An Object-centric Profiler for Java

Anonymous Author(s)

ABSTRACT

This document provides the supplementary materials for the PPoPP 2020 submission. First, it shows the overall overhead analysis. Then this document shows the accuracy analysis, which the existing issues reported by prior work [10] (four DaCapo 2006 benchamrks [7] luindex, bloat, lusearch, xalan, and SPECJbb2000 [1]), are also detected by DJXPerf. Finally, this document describes the optimizations under the guidance of DJXPerf for Cache2k 1.2.0 [9], Apache SAMOA [5], Apache Commons Collections [6] and Java Grande Benchmark JGFMolDynBench [3] in the Table 1 of the submitted paper.

1 OVERALL OVERHEAD ANALYSIS

In figure 1, we run DJXPerf with Renaissance benchmark suite [8], Dacapo2006 [7], Dacapo 9.12 [2], and SPECjvm2008 [4] 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 [7]. DJXPerf reports a problematic object—the `Posting` array—allocated at line 235 in method `sortPostingTable` of class `DocumentWriter` reported by prior work [10], shown in Listing 1, which accounts for 23.3% of total cache misses. We followed the proposed optimizations to fix this issue.

2.2 DaCapo 2006 bloat

bloat performs a number of optimizations and analysis on Java bytecode files [7]. 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 [10] 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.

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 [7]. DJXPerf reports a problematic object—the `QueryParser` object—allocated at line 119 in method `parse` of class `QueryParser` reported by prior work [10], 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 [7]. DJXPerf reports a problematic object—the `Transformer` object—allocated at line 100 in method `run` of class `XSLTBench` reported by prior work [10], shown in Listing 4, which accounts for 16.7% of total cache misses. We followed the proposed optimizations to fix this issue.

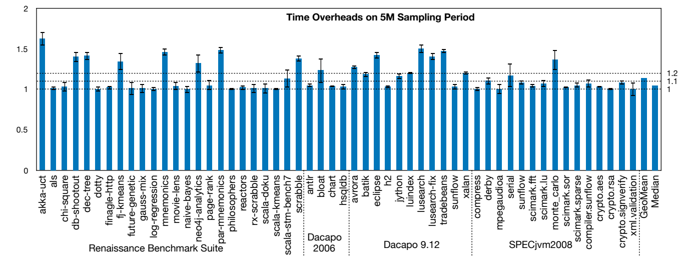
2.5 SPECJbb2000

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.

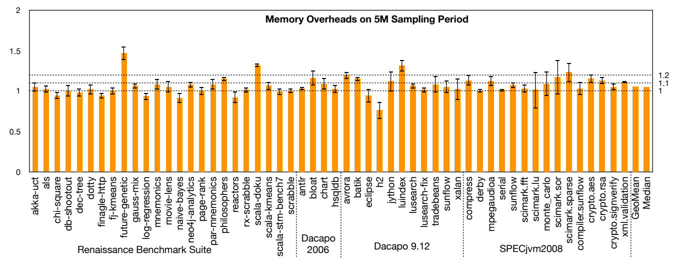
3 CASE STUDIES

3.1 Locality Issues due to Memory Bloat

3.1.1 Cache2k 1.2.0. Cache2k [9] provides an in-memory object cache implementation for Java applications. We run Cache2k with provided Cache2k JMH benchmarks as input. DJXPerf reports a problematic object allocated at line 313 in method `rehash` of class tt Hash2 as shown in Listing 6. This object—an array of `Entry` objects that contain key-value pairs—accounts for 85.6% of total cache misses. Further investigation shows that the code repeatedly creates `Entry` object to rehash the entries. Because the Entry object never escapes to the heap, each thread needs only one instance of this data structure at any point during the execution. To address the problem, we create a static `Entry` array that maintains one



(a) Runtime overheads.



(b) Memory overheads.

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

```
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.

```
------------------------------------------------------------
...
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.purdue.cs.bloat.tree.TreeVisitor.visitIfZeroStmt(TreeVisitor.java:124)
EDU.purdue.cs.bloat.tree.TreeVisitor.visitIfStmt(TreeVisitor.java:114)
EDU.purdue.cs.bloat.tree.TreeVisitor.visitStmt(TreeVisitor.java:219)
EDU.purdue.cs.bloat.tree.TreeVisitor.visitNode(TreeVisitor.java:369)
EDU.purdue.cs.bloat.tree.Node.visitChildren(Node.java:111)
...
EDU.purdue.cs.bloat.ssa.SSAConstructionInfo(SSAConstructionInfo.java:86)
EDU.purdue.cs.bloat.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.

```
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.

Entry object per thread. Also, each time an Entry object is needed, we will reset this Entry array. This optimization yields a (1.09 ± 0.02)× speedup.

