Efficient Dynamic Analysis of the Synchronization Performance of Java Applications

Peter Hofer David Gnedt

Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems, Johannes Kepler University Linz, Austria
peter.hofer@jku.at, david.gnedt@jku.at

Hanspeter Mössenböck

Institute for System Software, Johannes Kepler University Linz, Austria
hanspeter.moessenboeck@jku.at

Abstract

Concurrent programming has become a necessity in order to benefit from recent advances in processor design. However, implementing correct and scalable synchronization in concurrent code remains a challenge. Dynamic analysis of synchronization behavior is vital to determine where more sophisticated but error-prone synchronization pays off.

We examine common approaches that developers use to identify and analyze concurrency bottlenecks in Java applications. We then describe key aspects of our ongoing research on a novel approach to Java synchronization analysis. Our approach provides developers with exhaustive information on the synchronization behavior of their application, but incurs such low overhead that it is feasible to use it for monitoring production systems. Unlike other methods, our approach can precisely show where optimizations have the largest impact.

Categories and Subject Descriptors C.4 [*Performance of Systems*]: Measurement Techniques

General Terms Measurement, Performance

Keywords Concurrency, Parallelism, Synchronization, Locking, Threading, Profiling, Monitoring, Java, Tracing

1. Introduction

Over decades, processor manufacturers have achieved exponential improvements in CPU performance primarily by increasing clock speeds, which has provided software developers with almost effortless performance increases. Around the year 2003, issues with heat dissipation, power consumption and current leakage have brought this development to a halt [15]. Since then, the focus has shifted toward multi-core processors, which have become the norm from server-class machines to phones. While additional cores continue to increase the computing power of processors, software developers can no longer effortlessly benefit from this improvement, but need to write explicitly concurrent code.

The main challenge in concurrent programming is the synchronization of accesses to shared resources. Subtle mistakes can cause anomalies that are difficult to detect and to debug, favoring coarse-grained synchronization which is easier to implement correctly. However, coarse-grained synchronization tends to serialize execution again and can negate any gains from parallelization. During development, it is difficult to judge in which cases more fine-grained synchronization would significantly improve performance and in which cases it would just make the application more complex and error-prone. Therefore, dynamic analysis is needed to measure the effect of synchronization on the performance of an application and to identify bottlenecks where more fine-grained synchronization would be worth the additional complexity.

Synchronization analysis is valuable not only during development, but also in production. Server applications in particular are deployed on machines with significantly more cores and memory than a developer workstation has, and must handle workloads that are often orders of magnitude larger than the workloads used for testing. Under such conditions, concurrent code can behave very differently, and it is difficult to reproduce and to debug performance bottlenecks on a smaller scale. A synchronization analysis approach that is feasible for use in a production environment should ideally have an overhead that is close to zero while still providing information suitable to identify and comprehend bottlenecks. The contributions of this paper are:

1. We examine different methods for analyzing synchronization at runtime, focusing on the Java platform.
2. We describe key aspects of our ongoing research on efficient synchronization analysis from within a Java virtual machine. Unlike other methods, our approach can precisely show where optimizations have the most impact.

WODA'15, October 26, 2015, Pittsburgh, PA, USA
© 2015 ACM. 978-1-4503-3909-4/15/10...$15.00
http://dx.doi.org/10.1145/2823363.2823367

2. Java Synchronization and its Analysis

The Java language has direct support for synchronization. Each object has a *monitor* associated with it, which is a mutual exclusion lock that can also be used for conditional waiting. Developers can insert *synchronized* blocks in their code and specify an object to synchronize on. The monitor of the specified object is then acquired before executing the statements in the block, and released when leaving the block. When threads contend for a monitor, i.e., when one thread has acquired a monitor and has entered a synchronized block and other threads are trying to enter a synchronized block using the same monitor, those threads are blocked until the owner leaves its synchronized block and releases the monitor. Monitor semantics are implemented entirely in the Java Virtual Machine (VM), usually in a very efficient way so their use only incurs significant overhead when threads actually contend for monitors [1, 12].

Java 5 introduced the *java.util.concurrent* package with classes that provide useful synchronization mechanisms, such as concurrent collections and read-write locks [8]. Most of these classes do not use monitors, but rather rely on the newly introduced *LockSupport* facility, which provides a *park* method that a thread can call to park (suspend) itself, and an *unpark* method which resumes a parked thread. Using these two methods and compare-and-set operations, the semantics of java.util.concurrent classes can be implemented entirely in Java. Because the LockSupport facility is public, application developers can also implement custom synchronization mechanisms on top of it.

