An Object-centric Profiler for Java

Anonymous Author(s)

ABSTRACT

This document provides the supplementary materials for the ESEC/FSE 2020 submission. First, it shows the overall overhead analysis. Then this document shows the accuracy analysis, which the existing issues reported by prior work [6] (four DaCapo 2006 benchamrks [4] luindex, bloat, lusearch, xalan, and SPECJbb2000 [1]), are also detected by DJXPerf.

1 OVERALL OVERHEAD ANALYSIS

In figure 1, we run DJXPerf with Renaissance benchmark suite [5], Dacapo2006 [4], Dacapo 9.12 [2], and SPECjvm2008 [3] with 5M sampling period. The figure 1 shows that DJXPerf typically incurs 5% runtime and 5% memory overhead.

2 ACCURACY ANALYSIS

2.1 DaCapo 2006 luindex

luindex uses lucene to indexes a set of documents: the works of Shakespeare and the King James Bible [4]. DJXPerf reports a problematic object—the `Posting` array—allocated at line 235 in method `sortPostingTable` of class `DocumentWriter` reported by prior work [6], shown in Listing 1, which accounts for 23.3% of total cache misses. We followed the proposed optimizations to fix this issue.

```
234 private final Posting[] sortPostingTable() {
235   ▶Posting[] array = new Posting[postingTable.size()];
236   ...
237   quickSort(array, ...);
238   return array;
239 }
```

Listing 1: DaCapo 2006 luindex: The hotspot object array (line 235) which suffers from memory bloat problem.

2.2 DaCapo 2006 bloat

bloat performs a number of optimizations and analysis on Java bytecode files [4]. DJXPerf reports two problematic objects allocated at line 86 and 91 in the constructor of class `SSAConstructionInfo`: the `LinkedList` object `reals` and the `PhiStmt` object `phis`, shown in Listing 2. Prior work [6] did not give the exact source code location they fixed and only mentioned that the issues came from the pervasive created objects in visitor patterns. DJXPerf detected such visitor patterns in Figure 2. By checking the calling context in Figure 2, we found that the bloat visits a graph iteratively by creating many visitor objects in nested loops. At the end of the calling context, the program gets into the two problematic objects, the `LinkedList` object `reals` and the `PhiStmt` object `phis`, which accounts for 13.6% of total cache misses. To address the problem, we created these two objects outside the constructor `SSAConstructionInfo`. This optimization yields a $(1.07 \pm 0.02)\times$ speedup.

```
...
EDU.purdue.cs.bloat.cfg.FlowGraph.visit(FlowGraph.java:2249)
 EDU.purdue.cs.bloat.tree.TreeVisitor.visitFlowGraph(TreeVisitor.java:94)
  EDU.purdue.cs.bloat.cfg.FlowGraph.visitChildren(FlowGraph.java:2235)
   EDU.purdue.cs.bloat.cfg.Block.visit(Block.java:167)
    EDU.purdue.cs.bloat.tree.TreeVisitor.visitBlock(TreeVisitor.java:99)
     EDU.purdue.cs.bloat.cfg.Block.visitChildren(Block.java:162)
      EDU.purdue.cs.bloat.tree.Tree.visit(Tree.java:3243)
       EDU.purdue.cs.bloat.ssa.SSA.visitTree(SSA.java:110)
        EDU.purdue.cs.bloat.tree.IfZeroStmt.visit(IfZeroStmt.java:78)
         EDU...TreeVisitor.visitIfZeroStmt(TreeVisitor.java:124)
         ...
          EDU...ssa.SSAConstructionInfo(SSAConstructionInfo.java:86)
          EDU...ssa.SSAConstructionInfo(SSAConstructionInfo.java:91)
```

Figure 2: DaCapo 2006 bloat: The hotspot call path to the allocation site line 86 and 91 in SSAConstructionInfo.java.

