Iceberg: Dynamic Analysis of Java Synchronized Methods for Investigating Runtime Performance Variability

Michael D. Shah
College of Computer and Information Sciences
Northeastern University
Boston, MA, United States
mshah.475@gmail.com

Samuel Z. Guyer
Computer Science Engineering
Tufts University
Medford, MA, United States
sguyer@cs.tufts.edu

Abstract

Writing multi-threaded code for both correctness and performance is difficult. Often programmers strive for correctness, which may lead to an overly conservative use of synchronization primitives. Even when correct, however, synchronized regions of code (critical sections) may introduce performance variability that causes unexpected *software hangs* and can be considered a performance bug. These performance bugs are occasional, hard to predict, and may negatively impact the overall user experience.

In this paper we present a tool called Iceberg for finding potential software hangs in Java programs. Our focus is on identifying synchronized methods with high variability in their execution time – in particular, if they occasionally run much longer than normal. The key feature of Iceberg is that it records the runtime of every individual execution of a synchronized method, looking for outliers among them, rather than aggregating the runtime the way previous profilers have done. We calibrate our tool using a suite of microbenchmarks with known variability in their critical sections and then tested it with three real world programs. Our results document several cases where our tool finds high variability that could cause real software hangs.

CCS Concepts • **Software and its engineering** → **Dynamic analysis**; • **Theory of computation** → *Concurrency*;

Keywords Critical Sections, Javassist, Concurrency, Performance bugs

ACM Reference Format:
Michael D. Shah and Samuel Z. Guyer. 2018. Iceberg: Dynamic Analysis of Java Synchronized Methods for Investigating Runtime Performance Variability. In *(ISSTA Companion/ECOOP Companion'18), July 16–21, 2018, Amsterdam, Netherlands.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236454.3236505

ISSTA Companion/ECOOP Companion'18 , July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5939-9/18/07…$15.00
https://doi.org/10.1145/3236454.3236505

1 Introduction

In all but trivial cases, writing correct multithreaded code is difficult for even adept programmers. Concurrent behavior is hard to reason about, leading to bugs such as race conditions and atomicity violations that are difficult to reproduce, diagnose, and fix.

There is a significant body of work on both static and dynamic techniques for helping programmers find correctness errors in concurrent programs (e.g., [13] and [10]). Unfortunately, even after these errors are corrected the program can still fail in a different way: by failing to provide the expected performance, either in terms of throughput, scaling, or responsiveness. Since performance is the primary reason for using concurrency, this situation is considered a kind of bug – a performance bug. To make matters worse, performance bugs and correctness bugs are often in tension. In order to avoid correctness problems programmers use synchronization defensively, adding more locks and enlarging critical sections even where it is not necessary. As an example, a programmer might add synchronization to every method in a class. The code is more likely to run correctly, but there is an additional overhead of locking each execution, which might not be necessary.

A number of tools exist to help programmers identify throughput and scaling problems in concurrent programs [5, 9]. These tools look for conditions, such as lock contention, serialization, or false sharing, that inhibit overall concurrent execution. In order to impact throughput, these conditions must happen frequently and pervasively, which allows sampling-based solutions to catch these errors. In the case of responsiveness problems, however, even a rare performance hiccup can produce an annoying "hang" in the

program. The user might experience it as a momentarily frozen user interface (e.g., in a game or video) or an unusually long latency for a transaction (e.g., after pushing a submit button on a web form). For some kinds of applications, like games, users might become frustrated enough to abandon them. Thus tools are needed to monitor performance variability, and at a very fine granularity.

In this paper we present Iceberg, a dynamic analysis tool for Java that measures variability in the runtime of synchronized methods to discover potential performance hangs. Unlike existing tools, which provide aggregate performance, such as total runtime or average runtime, Iceberg records each individual execution of a synchronized method, looking for outliers. Such outliers might occur, for example, when using amortized-time data structures such as hash tables. The average runtime, and even the typical runtime (as might be measured by a sampling tool) could be fast, but occasionally the hash table will resize its underlying array, incurring a high cost. We focus on hangs in critical sections because they can impede other threads, causing a cascade of slowdowns that are difficult to diagnose[7].

