Efficient Sampling-based Lock Contention Profiling for Java

Andreas Schörgenhumer
andreas.schoergenhumer@jku.at

Peter Hofer
peter.hofer@jku.at

David Gnedt
david.gnedt@jku.at

Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems
Johannes Kepler University Linz, Austria

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

ABSTRACT

Concurrent code commonly uses locks. Choosing between simpler but less scalable and more sophisticated but error-prone locking mechanisms is difficult during development. Therefore, lock contention analysis at run-time is crucial to aid such decisions.

We present a novel sampling-based approach for collecting detailed information on lock contention in Java applications by using the Java Virtual Machine Tool Interface (JVMTI) and bytecode instrumentation. We support both intrinsic locks as well as java.util.concurrent locks. Moreover, we can determine not only where contention *occurs* but also where it is *caused*. With a mean run-time overhead of about 5%, we consider our approach suitable for use in production environments.

Keywords

Locking, Contention, Java, Concurrency, Parallelism, Threading, Synchronization, Profiling, Monitoring

1. INTRODUCTION

Multi-core hardware has become widely available and offers a considerable performance improvement over single-core systems. However, developers have to explicitly write concurrent software to make use of these capabilities. With concurrent programming, synchronization is necessary to safely access shared resources, which is commonly achieved with locks. When using locks incorrectly, anomalies and bugs can occur that are difficult to detect, locate and fix. Coarse-grained locking is simpler and less error-prone, but it can lead to higher lock contention, which is when multiple threads try to acquire the same lock at the same time. Fine-grained locking can alleviate this problem, but it may be more difficult to implement correctly. Deciding on an appropriate locking mechanism when developing concurrent software is not an easy task. Lock contention analysis at run-time is crucial to reveal performance bottlenecks. These can then be resolved by opting for a more complex but also more scalable locking mechanism.

In earlier work, we presented an efficient lock contention profiler directly in the OpenJDK HotSpot Virtual Machine (VM) that provides detailed information on lock contention in Java applications while imposing only a small performance overhead [5]. However, this approach relies on specific capabilities of the VM, which might not be available in an existing environment. In this paper, we describe an approach to lock contention analysis which relies only on the commonly available Java VM Tool Interface (JVMTI) and on bytecode instrumentation.

The main contributions of this work in progress paper are:

1. We describe a novel approach for collecting lock contention information at run-time by only using JVMTI and bytecode instrumentation. Our approach is capable of recording similar information as our previous VM-internal approach that we described in [5]. The collected data shows both where lock contention occurs and where it is caused.

2. We provide a preliminary evaluation of the run-time overhead as well as the amount of generated data of our implementation. The results demonstrate that both of these metrics are low enough so that we consider our approach to be feasible for the use in production environments.

The rest of this paper is organized as follows: Section 2 characterizes locking in Java. Section 3 presents our novel approach and describes how we collect and record information on lock contention. Section 4 evaluates the run-time overhead and the amount of generated data with our approach. In Section 5, we discuss related work and Section 6 concludes this paper.

2. LOCKING IN JAVA

Every object in Java has exactly one intrinsic lock (also known as the monitor of the object) which can be used for mutual exclusion of threads in *synchronized* code blocks. The

ICPE'17, April 22-26, 2017, L'Aquila, Italy

DOI: http://dx.doi.org/10.1145/3030207.3030234

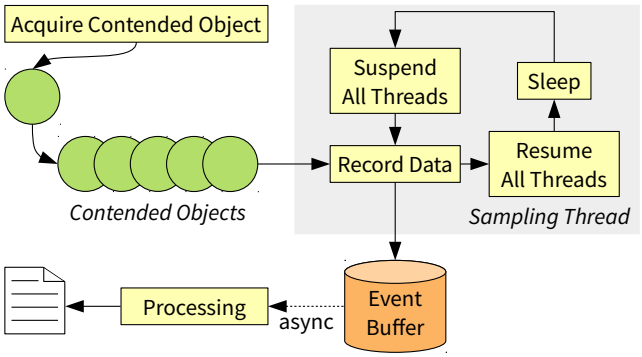

Figure 1: Overview of the sampling approach.

synchronized keyword can also be used for entire methods. Every time a thread wants to enter such a synchronized block, it must first acquire the object's monitor. All other threads that also want to enter the same block or a block protected with the same synchronization object have to wait until the lock is released again by the thread currently holding it. Intrinsic locks are always automatically released upon exiting the synchronized block, whether it is due to the normal program execution or due to an exception. Their implementation in the VM typically imposes a significant overhead only when there is actually lock contention [1, 10].

