** (-XX:OldPLABSize=1024)
  - **mark** previous object with **forwarding address** using CAS
    - failed – de-allocate back, read other thread updated forwarding address
    - success
      - for forwarding in *Young* update **age** of new object
      - DFS queuing of processing of object’s addresses of **old-to-young refs**
  - **update reference** to forwarding address

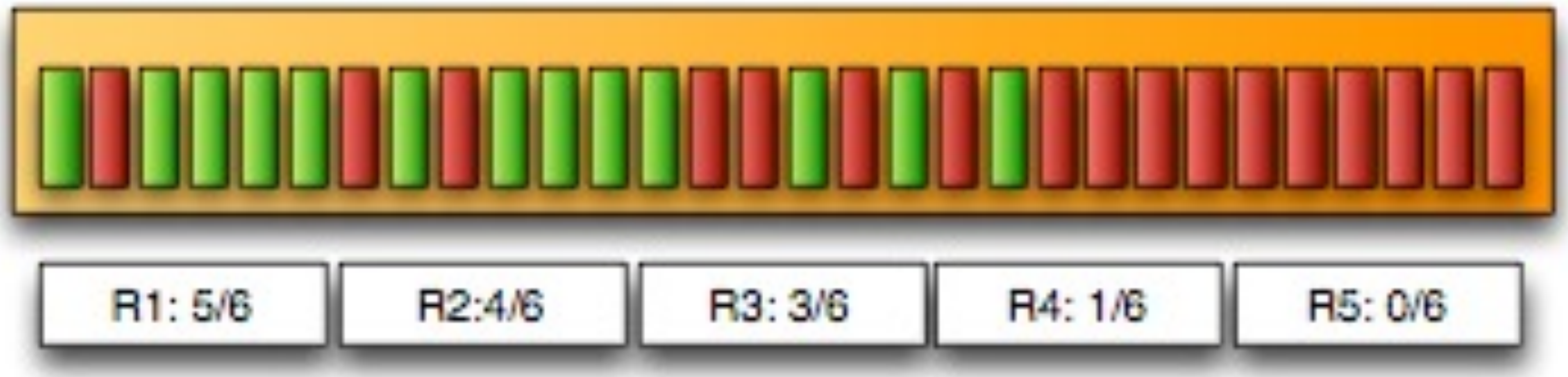
Note: all reference changes update card table iff address is in “to” *Survivor*

