OJXPerf: Pinpointing Object-level Redundancies in Java Programs

Anonymous Author(s)

ABSTRACT

This document provides the supplementary materials for the PLDI 2020 submission. Iit shows the overall overhead analysis. Then this document describes the optimizations under the guidance of OJXPerf for JGFSerialBench [2] and RoaringBitmap [1].

1 CASE STUDIES

1.1 JGFSerialBench

JGFSerialBench is a benchmark of Java Grande benchmark suite, which are designed to be representative of scientific and other numerically intensive computation [2]. We run JGFSerialBench using its default input size. After profiling JGFSerialBench, one object, `arraybase`, which is allocated at line 371 in method `JGFrun` of class `JGFSerialBench` stood out, as shown in Listing 1. Listing 1 shows the allocation site of this `arraybase` is in a while loop. In each loop, the program creates a new `arraybase` and puts some contents info this `arraybase`. As this `arraybase` is a two-dimensional array, `arraybase` uses writeObject method (line 375) to put its one-dimensional elements (can be seen as an object) into an object `arrayout`. OJXPerf reports that the redundancy factor θ of the created `arraybase` is 68.8%, and its lower bound factor ω indicates that the largest identical `arraybase` group size is larger than 15.2%. To avoid such redundant object copy operations, we redesign the code as we first check whether the program has a different `arraybase` compared with the one from the previous iteration. If the `arraybase` keeps the same, then copy of `arraybase`'s contents is unnecessary. This optimization yields a $(1.1 \pm 0.03)\times$ speedup to the entire program.

```
368 public void JGFrun(){
369   ...
370   while  (time < TARGETTIME && size < MAXSIZE){
371     ▶arraybase = new arrayitem [size][LENGTH];
372     arrayout = new ObjectOutputStream(arrayfout);
373     ...  //put contents in array base
374     for (j=0; j<size; j++) {
375       ▶arrayout.writeObject( arraybase[j]);
376     }
377   }
378   ...
379 }
```

Listing 1: OJXPerf pinpoints the `arraybase` array suffering from object-level redundancy in JGFSerialBench

```
284 public void deserialize(DataInput in) {
285   ...
286   final short keys[] = new short[this.size];
287   for (int k = 0; k < this.size; ++k) {
288     keys[k] = Short.reverseBytes(in.readShort());
289     ...
290     ▶this.keys[k] = keys[k];
291   }
292   ...
293 }
```

Listing 2: The source code highlighed by OJXPerf shows the `keys` array suffering from object-level redundancy in RoaringBitmap.

1.2 RoaringBitmap

Roaring bitmap are compressed bitmaps which tend to outperform conventional compressed bitmaps such as WAH, EWAH or Concise [1]. We run RoaringBitmap using SerializationBenchmark as input. OJXPerf reports an object, array `keys`, which is accessed at line 290 in method `deserialize` of class `MutableRoaringArray`, as shown in Listing 2. In this `deserialize` method, the program declares a short array `keys` (line 286) and fills every slots of this array (line 288). For each iteration, after the program obtained the value of `keys[k]`, it will update the class member array `keys[k]`'s value at end of the loop. OJXPerf reports that the array `keys` always equals with the array `keys` from the last iteration (redundancy factor θ is 100%), which means that the method `deserialize` often generates the exactly same array `keys` comparing with the one from the last call of this method. We redesign the code as instead of updating the array `keys` value at end of every loop, the program only updates the array `keys` outside this loop using the `Arrays.copyOf` method if the `deserialize` method generates a different array `keys`. The score's unit used in RoaringBitmap is microseconds per operation (throughput). While this simple fix did not lead to a significant running time reduction (only 2.1%), the optimization increases the throughput by $(1.09 \pm 0.01)\times$.

REFERENCES

[1] RoaringBitmap authors. 2019. RoaringBitmap. https://github.com/RoaringBitmap/RoaringBitmap.

[2] Mark Bull, Lorna Smith, Martin Westhead, David Henty, and Robert Davey. 2001. Java Grande benchmark suite. https://www.epcc.ed.ac.uk/research/computing/performance-characterisation-and-benchmarking/java-grande-benchmark-suite.