Effective Software

Lecture 11: Memory Management in JVM – Memory Layout, Garbage Collectors

David Šišlák
david.sislak@fel.cvut.cz

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**
    - discover unreachable objects (not transiently reachable from **roots** – variables and stack operands in frames, static fields, special native references from JNI)
    - reclaim storage

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

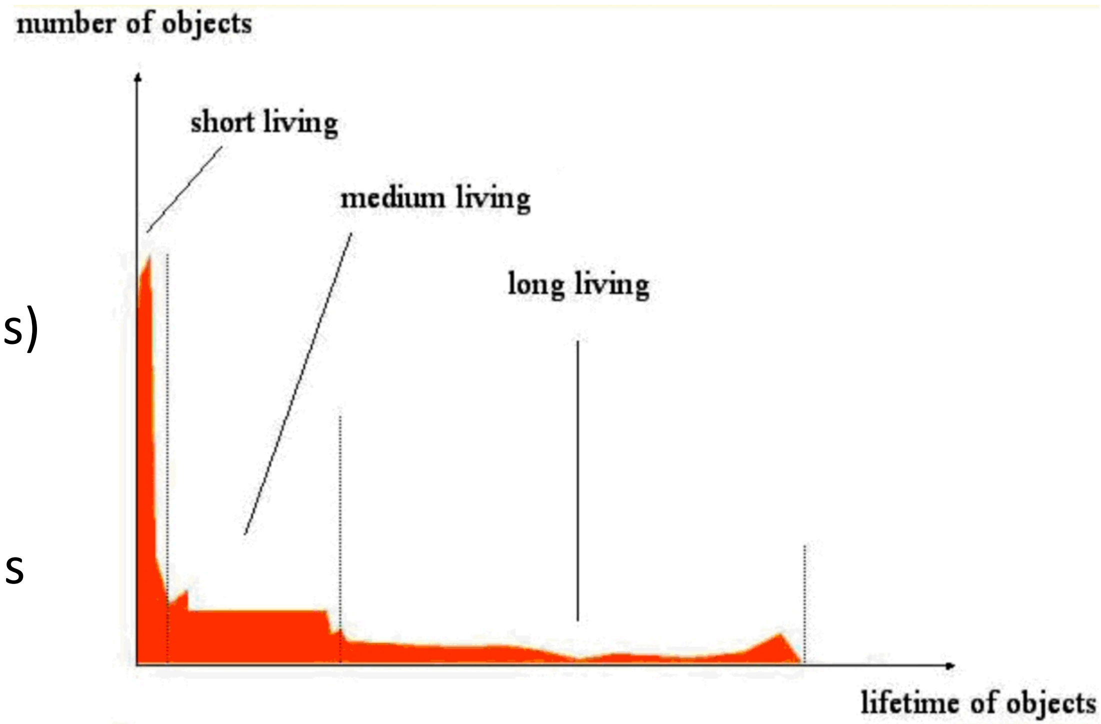
Automatic Memory Management

» *desired characteristics*

- **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 occupy 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** – even not newly created 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)
  - **tenured**
    - large size
    - rare slow **full** collections (seconds)
- promotion of objects during minor collections