The java.util.concurrent package of Java 5 offers another way of implementing synchronization. The *LockSupport* class provides the methods *park* and *unpark* for the current thread and a given blocker object. In contrast to intrinsic locks, which are implemented in the VM, *park* and *unpark* can be used together with compare-and-set operations to implement synchronization directly in Java. The *AbstractQueuedSynchronizer* (AQS) class provides a framework for lock implementations relying on wait queues on which many ready-to-use *synchronizers* in Java are based. The main goal of synchronizers is to minimize performance overhead, most notably under contention and especially in heavily multi-threaded applications, where the overhead of intrinsic locks would be high [7]. An example for an implementation based on AQS is the *ReentrantLock* class which provides similar semantics as intrinsic locks.

3. APPROACH

Initially, we attempted to recreate our VM-internal profiler [5]. We used JVMTI callbacks for recording lock contentions and combined them with Java bytecode instrumentation to signal when a monitor is released. However, this required instrumenting every possible monitor exit, regardless of whether actual contention occurred. As a result, the overhead was infeasible and we decided to implement a sampling-based approach. We will use the phrase *contended objects* throughout this paper to conveniently address objects for which there is contention.

Figure 1 shows an overview of our sampling-based approach. We manage a list of possibly contended objects for both intrinsic locks as well as for java.util.concurrent locks. For intrinsic locks the contended objects are the lock objects used in the synchronized blocks, for java.util.concurrent they are the blocker objects. In the loop of the sampling thread, we periodically inspect each of the objects in this list. If an object is still contended at that time, we record contention data which consists of the object itself, the owner thread, all threads that are currently waiting on this object and the stack traces of these threads. While taking the sample we suspend all threads in order to guarantee that the recorded stack traces correspond to the locations where the contentions occurred. The collected data is stored in events that are written into a buffer. This buffer is asynchronously processed and the results are written to a file which can then be analyzed with the same visualization tool as described in [5]. The processing includes an optional online analysis which can be viewed in the visualization tool at run-time.

Our analysis not only reveals which threads were blocked (where and how long) but also which threads were responsible for the blocking. This is possible because our samples record the contended object and its owner thread for all waiting threads. A description of this analysis can be found in [5].

3.1 Intrinsic Locks

For intrinsic locks we get currently contended objects from the JVMTI callback *MonitorContendedEnter* (MCE). This callback is invoked whenever a thread cannot acquire an object's lock and must wait because another thread currently holds this lock. We add every newly encountered object to a list of intrinsic lock objects and assign an additional data structure to it using JVMTI's object tagging capability. The data structure contains a unique identifier (see Section 3.4) and a contention flag which indicates whether the object is contended. The flag is initially set. The sampling loop periodically inspects each object in the list by calling JVMTI's *GetObjectMonitorUsage* (GOMU). This function returns the object's owner thread as well as all threads that are currently waiting for this object's lock. Since we do not capture when a contention is resolved, the list can contain objects which are no longer contended, in which case GOMU returns no waiting threads. If an object is no longer contended, we clear its contention flag and remove it from the list. In case the MCE callback is invoked again for this object, we simply set the contention flag and reinsert it into the list. We could also track uncontended objects with the *MonitorContendedEntered* callback which signals when a waiting thread finally acquires the lock. However, we decided against this because executing code in this callback would increase the duration of holding the lock which could cause more lock contention and affect our analysis results.

3.2 java.util.concurrent

The java.util.concurrent part requires bytecode instrumentation to get the list of contended objects since JVMTI does not provide the means. As explained in Section 2, AQS provides a basis for locking implementations. Hence, we modified its method *parkAndCheckInterrupt*, which invokes the *LockSupport.park* method. In the method, we introduced additional bytecode that inserts the blocker object into a list (separate from the list used for intrinsic locks). The blocker object is always the *this* object, that is, an instance of AQS. Each object in the list is inspected in the sampling loop by retrieving its owner thread and all waiting threads via the AQS interface. If an object is no longer contended, it is removed from the list.

3.3 Sampling

Using JVMTI we record stack traces of the owner thread and all waiting threads for every contended object used in

intrinsic locks and java.util.concurrent locks. After recording the stack traces all data is packed into a so-called *contention event* which is inserted in a global event buffer. An asynchronous Java thread periodically retrieves all events in this buffer, processes them and writes the result into a file, optionally using compression.

3.4 Metadata

In order to efficiently create the contention event mentioned above, we do not include information like the thread name, the class signature or the stack trace data in this event itself but rather use unique identifiers that refer to additional metadata events. This way we significantly reduce the amount of output data because we need to write the actual metadata only once and not repeatedly in every contention event. Each time a thread is started, we are notified via the corresponding JVMTI callback and record a *thread start event* with the name of the thread. When we encounter a contended object for the first time, we create a *new object event*, both for intrinsic locks and for java.util.concurrent locks. This event consists of the identity hash code and the class signature of the object. The *new stack event* stores the method signature as well as the class signature of every stack frame. It is only created once for every newly encountered stack trace, because previously recorded stack traces are already stored in a stack trace cache. All events are written to the same global buffer.