2 Our Tool - Iceberg

Our tool Iceberg instruments and monitors methods qualified with the keyword *synchronized*, which is a Java keyword that applies an intrinsic lock (also called a monitor lock) to that objects method. Our technique extends easily to synchronized blocks, but we would need to change our implementation strategy to support the ad hoc synchronization primitives provided in the `java.util.concurrent` package.

Our focus is not on program-scale properties, such as contention or serialization. Rather, our approach is unique because we measure how much time the programs spends *inside* a critical section each time it is executed to then build a statistical model of its behavior so we can understand its variability. Thus the focus is on latency variability within critical sections which have potential to be bottlenecks in programs.

2.1 Architecture and Implementation

In order to collect dynamic information from our benchmarks, we used the Javassist library to implement Iceberg [1]. Javassist is a bytecode engineering library that allows users to statically instrument Java source code before the program runs. Javassist is based off the idea of aspect oriented programming (AOP) where we modify programs to profile code. Similar frameworks have been used by Moret et al. to develop profilers for the JVM [6].

2.1.1 Instrumentation

Iceberg instruments methods by adding probes to the entry point and at each point where a method exits. These probes log information about each execution of the method and capture every execution of the method. Sampling-based strategies may miss executions, and thus not capture outlier executions in which something unusual happens. In Figure 1 we demonstrate how the time spent in a synchronized method is summed, such that timers capture precisely how much time is spent per critical section.

Iceberg uses the JVM's classloader to find each synchronized method in the program automatically. We allow a user to read in a subset of critical sections (or even non-synchronized methods if a user wanted to analyze additional methods) through a configuration file, but for the experiments in this paper we analyze all of the synchronized methods by default.

2.1.2 Instrumentation Settings

Iceberg has varying levels of instrumentation for gathering data. In this paper, we primarily used timers only. Different levels of instrumentation can impact performance and perturb the system more as a trade off. For each of our real world benchmark experiments we report the overhead and what instrumentation is used in Table 2. The following instrumentation settings are available:

Timers - The core instrumentation is to add only timers to each synchronized method. Time spent in each method is stored in a list data structure outlined in section 2.1.1 Instrumentation

Call Tree - Iceberg can construct a call tree of the methods instrumented. By default, these are only synchronized methods. However, the user may choose to instrument all methods if they want a more complete call tree for the program.

Thread Contention - Iceberg has lightweight reporting to tell if the executing synchronized method is being blocked by other methods (i.e. does it hold the lock).

Calling context - Iceberg records the calling method of each other method (i.e. retrieve a stack trace).

Stream Output - Iceberg writes data immediately, such that long running applications (i.e. running for hours or days) information can be collected immediately. The trade-off is introducing additional file I/O calls could perturb the system, and by default this option is disabled.

2.1.3 Collected Metrics

After the program has terminated, information from each of the dynamic invocations of a synchronized method is output. Iceberg then aggregates this information into the following list of metrics automatically:

Metrics 1-6: The thread id at which the method was executing, number of times an individual method executed, time spent in each execution of the method (using nanoTime()), maximum time spent in a single execution of a method, minimum time spent in a single execution of a method, average time spent across all executions of a method.

Metric 7: The standard deviation among all executions of a method.
Metric 8: The number of diverging points from the standard deviation. This is the basis of our statistical model described in greater detail in the next subsection.
Metric 9: Percentage of divergent points. This is used to see the extent to which a critical section has variability in its execution time. A method with lots of diverging execution time points and many conditional branches may be an interesting case to start optimizing.
Metric 10: Time it took the entire program to execute (Absolute Program Time). This can be used to measure the stability of each total program run, or otherwise compared to different versions of the program after optimizations are made.
Metric 11: Total time spent in critical sections alone (Critical Section Time). This gives the user a sense of how much concurrency is being used within the program.
Metric 12: Total time spent in instrumentation is totaled to give an estimate of how much Iceberg's instrumentation perturbs an actual run of the program.