Identify Reachable Objects

» **reference counting**

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

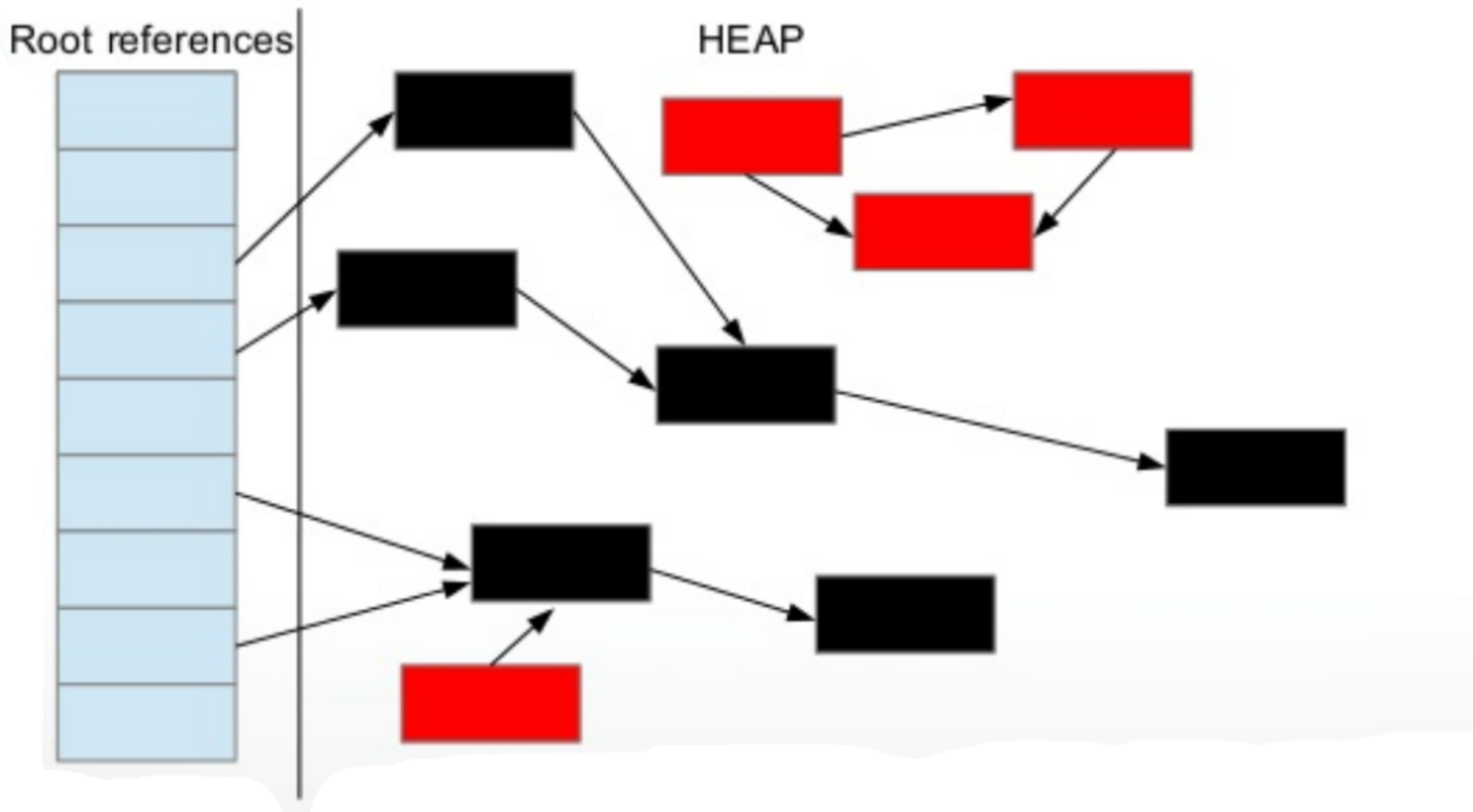
Identify Reachable Objects

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

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

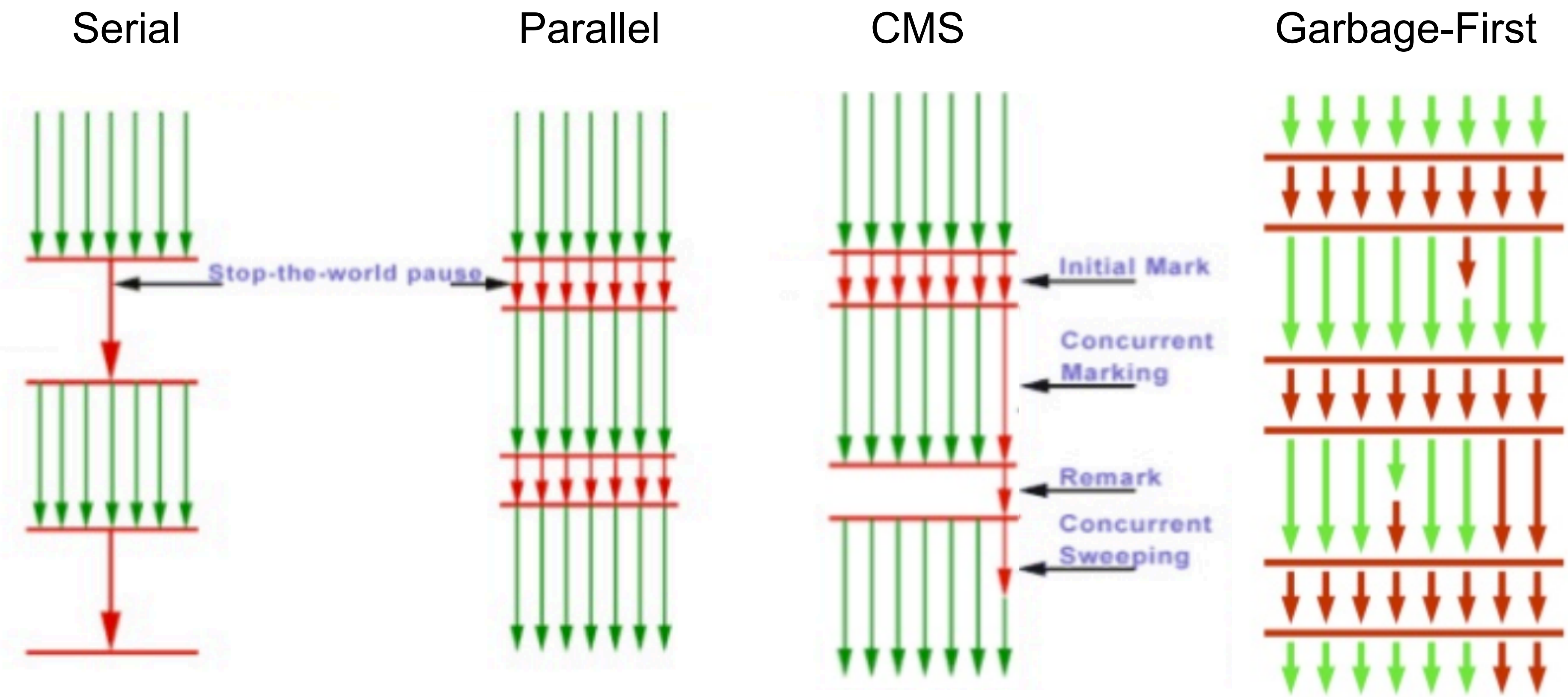
  - in different object using object type
    - reference positions in klass VM structure
- **marking** traverse all objects from **roots**
  - depth-first search, breath-first search
  - dominates collection time due to random access to memory
    - cache prefetching to reduce cost
- use marks to avoid 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

