Memory-mapped Files

> Exceeding NIO Low-latency Capabilities
> Memory-mapped Files

Intro

Stream I/O has been improved by leveraging NIO; however, a Memory-mapped File's data access is dramatically faster (by one to two orders of magnitude).

Although Memory-mapped Write operations seem to utilize an instance of `FileOutputStream`, in actuality all of a Memory-mapped File's output uses an instance of `RandomAccessFile`.

The demo application being discussed in this document (MappedIO) is running in Java 11, it has a side benefit of proving the Java 11 JVM's viability.

Because Memory-mapped Files are related to the Java Memory Model and to GC, these topics are covered in some detail, prior to their application in the demo app.

The MappedIO demo application is deceptively straightforward, so the underlying concepts will foster a better understanding of the application's findings.

Application Overview

The `start()` *Template Method* in this class iterates through various implementations of the `conductTest()` method, providing a test battery, as defined in the several anonymous inner classes.

Each `TestModule` instance performs one kind of I/O test. The formulation of each of the `conductTest()` methods measures the relative performance of the two I/O approaches. There is a test for both read and write modalities:

Sample output

Stream Write:	36
Mapped Write:	63
Stream Read:	81
Mapped Read:	11
Stream Read/Write:	1673
Mapped Read/Write:	18

Note that the implementation of each `conductTest()` method includes *Wall Time* (this includes the time expended for the initialization of the various I/O objects .

After accounting for scaffolding activities – Memory-mapped Files can be expensive to initialize – the overall throughput gain realized by adopting Memory-mapped Files as compared to standard NIO Streams is significant.

There is also some memory overhead to consider when looking at the effects on the heap in the aftermath of any GC cycle. For instance, you must factor-in *Wired/Resident Memory*, which is reserved for kernel code and for Operating System data structures.

Garbage Collection

Because Memory-mapped Files are intimately related to a JVM's Heap, this this demo application allows you to conditionally monitor GC cycles as they occur.

Allowing GC cycles to be reported (they are always running) also messes with the presentation (the console-based representation) of the application's test results. Additionally, printing to console (the `System.out` stream) incurs a time expense/distortion.

For these reasons, the user can elect to publish the GC cycles, via a CLI switch for the ßash script named gc, shown next:

Conditionally enabling GC...

```
#!/bin/bash -p

if [[ $1 == *"t"* ]]; then
  export GC="true"
else
  export GC="false"
fi

clear; echo; export; echo

~/stage/memorymappedfile
```

The (12-Factor aware) ßash script named (ok), shown next, recognizes the value stored in the `$GC` environment variable, whose value is toggled by the (gc) ßash script that was shown above.

Conditionally-enabling GC - 12-Factor App

```
#!/bin/bash -p

echo;echo
krush=$( top -l 1 | grep PhysMem | sed 's/, /n /g' )
echo $krush;echo


if [[ $GC != "true" ]]; then
  java -jar ./target/memorymapped.jar $@

else
  java -verbosegc -jar ./target/memorymapped.jar $@
  # java -verbosegc -Xlog:gc* -jar ./target/memorymapped.jar $@
  # java -verbosegc -Xlog:gc* -XX:BiasedLockingStartupDelay=0 -jar ./target/memorymapped.jar $@
fi


~/stage/memorymappedfile cls; echo;echo;cat ok; echo
```

Garbage Collection – a deep-dive

As part of this section – aside from discussing GC in the MappedIO demo application – the topic of Java 11-12 JVMs' support for Compressed Oops, the Virt-zero address, and Colored 64-bit Pointers (an unfortunate naming choice) are explored, because the demo is running JVM 11.

Those concepts relate to the Java Memory Model and to GC (the ZGC garbage collector).

1/3 The App and GC

The MappedIO demo application conditionally allows reporting of (semi) verbose GC activities to be presented alongside the application-oriented readout. Because it is delivered as another thread of execution, the gc statements "crash" into the `System` class' `out/err` Streams (console output).

Below is a representation of a typical (console) gc cycle, *running under the Java 11 JVM*.