In the following sections, we examine different approaches for analyzing Java synchronization and detecting bottlenecks.

2.1 Java VM Performance Counters

Some Java Virtual Machines have performance counters for synchronization-related events. This is also true for the HotSpot virtual machine [9], which is part of the popular Oracle Java Development Kit (JDK) and its open-source foundation, OpenJDK. HotSpot has counters that measure how often each thread was blocked when trying to acquire a monitor, how long it was blocked in total, and how much time it spent in conditional waiting. The cost of these counters is almost negligible. The counters can be queried using the Java Management Extensions (JMX), for example with the JConsole tool that comes with OpenJDK and the Oracle JDK.

These counters are useful for assessing whether the application or specific threads (such as the user interface thread) suffer from substantial monitor contention. However, the counters do not show which objects and classes are involved, and most significantly, they do not reveal which methods and call chains suffer from contention. Therefore, the counters provide no hints on where optimizations could be applied. Also, time that is spent parking with LockSupport is accounted as conditional waiting, which is inaccurate in many cases.

2.2 Thread Dumps

A thread dump is a snapshot of all threads of an application at a specific point in time and can be obtained from a running Java VM with tools such as *jstack.* In HotSpot, a thread dump includes each thread's state, its call chain, and even which methods in the call chain hold monitors and which java.util.concurrent locks are held by a thread. Thread dumps are useful means of diagnosis when the application has stopped responding, for example to identify a deadlock. However, because a thread dump is only a snapshot of the application, it does not show how frequently contention occurs and how much time the application spends in synchronization.

2.3 Call Profilers

Call profilers inject code into the application that records method calls and their execution times, or they periodically take samples of the current call chain in each thread. The collected data is commonly represented in a calling context tree, which shows the total time that was spent in a particular method when invoked by a particular chain of callers. Sampling profilers for Java can be implemented with very low overhead, as we have shown in earlier research [5].

To find synchronization bottlenecks in an application, a user would look at the methods where the most time was spent, and would identify those methods which use synchronization and thus likely spent a significant portion of their time being blocked. However, those methods which were being blocked are not necessarily the same methods that blocked other methods. Consider an application with two methods *lookup* and *insert* that both access the same data structure and use the same lock to synchronize accesses. The *lookup* method is used very frequently from different threads, but it is fast and holds the lock only for a short period of time. The *insert* method is called less frequently and only from a single thread, but it performs expensive operations, so the method holds the lock for a long period of time, during which it blocks the execution of *lookup* in other threads. The call profile would show significantly more time spent in *lookup* than in *insert,* while the contention is actually caused by *insert* and optimizing the *insert* method would yield the most speedup.

Because call profilers often cannot show only the time that is spent in synchronization, a user must be sufficiently familiar with the application to recognize those methods which use synchronization, or must examine the source code of methods. Call profilers also typically do not allow the user to view the calling context tree of an individual thread or of a group of threads, and provide no information on the lock objects or their classes.

2.4 Monitor Profilers

The Java Virtual Machine Tool Interface (JVMTI) is a native programming interface that allows debuggers, profilers, and similar tools to interact with the Java VM and the application running on top of it [11]. JVMTI provides specific

functionality to query and observe the use of monitors in a Java application, in particular the two events *MonitorContendedEnter* and *MonitorContendedEntered*. Profilers can register callbacks for these events, which are called when a thread fails to acquire a monitor and is blocked, and later when the thread has acquired the monitor after being blocked, respectively. In the callbacks, a profiler can collect information on the contending thread, the thread's current call chain, the object to which the monitor belongs, and its class. A major advantage of the two events is that they are only fired when contention occurs and that they cause hardly any overhead when contention is low. JVMTI also provides events for tracking conditional waiting with monitors, but not for parking using LockSupport.