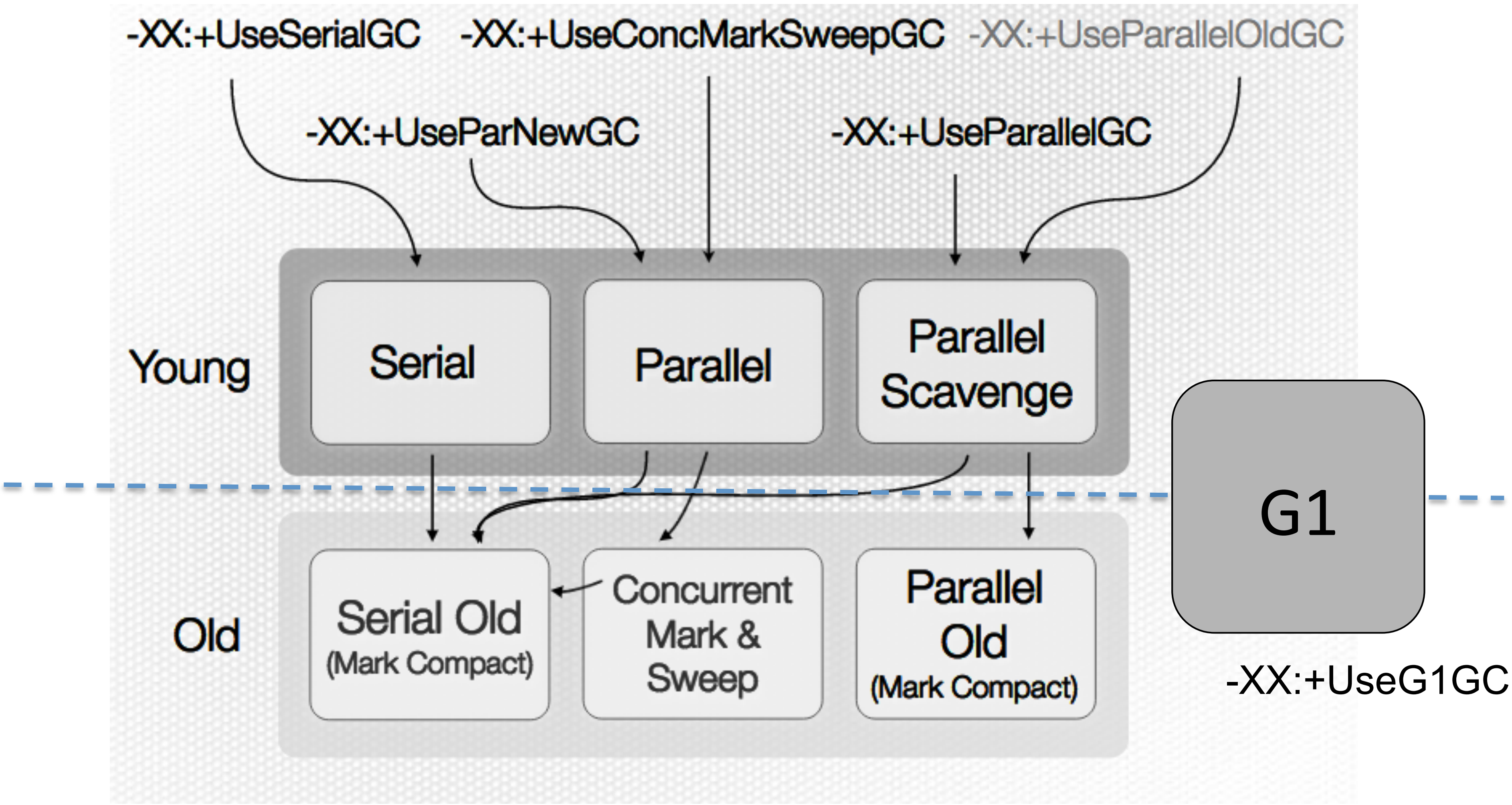


Collector Design Architecture

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

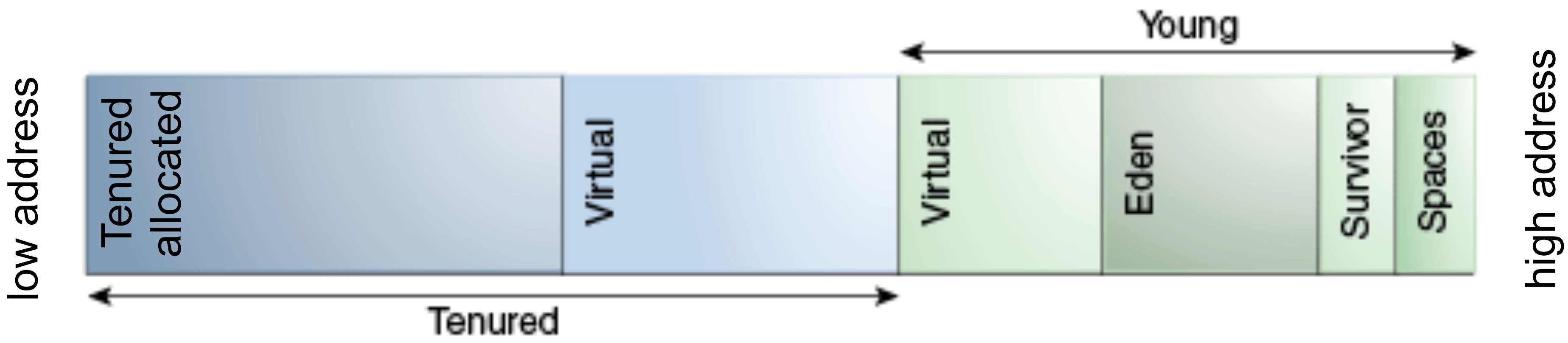


Collector Design Architecture



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 32 GB if there is >=128 GB RAM

» **initial heap** size (really allocated) –Xms
  - default 1/64 RAM up to 1 GB if there is >=128 GB RAM

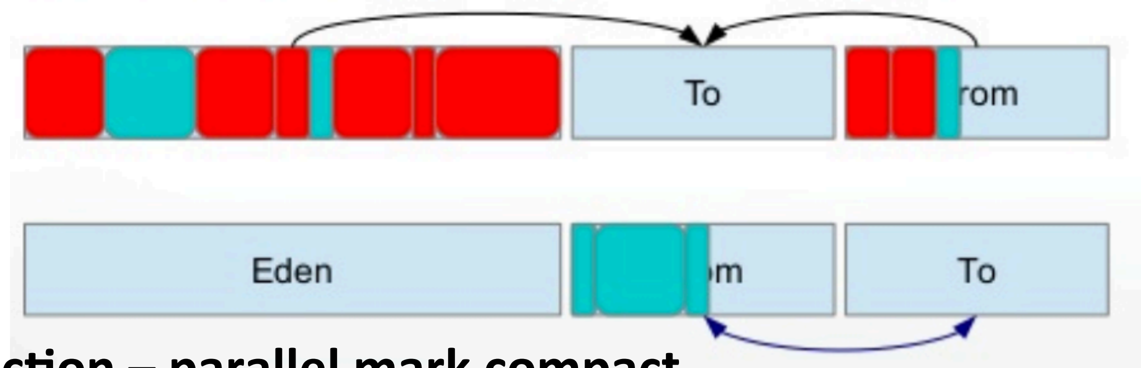
» **young** vs. **tenured** ratio –XX:NewRatio=<n>
  - default 2 – thus tenured is 2x larger than young