The console representation, by the way, depicts a minor GC, as specified by the following canonical CLI switch:

```
-verbosegc
```

```
[0.008s][info   ][gc,heap] Heap region size: 1M
[0.010s][info   ][gc     ] Using G1
[0.010s][info   ][gc,heap,coops] Heap address: 0x0000000700000000, size: 4096 MB, Compressed Oops mode: Zero based, Oop shift amount: 3
```

Breakdown

[0.008s]	[info]	[gc,heap]	Heap region size: 1M
[0.010s]	[info]	[gc]	Using G1
[0.010s]	[info]	[gc,heap,coops]	Heap address: 0x0000000700000000, size: 4096 MB, Compressed Oops mode: Zero based, Oop shift amount: 3

2/3 Virt-zero Address

If the OS supports it (post Java 6 does), heap allocation can begin at a virt-zero address – then, Compressed Oops Mode (COop) can be used. COop eliminates the need to add to the base heap address when decoding a 64-bit pointer.

With COop (on a 64-bit architecture) decoding a 64-bit pointer starts with a 32-bit *object* offset, and uses a (3) byte offset as an alignment (the Oop shift amount shown in the **1/3 The App and GC** section, above).

Since a 64-bit pointer uses only 42 bits – and the 3-byte offset, the Oop shift amount – the extra bits are used for GC metadata (in a version 11/12 JVM).

Because they're object offsets rather than byte offsets, a 64-bit pointer can address up to four billion *objects* (not bytes) – this equates to a heap size of up to about 32 gigabytes.

Using the COop scheme within an ILP64 architecture, the size of 64-bit pointers are *comparable* to pointers used in ILP32 mode.

To use COop, pointers must be scaled by a factor of 8 and added to the Java heap base address to find the object to which they refer.

The computational savings – of decoding/deriving an effective 64-bit native address (pointer) from a 32-bit COop (compressed) address requires the OS being capable of leveraging a virt-zero address:

```
[0.008s]  [info]  [gc,heap]         Heap region size: 1M
[0.010s]  [info]  [gc]              Using G1
[0.010s]  [info]  [gc,heap,coops]   Heap address: 0x0000000700000000,
                                    size: 4096 MB,
                                    Compressed Oops mode: Zero based,
                                    Oop shift amount: 3
```

3/3 Colored 64-bit Pointers

ZGC is the latest (low-latency, 10-ms) garbage collector. It trades disk usage for speed. It uses 64-bit colored pointers. It supports varying sized JVMs. *It is experimental.*

Recall that a 64-bit pointer has extra, unused bits (only 42 bits are used).

Hence, the 11-12 versions of the JVM appropriate the extra bits to provide the concept of colored pointers (an unfortunate name). Colored pointers utilize the extra memory (bits) to model the GC-related meta-properties that are listed (in high-order, to low-order bits) below:

```
finalizable
remapped
marked 1
marked 0
```

Marked bits are used for the swapping that occurs between the two Survivor GC spaces

---

Note about Javadoc Generation

When running w/ Java 11, the Javadoc Tool no longer supports links that toggle the familiar frames/noframes feature.

Instead, it creates a Search box (which appears in the upper-right corner of all generated Javadoc pages).

ref: https://bugs.openjdk.java.net/browse/JDK-8215599

Below is a "garden" example of the CLI way to generate Javadoc (Lines 10-12, 14-17).

A Portable CLI Javadoc Generation Script

```
#!/bin/bash -p

# Detect the source directory of a ßash script from within the script itself
# https://bit.ly/2NokAsM

# Running Javadoc
# https://bit.ly/31XVxSr


ORIGIN="$( cd "$( dirname "${BASH_SOURCE[0]}" )" >/dev/null 2>&1 && pwd )"

cd $ORIGIN/src/main/java

javadoc -private -d $ORIGIN/html \
-windowtitle 'Memory-mapped Files, Demo' \
-doctitle 'Memory-mapped Files, Demo' \
org.ascension.util.memorymappedfile.demo

# open $ORIGIN/html/org/ascension/util/memorymappedfile/demo/MappedIO.html

cd $ORIGIN


# NB
# Under Java 11, the Javadoc tool no longer supports links that
# toggle 'frames/noframes'; instead, it supports a 'Search' box,
# located in the upper-right corner to replace the links.
#
#   ref:  https://bugs.openjdk.java.net/browse/JDK-8215599

~/stage/memorymappedfile
```

Application Details

The MappedIO application contrasts two approaches to reading and writing data (files). One approach is streaming using standard NIO; the other approach leverages Memory-mapped Files.