```
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.

```
310 private void rehash() {
311   Entry<K,V>[] src = entries;
312   int i, sl = src.length, n = sl * 2, _mask = n - 1, idx;
313   ▶Entry<K,V>[] tab = new Entry[n];
314   ...
315   entries = tab;
316   calcMaxFill();
317 }
```

Listing 6: cache2k: The hotspot object tab (line 313) which suffers from memory bloat problem.

3.1.2 Apache SAMOA. Apache SAMOA [5] (Scalable Advanced Massive Online Analysis) is a platform for mining big data streams. We run Apache SAMOA with the covtypeNorm.arff dataset as its input. DJXPerf reports a problematic object—Instance—allocated at line 165 in method readIns-

```
163 public Instance readInstanceDense() {
164   ...
165   ▶Instance instance = newDenseInstance(this.
          instanceInformation.numAttributes());
166   int numAttribute = 0;
167   instance.setValue(numAttribute, streamTokenizer.nval);
168   ...
169 }
```

Listing 7: Apache SAMOA: The hotspot object instance (line 165) which suffers from memory bloat problem.

```
1 protected AbstractHashedMap(int initialCapacity, final
      float loadFactor) {
2   ...
3   this.loadFactor = loadFactor;
4   this.threshold = calculateThreshold(initialCapacity,
        loadFactor);
5   ▶this.data = new HashEntry[initialCapacity];
6   ...
7 }
```

Listing 8: Apache Commons Collections: The hotspot object data (line 5) which suffers from memory bloat problem.

tanceDense of class ArffLoader. This object accounts for 26% of total cache misses. With code investigation as shown in Listing 7, we find that this Instance object is repeatedly formed (line 165) every time the program reads a dense instance from a file, and different instances of this object do not overlap in their life intervals. Thus, we hoist this object outside of the loop and put it into a static location to avoid repeated allocation and. This optimization yields a (1.17 ± 0.04)× speedup to the entire program execution.

3.1.3 *Apache Commons Collections.*

Apache Commons Collections [6] include many powerful data structures that accelerate the development of most significant Java applications. We run it using its provided commons collections4 map tests as input. DJXPerf reports a problematic object—the HashEntry array—in the constructor of class AbstractHashedMap, which is used as a map to store key-value entries. Line 5 of listing 8 shows this object allocation, which accounts for 22% of total cache misses. Similar to other cases that different instances of this object have disjoint life intervals, so we change this object as a static one. When to use a new HashEntry object, we clear and reuse the static object to avoid repeated allocation upon each iteration. The optimization yields a (1.08 ± 0.01)× speedup to the entire program.

3.2 Issues due to Traditional Locality

3.2.1 *Java Grande: JGFMolDynBench.*

JGFMolDynBench is a simple N-body code modeling the behavior of N argon atoms interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions [3]. We run JGFMolDynBench with input size B. DJXPerf reports the array md.one accounts for 83.8% of the total cache misses and the problematic accesses to this array are highlighted in

```
1 public void force(double side, double rcoff,int mdsize) {
2   ...
3   for (int i=0; i<mdsize; i++) {
4     ▶xx = xi - md.one[i].xcoord;
5     ▶yy = yi - md.one[i].ycoord;
6     ▶zz = zi - md.one[i].zcoord;
7     ...
8   }
9   ...
10 }
```

Listing 9: JGFMolDynBench: The hotspot array md.one (line 4, 5 and 6) which suffer from high cache miss.

the method force of the class md, as shown at line 4, 5, and 6 of Listing 9. The method force is called multiple times in a loop (not shown). The loop in method force (line 3) has a streaming access pattern on array md.one, which shows up as poor temporal locality. Like JGFMonteCarloBench, we apply loop tiling to improve the temporal locality. The optimization reduces cache misses by 42%, yielding a 1.25 ± 0.13× speedup to the entire program execution.

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] 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.
[4] Standard Performance Evaluation Corporation. 2008. SPECjvm2008 benchmark suite. https://www.spec.org/jvm2008.
[5] Apache Software Foundation. 2017. Apache SAMOA: Scalable Advanced Massive Online Analysis. https://samoa.incubator.apache.org.
[6] Apache Software Foundation. 2019. Apache Commons Collections. https://github.com/apache/commons-collections.
[7] 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.
[8] 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
[9] Jens Wike, Ayman Abdel Ghany, Filipe Manuel, Tatsiana Shpikat, and Andreas Worm. 2013. cache2k. https://cache2k.org.
[10] Guoqing Xu. 2012. Finding reusable data structures. ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 1017–1034.