» **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 - no space in TLAB left, new TLAB allocated
- in eden directly for objects larger than TLAB
- tenured directly for objects larger than eden

» **minor collection – parallel scavenge**
- triggered when no space for new TLAB/object in eden
- collection in young generation only, promote to survivor or tenured
- results into clean eden, **swap** of survivor spaces (one empty)



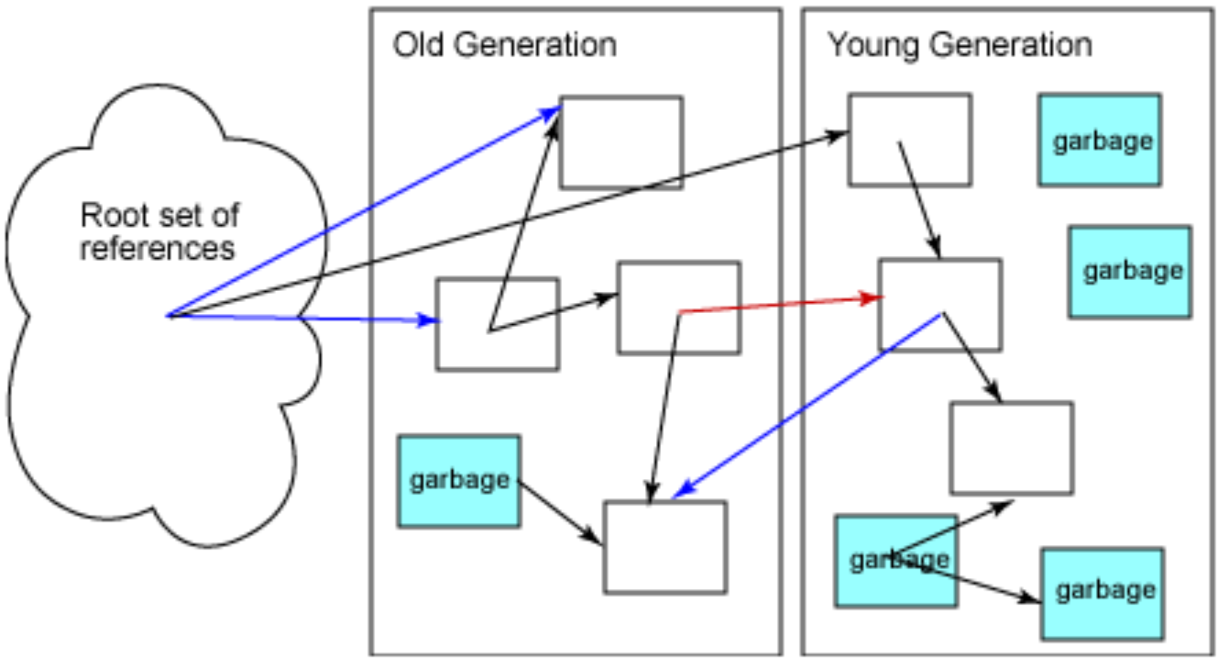
» **full collection – parallel mark compact**
- triggered when there is no space for promotion or new object in tenured
- collection in young and tenured generations
- results into completely clean young (eden, both survivor spaces)

Remembered Set

» **track tenured-to-young references**

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

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



red – tenured-to-young, blue – to old (*don't need 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
  - avoid 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* imply **assembly code write barrier**
  - no tracking for null writes or reference writes in newly allocated
  - track **standard object start address** CARD_TABLE[object address >> 9] = 0;
  - track **real element address for native reference arrays**
  - **imprecise but very fast** without any condition
    - cards for young, all reference writes

CARD_TABLE[array slot address >> 9] = 0;

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)
```

- store reference in RAX to the first field in object
- compute card offset from obj. start (R11) directly
- **card table** start address to R9
- store **dirty** to 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)
```

- count address of slot in array to R11
- store reference in RAX to array slot
- compute card offset from slot address (R11)
- **card table** start address to R9
- store **dirty** to 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 card table)

- object can overlap over chunk boundary

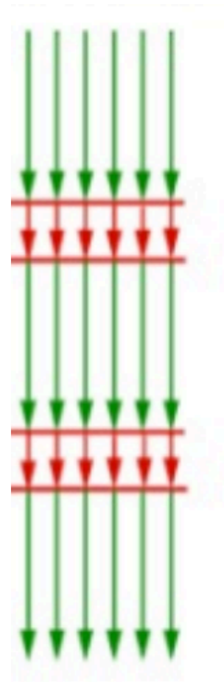
» **false sharing** in contended multi-thread writes (even worse on multi-CPU)

- 64B cache line implies sharing of cards for 32kB (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