This app shows that the improvement in throughput for Memory-mapped Files over NIO Streams varies from one order of magnitude (for reads) to two orders of magnitude (for a read, then a write).

Besides realizing better throughput, Memory-mapped Files allow your application to access (to page-in) more data than you have RAM.

Additionally – synchronization costs aside – Memory-mapped Files allow multiple threads to share the (paged-in) data. Memory-mapped Files can rapidly access (page-in) other non-resident parts of the file.

The Memory-mapped arrangement, from a throughput standpoint, is a competitive alternative to using Event-sourcing/Ring Buffers, which requires an Event-Processing architecture topology.

Referring to Figure One below, (Lines 75, 84) establish the capacity of the data which is to either be read and/or to be written. These are the three use cases being illustrated by the app.

Note that the write capacity (Line 75) is twice that of the read (capacity, Line 84). This provision avoids incurring an `java.nio.BufferOverflowException` at runtime.

The actual value of the write capacity variable is deliberately tagged close to the disk allocation value of the test file. This deliberate sizing level-sets the relative performance of all six (6) tests.

This consideration eliminates any testing bias due to the fact that some tests only write data, some tests only read data (from the test data file), and some test do both.

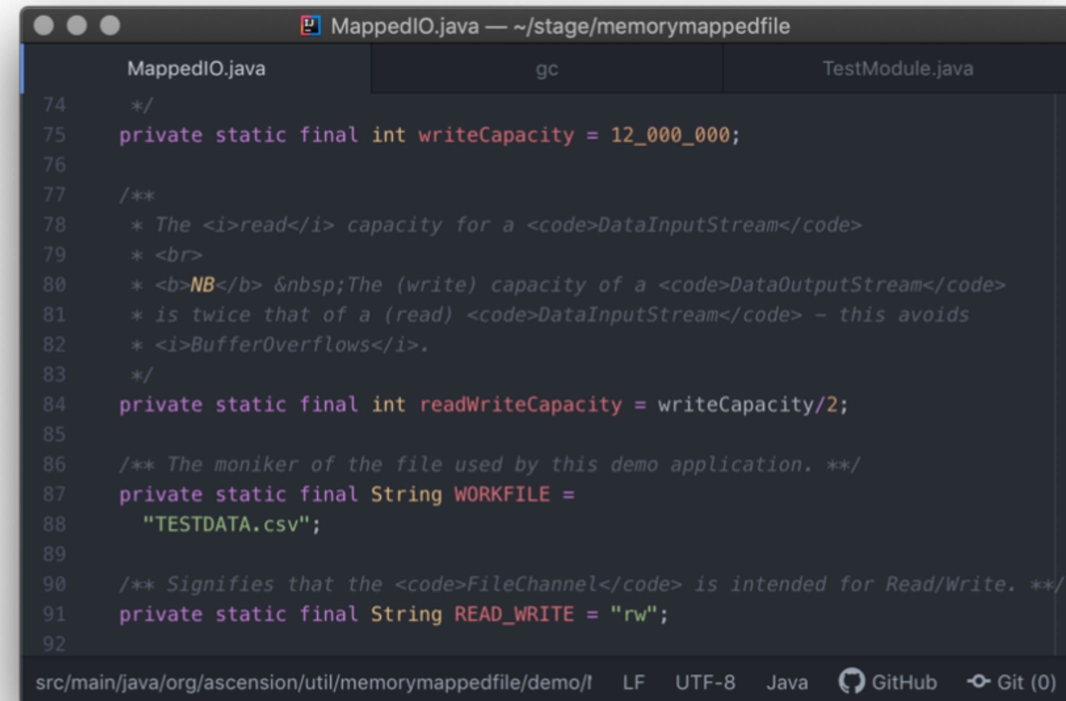


Figure One

MappedIO.java — ~/stage/memorymappedfile

MappedIO.java | TestModule.java

```java

/** ßattery of tests. **/
private static TestModule[] testBattery = {

  new TestModule(
      "\n\n   [0] Stream Write\n           Capacity  " +
      writeCapacity/1_000_000 + " Miß") {

    public void conductTest() {

      try (

        DataOutputStream dataOutStream =
          new DataOutputStream(
            new BufferedOutputStream(
              new FileOutputStream(
                new File(WORKFILE))))) {

        for (int i = 0; i < writeCapacity; i++) {
          dataOutStream.writeChar(i);
        }
      }
      catch (FileNotFoundException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
    }
  },
  new TestModule(
      "\n   [1] Mapped Write\n           Capacity  " +
      writeCapacity/1_000_000 + " Miß") {

    public void conductTest() {

      try (

        FileChannel fileChannel =
          new RandomAccessFile(WORKFILE, READ_WRITE).
            getChannel()) {

        CharBuffer charBuffer = fileChannel.map(
          FileChannel.MapMode.READ_WRITE, 0, fileChannel.size()).
            asCharBuffer();

