# Iceberg: A Tool for Static Analysis of Java Critical Sections

Michael D. Shah
Tufts University, USA
mshah08@cs.tufts.edu

Samuel Z. Guyer
Tufts University, USA
sguyer@cs.tufts.edu

## Abstract

In this paper we present a static analysis tool called Iceberg that helps programmers find potential performance bugs in concurrent Java programs. The focus of our work is on identifying critical sections with high variability in their latency: in most cases they execute quickly, but occasionally they stall, holding a lock for an unusually long time, and preventing other threads from making progress. The end user experiences such behavior as transient program "hangs". These performance bugs are difficult to find because they are infrequent, transient, and hard to reproduce.

This paper describes our initial results running Iceberg on 24 real-world concurrent programs. Our current approach is to identify all of the code that *could be* executed inside each critical section, including all methods potentially called by it. We collect a number of code metrics that might indicate potential performance problems. These metrics include counts of variable-latency operations such as I/O and memory allocation, as well as overall measurements of critical section size. Using our tool we are able to find critical sections with unusual behavior compared to the other critical sections. Our future work includes a more detailed analysis of control-flow through critical sections, as well as a dynamic analysis to measure the critical section latencies directly.

***Categories and Subject Descriptors*** D.2.8 [*Software Engineering*]: Metrics—complexity measures, performance measures

***General Terms*** Concurrency, Program Analysis

***Keywords*** Critical Sections, Soot, Concurrency

## 1. Introduction

In the modern world of multicore processors and highly responsive applications, concurrent programming is now the norm. The difficulties of writing concurrent programs, however, have been well-documented and led to a large body of work on analysis-based debugging tools. The vast majority of this work is focused on correctness: eliminating vexing bugs such as race conditions and atomicity violations. Much less effort has been spent on understanding performance, and a growing body of evidence suggests that many concurrent programs have lower-than-expected performance, or have unpredictable performance characteristics, such as sporadic *software hangs* [13].

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive version was published in the following publication:

*SOAP'16*, June 14, 2016, Santa Barbara, CA, USA
© 2016 ACM. 978-1-4503-4385-5/16/06...
http://dx.doi.org/10.1145/2931021.2931025

In this paper we present a tool, called Iceberg, that uses static analysis to help programmers identify potential performance problems in concurrent Java programs. Specifically, Iceberg analyzes each critical section and generates a report about code that might be executed while holding a lock. The goal is to alert the programmer to potential problems by identifying code constructs with very high variability in latency. Simple cases include I/O (which could block), memory allocation (which could trigger a garbage collection), or code that is irrelevant to concurrency correctness (as a result of overly-conservative synchronization). More complex cases include potentially long control-flow paths, for example, when an algorithm or data structure has a low amortized cost, but individual operations might have much higher worst-case costs. Also of concern are cases where one synchronized method calls another, potentially causing additional performance penalties.

While such cases might be easy to identify when they appear directly in the synchronized method itself, in many cases potential problems lurk inside methods called indirectly from it. As an example, a critical section might include a call to insert an element in HashMap (which is not synchronized itself). Most of the time this operation is very fast, but occasionally it will resize the underlying table and need to rehash and reinsert all of the elements, causing a much longer delay. Recent work by Kurtsinger and Berger has shown that such delays can cascade to other parts of a system [1].

This paper represents our initial work on this problem, with the following specific contributions:

- Our static analysis tool, called Iceberg, which collects information about critical sections of code.
- A summary of code metrics of Critical Sections gathered from real world programs that may negatively effect performance.
- A description of our ongoing and future work extending this tool in several ways, including a more detailed analysis of the control flow through critical sections, and a complementary dynamic analysis to measure real latencies.

## 2. Related Work

### 2.1 Static Analysis

Much of the prior work on static analysis for concurrency has focused on program correctness. Recently, however, more effort has been turned towards identifying performance problems.

LUPA (Lock Usage Pattern Analysis) is a static analysis tool for understanding the different lock patterns that occur in a program [11]. The focus of this work is similar to ours in that they are primarily looking at lock-related functions only, but with a focus on finding correctness bugs. In our work, we are considering performance implications of code that is assumed otherwise bug free. The benchmarks analyzed by LUPA showed that 80.5% of methods only acquire one lock. This may indicate methods holding more than one lock add additional complexity and require further investigation regarding performance implications of having multi-

ple locks. We make use of this finding in our work by reporting the extent at which methods may have a potential control flow that leads to nested synchronized method calls.