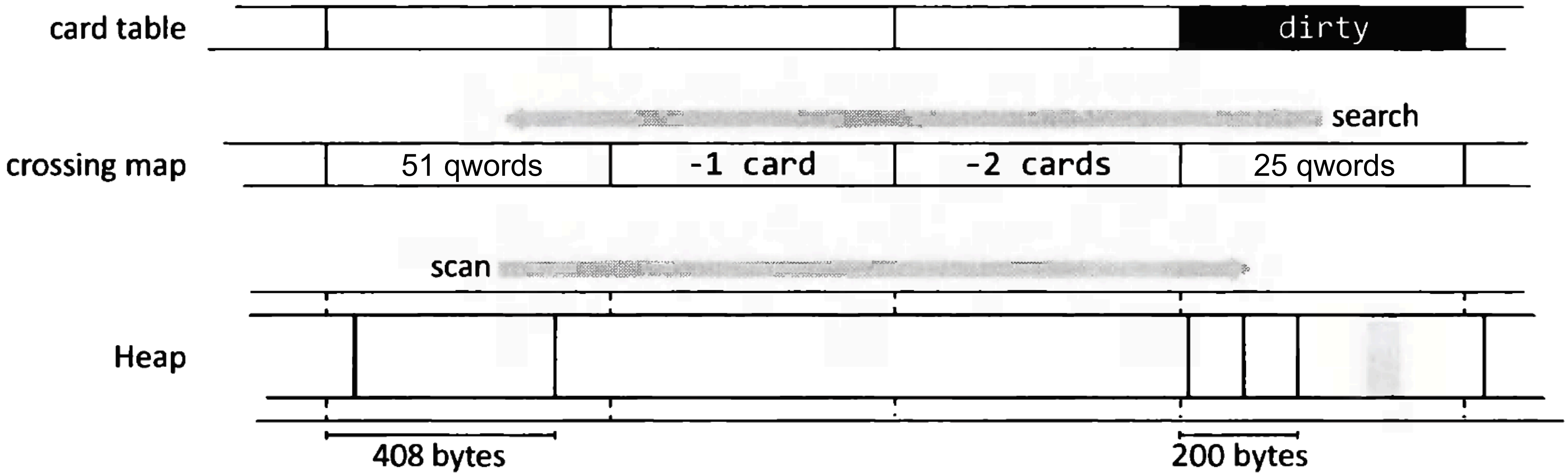
Minor Collector – Parallel Scavenge



- known also as **throughput garbage collector**
- currently default for Oracle JVM
- 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”, 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 minor collection

- **parallel processing** of **task queue** initially filled with
  - add stripes of cards for scanning for old-to-young references (only allocated)
  - add JNI handles and VM internals
  - add frames from stacks
  - add static references

Minor Collector – Scan Tenured for References to Young



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

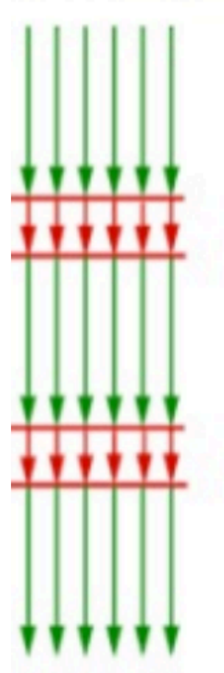
- target is already **marked/forwarded** – mark word (forwarding address | 0b11)
  - **update reference** to forwarding address
- target **not marked** yet
  - current **age** < tenuring threshold
    - copy object to “to” survivor using 32k **PLAB** (-XX:YoungPLABSize=4096)
  - older or no space in young
    - copy object to tenuring using 8k **PLAB** (-XX:OldPLABSize=1024)
  - **mark** previous object with **forwarding address** using CAS
    - failed – de-allocate back, read other thread 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

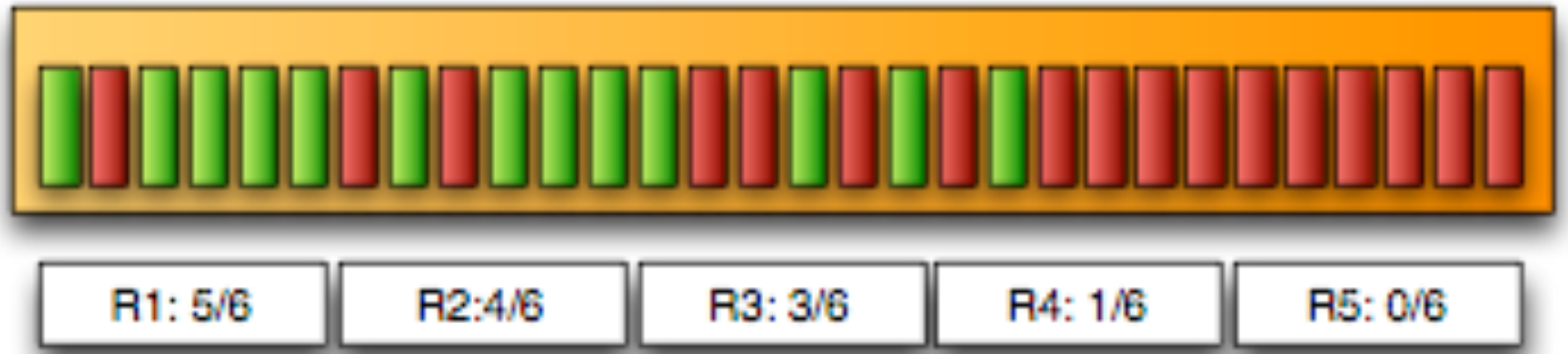
Note: all reference changes update card table if in “to” survivor

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

Full Collector – Parallel Mark Compact

- default for Oracle JVM
- stop-the-world manner
- multiple threads as parallel scavenge
- tenured generation logically divided into fixed-size regions
- use **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 reference not just those targeting to young
  - info about reachable objects (location & size) are propagated to corresponding region data