2.1.4 Finding outliers

Metric 8 captures how Iceberg identifies to users which executions are potential problems. We are specifically looking for methods with fairly uniform runtimes *except* for some occasional executions with very different behavior. We want to ignore methods with a wide range of runtimes – for example, methods that are highly input-dependent. Iceberg's algorithm to pick out divergent points is the following:

1. Iceberg retrieves every execution of a particular synchronized method.
2. The standard deviation is computed from all of the executions to give an indicator of variance.
3. The number of executions and elapsed time spent in each execution is used to model a line of best fit using the least square method. Iceberg by default generates the least square best-fit line.
4. Iceberg then iterates over each of the actual executions and compares them to the difference of the best-fit line generated in the prior step.
5. If the difference in time is greater than one standard deviation, then this is reported as a divergent point. At this point, a programmer can expect to see what happened.

An example of our statistical method is highlighted in Figure 2 to show that two points are outliers. In this case, both outliers are greater than one standard deviation above the line. This shows they go against the trend for time spent in this synchronized method, and those particular executions should be investigated. In order to debug more precisely which executions caused variability, Table 1 lists all of the diverging executions.

At this point the programmer has two options:

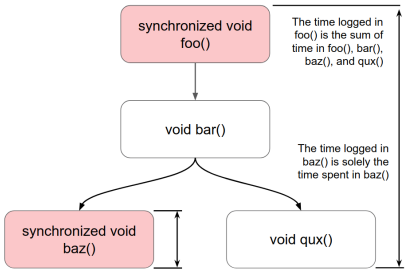

Figure 1. This diagram represents a snapshot of a call graph. Only synchronized methods (in red) are instrumented, and the time spent in that synchronized method is the sum of the time spent until the synchronized method returns and the monitor lock is released.

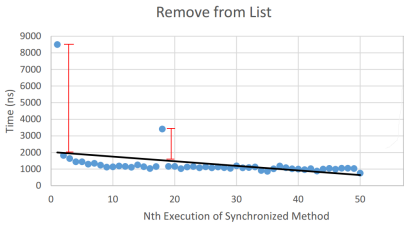

Figure 2. This scatter plot shows a linear trend line that is plotted for a synchronized method which removes an item from an arraylist. Two points are observed as outliers which are more than one standard deviation away from the trend. The first is the initial allocation. The second involves the array resizing and copying its contents. The story from the plot is a use case where an amortized data structure may not be great for responsiveness,

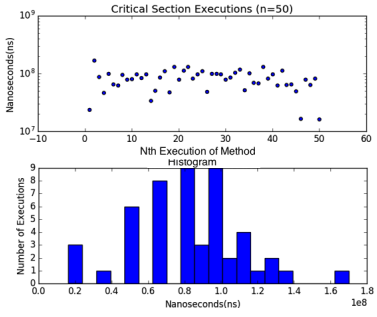

Figure 3. Example visual output of variability in sorting a list in a synchronized method

1. The programmer may investigate the source code (if available) to guess why variability exists.
2. The programmer may turn on another feature in Iceberg to output a call tree that gives a more accurate trace (at the expense of performance).

Table 1. Raw data from microbenchmarks divergent executions automatically output by Iceberg. The rightmost column shows the actual execution that diverged using our statistical model, and this data point can be used when analyzing call tree information to find root causes of suspected performance errors.

Method Name	Total Runs	Runs Avg(ns)	Std Dev (ns)	% of Diverging Executions	Diverging Execution
ArrayListOfArraylistTest.increment()	50	11823111	10408412	0.1	1,5,15,24,42
hashmapTest.append(int,int)	50	1490.12	4232	0.06	0,25,48
SortRandomListTest.generateAndSortRandomLists(int)	50	3669.64	3308	0.06	0,44,45
RandomFastSlowTest.fastPath()	251	1531	4709	0.059	20,26,47,55,61,73,81,82,88,102,206,226,238, 239,250
AllocationTest.remove()	50	1699.4	2829	0.04	0,32
appendToList.append()	50	348.42	695	0.02	0
ArrayDataLocalityTest.increment()	50	2277257	1427820	0.02	0
NetworkIPReachableTest.getHost()	50	292897.8	1761967	0.02	0
RandomFastSlowTest.slowPath()	249	16981037	2701184	0.008	0,2

3 Experimental Methodology

We run our benchmark experiments using the protocol outlined in this section.