Prior work on whole-program analysis by Yan et al. shows techniques for reusing code metrics collected from a static analysis for libraries [2]. In our tool, we have run our static analysis on the Java Standard Library in order to gather code metrics about the synchronization of each method. This allows developers to then compare with their own data structures to investigate any overhead regarding synchronization.

### 2.2 Critical Sections

Work by Gu et al. studied a programs revision history to see when locks are introduced, modified, or removed in code [10]. Their benchmarks show that 10% of critical section changes are performance related and that over synchronization is a problem real world developers have in C/C++ programs. The metrics they measured include how often lock variable and boundaries are adjusted for critical sections throughout the revision history. Our work aims to find useful metrics of critical sections that can be brought to the attention of programmers through empirical evidence.

### 2.3 Dynamic Analysis

While our tool a static analysis, there exist many other relevant dynamic analysis tools that report on the actual time spent in critical sections.

The tool COZ by Curtsinger and Berger investigates where users should spend time focusing optimization efforts through a methodology called casual profiling [1]. The central idea is to select one thread at random to proceed and pause all other threads and then measure the slowdown of the program. This measured slowdown can then emulate the potential virtual speedup by optimizing that method. This methodology is similar to how critical sections work when any number of threads try to access a shared resource but must wait to acquire a lock. Casual profiling may provide insights into how to effectively measure slowdown of critical sections in our future dynamic analysis work.

The work by David et al. on the Free-Lunch Profiler measures critical section pressure (CSP) to see how much a threads progress is impeded by locks [5]. The work states that the longest time spent in critical sections may not always impede thread progress, however this will vary depending on the task and requires further analysis with tool support. In our work we are looking specifically at the code inside critical sections that may cause unexpected performance problems making a thread spend more time in a critical section.

The work by Yan et al. uncovers performance problems in Java applications with Reference Propagation Profiling[3]. This work presents insights into the performance cost of how often an object is written to versus how often it is read. Different data structures will provide different performance benefits. In our future work we would like to measure time spent in a critical section versus the cost of acquiring that lock. We currently collect what types of data structures are used in a critical section, and we can further use this information to draw conclusions about performance based on the data structures used.

## 3. Our Tool - Iceberg

Iceberg is our tool for statically analyzing the composition of the code inside critical sections. For each critical section it visits all of the bytecodes that *could be* executed while holding a lock, including all code reachable transitively through method calls, and computes a set of metrics. Currently, these metrics consist of counts of operations that often have variable latency, such as I/O, memory

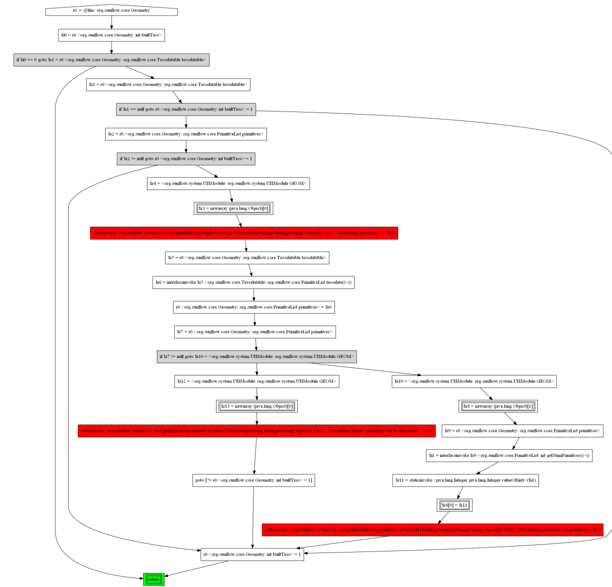

**Figure 1.** Zoomed out view of control flow graph of tesselate method generated with Iceberg. Red boxes indicate other synchronized sections nested.

allocation, and some standard library routines. In the future we will extend this analysis to compute these metrics along individual control-flow paths, which will provide a more detailed account of the overall potential behavior of the critical section.

### 3.1 Implementation

Iceberg is built on top of the Soot Java analysis framework [9]. Our analysis pass operates on the jimple intermediate representation, and relies on Soot's existing callgraph construction algorithm to identify which methods could be called at any given call site. Static call graph construction is necessarily conservative in the presence of dynamic dispatch, so some of the metrics presented here are overestimated because they include all possible callees. Since this analysis is not used for optimization purposes, we could consider a less conservative, or even unsound analysis to prune the call graph.