all PLAB or object re-allocations are **NUMA** aligned to speed-up collection

Full Collector – Parallel Mark Compact



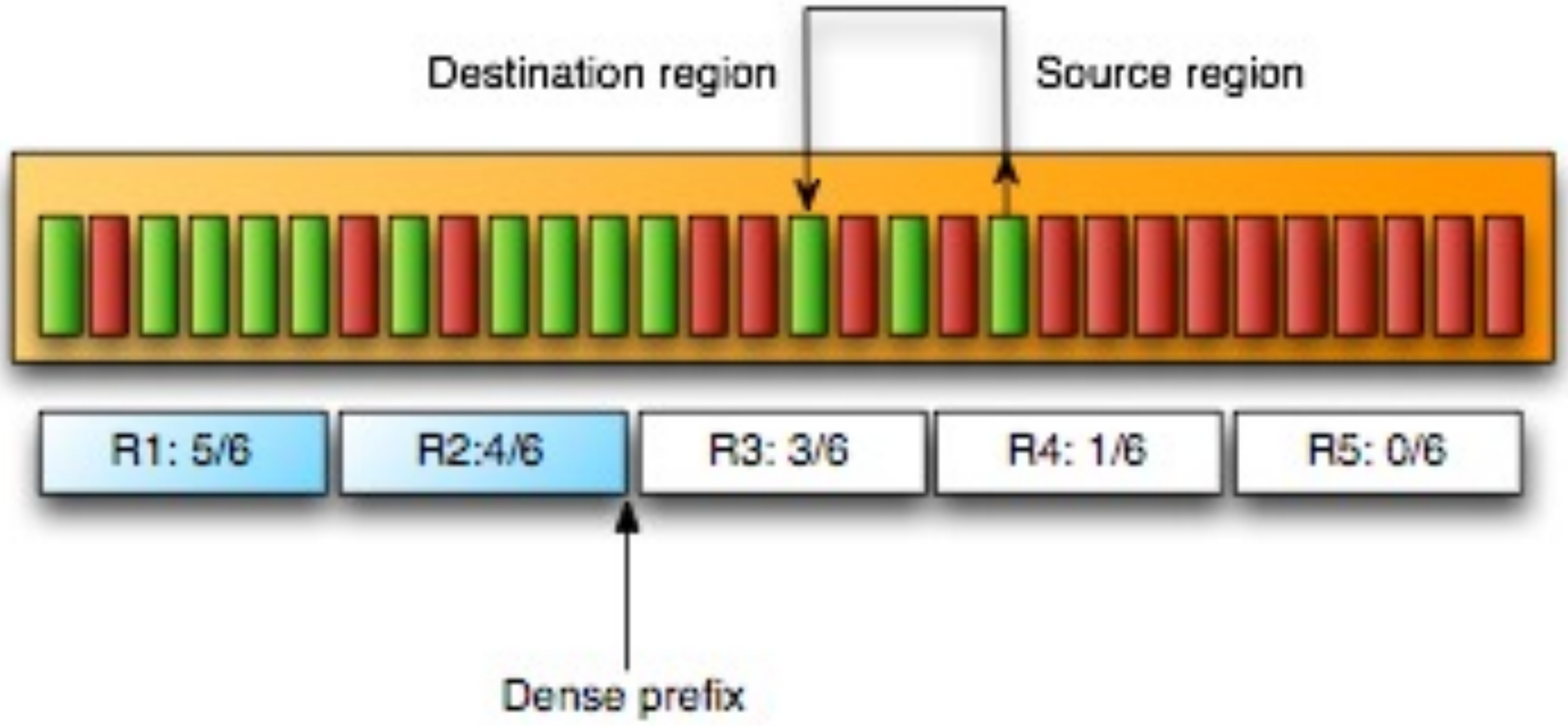
- default for Oracle JVM 8
- stop-the-world manner
- multi-threaded
- *Old* generation logically divided into fixed-size regions
- uses **sliding compaction** - clean *Eden* and both *Survivors* as well
  - doesn't need additional memory, but is slower than copying
- **parallel mark** phase
  - initiated with all roots (not using card table)
  - track all references (not just those pointing to *Young)*
  - info about reachable objects (location & size) is propagated to the corresponding region data



Full Collector – Parallel Mark Compact

» **serial summary** phase

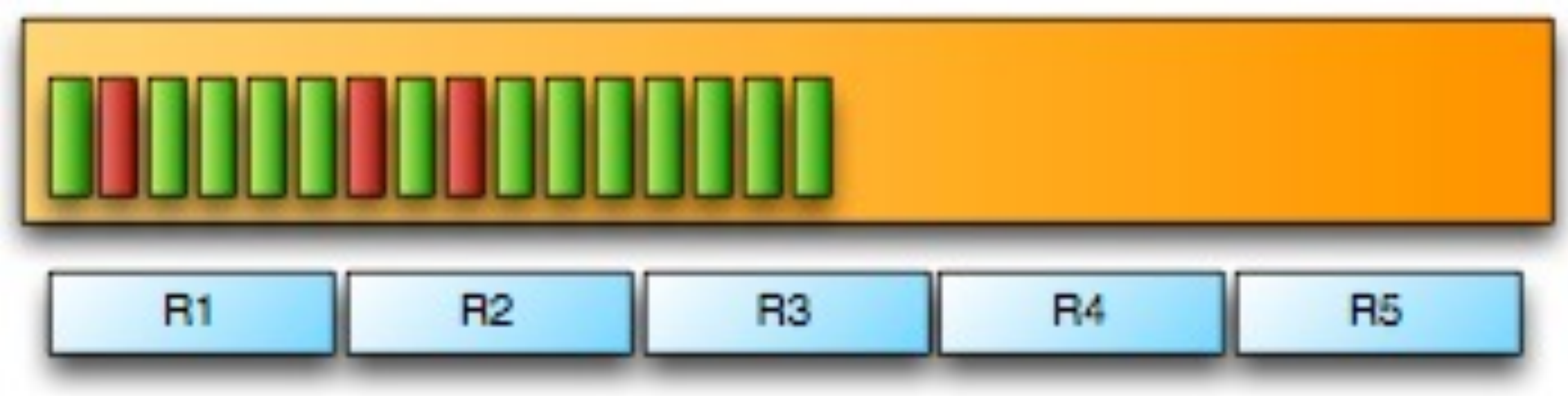
- identify density of regions (due to previous compactions, older objects should be on the left, younger to right side)
- find from which region (starting from the left side) it has sense to do compaction based on recovered space from a region
  - *dense prefix* – left regions are not collected
- calculate new location of each live object; **most right regions will fill most left ones**; pretend data locality keeping their order