HPROF is a profiling tool which is bundled with OpenJDK and with the Oracle JDK and has a monitor contention profiling mode that uses these capabilities of JVMTI [10]. It collects data on the time that the application spent being blocked on monitors, broken down by the thread being blocked, by the call chain, and further by the object class (but not by individual objects). HPROF also writes a log of when threads started, when they ended, and when time was spent in conditional waiting. This information is comprehensive enough to assess whether bottlenecks occurred and which threads, methods and object classes were affected. However, HPROF also only records which threads and call chains were being blocked, but not those that blocked other threads and call chains by holding a contended monitor. This can lead to wrong conclusions, as shown in the earlier example. We used HPROF with the benchmarks of the DaCapo benchmark suite [2] and Scala Benchmarking Project [14] and limited the length of recorded call chains to 32 frames. We found that the overhead was low or negligible for benchmarks with little contention, but also saw overheads from 20% to 250% in the five most affected benchmarks (avrora, scalatest, actors, specs, and tradebeans).

2.5 Instrumentation

Another approach to observe synchronization overhead is by instrumenting the application to measure how much time it spends on synchronization. Instrumentation is typically done on the Java bytecode level when a class is loaded by the Java VM. The injected code can collect arbitrary information including threads, call chains, lock objects, and classes. Instrumentation is well-suited for most java.util.concurrent classes because their synchronization semantics are implemented in Java. Therefore, the collection of data can be restricted to when contention actually occurs to decrease the overhead. One example for an instrumenting profiler for java.util.concurrent is *jucprofiler [13],* which is part of IBM's Multicore SDK.

Unlike the java.util.concurrent classes, monitor semantics are implemented entirely in the Java VM, and instrumentation on the Java level cannot determine when contention occurred. Hence, each time when a monitor is acquired must be measured, which incurs a significant performance penalty.

2.6 Java Flight Recorder

Java Flight Recorder (JFR) is a commercial, closed-source feature of the Oracle JDK that was ported from the discontinued JRockit Java VM [4]. It adds a subsystem to the HotSpot VM that efficiently records performance-related events in an application. JFR collects information on contending threads, their call chains, and the classes of lock objects. To keep the overhead low, JFR records only long contentions (more than 10ms) by default. The individual events as well as profiles created from the events can be viewed with the Java Mission Control (JMC) application that comes with the Oracle JDK. JMC can break down the recorded monitor contention time by object class, by thread, and by call chain. It can also show how much contention a thread caused by blocking other threads.

Unfortunately, JFR does not record call chains for those threads that block other threads, which would reveal where optimizations would yield the most speedup. Moreover, we found that JFR records only which thread most recently owned the monitor at the end of a contention, and considers the entire time that is spent blocking to be caused by that thread. Because more than one thread can own a monitor over the time that another thread is blocked on that monitor, we are not convinced of the accuracy of the derived caused contention times. Currently, JMC also does not provide contention statistics for java.util.concurrent locks.

3. Java Synchronization Event Tracing

We are working on a novel approach to Java synchronization analysis that provides developers with exhaustive information to understand and resolve synchronization bottlenecks in an application, yet incurs very low overhead which makes it feasible for production use. Most notably, our approach not only records the call chains of threads that are blocked, but also accurately reports the call chains of threads that block other threads by holding a contended monitor, and therefore shows where optimizations would have the largest impact. We can collect this information efficiently because we implemented our approach directly in the HotSpot VM.

Our approach observes monitor contention in the application that is running on top of the Java VM. When a thread is blocked when trying to acquire a monitor, we record an event. The event includes the duration of the contention, the blocked thread, the object and its class, and a call chain. We collect as much of this data as possible while the thread is still blocked, and encode it in an efficient binary representation. Furthermore, we avoid writing events to a shared trace buffer, which itself would require synchronization that would create contention. Instead, we assign separate trace buffers to the threads so each thread can write events without synchronization.

Figure 1 shows an overview of our event tracing approach. Initially, we assign each thread T_i a trace buffer to write

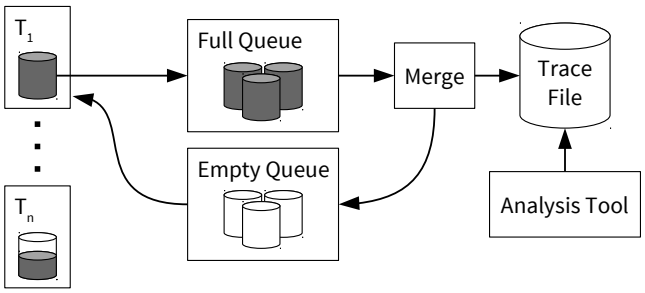

Figure 1. Event Tracing Overview

events to. When a thread's trace buffer is full, the thread submits it to the Full Queue, and then obtains a new buffer from the Empty Queue. A background thread retrieves the trace buffers from the Full Queue to merge their events into a single trace file, and places the processed trace buffers in the Empty Queue. The generated trace file can be opened in an analysis tool for offline analysis. However, our design also allows for an online analysis of the recorded events, either in the background thread that processes the trace buffers, or by streaming the trace data to another process or computer.