The data is collected in Iceberg in the following process:

1. First all of the jar files that make an application must be loaded.
2. We setup our analysis in whole program mode (-w) with soot outputting Jimple code (-f J). For classes that Soot cannot resolve, we use the option -allow-phantom-refs so our analysis may proceed with partial results if class files are missing.
3. We created a new *SceneTransformer* that runs on the whole program during Soots Whole program Transformation Phase (wjtp). In this step, we gather all entry points into a program, and then perform a breadth-first traversal from the entry points to build a call graph. The call graph is stored in an adjacency list.
4. For each method in the call graph we store any of the 17 jimple bytecode statements that make up this method.
5. For every synchronized method we generate two visualizations which are described in section 3.4.
6. All of the code metrics collected are finally output to a table summarizing the features of the program and each critical section where programmers may then use in their analysis.

### 3.2 Code Metrics Collected

We collect code metrics for each critical section that show properties of critical sections that would be difficult for programmers to manually tally. These metrics may also indicate performance problems that programmers were not aware of relating to the size and possible execution paths of critical sections. We remind our readers that our metrics are overly conservative so this means the code metrics collected may not be representative of the actual run of a program. We do believe the features collected bring attention to programmers performance problems that could be focused on during an actual run of a program.

- Nested Methods and Nested Synchronized Methods: If the number of methods called from a critical section grows significantly this may indicate more time spent in this critical section then expected by the programmer. Finding nested critical sections may indicate more locks being contended from a single method than expected.
- Nested Loops: Nested loops that iterate over collections of variable size could cause unexpected software hangs. This is especially important if the input size of the collection is unknown or has a large distribution of sizes during program runs. Data structures and algorithms with amortized costs could suffer.
- Contains Allocations: Lots of memory allocations, especially in a loop, could trigger a garbage collection that causes halting. This is an unwanted bottleneck in interactive applications where it can effect the user experience negatively.
- Large Line Count: Very large critical sections may indicate many other threads have to wait when this method is called. This may also indicate a potential site for code optimization if there are many branches and the control flow of a critical section varies often. From a software engineering standpoint, a large critical section may also be a point of interest for code refactoring.
- Branches: The number of branches within a critical section may indicate variability in each execution of the critical section.
- Java Library Method: Java Library methods may call other synchronized methods and use locks more liberally for additional safety. General purpose library methods may be good candidates for users to rewrite under specific conditions to improve performance.

### 3.3 Visualization

To aid programmers in analysis, Iceberg outputs call graphs and a control flow graph in DOT format which can be viewed as node-link diagrams in external tools like graphviz to aid in program comprehension [6]. Our visualizations have simple color encodings to help programmers find regions of interest. In order to save space, we have only displayed one sample of a control flow graph in Figure 1. The three outputs are the following:

1. A whole program call graph with critical sections highlighted in red. This allows us to see clusters of critical sections for instance.
2. We output a control flow graph of the critical section so that we can view control flow and see branching behavior. This allows us to see if there are possibly very long or short paths in a methods call tree.
3. We output an additional call graph with the critical section as the root, so we can see the behavior from the critical section in a scaled down version.

```
1   private synchronized void tesselate() {
2     // double check flag
3     if (builtTess != 0){
4         return;
5     }
6     if (tesselatable != null && primitives == null) {
7       UI.printInfo(Module.GEOM, "Tesselating geometry ...");
8       primitives = tesselatable.tesselate();
9       if (primitives == null)
10      UI.printError(Module.GEOM, "Tesselation failed - geometry will be discarded");
11    }else{
12      UI.printDetailed(Module.GEOM, "Tesselation produced %d primitives",
13                      primitives.getNumPrimitives());
14    }
15    builtTess = 1;
16  }
```

**Figure 2.** Code Listing 1 - Sunflow Synchronization Code for Tesselate Method

## 4. Critical Section Use Cases

In this section we will present three use cases where synchronization may show performance bugs. The benchmark for each use case was derived by studying a multi-threaded ray tracer benchmark program called Sunflow [12]. We have provided a code sample and our analysis with the aid of Iceberg of why each of the critical sections could be a performance bug based on information gathered from our tool. The critical sections and some of the code metrics we collect are displayed in Table 1.