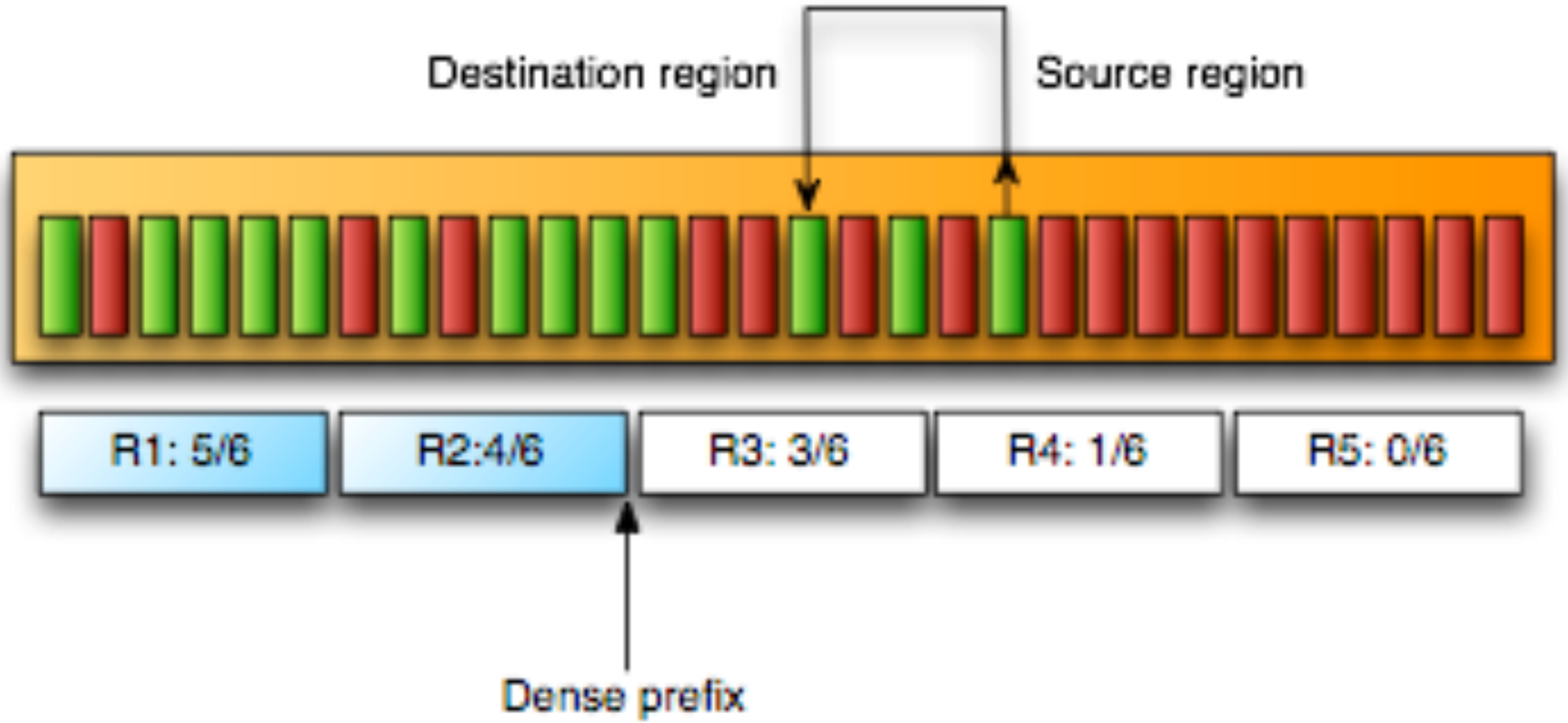




Full Collector – Parallel Mark Compact

» **serial summary** phase

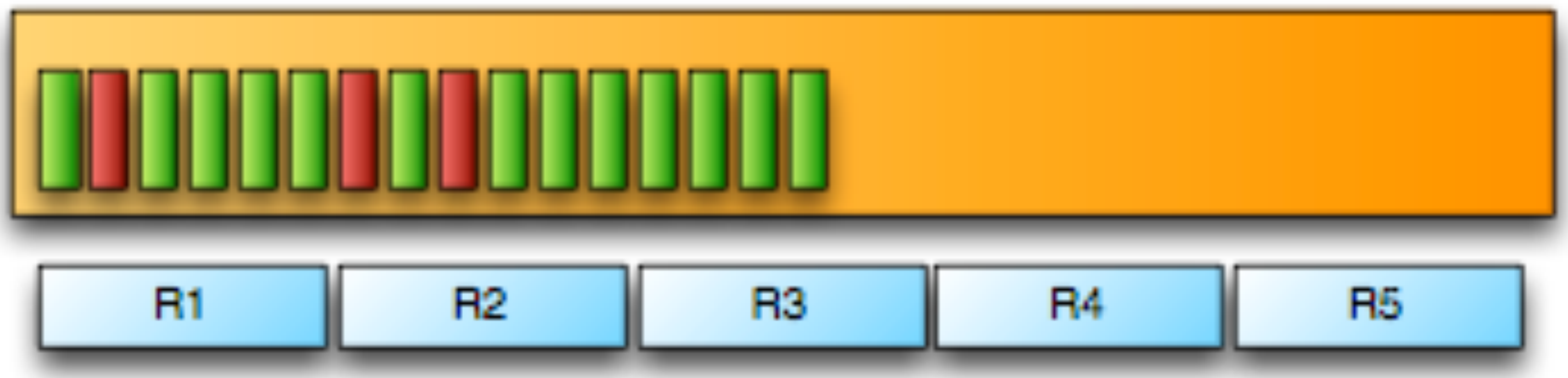
- identify density of regions (due to previous compactions, older objects should be on the left, younger to right side)
- find from which region (starting from the left side) it has sense to do compaction regarding recovered from a region
  - *dense prefix* – left regions which are not collected
- calculate new location of each live data for each regions; **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