To determine the causes of contention, we record events not only in those threads that are blocked, but also in those threads which block other threads by holding a contended monitor. A thread which is blocked on a monitor registers on an internal queue of the monitor. Before a thread releases a monitor, we check whether that monitor's queue contains any blocked threads. If so, we write an event to record which thread held the monitor (and thus caused the contention) and the thread's call chain. We delay writing this event until after the thread has released the monitor to avoid unnecessary further contention.

During analysis, we correlate the events that are recorded in the blocked threads with the events that are recorded in the owner threads. Therefore, we can determine which thread blocked which other threads, and for how long. With the recorded call chains, we can infer which call chains blocked which other call chains for how long, and therefore see where in the code optimizations would have the largest impact. Moreover, we can also determine which objects are heavily contended, or which threads and call chains experience contention for a particular object.

While we consider call chains to be vital for understanding and fixing synchronization bottlenecks, we found that walking the stack to construct them is one of the most expensive operations in our approach. However, we have been able to adapt our incremental stack tracing technique [6] to significantly reduce the overhead of constructing call chains. Moreover, we found that there are typically very few distinct call chains in an application where contention occurs. Therefore, we maintain a set of known call chains to each of which we assign unique identifiers. When an event's call chain matches a known call chain, we only record its identifier in the event instead of writing the entire call chain.

We measured the overhead of our current implementation that is based on OpenJDK 8u45. We used the DaCapo benchmark suite[1] [2] and the Scala Benchmarking Project [14]. We executed the benchmarks under openSUSE Linux 13.2 with a quad-core Intel Core i7-4790K processor and 16 GB of memory. To get more reproducible results, we disabled hyperthreading, turbo boost and dynamic frequency scaling and used a fixed Java heap size of 8 GB. With the exception of system services, no other processes were running during our measurements. We executed each benchmark several times and accounted for warmup of the Java VM by executing 35 warmup iterations before measuring 10 iterations. Moreover, we reinitialized event tracing at the start of each iteration.

Our measurements showed geometric mean overheads of only 1.12% for the multi-threaded benchmarks when writing a compressed trace file. Only the *pmd, tradebeans, actors, tomcat* and *scalatest* benchmarks showed significantly higher overheads between 2.4% and 5.6%. For the single-threaded benchmarks, we measured a mean overhead of around 0.42%, which is caused in part by the trace buffer management, and in part because the JDK itself uses multi-threading and synchronization, which we monitor as well. Even when processing the recorded events online in the same process to compute basic contention statistics, the mean overhead for the multi-threaded benchmarks is only 1.84%, while the mean overhead for the single-threaded benchmarks is 0.54%. We consider these overheads low enough for production use.

Space overhead is also an important factor for production use. We chose a size of 16 KB for our trace buffers and found that we never use more than 8 MB of memory for trace buffers, and typically even less than 1 MB. For most benchmarks, we generate less than 50 KB of raw trace data per second. Tracing *avrora, tradebeans* and *pmd* generates the most data at 26 MB, 12 MB, and 6 MB per second, respectively. However, we support on-the-fly compression that typically saves between 60% and 70% of space and only increases the mean runtime overhead for the multi-threaded benchmarks to 1.55%. For *avrora,* the compression reduces the data rate to less than 10 MB/s. In future work, we plan to support limiting the trace file size, and automatically discarding old events. Hence, the most recent 24 hours of trace data from an *avrora*-type application could be retained on a 1 TB hard disk, which should suffice to analyze recent performance problems.

The research we described in this paper is work in progress. In addition to refining and tuning event recording and analysis, we are working on a versatile visualization tool that can aggregate contentions by multiple aspects at once. For example, a user could choose to aggregate contentions first by *object*, then by the *contender's call chain,* and then by the *lock owner's call chain*. This could reveal that there is a single HashMap object that is heavily contended, that there are three call chains where that contention occurs, and that there is a

[1] We did not use the DaCapo suite's *batik* and *eclipse* benchmarks because they do not run on OpenJDK 8.

single call chain that causes almost all of the contention of those three call chains. The user could then choose to optimize the code that that particular call chain executes, or perhaps select a different data structure or implementation, such as ConcurrentHashMap. We expect that this versatile form of aggregation enables users to identify and isolate synchronization bottlenecks and their characteristics in an effective way.