3.1 Equipment

Each benchmark was run with and without instrumentation on the same Lenovo Ideapad Y700 with an Intel 6th generation core I7 with 16 GB DDR4 2133 MHz of DDR4 memory and AMD R9 with 4 GB of Video Ram under Ubuntu Linux 16.04.

3.2 JVM Consistency

Our analysis includes looking at each execution of the program, and we remind the reader that we are looking at variability within one run of a single program as opposed to comparing an average of runs. Thus, after each run of a program we look to see if we have captured any interesting variability. Hence, we do not try to warm up the JVM in any way by running the application multiple times. This is in an effort to capture real world use cases in which variability occurs infrequently, thus every run of the program matters.

3.2.1 Minimizing external effects

Many factors influence the runtime of a piece of code, including external factors that are not helpful in performance analysis, such as execution of other processes and services. We took the following steps to try to minimize these effects:

- External applications (e.g. web browsers, terminals, editors) and additional operating system services (e.g. cloud sync, user `chron` jobs) that were not necessary to run the benchmarks were terminated.
- We initially attempted to warm up the JVM with the-XX:CompileThreshold=1 option to ensure that every critical section was compiled rather than being JIT interpreted. This would make the first execution always take longer than normal time, as compilation time would also be captured in our results. We thought this would ease analysis and allow for more consistency in our distribution of each critical section that executes. This idea was later removed in order to reflect more closely real world application runtime performance. Thus, no additional compiler flags were modified not found in each benchmark. Care is instead taken during the analysis to understand that one time jumps in execution time may be do to JIT compilation occurring during any execution of a method.

3.3 Threats to Validity

While many precautions were taken, there may exist noise that is difficult to account for. Mytkowicz et al. discuss several of the difficulties of making careful measurements [12]. Different operating systems policies for scheduling services and processes can impact variability within a single application. The Java run-time environment for example has fixed-priority scheduling for its threads where higher priority threads are chosen to run first. However the policy for threads with the same priority is chosen to execute in arbitrary order. On occasion the thread scheduler will also execute a lower priority thread to avoid starvation. Additionally, any instrumentation that is added from our tool can disrupt thread timing and scheduling.

In order to get as honest analysis as possible on real world programs, we ran benchmarks as they were packaged by the real world developers, only adding in our probes through a Java agent. The goal was to sample critical sections as they are used in the real world and avoid fiddling with too many compiler optimizations. Our goal is to identify performance variability that may cause disruptive *software hangs* that disrupt a smooth user experience.

3.4 Microbenchmarks for Calibration

We have summarized a set of the microbenchmarks (prefixed M, followed by a number) we hand crafted to test our tool. Calibration included making fast and slow executing synchronized methods, and testing if we could catch and output divergent executions. The basis of these results also help define some of the behaviors (Behavior A-H in the next section) that we hypothesize may contribute to variability within a single critical section. A sample of our microbenchmarks results generated from Iceberg are captured in Figure 3 and Table 1 show the tabular output.

3.4.1 Microbenchmarks

M1 - Arithmetic - a series of arithmetic operations.

Table 2. Benchmark Overheads - Each of the different timing modes offers a way to investigate performance variability with different overheads. A note that running Sunflow with all methods instrumented + full call tree information makes the program too slow to use; as there are a large number of small method calls made per frame that slow the application when logging with our instrumentation.