Full Collector – Parallel Mark Compact

» **parallel compaction/sweeping** phase

- divide regions with some targets (start of objects)
- each thread first compact the region itself and fill it by designated right regions
  - all references are updated based on summarized data (read only)
  - crossing map is updated to track the last object start in chunk
- *no synchronization* needed, only one thread operate per each region
- update root references and clean empty in parallel
- finally, heap is packed, and large empty block is at the right end



Full Collector – Parallel Mark Compact

- support **strong generational hypothesis** – younger object dies earlier then older
  - objects with highest probability to survive are located on the left side of old generation (because of previous GC runs)
  - **dense prefix** completely *avoids their costly copying*
  - 50% of full collection work reclaim 82% of garbage
  - reclaim of additional 18% of garbage cost as much as previous work
- dense prefix is adaptively updated
  - considering *used* to *total heap* ratio
  - affects pause time of full collection
- after full collection
  - the whole *Young* is empty
  - the card table is cleaned (there are no references to *Young*)

Parallel Collector - Ergonomics

» **adaptive mechanism resizing** generations (-XX:+UseAdaptiveSizePolicy)

- **max pause time** goal (-XX:MaxGCPauseMillis=<undef>)
  - if not met - shrink *Young* size
- **throughput** goal (-XX:GCTimeRatio=99) – applied when the previous is met
  - if not met – increase *Young* and *Old* generations
    - *Young* increased according to its time portion in total time
- **minimum footprint** goal – applied if all previous are met
  - shrink heap size

-XX:YoungGenerationSizeIncrement=20 ; -XX:TenuredGenerationSizeIncrement=20

-XX:AdaptiveSizeDecrementScaleFactor=4 (default 5%)

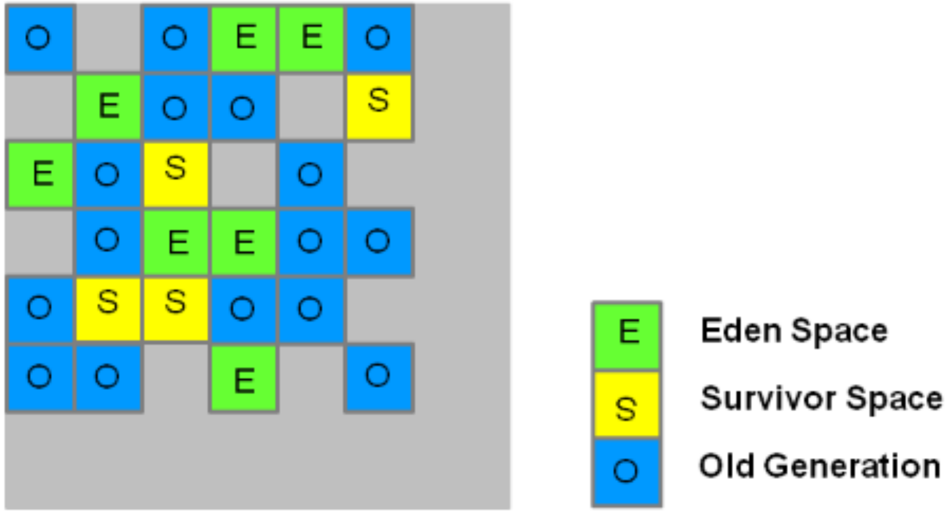
-XX:YoungGenerationSizeSupplement=80 (similar for tenured)

-XX:YoungGenerationSizeSupplementDecay=8 (8 times added)

-XX:TenuredGenerationSizeSupplementDecay=2 (2 times added)

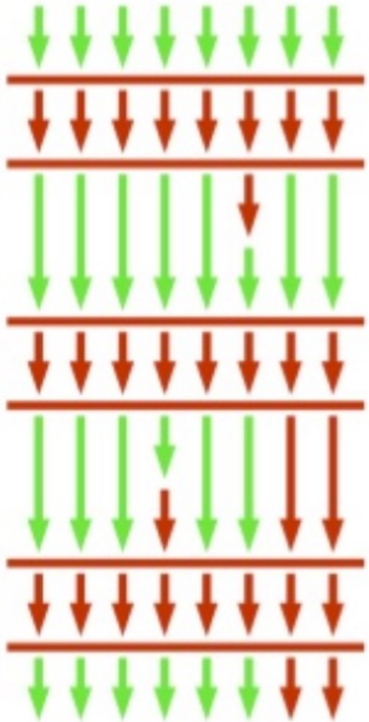
Garbage First (G1) Collector

- **dynamic generational collector** called G1GC (-XX:+UseG1GC)
- **concurrent** collector for large heaps (replacement for older CMS)
- the whole heap is divided into **regions** (by def. to be close to 2048 regions 1-32MB)
- no explicit separation between generations, only regions are mapped to generational spaces (generation is set of regions, changing in time)
- set of regions defines
  - *Young* generation
  - *Old* generation