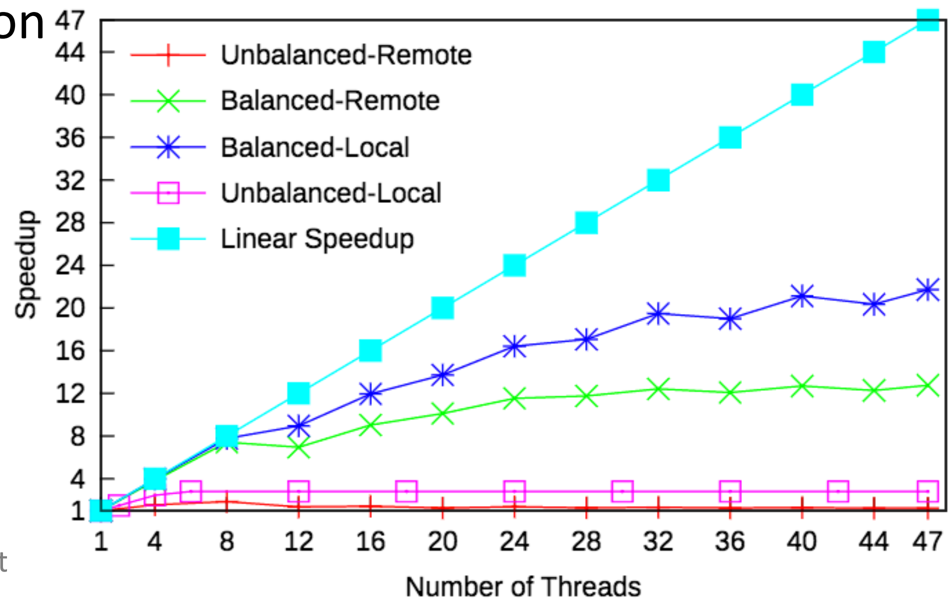


Full Collector – Parallel Mark Compact

» support **strong generational hypothesis** - even not newly created object dies earlier then older
  - the objects with highest probability to survive are located on the left side (because of previous GC runs)
  - **dense prefix** completely *avoid 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
  - whole young is empty
  - 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 generation size where the pause time is longest and at least above the goal
- **throughput** goal (-XX:GCTimeRatio=99) – applied when previous is met
  - if not met – increase both 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)

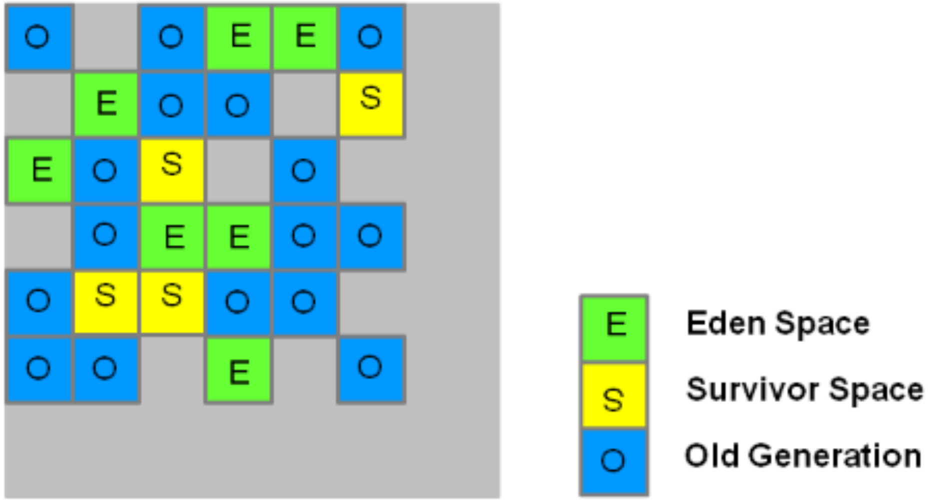
-XX:YoungGenerationSizeSupplementDecay=8 (8 times added)

-XX:TenuredGenerationSizeSupplementDecay=2 (2 times added)

Garbage First Collector

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

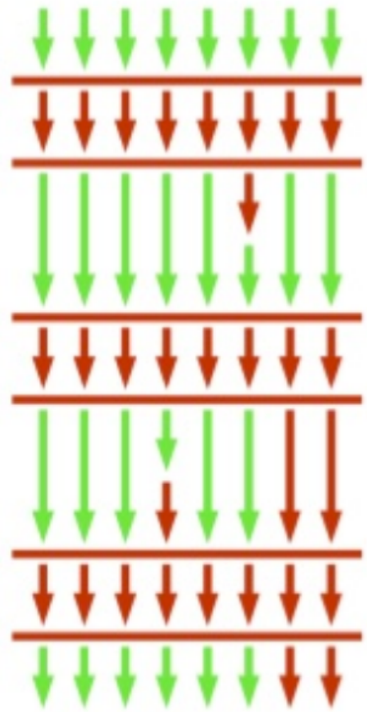
- set of regions defines
  - young generation
  - tenured generation



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

Garbage First Collector

» **activities** in garbage first collector
  - parallel with **global safe point**
    - minor collection
      - initial mark
    - mixed collection
    - full collection
  - **concurrent** with multiple threads
    - remember set refinement
    - scanning
    - marking
    - clean-up



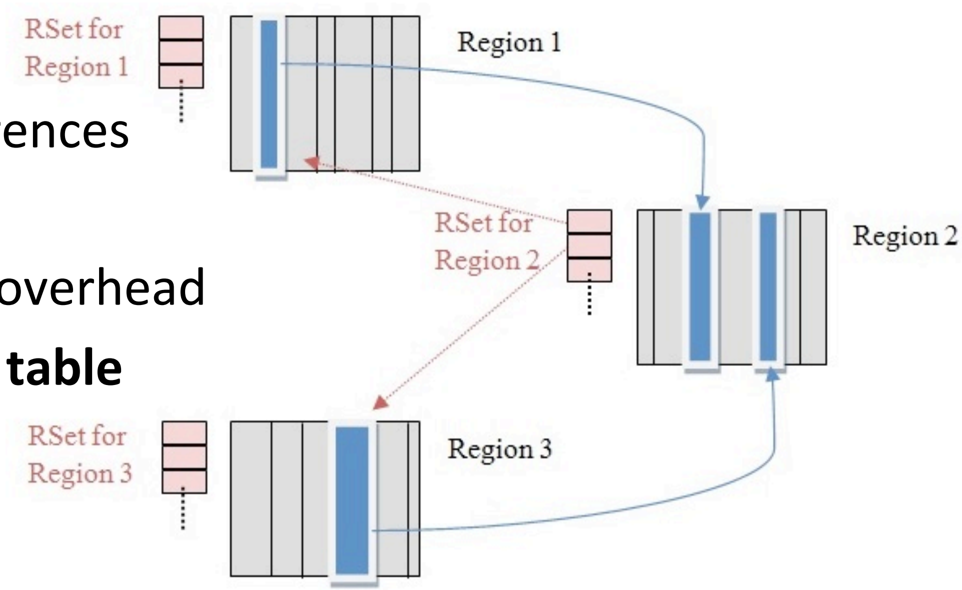
» **major speed-up** is that **fast copying** collection applied incrementally to tenured
  - requires more heap than parallel due to concurrent activities