4. EVALUATION

We evaluated the run-time overhead and the amount of generated data using OpenJDK version 8u45 on a set of real-world benchmarks. We also looked at the accuracy of our approach using a synthetic benchmark.

4.1 Overhead

For simulating real-world tasks, we decided to use the multi-threaded benchmarks from both the DaCapo 9.12 [2] benchmark suite[1] and the Scala Benchmarking Project 0.1.0 [11]. Each benchmark was run with 45 iterations of which we only used the last ten iterations for our evaluation to exclude the VM's startup phase. We repeated this process 40 times to avoid other biases in the results.

All benchmarks were executed on a hyper-threaded quad-core processor Intel Core i7-4770 with 16 GB of main memory and operating under Ubuntu Linux 15.10. Dynamic frequency scaling and turbo boost were disabled for more stable and reliable results. The benchmark suite was started while no processes other than essential system services were running. We chose sampling frequencies of 20 samples/s and 100 samples/s, which we consider a reasonable trade-off between overhead and accuracy. We measured the run-time overhead by comparing the execution times of the individual benchmarks with and without our profiler.

Figure 2 shows the median execution times for all multi-threaded benchmarks. The first and third quartiles are represented by the error bars. The *G.Mean* shows the geometric mean over all benchmarks with a 50% confidence interval displayed as its error bars. The mean run-time overhead is 1.5% when sampling with 20 samples/s and 5.1% when sampling with 100 samples/s.

[1]We excluded *batik* and *eclipse* (do not work on OpenJDK 8) and *tradesoap* which constantly timed out on our system.

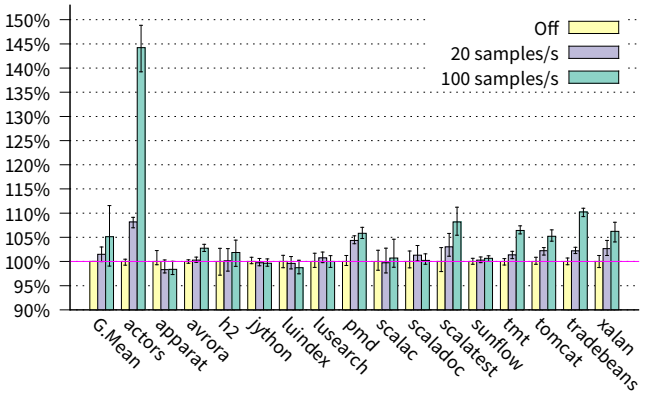

Figure 2: Run-time overhead with different sampling frequencies compared to using no profiler.

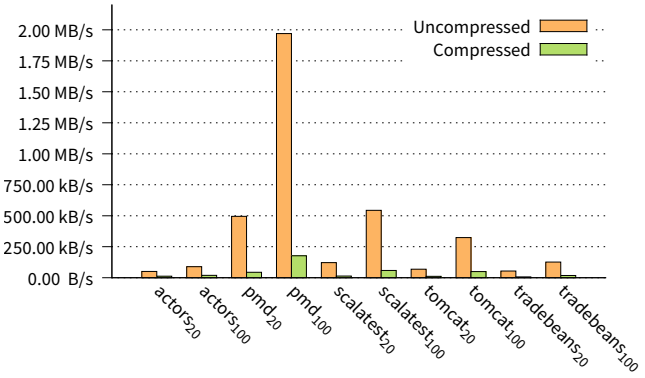

Figure 3: Output data written per second. The subscripts show the sampling frequency in samples/s.

As expected, the higher sampling frequency has a greater impact on performance. The highest overhead is caused by *actors* with 44% at 100 samples/sec. This is because the number of possibly contented objects to inspect in the sampling loop is four to 20 times larger than in all other benchmarks. On the other hand, some benchmarks like *apparat* or *luindex* even slightly gain performance. This can be attributed to small influences on thread scheduling and on garbage collection.

Figure 3 shows the average amount of data written per second for the benchmarks that yield the highest values. The largest output when sampling with 100 samples/s is produced by *pmd* with slightly below 2 MB/s, followed by *scalatest* with about 540 KB/s. For 20 samples/s the same benchmarks drop to 500 KB/s and 120 KB/s, respectively. All other benchmarks produce significantly less output, especially those that are not part of Figure 3 for which we recorded less than 50 KB/s. Enabling compression substantially decreases the amount of generated data by typically 70% to 90% and reduces it to under 200 KB/s in all cases. Moreover, the difference in run-time overhead is negligible (below 0.1%).