- compacting -> enables bump-the-pointer, TLABs, uses CAS
- **copying** = copy live from a region to an empty region
- keep **Humongous regions** (sequences) for objects >=50% regions size
- maintain list of free regions for constant time

Garbage First (G1) Collector



» **activities** in garbage first collector – **minor**, **mixed** and **full** collections

- **parallel** with **global safe point** (stop the world)
  - initial marking pause – end of previous evacuation
  - final marking pause with data counting
    - prepare candidate regions for mixed
  - copying (evacuation)
- **concurrent** with multiple threads
  - remember set refinement
  - marking + write barriers for concurrent modifications
  - clean-up

» **major speed-up** is that **fast copying** approach is applied incrementally to *Old*

- requires more heap than parallel due to concurrent activities

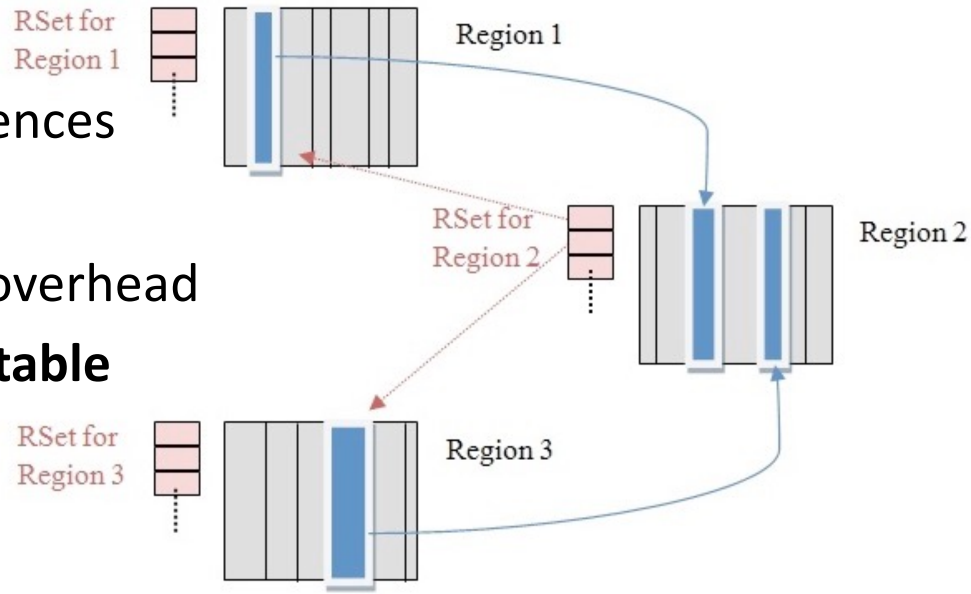
» **controlled pause time** up to MaxGCPauseMillis

» **poor handling of larger objects (humongous objects)**

» **NUMA aware –** new Eden region from local memory

» default since JVM 9

Garbage First (G1) Collector – Remember Set



- **track references into a region**
  - **ignore** null and inter-region references
  - old-to-young and old-to-old
- additional structures with ~5% heap overhead
- use **per-region-table** (PRT) with **card table** updated *asynchronously* using *update thread log buffers*
  - processed by refinement threads
    -XX:G1ConcRefinementThreads=<n> (max threads)
  - filled by *compiled write barrier* (pseudo code shown for simplification)