### 4.1 Over Synchronization

Over synchronization occurs when a method is synchronized that does not need to be. A clear example is a synchronized method that contains immutable data which by definition cannot be interfered with. If all fields of a class are final and private for example, then it is unlikely an object needs to be synchronized. If a class has no *setter* methods, then we may be able to get away with less synchronization.

In order to use our static analysis to help programmers mitigate over synchronization, the following steps are performed:

1. We collect information about classes and record attributes about the quantifiers of each of the fields and methods in that class using Soot. Soot provides a way to return if a method isPrivate, isPublic, or isProtected for example.
2. We can use this information to find classes that are interesting candidates to optimize if they have many public members that were not final for instance.

The performance gain by avoiding over synchronization is that we no longer have to obtain a lock each time we access an object.

### 4.2 Large Critical Sections

One notable feature of Table 1 is that the numbers grow much larger when considering all of the code potentially reachable from a critical section, not just the top-level calls directly in the synchronized method itself. This feature is what makes static analysis such a valuable tool for this problem: it can reveal properties of the code that might not be readily apparent to the programmer. As an example, consider memory allocation. Just looking at the synchronized methods themselves (column 8) it might look like there are very few allocations (or none at all). But once we consider all the potential callees, the number is actually quite large – large enough to potentially trigger at least a minor garbage collection.

### 4.3 Nested Critical Section

Nested critical sections may indicate a performance problem as the more deeply a series of locks is, the longer other threads may be contended. In Figure 2, we have three methods that are synchronized within our method. While this is functionally correct, it is conservative enough, that some of the methods may not need to be synchronized and we may save some performance. The opportunity

| Critical Section Name: | # of method calls in first level | total # of method calls | # of synchronized method calls in first level | total # of sync method calls | # of Java Library Calls in first level | Total Number of Java Library Calls | Number of allocations in first level | Total # of allocations | total # lines of IR |
|---|---|---|---|---|---|---|---|---|---|
| String | 9 | 212 | 0 | 12 | 8 | 190 | 0 | 81 | 1894 |
| tesselate | 6 | 11673 | 3 | 240 | 2 | 11664 | 0 | 3030 | 128265 |
| build | 6 | 7797 | 1 | 161 | 2 | 7783 | 1 | 2031 | 85646 |
| run | 4 | 3894 | 2 | 81 | 0 | 3887 | 0 | 1009 | 42767 |
| lookupShadingCache | 7 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 32 |
| addShadingCache | 10 | 14 | 0 | 0 | 0 | 2 | 2 | 4 | 43 |
| load | 6 | 11876 | 2 | 245 | 2 | 11864 | 1 | 3094 | 130074 |
| getTexture | 6 | 7807 | 2 | 160 | 6 | 7799 | 1 | 2021 | 85795 |
| flush | 2 | 3890 | 1 | 80 | 2 | 3887 | 0 | 1009 | 42761 |
| imageBegin | 14 | 33064 | 1 | 911 | 24 | 33050 | 4 | 7764 | 369933 |
| imageEnd | 10 | 5479 | 0 | 108 | 16 | 5469 | 0 | 1382 | 59450 |
| paint | 1 | 1 | 0 | 0 | 2 | 0 | 0 | 0 | 0 |
| imageBegin | 3 | 1243 | 0 | 16 | 4 | 1239 | 0 | 289 | 13808 |
| imagePrepare | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| imageUpdate | 5 | 2477 | 0 | 31 | 4 | 2469 | 0 | 575 | 27570 |
| imageFill | 5 | 2477 | 0 | 31 | 4 | 2469 | 0 | 575 | 27570 |
| imageEnd | 2 | 1242 | 0 | 16 | 2 | 1239 | 0 | 289 | 13808 |
| imageUpdate | 4 | 6375 | 1 | 110 | 4 | 6364 | 0 | 1588 | 70593 |
| store | 1 | 10 | 0 | 0 | 2 | 9 | 0 | 0 | 62 |
| store | 5 | 18 | 0 | 0 | 4 | 12 | 1 | 1 | 170 |
| store | 5 | 18 | 0 | 0 | 4 | 12 | 1 | 1 | 170 |
| getRadiance | 33 | 2638 | 0 | 33 | 20 | 2596 | 1 | 612 | 28652 |
| store | 7 | 13 | 0 | 0 | 0 | 3 | 1 | 5 | 131 |
| run | 9 | 10 | 1 | 0 | 0 | 0 | 0 | 0 | 31 |
| run | 5 | 6 | 1 | 0 | 0 | 0 | 0 | 0 | 19 |
| run | 7 | 7 | 0 | 0 | 0 | 0 | 0 | 0 | 23 |
| drag | 1 | 1544 | 0 | 25 | 2 | 1543 | 0 | 357 | 16420 |
| zoom | 11 | 1560 | 0 | 25 | 22 | 1549 | 0 | 357 | 16482 |
| reset | 3 | 1548 | 0 | 25 | 6 | 1545 | 0 | 357 | 16434 |
| fit | 9 | 1558 | 0 | 25 | 18 | 1549 | 0 | 357 | 16474 |
| imageBegin | 7 | 3496 | 1 | 52 | 14 | 3489 | 1 | 897 | 39971 |
| imagePrepare | 3 | 1564 | 2 | 27 | 6 | 1561 | 0 | 367 | 16516 |
| imageUpdate | 5 | 1563 | 1 | 26 | 4 | 1555 | 0 | 364 | 16559 |
| imageFill | 5 | 1565 | 1 | 26 | 2 | 1557 | 0 | 364 | 16568 |
| paintComponent | 13 | 36 | 0 | 0 | 26 | 23 | 0 | 0 | 117 |
| printDetailed | 2 | 3888 | 0 | 80 | 2 | 3886 | 0 | 1009 | 42738 |
| printInfo | 2 | 3888 | 0 | 80 | 2 | 3886 | 0 | 1009 | 42738 |
| printWarning | 2 | 3888 | 0 | 80 | 2 | 3886 | 0 | 1009 | 42738 |
| printError | 2 | 3888 | 0 | 80 | 2 | 3886 | 0 | 1009 | 42738 |
| taskStart | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taskUpdate | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taskStop | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| taskCancel | 1 | 3889 | 1 | 80 | 0 | 3887 | 0 | 1009 | 42748 |
| taskCanceled | 1 | 3889 | 1 | 80 | 0 | 3887 | 0 | 1009 | 42748 |
| Totals | 241 | 136126 | 21 | 2866 | 220 | 135809 | 14 | 33825 | 1498508 |