2.3 DaCapo 2006 lusearch

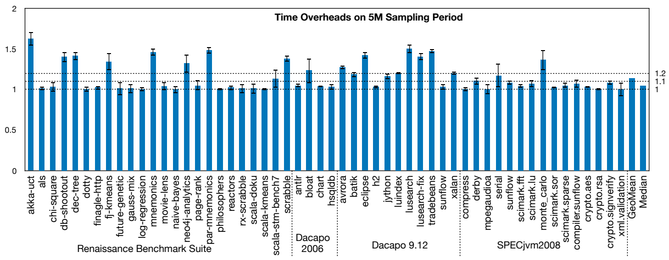
lusearch uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible [4]. DJXPerf reports a problematic object—the `QueryParser` object—allocated at line 119 in method `parse` of class `QueryParser` reported by prior work [6], shown in Listing 3, which accounts for 9.2% of total cache misses. To address the problem, we pulled this allocation site out of the method `parse`. We followed the proposed optimizations to fix this issue.

2.4 DaCapo 2006 xalan

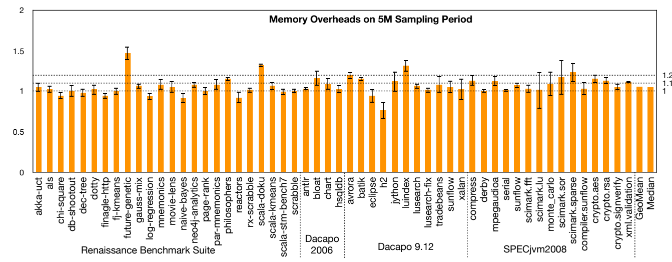
xalan transforms XML documents into HTML [4]. DJXPerf reports a problematic object—the `Transformer` object—allocated at line 100 in method `run` of class `XSLTBench` reported by prior work [6], shown in Listing 4, which accounts for 16.7% of total cache misses. We followed the proposed optimizations to fix this issue.

2.5 SPECJbb2000

SPECjbb2000 is SPEC's first benchmark for evaluating the performance of server-side Java [1]. DJXPerf reports a problematic object—the `Hashtable` object—allocated at line 173 in method `process` of class `StockLevelTransaction` as shown in Listing 5, which accounts for 4.7% of total cache misses. To address the problem, we pulled this allocation site out of the method `process`. This optimization increases the overall throughput by $(1.02 \pm 0.01)\times$ and no running time speedup.



(a) Runtime overheads.



(b) Memory overheads.

Figure 1: DJXPerf's runtime and memory overheads in the unit of times (×) on various benchmarks.

```
84 public SSAConstructionInfo(FlowGraph cfg, VarExpr expr) {
85   ...
86   ▶reals = new LinkedList[cfg.size()];
87   allReals = new LinkedList();
88
89   defBlocks = new HashSet();
90
91   ▶phis = new PhiStmt[cfg.size()];
92 }
```

Listing 2: DaCapo 2006 bloat: The hotspot object reals and phis (line 86 and 91) which suffer from memory bloat problem.

```
118 static public Query parse(String query, String field,
        Analyzer analyzer) {
119   ▶QueryParser parser = new QueryParser(field,analyzer);
120   return parser.parse(query);
121 }
```

Listing 3: DaCapo 2006 lusearch: The hotspot object parser (line 119) which suffers from memory bloat problem.

```
96 public void run() {
97   ...
98   while (true) {
99     ...
100    ▶Transformer transformer=_template.newTransformer();
101    ...
102  }
103  ...
104 }
```

Listing 4: DaCapo 2006 xalan: The hotspot object transformer (line 100) which suffers from memory bloat problem.

```
171 boolean process() {
172   ...
173   ▶Hashtable stockList = new Hashtable(200);
174   ...
175 }
```

Listing 5: SPECJbb2000: The hotspot object stockList (line 173) which suffers from memory bloat problem.

REFERENCES

[1] 2000. SPEC JBB2000. https://www.spec.org/jbb2000/.

[2] 2018. DaCapo Benchmark Suite 9.12. https://sourceforge.net/projects/dacapobench/files/9.12-bach-MR1/.

[3] Standard Performance Evaluation Corporation. 2008. SPECjvm2008 benchmark suite. https://www.spec.org/jvm2008.

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: java benchmarking development and analysis. OOPSLA '06 Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, 169–190.

[5] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. ACM, New York, NY, USA, 31–47. https://doi.org/10.1145/3314221.3314637

[6] Guoqing Xu. 2012. Finding reusable data structures. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1017–1034.