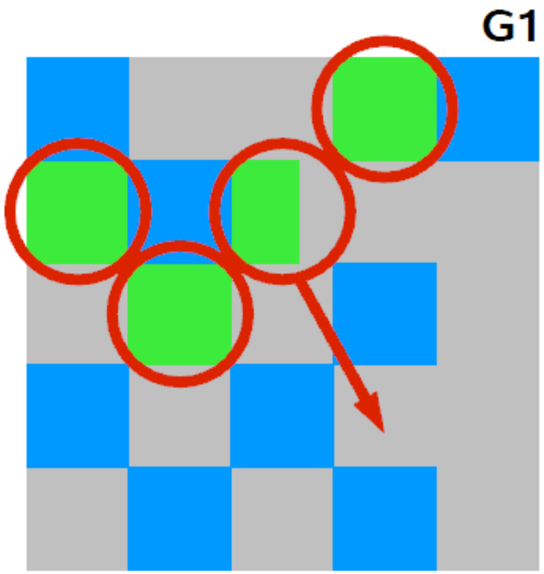
```
oop oldFooVal = this.foo;
if (GC.isMarking != 0 && oldFooVal != null){
  g1_wb_pre(oldFooVal);
}
this.foo = bar;
if ((this ^ bar) >> 20) != 0 && bar != null) {
  g1_wb_post(this);
}
```

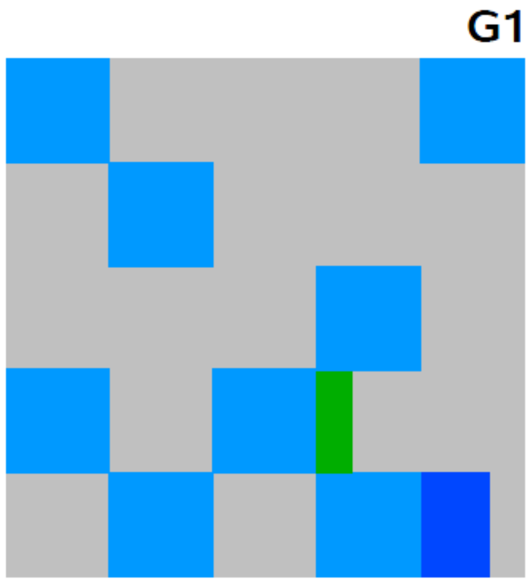
log2 of region size (1MB)

-XX:+G1SummarizeRSetStats -XX:G1SummarizeRSetStatsPeriod=1

Garbage First (G1) Collector – Minor and Mixed Collection

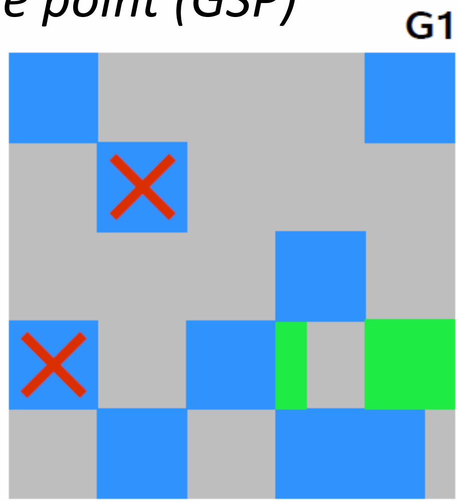
- » stop-the-world approach with **parallel threads**
- » **triggered** when no more allocation in *Young* regions possible
- » **collection set** (CSet)
  - *Eden* and “from” *Survivor* regions for pure **minor collection**
  - *Eden*, “from” *Survivor* and *candidate Old regions* for **mixed collection**
- » reachable objects identified from roots + RSet for the regions + card table
- » reachable objects are copied (from *Eden* and *Survivor* regions) into one or more new *Survivor* regions
  - using forwarding address with marking like parallel scavenge
- » if aging threshold is met => promoted into *Old* regions

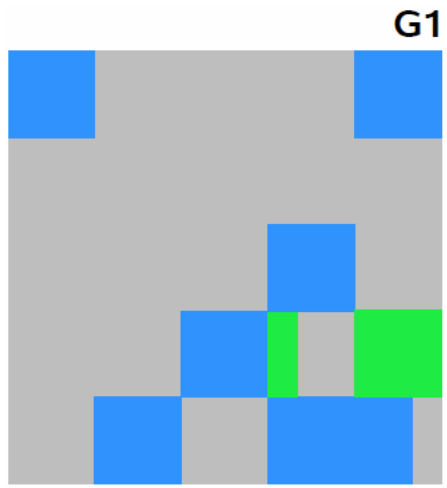




Garbage First (G1) Collector – Marking Phase

- **triggered by** heap occupancy percent (-XX:InitiatingHeapOccupancyPercent=45)
- outcomes
  - **candidate *Old* regions** with a lot of garbage for mixed collection
  - cleanup completely empty *Old* regions
- **initial mark** – done after minor collection *utilizing global safe point (GSP)*
  - snapshot-at-the-beginning (SATB) – scan roots
- **marking** and **region-based statistics collection** – *concurrent*
  (-XX:ConcGCThreads=<n>)
  - can be interrupted by minor GC
  - pre-write barrier keeps previous reference in SATB
- **final mark -** right after the next minor collection *in GSP*
  - reflect changes in previous minor collections and allocations utilizing modifications in card tables
  - summarize and prepare ordered candidates