**Table 1.** Sunflow Critical Section output - Note that method parameters have been omitted to save space, such that repeated method names represent polymorphic functions.

here is that these methods could be combined, and also simplify the concurrency in the program.

## 5. Summary of Results

In order to understand critical sections, we have analyzed 24 real world programs listed in Table 2. This table shows that Java programmers are using concurrency, and that there is also more concurrency than they might originally anticipate. We have additionally shown a few use cases in the previous section that analyze all of the critical sections for Sunflow in Table 1.

## 6. Future Work

### 6.1 Static Analysis

For future work we hope to add more detailed analysis of the control flow properties of critical sections. One kind of analysis we are developing is called *latency variability analysis*, a flow-sensitive analysis that attempts to estimate the range of latencies possible through a block of code. The goal is not a precise prediction, but rather a gross characterization that would, for example, be able to distinguish so-called fast paths from slow-paths.

We would also like to add support to text editors such that they could highlight where problems may occur based on our analysis. We believe there can also be interesting work in finding patterns in critical sections that could indicate performance bugs. For example, if too many nested critical sections from different objects could be called along a path, that should flag as a potential performance bug.

### 6.2 Dynamic Analysis

Extending upon the static analysis and performing a more rigorous dynamic analysis is the next step in this research. Our static analysis can inform the dynamic analysis about where performance problems may occur, but there also may be many false positives for program execution paths displayed. We would like to add support for instrumentation and timing of critical sections that may have performance problems. Using techniques as discussed in the Casual Profiling work by Curtsinger and Berger for critical sections could be useful for determining the overall impact on program performance [1].

Zaparanuks and Hauswirth describe a method for developing a cost function for an algorithm as opposed to the actual time spent [4]. This allows a programmer to understand why a program is running slow under a set of given inputs. The goal of our tool is to understand specifically in critical sections what features may slow down a piece of concurrent code, and we believe the work in this paper may provide a framework for how highly variable sized inputs in critical sections affect a program's overall performance.