Benchmark	Percent of Program Spent in Critical Section	Instrumentation Overhead (Timers Only)	Timers + Full Call Tree Instrumentation	Timers + Thread Contention	Timers + Calling Context
Sunflow	23.97%	0.56%	N/A	0.60%	1.7%
Skulls	17.96%	0.21%	2.63%	12.77%	11.73%
Movie Player	0.19%	0.21%	15.0%	0.07%	0.27%

M2 - appendToList - behavior of growing list.
M3 - HashmapTest - adding and removing of elements.
M4 - AllocationTest - memory pressure on the heap.
M5 - StringTest - behavior of appending strings.
M6 - SortLinkedListTest - various sized linked lists.
M7 - SortArrayListTest - various sized arrays.
M8 - StackTest - standard library stack operations.
M9 - ArrayDataLocalityTest - updating values in an array.
M10 - ArrayListOfArraylistTest - updating values in multi-dimensional data structures.
M11 - RandomFastSlowTest - branching behavior in critical section to faster and slower code blocks.
M12 - NestedSynchronizedClassesTest - nested locks.
M13 - GrowingIntegerListTest - growing list of integers.
M14 - SortRandomListTest - sorting of randomly generated strings.
M15 - NetworkIPReachableTest - I/O latency.

3.5 Real World Programs as Benchmarks

In addition to our microbenchmarks we selected three representative benchmarks as computer intensive applications that need consistent performance. These projects are real world projects sourced from GitHub and SourceForge. Each of the benchmarks have different workloads, and make use of critical sections. The benchmarks are the following: Sunflow (a multi-threaded ray tracer program), Skulls (an interactive video game), and MoviePlayer (a media player built on top of the JavaFX framework) [2–4] .

4 Results and Discussion

Based on the results found in our real world benchmarks and guided by our microbenchmarks, we have come up with several behaviors of critical sections which may indicate performance variability in general. These insights are in part from the results in Table 1 from our microbenchmarks.

The tests involving hashmap indicate that amortized data structures can cause variability when the implementation resizes and rehashes. Standard Java Library data structures (such as lists or stacks) also have some upfront cost when first performing operations such as adding or removing items. We tested allocation and were able to detect some divergence where garbage collection took place or different memory sizes were allocated. Locality also appears to have an effect, such as an arraylist of arraylists where the hit rate in the cache may drop. Tests that involve *fast and slow* code paths indicate performance variability based on the branch that executes within a critical section. And finally, the choice of algorithm, such as in a sort where the input can effect overall time also adds more variability. Observing the rightmost column of Table 1 gives a hint of which types of programming patterns and data structures may be more at risk for having higher variability. Further behaviors are described in the next subsection.

4.1 Behaviors that may cause variability summary

Behavior A: A critical section contains several execution paths which vary in time duration when executed.
Behavior B: In a critical section there exists an object that executes in several threads, and is highly contended for and blocked along its execution. Particular instances of its execution may be unfairly scheduled or starved further adding variability.
Behavior C: Within a critical section, amortized data structures and algorithms such as hashmap are heavily used that have good average-case performance, but in the worst-case have to rehash or otherwise reallocate resources.
Behavior D: Data locality (use of linked lists versus arrays) and cache misses may also play a role in time spent in a critical section. Lists of lists that grow independently and with objects of varying size may cause additional variability.
Behavior E: If a large number of allocations are occurring within a critical section, this can trigger a garbage collection which increases the time spent for an instance of that methods execution.
Behavior F: Input and output events in critical sections such as keyboard, mouse, networking events, or file related commands could introduce unexpected latency.
Behavior G: Generic getter and setter methods often showed the largest amount of divergent executions. We suspect after analyzing the benchmarks' that these operations are often inexpensive and depend primarily on if these values are in the cache or not.
Behavior H: Critical sections which only execute once are common. We observed them in configuration screens or before the program was loading. The variability within an execution is often proportional to loading external assets.

5 Related Work

The framework VarCatcher by Zhao et al. was developed to study the performance of parallel workloads on multi-threaded programs [15]. Their tool uses a parallel characteristic vector (PCV) to analyze the results and find the impact of features of a program and how they effect the variability. Their analysis focuses on looking at the different execution patterns, and finding across multiple runs where performance variability across multiple program execution occurs with a focus on how the system (i.e. thread scheduler and caching) effect program performance where our work focuses at a different granularity on the code itself.

The tool COZ by Curtsinger and Berger investigates where users should spend time focusing optimization efforts through a methodology called causal profiling[7]. In this system, a thread is selected at random and paused to see the impact on the overall program performance giving an estimated virtual speedup that could occur. Our tool could be complementary: our technique can identify potentially high-latency critical sections, and Coz can then be used to determine the potential impact on overall program performance and better modeling the cost of a method call in a program.

