

PAUSELESS SCALA WITH SCALA-OFFHEAP


Denys Shabalin, LAMP/EPFL

[@den_sh](#)

ABOUT ME

- Born in Ukraine
- Research assistant at EPFL

- Author of [quasiquotes](#) in 2.11




[densh](#)

170 commits / 11,732 ++ / 4,597 --

#14

- Co-author of [scala.meta](#)




[densh](#)

114 commits / 46,284 ++ / 38,934 --

#2

- Author of [scala-offheap](#)



[densh](#)

267 commits / 18,190 ++ / 13,605 --

#1

ABOUT ME

Personal goal: exposing lower-level APIs in Scala to make writing predictable performance-sensitive code easy.

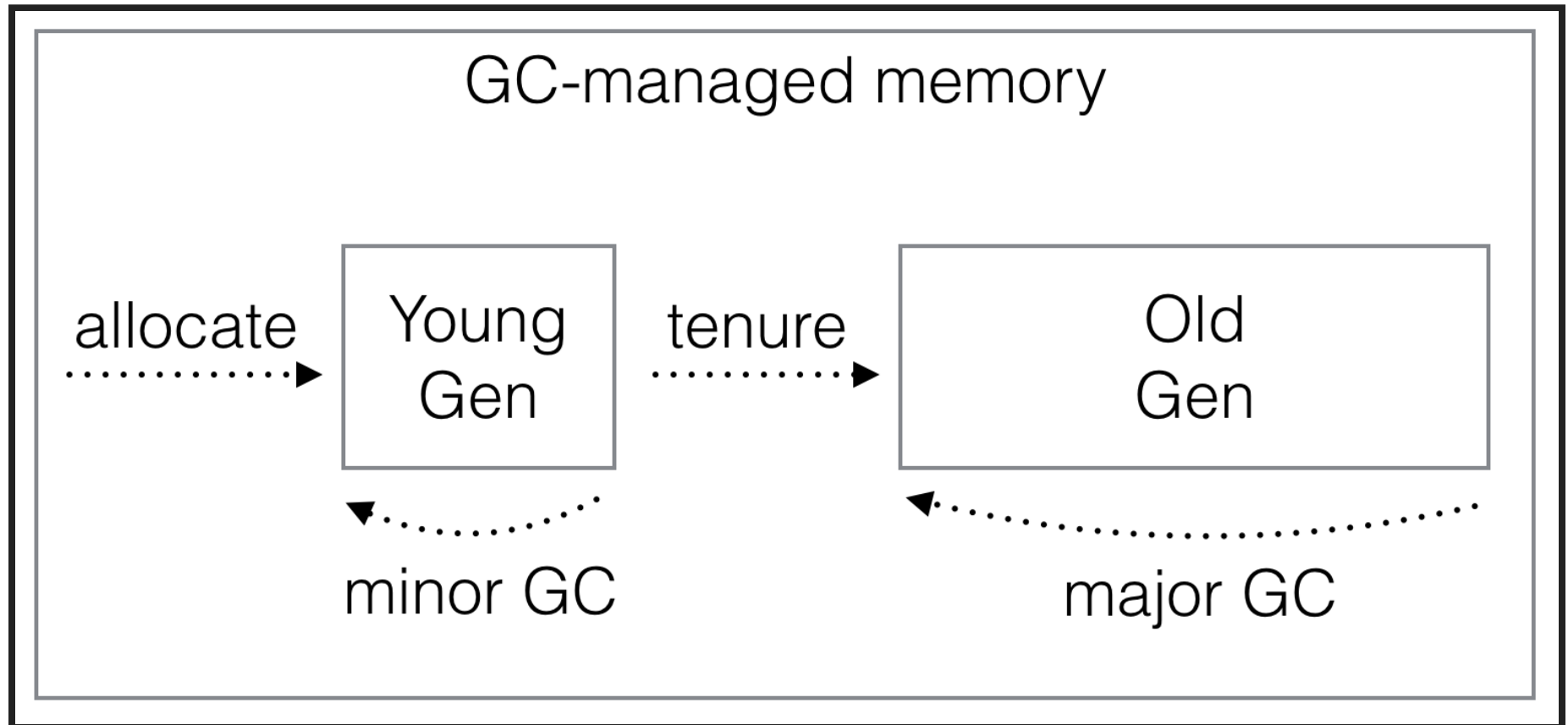
TODAY

1. Generational GCs overview
2. Avoiding garbage
3. Using off-heap memory

GENERATIONAL HYPOTHESIS

"Most object die young."