Garbage First (G1) Collector – Full Collection

- multiphase **full tracking with compact of all regions** during global safe point
- triggered by
  - **concurrent mode failure** – *Old* fill-up before concurrent marking is complete
    - increase heap, decrease trigger threshold, more concurrent threads
  - **promotion failure** – mixed collection runs-of space in *Old*
    - trigger sooner
  - **evacuation failure** – minor collection has no more space for promotion
    - increase heap
  - **humongous allocation failure** – no space for large objects
    - avoid large objects (>50% of region size)
    - increase region size (alternatively increase heap)
      - max region size is 32MB

Garbage First (G1) Collector – Humongous Objects

- objects larger than ½ of the region are considered as **humongous**
  - with 1MB region it is just 500kB -> there can be a lot of such objects
- **allocation**
  - check concurrent trigger and optionally start concurrent marking
  - one set of humongous regions contain just one such object
    - **waste** up to region size – 1 + allocated **out of *Young*** generation
  - not having sequence of free regions for allocation of an object trigger **expensive full collection**
- **reclamation** of non-reachable humongous objects during
  - cleanup phase of concurrent cycle
  - full collection (can compact free space)
- **debug** humongous allocations
  - -XX:+UnlockExperimentalVMOptions –XX:G1LogLevel=finest –XX:+PrintAdaptiveSizePolicy
  - use **Java Flight Recorder** in Java Mission Control
    - all allocations tracked in runtime like TLAB allocations

Garbage First (G1) Collection – Tuning Options ☺

G1ConcMarkForceOverflow
G1ConcMarkStepDurationMillis
G1ConcRSHotCardLimit
G1ConcRSLogCacheSize
G1ConcRefinementGreenZone
G1ConcRefinementRedZone
G1ConcRefinementServiceIntervalMillis
G1ConcRefinementThreads
G1ConcRefinementThresholdStep
G1ConcRefinementYellowZone
G1ConcRegionFreeingVerbose
G1ConfidencePercent
G1DummyRegionsPerGC
G1EvacuationFailureALot
G1EvacuationFailureALotCount
G1EvacuationFailureALotDuringConcMark
G1EvacuationFailureALotDuringInitialMark
G1EvacuationFailureALotDuringMixedGC
G1EvacuationFailureALotDuringYoungGC
G1EvacuationFailureALotInterval
G1ExitOnExpansionFailure
G1FailOnFPError
G1HRRSFlushLogBuffersOnVerify
G1HRRSUseSparseTable
G1HeapRegionSize
G1HeapWastePercent
G1MarkingOverheadPercent
G1MarkingVerboseLevel
G1MaxVerifyFailures
G1MixedGCCountTarget
G1PrintHeapRegions
G1PrintRegionLivenessInfo
G1RSBarrierRegionFilter
G1RSScrubVerbose
G1RSetRegionEntries
G1RSetRegionEntriesBase
G1RSetScanBlockSize
G1RSetSparseRegionEntries
G1RSetSparseRegionEntriesBase
G1RSetUpdatingPauseTimePercent
G1RecordHRRSEvents
G1RecordHRRSOops
G1RefProcDrainInterval
G1ReservePercent
G1SATBBufferEnqueueingThresholdPercent
G1SATBBufferSize
G1SATBProcessCompletedThreshold
G1ScrubRemSets
G1SecondaryFreeListAppendLength
G1StressConcRegionFreeing
G1StressConcRegionFreeingDelayMillis
G1SummarizeConcMark
G1SummarizeRSetStats
G1SummarizeRSetStatsPeriod
G1TraceConcRefinement
G1TraceHeapRegionRememberedSet
G1TraceMarkStackOverflow
G1UpdateBufferSize
G1UseAdaptiveConcRefinement
G1VerifyBitmaps
G1VerifyCTCleanup
G1VerifyHeapRegionCodeRoots
G1VerifyRSetsDuringFullGC
G1YoungSurvRateNumRegionsSummary
G1YoungSurvRateVerbose
PrintCFG1