        for (int i = 0; i < writeCapacity; i++) {
          charBuffer.put(Character.valueOf((char)i));
        }
      }
      catch (FileNotFoundException e) { e.printStackTrace(); }
      catch (IOException e) { e.printStackTrace(); }
    }
```

src/main/java/org/ascension/util/memorymappedfile/demo   LF   UTF-8   Java   GitHub   Git (0)

Figure Two

Line 113 in the screen shot above depicts a collection – a `static Object` array of type:

`org.ascension.util.memorymappedfile.demo.TestModule.java`

Each of the successive `TestModule` instances are required to implement the `abstract conductTest()` method. The `conductTest()` method is where each instance fulfills their respective test objectives.

The `TestModule` instances are launched inside a `for(...)` loop, within the MappedIO class.

On lines 121-7, 143-7 the canonical `try-with-resources` feature instantiates the particular data structure being utilized for the specific test.

The first code block (Lines 121-7) instantiates a `DataOutputStream`.

The second code block (Lines 143-7) wraps an instance of (NIO) `java.io.RandomAccessFile` with an instance of `java.nio.channels.FileChannel`.

These two test modules perform "writes" (with identical capacity) – later, the values of their respective throughput characteristics are compared/reported.

This paring of test modules is repeated two more times: a. Reads; b. Reads/Writes.

The Read-Write scenario is the most realistic/common use case for streaming NIO, and for Memory-mapped Files.

The Read-Write scenario shows the most divergence in performance between the two approaches – two orders of magnitude.

MappedIO.java — ~/stage/memorymappedfile

MappedIO.java | TestModule.java

```java
278     System.err.println("Signal " + signal);
279
280     System.err.println("  ... ");
281
282     for (int i = 0; i < testBattery.length; i++) {
283
284       if (signal.toUpperCase().contains("BOTH")) {
285
286         long result = testBattery[i].start();
287         results.add(result);
288         System.err.println(result + " ms.");
289       }
290       else if (signal.toUpperCase().contains("EVEN") && i % 2 == 0) {
291
292         long result = testBattery[i].start();
293         results.add(result);
294         System.err.println(result + " ms.");
295       }
296       else if (signal.toUpperCase().contains("ODD") && i % 2 != 0) {
297
298         long result = testBattery[i].start();
299         results.add(result);
300         System.err.println(result + " ms.");
301       }
302     }
303
304     summarize();
305   }
306
```

src/main/java/org/ascension/util/memorymappedfile/dem LF UTF-8 Java GitHub Git (0)