GENERATIONAL GC



(Grossly oversimplified illustration.)

SOURCES OF GARBAGE

1. Closures
2. Boxing
3. Implicit decorators
4. Varargs
5. Actual allocations

PUZZLE #1

```
val arr = Array.fill(100500)(99)  
arr.map(_ + 1).map(_ * 3)
```

PUZZLE #2

```
val arr = Array.fill(100500)(999999)  
arr.map(_ + 1).map(_ * 3)
```

PUZZLE #3

```
val arr = List.fill(100500)(999999)  
arr.map(_ + 1).map(_ * 3)
```

MEASURE, DON'T GUESS

- guessing is hard in the presence of optimising compilers.
- `jvisualvm` is a reasonable built-in tool
- plenty of (better) third-party alternatives

SOURCES OF GARBAGE

1. ~~Closures~~
2. ~~Boxing~~
3. ~~Implicit decorators~~
4. ~~Varargs~~
5. Actual allocations

TUNING VM FLAGS

Performance Options	
Option and Default Value	Description
-XX:+AggressiveOpts	Turn on point performance compiler optimizations that are expected to be default in upcoming releases. (Introduced in 5.0 update 6.)
-XX:CompileThreshold=10000	Number of method invocations/branches before compiling [-client: 1,500]
-XX:LargePageSizeInBytes=4m	Sets the large page size used for the Java heap. (Introduced in 1.4.0 update 1.) [amd64: 2m.]
-XX:MaxHeapFreeRatio=70	Maximum percentage of heap free after GC to avoid shrinking.
-XX:MaxNewSize=size	Maximum size of new generation (in bytes). Since 1.4, MaxNewSize is computed as a function of NewRatio. [1.3.1 Sparc: 32m; 1.3.1 x86: 2.5m.]
-XX:MaxPermSize=64m	Size of the Permanent Generation. [5.0 and newer: 64 bit VMs are scaled 30% larger; 1.4 amd64: 96m; 1.3.1 -client: 32m.]
-XX:MinHeapFreeRatio=40	Minimum percentage of heap free after GC to avoid expansion.
-XX:NewRatio=2	Ratio of old/new generation sizes. [Sparc -client: 8; x86 -server: 8; x86 -client: 12.]-client: 4 (1.3) 8 (1.3.1+), x86: 12]
-XX:NewSize=2m	Default size of new generation (in bytes) [5.0 and newer: 64 bit VMs are scaled 30% larger; x86: 1m; x86, 5.0 and older: 640k]
-XX:ReservedCodeCacheSize=32m	Reserved code cache size (in bytes) - maximum code cache size. [Solaris 64-bit, amd64, and -server x86: 2048m; in 1.5.0_06 and earlier, Solaris 64-bit and amd64: 1024m.]
-XX:SurvivorRatio=8	Ratio of eden/survivor space size [Solaris amd64: 6; Sparc in 1.3.1: 25; other Solaris platforms in 5.0 and earlier: 32]
-XX:TargetSurvivorRatio=50	Desired percentage of survivor space used after scavenge.
-XX:ThreadStackSize=512	Thread Stack Size (in Kbytes). (0 means use default stack size) [Sparc: 512; Solaris x86: 320 (was 256 prior in 5.0 and earlier); Sparc 64 bit: 1024; Linux amd64: 1024 (was 0 in 5.0 and earlier); all others 0.]
-XX:+UseBiasedLocking	Enable biased locking. For more details, see this tuning example . (Introduced in 5.0 update 6.) [5.0: false]
-XX:+UseFastAccessorMethods	Use optimized versions of Get<Primitive>Field.
-XX:-UseISM	Use Intimate Shared Memory. [Not accepted for non-Solaris platforms.] For details, see Intimate Shared Memory .

(Only a fraction of [vm flags](#) is shown here.)