» poor handling of larger objects (humongous objects)

» not NUMA aware

» proposed to be default in JVM 9

Garbage First 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 Collector – Minor and Mixed Collection

» stop-the-world approach with **parallel threads**

» **triggered** when no more allocation in Young regions possible

» **collection set** (CSet)

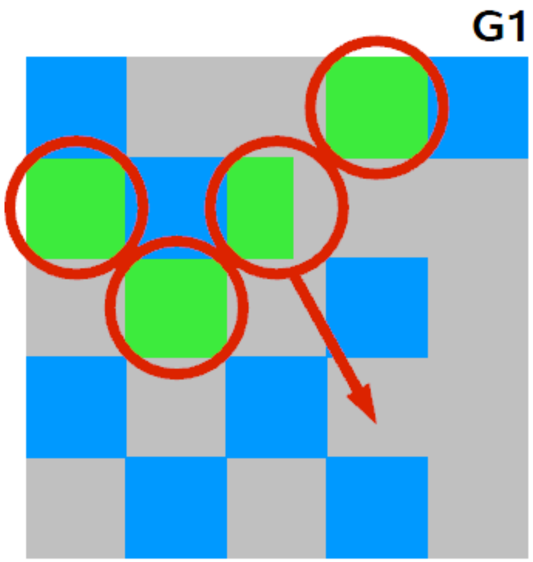
- eden and from survivor regions for pure **minor collection**
- eden, from survivor and *candidate tenured* regions for **mixed collection**

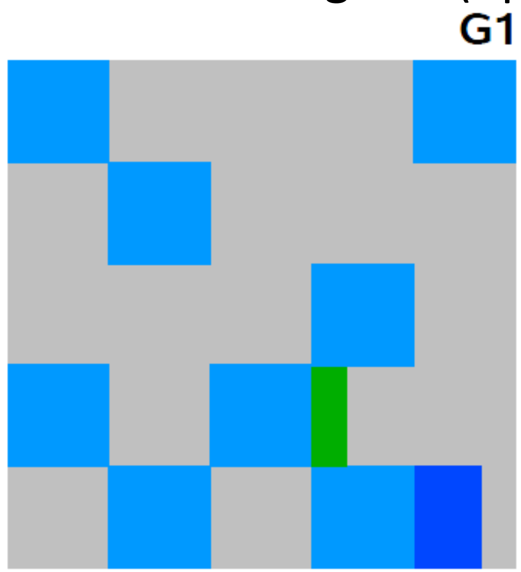
» reachable objects identified from roots + Rset for the regions + card table

» reachable objects are copied (from eden and survivor regions) into one or more new survivor regions

- using forwarding address with marking similar to parallel scavenge

» if aging threshold is met => promoted into tenured regions (optionally new)





Garbage First Collector – Concurrent Phase

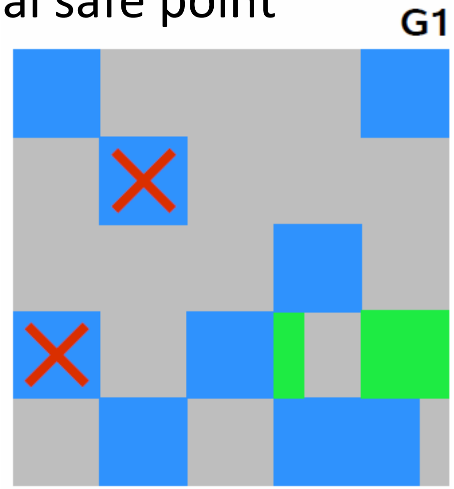
» **triggered by** heap occupancy percent (-XX:InitiatingHeapOccupancyPercent=45)

» outcomes

- **candidate tenured regions** with a lot of garbage for mixed collection
- cleanup completely empty tenured regions

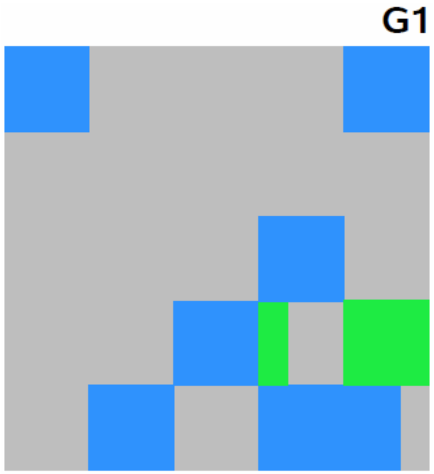
» **initial mark** – done right after minor collection utilizing global safe point

- snapshot-at-the-beginning (SATB)



» **concurrent** phases (-XX:ConcGCThreads=<n>)

- *scan roots* – minor GC is prohibited
  (if needed => global safe point)
- *marking* and *region-based statistics collection*
  - can be interrupted by minor GC
  - pre-write barrier keeps previous reference in SATB
- *re-marking* after minor GC and *final marking*
  - right after the next minor collection utilizing modifications in card tables
- *final output* (cleanup + candidates)



Garbage First Collector – Full Collection

- multiphase **full tracking with compact of all regions** during global safe point
- triggered by
  - **concurrent mode failure** – tenured fill-up before concurrent complete
    - increase heap, decrease trigger threshold, more concurrent threads
  - **promotion failure** – mixed collection but run-of space in tenured
    - 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)

Garbage First 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 a object trigger **expensive full collection**
- **reclamation** of non-reachable during (compacted during full collection only)
  - cleanup phase of concurrent cycle
  - full collection
- **debug** humongous allocations
  - -XX:+UnlockExperimentalVMOptions –XX:G1LogLevel=finest –XX:+PrintAdaptiveSizePolicy
  - use **Java Flight Recorder** in Java Mission Control
    - all allocations tracked in runtime routines like TLAB allocations

Garbage First Collection – Tuning Options ☺

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

Conclusion

Google

When is the best time to do a GC?

When nobody is looking.

> Using camera to track eye movement
> When subject looks away do a GC.