4.2 Accuracy

For determining the accuracy, we built a test suite using the Java Microbench Harness [8]. Our suite creates lock contention in a predefined way by varying the number of threads, lock sites and the duration of lock holding times (ranging from very short to very long contentions). This

means that we can predict which threads are waiting for how long at what lock site. We found that our sampling approach yielded the predicted results. Moreover, we executed the test suite with our VM-internal profiler which resulted in similar output. A more detailed analysis of the accuracy of our sampling approach will be part of our future work.

5. RELATED WORK

In earlier work [5], we proposed a modification of the OpenJDK HotSpot VM to record lock contention for both intrinsic locks and java.util.concurrent locks by tracing contention events. We combined and analyzed these events to identify where contention occurs and also by which threads it is caused, including detailed information on the contending object, its owner thread, all waiting threads and their call chains. Contrary to our sampling-based approach, our previous profiler records all contention with high accuracy and still causes an acceptable mean overhead of just 7.8% for multi-threaded applications. However, it relies on a modified VM which may be unsuitable for existing environments.

Another modification of the HotSpot VM is described by David et al. [3]. Their profiler monitors the so-called *critical section pressure* (CSP) for each lock. This metric represents the time threads have to wait for lock acquisition as a function of the number of threads that are running. If the CSP of a lock reaches a threshold, information about this lock as well as a stack trace from one blocked thread are collected. The authors report a worst-case overhead of 6%.

Tallent et al. [12] proposed a sampling-based lock contention profiler for C programs which associates a counter with each lock. The profiler periodically inspects all threads and for each thread that is blocked on a lock, it increases the lock's counter. When the lock is later released by its owner thread and the lock's counter is non-zero, the owner thread records the contention and its call chain. While this approach has an overhead of only 5%, it does neither record which threads had to wait nor their call chains.

Using hardware performance counters can result in even less overhead. Inoue and Nakatami [6] presented a sampling profiler that utilized such counters in a Java VM to collect information on where locks are acquired and where blocking occurs. They use a technique to record call chains using the stack depth and achieve an overhead of typically below 2.2%. However, they cannot determine the cause of contention and java.util.concurrent locks are not supported.

Another approach is used in the Java Flight Recorder (JFR) [4], a commercial tool built into the Oracle JDK. It mostly imposes only about 1% run-time overhead and provides data on the objects used in locking, which threads were blocked and their call chains. However, only contentions longer than 10 ms are recorded by default. Furthermore, JFR blames the last thread that owned the lock for the contention. This thread's call chain, however, is not recorded and threads that held the same lock before this last thread are not taken into account. java.util.concurrent locks are not supported.

Research investigating java.util.concurrent locks was conducted by Patros et al. [9]. They modified the IBM Java VM to record park contention data whenever threads are parked. The records include class data of the blocker object, thread names and stack traces. They measured where locks are held for how long and how many threads have to park. The modifications result in a overhead below 0.5%. However, they collect no information about where contention is caused.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a sampling-based lock contention profiler for Java applications that works outside the VM. Using only JVMTI and Java bytecode instrumentation, we can collect detailed contention data for both intrinsic locks as well as for locks of the java.util.concurrent framework. This data does not only show where contention occurs but also where it is caused. Our novel approach incurs only 1.5% overhead when running with a sampling frequency of 20 samples/s and 5.1% with 100 samples/s. Therefore, we consider this approach to be feasible for the use in production environments.

Future work includes a statistical evaluation of accuracy of our new profiler by comparing its contention output against the results of our VM-internal approach [5]. Furthermore, we intend to add capabilities to determine the actual lock site of java.util.concurrent locks which we currently cannot obtain from the stack traces.

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *ACM SIGPLAN Not.*, 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 et al. Efficient tracing and versatile analysis of lock contention in Java applications on the virtual machine level. ICPE '16, pages 263–274, 2016.

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

[7] D. Lea. The java.util.concurrent synchronizer framework. *Sci. Comput. Program.*, 58(3):293–309, Dec. 2005.

[8] Oracle. OpenJDK Code Tools: Java Microbench Harness (JMH). http://openjdk.java.net/projects/code-tools/jmh/.

[9] P. Patros, E. Aubanel, D. Bremner, and M. Dawson. A java util concurrent park contention tool. PMAM '15, pages 106–111, 2015.

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

[11] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da capo con scala: Design and analysis of a scala benchmark suite for the java virtual machine. OOPSLA '11, pages 657–676, 2011.

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