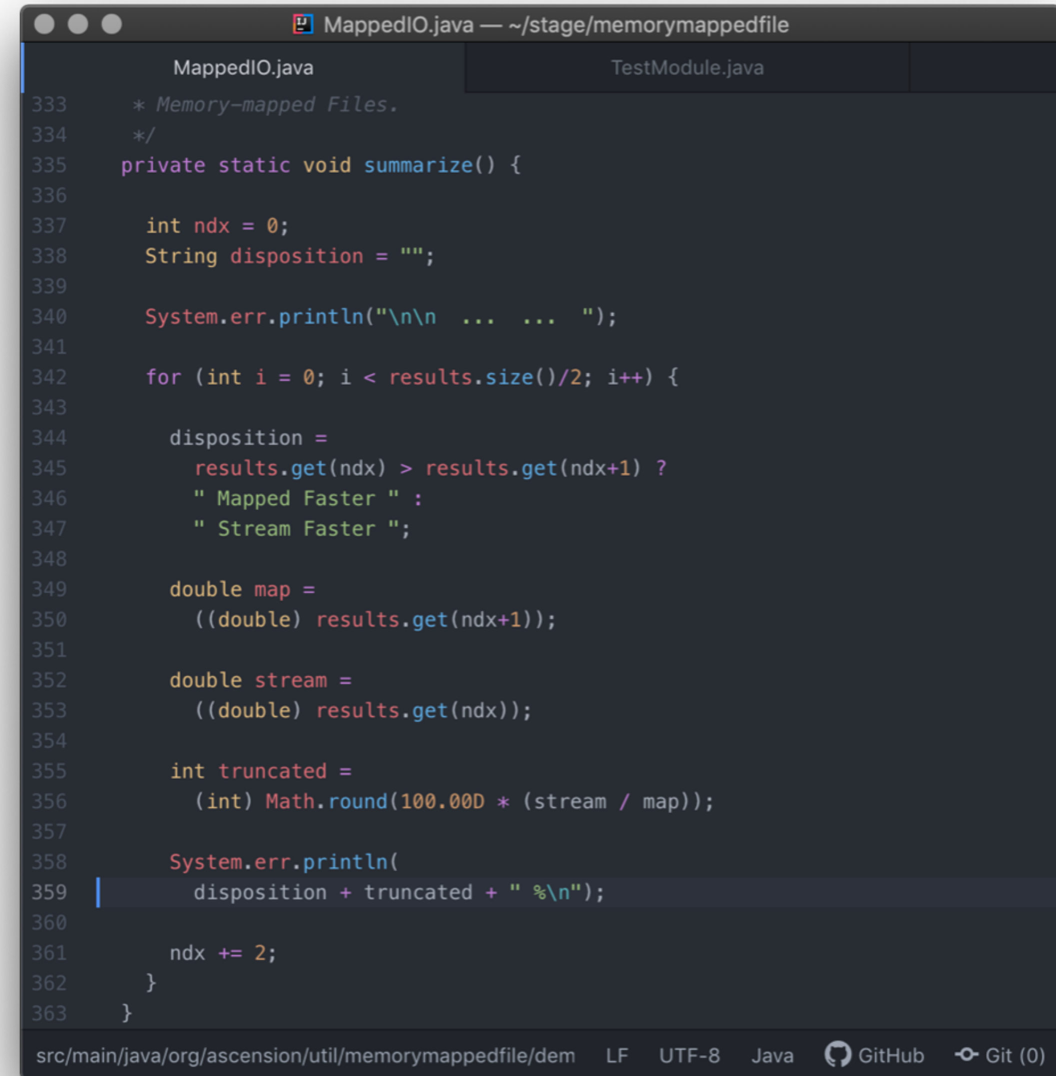
Figure Three

Above, the filtering logic that determines which tests are run (according to CLI input) is shown.

The modulo arithmetic – with a divisor of 2 – determines which pair of tests are executed.
If you specify nothing, all six tests are executed.

The `summarize()` method (Line 304) produces a report of the performance differential (as a percentage).

Filtering allows for the elimination of distortion
due to GC which may occur and incur CPU cycles




```java
     * Memory-mapped Files.
     */
    private static void summarize() {

      int ndx = 0;
      String disposition = "";

      System.err.println("\n\n  ...   ...   ");

      for (int i = 0; i < results.size()/2; i++) {

        disposition =
          results.get(ndx) > results.get(ndx+1) ?
          " Mapped Faster " :
          " Stream Faster ";

        double map =
          ((double) results.get(ndx+1));

        double stream =
          ((double) results.get(ndx));

        int truncated =
          (int) Math.round(100.00D * (stream / map));

        System.err.println(
          disposition + truncated + " %\n");

        ndx += 2;
      }
    }
```

Figure Four

Above, the logic that compiles the comparison of paired test results.