The work by David et al. on the Free-Lunch Profiler measures Critical Section Pressure(CSP) to see how much a thread's progress is impeded by locks [8]. In our work we are analyzing the code inside critical sections and looking for reasons why it might execute longer than expected. Holding a lock for longer than expected can be an underlying cause for contention and we think this could be another behavior that can be used to predict and mitigate root causes of contention that happen sporadically.

The work by Schorgenhumer et al. presents a way to use the Java Virtual Machine Tool Interface (JVMTI) to perform sampling based profiling with locks [14]. Their work takes the monitor lock associated with every object, and instruments each monitor exit (This extends on previous work by Hofer et al. which uses the OpenJDK HotSpot VM directly to record lock contention events [11]). If a contention exists, threads are suspended, the contention is recorded, and data is collected on how long and where the contention occurred.

6 Conclusion and Future Work

We have presented our dynamic analysis tool Iceberg which reports outliers in executions of synchronized methods in Java programs. We used a latency variable analysis that shows where synchronized methods may deviate from the expected performance and create unexpected software hangs. In future work, we will explore how can we find critical sections that may be susceptible to performance bugs statically.

References

[1] 2016 (accessed July 14, 2016). *Java Assist.* http://jboss-javassist.github.io/javassist/.

[2] 2017. Movie Player Code Archive. https://code.google.com/archive/p/paul-gramming/source/default/source. (2017). [Online; accessed 30-June-2017].

[3] 2017. Sunflow - Global Illumination Rendering System. http://sunflow.sourceforge.net/. (2017). [Online; accessed 10-Jan-2017].

[4] 2017 (accessed June 30, 2017). *Skulls Video Game.* https://hub.jmonkeyengine.org/t/skulls-demo-available-for-download/31984.

[5] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. 2010. Performance Analysis of Idle Programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, New York, NY, USA, 739–753. https://doi.org/10.1145/1869459.1869519

[6] Danilo Ansaloni, Walter Binder, Alex Villazón, and Philippe Moret. 2010. Rapid Development of Extensible Profilers for the Java Virtual Machine with Aspect-oriented Programming. In *Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*. ACM, New York, NY, USA, 57–62. https://doi.org/10.1145/1712605.1712616

[7] 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. https://doi.org/10.1145/2815400.2815409

[8] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously Measuring Critical Section Pressure with the Free-lunch Profiler. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14)*. ACM, New York, NY, USA, 291–307. https://doi.org/10.1145/2660193.2660210

[9] Kristof Du Bois, Jennifer B. Sartor, Stijn Eyerman, and Lieven Eeckhout. 2013. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &; Applications (OOPSLA '13)*. ACM, New York, NY, USA, 355–372. https://doi.org/10.1145/2509136.2509529

[10] Dawson Engler and Ken Ashcraft. 2003. RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP '03)*. ACM, New York, NY, USA, 237–252. https://doi.org/10.1145/945445.945468

[11] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. 2016. Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (ICPE '16)*. ACM, New York, NY, USA, 263–274. https://doi.org/10.1145/2851553.2851559

[12] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2009. Producing Wrong Data Without Doing Anything Obviously Wrong! *SIGPLAN Not.* 44, 3 (March 2009), 265–276. https://doi.org/10.1145/1508284.1508275

[13] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. 1997. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 391–411. https://doi.org/10.1145/265924.265927

[14] Andreas Schörgenhumer, Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2017. Efficient Sampling-based Lock Contention Profiling for Java. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering (ICPE '17)*. ACM, New York, NY, USA, 331–334. https://doi.org/10.1145/3030207.3030234

[15] W. Zhang, X. Ji, B. Song, S. Yu, H. Chen, T. Li, P. C. Yew, and W. Zhao. 2017. VarCatcher: A Framework for Tackling Performance Variability of Parallel Workloads on Multi-Core. *IEEE Transactions on Parallel and Distributed Systems* 28, 4 (April 2017), 1215–1228. https://doi.org/10.1109/TPDS.2016.2613524