TUNING VM FLAGS



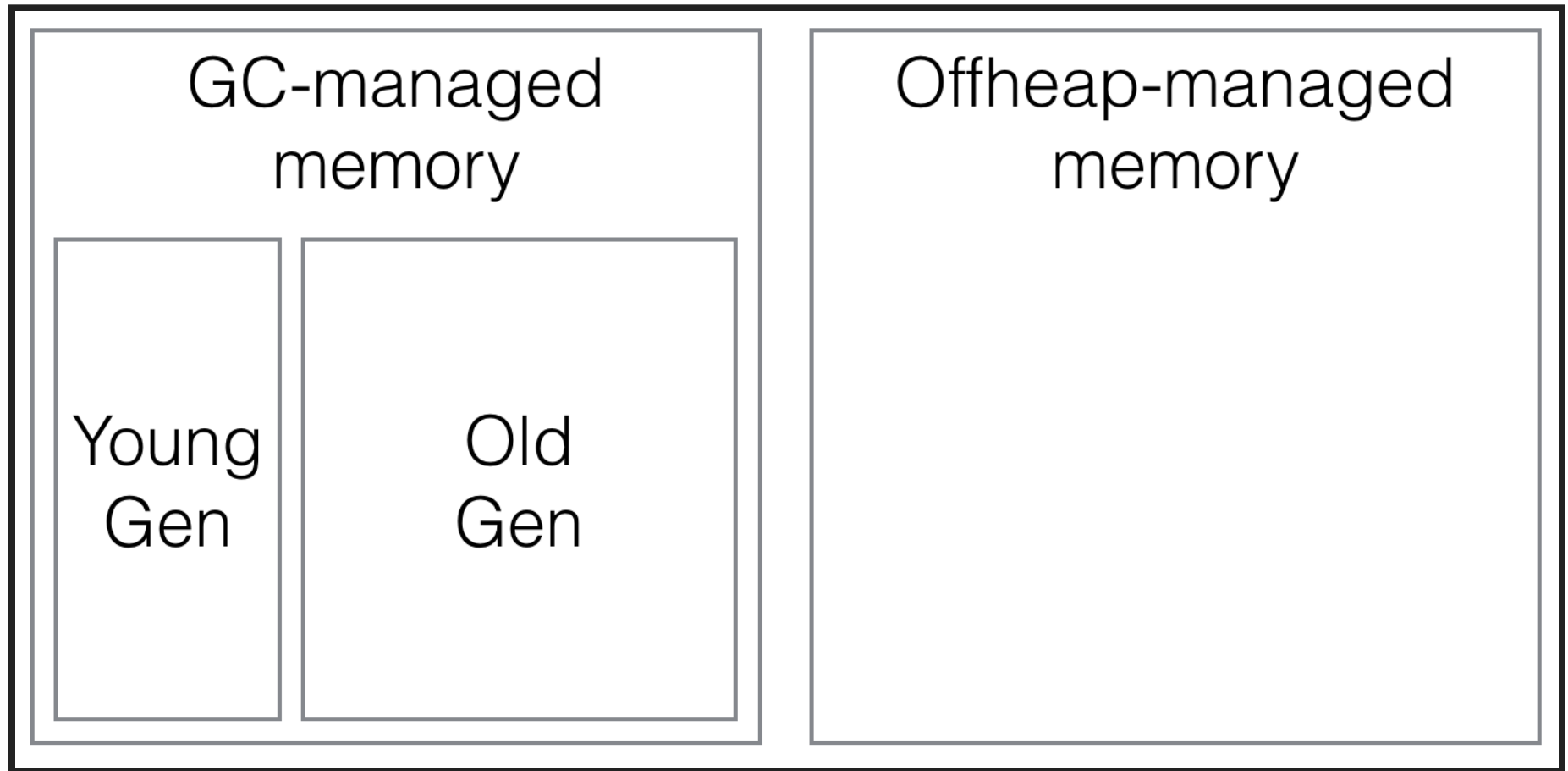
TUNNING VM FLAGS

- Requires understanding of VM internals
- Different GC implementations respond to them differently
- Brittle to changes as application evolves

SCALA-OFFHEAP

<https://github.com/densh/scala-offheap>

OFFHEAP MEMORY



OFFHEAP ALLOCATIONS

- `@data` class instances
- `@variant` class instances
- `Array[T]`

OFFHEAP ALLOCATORS

- `malloc`
- `Region { implicit r => ... }`

PUZZLE #4

```
import scala.offheap._

implicit val props =
  Region.Props(Pool(...))

Region { implicit r =>
  val arr = Array.fill(100500)(999999)

  arr.map(_ + 1).map(_ * 3)
}
```

SOURCES OF GARBAGE

1. ~~Closures~~
2. ~~Boxing~~
3. ~~Implicit decorators~~
4. ~~Varargs~~
5. ~~Actual allocations~~

NO GARBAGE, NO PAUSES



(Image courtesy of [B. Dehler](#))

SUMMARY

- Don't allocate garbage in performance-sensitive code.
- Measure, don't guess, you're probably wrong.
- Allocate domain-specific objects offheap.

QUESTIONS?