## 7. Conclusions

In this paper we have presented our static analysis tool Iceberg with the following contributions:

- Our static analysis tool Iceberg which can gather and collect information about critical sections.
- A report of code metrics for critical sections of real world Java programs.
- Future work on using our static analysis results to develop focused dynamic analysis tools for identifying top candidate methods that may have performance problems in critical sections.

## Acknowledgments

We would like to thank members of the Redline Research group at Tufts University for their feedback on this tool.

## References

[1] Charlie Curtsinger and Emery D. Berger. 2015. Coz: finding code that counts with causal profiling. In Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15). ACM, New York, NY, USA, 184-197.

[2] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Rethinking Soot for summary-based whole-program analysis. In Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis (SOAP '12). ACM, New York, NY, USA, 9-14.

[3] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. Uncovering performance problems in Java applications with reference propagation profiling. In Proceedings of the 34th International Conference on Software Engineering (ICSE '12). IEEE Press, Piscataway, NJ, USA, 134-144.

[4] Dmitrijs Zaparanuks and Matthias Hauswirth. 2012. Algorithmic profiling. In Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12). ACM, New York, NY, USA, 67-76.

[5] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously Measuring Critical Section Pressure with the Free-Lunch Profiler. SIGPLAN Not. 49, 10 (October 2014), 291-307.

[6] GraphViz. http://www.graphviz.org/

[7] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. 2012. Understanding and detecting real-world performance bugs. SIGPLAN Not. 47, 6 (June 2012), 77-88.

[8] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. 2015. A large-scale study on the usage of Java's concurrent programming constructs. J. Syst. Softw. 106, C (August 2015), 59-81.

[9] Raja Valle-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 1999. Soot - a Java bytecode optimization framework. In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research (CASCON '99), Stephen A. MacKay and J. Howard Johnson (Eds.). IBM Press 13-.

[10] Rui Gu, Guoliang Jin, Linhai Song, Linjie Zhu, and Shan Lu. 2015. What change history tells us about thread synchronization. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). ACM, New York, NY, USA, 426-438.

[11] Rui Xin, Zhengwei Qi, Shiqiu Huang, Chengcheng Xiang, Yudi Zheng, Yin Wang, and Haibing Guan. An automation-assisted empirical study on lock usage for concurrent programs. In 29th IEEE International Conference on Software Maintenance, 2013.

[12] Sunflow - http://sunflow.sourceforge.net/

[13] Xi Wang, Zhenyu Guo, Xuezheng Liu, Zhilei Xu, Haoxiang Lin, Xiaoge Wang, and Zheng Zhang. 2008. Hang analysis: fighting responsiveness bugs. SIGOPS Oper. Syst. Rev. 42, 4 (April 2008), 177-190.

| Benchmark | Non-Java Library Synchronized Methods | Java Library Synchronized Methods | Java Library Methods | Non-Java Library Methods |
|---|---|---|---|---|
| Avrora | 61 | 50 | 1422 | 248 |
| Barbecue | 0 | 85 | 2920 | 96 |
| Dom4J | 236 | 177 | 4168 | 2146 |
| Fop | 461 | 220 | 5977 | 3446 |
| Gravity2D | 10 | 99 | 2732 | 20 |
| H2 | 138 | 118 | 3585 | 917 |
| httpunit | 189 | 168 | 4337 | 1304 |
| Jace | 18 | 138 | 3984 | 245 |
| JAG3D | 656 | 187 | 5131 | 1263 |
| jmonkeyengine | 247 | 157 | 4915 | 3456 |
| Jython | 317 | 81 | 2636 | 2470 |
| kryonet | 24 | 53 | 1699 | 262 |
| lwjgl | 176 | 149 | 4075 | 1845 |
| Microemulator | 60 | 166 | 4399 | 433 |
| Paw | 171 | 164 | 4038 | 741 |
| Peers | 36 | 56 | 1637 | 11 |
| PMD | 202 | 161 | 4410 | 3178 |
| Simbad | 19 | 119 | 3210 | 166 |
| Sunflow | 44 | 150 | 3584 | 954 |
| treemap | 4 | 95 | 2689 | 19 |
| Ttorrent | 192 | 180 | 4628 | 638 |
| xalan | 197 | 148 | 3483 | 1731 |
| YCad | 44 | 101 | 2515 | 172 |
| zgrv | 175 | 162 | 3846 | 1310 |

**Table 2.** Real world Java Programs and total number of methods that could be called in a program.