Furthermore, we are working on extending our approach to support the *java.util.concurrent* classes. We intend to do so efficiently by implementing recording trace data partially in the VM code for the primitive *LockSupport* operations, while collecting additional information on synchronization semantics in the Java code through instrumentation.

4. Related Work

Tallent et al. [16] describe a sampling profiler which, like our approach, identifies which threads and call chains block other threads and call chains by holding a contended lock. The profiler associates a counter with each lock and periodically takes samples. When it samples a thread that is blocked on a lock, it increases that lock's counter. When a thread releases a lock, the thread inspects the lock's counter, and if it is non-zero, the thread "accepts the blame" and records its own call chain. The profiler was implemented for C programs and is reported to have an overhead of 5%, but determines *only* which threads and call chains blocked other threads. Our approach also records which threads and call chains were blocked by a specific thread or call chain, which we consider to have diagnostic value when reasoning about performance problems that *occur* in a specific part of an application.

David et al. [3] propose a profiler that observes *critical section pressure (CSP),* a metric which correlates the progress of threads to individual locks. When the CSP of a lock reaches a threshold over a one-second period, the profiler records the identity of the lock and a call chain from one thread that was blocked. They implemented their profiler in HotSpot and report a measured worst-case overhead of 6%. We consider this approach complementary to ours because the CSP can be computed from the events that we record.

Inoue et al. [7] describe a sampling profiler in a Java VM which uses hardware performance counters to observe where the application acquires locks and where it blocks. It constructs call chains with a probabilistic method that uses the stack depth. The profiler is claimed to have an overhead of less than 2.2%, but does not determine which threads and call chains block other threads and call chains.

5. Conclusion

We examined and compared common approaches for identifying and analyzing synchronization bottlenecks in Java applications. We further presented key aspects of our ongoing research on a novel event tracing approach for Java synchronization analysis. Unlike the other approaches that we examined, our approach accurately pinpoints the *cause* of contention to call chains and thus shows where optimizations have the largest impact. Nevertheless, our prototype implementation has such low overhead that it is feasible to use it for production systems. We are currently working on a versatile offline analysis tool as well as on support for also analyzing java.util.concurrent classes.

Acknowledgments

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria.

References

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *ACM SIGPLAN Notices*, volume 33, pages 258–268, 1998.

[2] S. M. Blackburn et al. The DaCapo benchmarks: Java benchmarking development and analysis. OOPSLA '06, pages 169–190, 2006.

[3] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously measuring critical section pressure with the free-lunch profiler. OOPSLA '14, pages 291–307, 2014.

[4] M. Hirt and M. Lagergren. *Oracle JRockit: The Definitive Guide*. Packt Publishing Ltd, 2010.

[5] P. Hofer and H. Mössenböck. Fast Java profiling with scheduling-aware stack fragment sampling and asynchronous analysis. PPPJ '14, pages 145–156, 2014.

[6] P. Hofer, D. Gnedt, and H. Mössenböck. Lightweight Java profiling with partial safepoints and incremental stack tracing. ICPE '15, pages 75–86, 2015.

[7] H. Inoue and T. Nakatani. How a Java VM can get more from a hardware performance monitor. OOPSLA '09, pages 137–154, 2009.

[8] D. Lea. The java.util.concurrent synchronizer framework. *Science of Computer Programming*, 58(3):293–309, 2005.

[9] Oracle. OpenJDK HotSpot group. `http://openjdk.java.net/groups/hotspot/`.

[10] Oracle. HPROF: A Heap/CPU Profiling Tool. `http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html`, 2015.

[11] Oracle. JVM™TI version 1.2. `http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html`, 2015.

[12] T. Pool. Lock optimizations on the HotSpot VM. Technical report, 2014.

[13] Y. Qi et al. Profiling java.util.concurrent locks. `http://www.infoq.com/articles/jucprofiler`, 2010.

[14] A. Sewe et al. Da Capo con Scala: design and analysis of a Scala benchmark suite for the Java virtual machine. OOPSLA '11, pages 657–676, 2011.

[15] H. Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[16] N. R. Tallent, J. M. Mellor-Crummey, and A. Porterfield. Analyzing lock contention in multithreaded applications. PPoPP '10, pages 269–280, 2010.