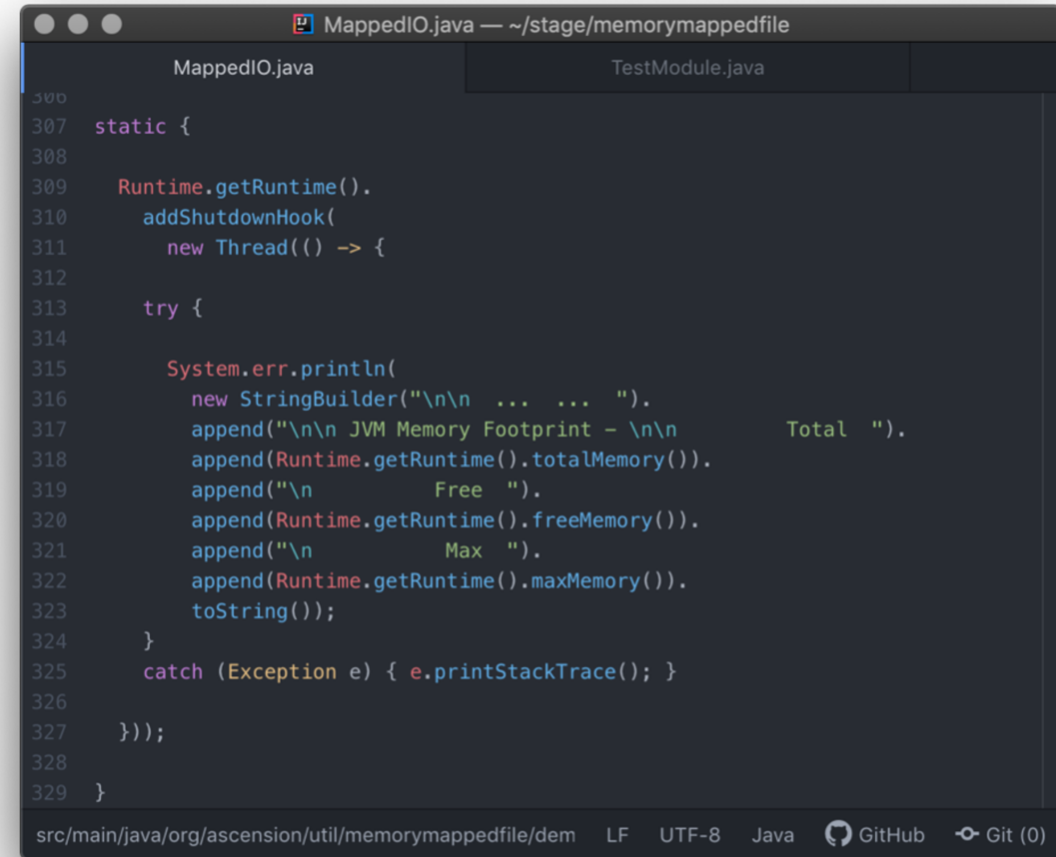


Figure Five

This screen shot shows a canonical Java `ShutdownHook`, containing logic that reports the JVM's memory footprint at the end of the application's execution.

Application, in-flights

```
memorymappedfile
[INFO] Finished at: 2019-11-07T15:24:12-06:00
[INFO] ------------------------------------------------------------------------


PhysMem: 15G used (2624M wired)n 1095M unused.

Signal BOTH
  ...


   [0] Stream Write
           Capacity  12 Miß   256 ms.

   [1] Mapped Write
           Capacity  12 Miß   76 ms.


   [2]  Stream Read
           Capacity   6 Miß   248 ms.

   [3]  Mapped Read
           Capacity   6 Miß   10 ms.


   [4] Stream Read/Write
           Capacity   6 Miß   39543 ms.

   [5] Mapped Read/Write
           Capacity   6 Miß   127 ms.


  ...  ...
 Mapped Faster 337 %

 Mapped Faster 2480 %

 Mapped Faster 31136 %


  ...  ...

 JVM Memory Footprint -

         Total  298319872
          Free  288025184
           Max  3817865216
```

Outro

The treatment of the Memory-mapped Files (using the MappedIO app as an example) covers GC topics and discusses salient parts of the Java Language's Memory-model.

The best part is that, as Java evolves, developers benefit from the underlying improvements with no attendant refactoring costs to the code; however, with a deep understanding of improvements, they can design more performant applications.

Appendix

https://mechanical-sympathy.blogspot.com/2011/11/biased-locking-osr-and-benchmarking-fun.html

https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html#compressedOop

https://www.baeldung.com/jvm-compressed-oops

https://www.baeldung.com/java-verbose-gc

https://codeahoy.com/2017/08/06/basics-of-java-garbage-collection/

https://www.opsian.com/blog/javas-new-zgc-is-very-exciting/

http://www.diva-portal.org/smash/get/diva2:754541/FULLTEXT01.pdf

https://subscription.packtpub.com/book/application_development/9781789133271/8/ch08lvl1sec45/colored-pointers