Submitted by
Andreas
Schörgenhumer, BSc

Submitted at
Institute for
System Software

Supervisor
o.Univ.-Prof.
Dipl.-Ing. Dr.Dr.h.c.
Hanspeter Mössenböck

Co-Supervisor
Dipl.-Ing. Dr.
Peter Hofer, BSc

February 2017

Efficient Sampling-based Lock Contention Profiling in Java

Master's Thesis
to obtain the academic degree of
Diplom-Ingenieur
in the Master's Program
Informatik

Master's Thesis

Portable Tracing of Lock Contention in Java Applications

Student: Andreas Schörgenhumer

Institute for System Software

Dipl.-Ing. Peter Hofer

Phone.: +43 732 2468-4369

Fax: +43 732 2468-4345

peter.hofer@jku.at

Linz, 8/3/2016

Concurrent programming has become necessary to benefit from current multi-core processors. The main challenge is safely accessing shared resources from multiple parallel threads, which is typically done with locks. Simple locking mechanisms are easier to implement correctly, but often suffer from high lock contention, which means that threads frequently have to wait until another thread releases a lock. During development, it is difficult to tell when a simple locking mechanism provides adequate performance and scalability, and when more sophisticated locking is worth the additional complexity. Therefore, a tool is needed that shows locking bottlenecks in an application and enables developers to make changes where they pay off.

Our research group has developed an efficient approach for recording lock contention in a Java application and implemented it in the HotSpot Java Virtual Machine [1]. In our approach, we record *events*, as shown in Figure 1. When a thread cannot immediately acquire an object's lock (monitor) and has to wait, we record a *contended enter event*, and when it finally acquires that lock, we record an *entered event*. This tells us when threads have to wait for a lock and for how long, but those are only symptoms. We also want to know which thread holds that lock, and is thus blocking those threads. Therefore, when a thread releases a lock, we check if there are any threads waiting for that lock, and if there are, we record a *contended exit event*. When we later analyze the recorded events from all threads, we can find out which threads were blocked by which other threads. More importantly, we also record stack traces in our events, so we can determine which methods cause the most lock contention and where they are called from, and enable a developer to locate the problems in the source code and make effective improvements.

Figure 1: Example of three contending threads, where T₂ is first blocked by T₁ and then by T₃

Since our approach has been implemented directly in the Java virtual machine, it has a very low overhead of typically below 10%. However, using our modified Java virtual machine is not always possible, especially in an existing production environment. The task for this thesis is to devise an approach that collects similar information, but does not require modifications to the virtual machine. Because the overhead is likely to be higher than that of our VM-internal approach, additional techniques might be beneficial to reduce the overhead, such as sampling individual contentions, or restricting tracing to individual packages or classes.

One possible approach is **Java instrumentation**, which modifies Java bytecode instructions when an application is loaded. However, Java provides only two bytecode instructions for using locks, which are *monitorenter* and *monitorexit*, and these do not expose to the application whether the lock has been contended. Therefore, a more promising approach is a **hybrid solution using instrumentation and the Java VM Tool Interface (JVMTI)**. An *agent* running in the Java VM can register for the JVMTI event *MonitorContendedEnter* and receive notifications when a lock is contended (and only then). There is no matching *MonitorContendedExit* event, but the agent could mark the lock as contended in *MonitorContendedEnter*. By additionally instrumenting *monitorexit* instructions, the code could check whether the lock has been marked, and write a corresponding event. This approach would allow for partial instrumentation. Efficiently marking locks and accessing those markings (from both native code and Java code) is a key challenge in this approach. The markings could also be stored in a weak hash map with a cache.

The expected result of this thesis is a reasonably efficient profiling tool for Java that is independent of the used Java virtual machine and provides useful information to resolve lock contention problems in the profiled application. Ideally, the profiling tool should work together with our existing analysis and visualization tools, if necessary by adapting our tool accordingly.

Peter Hofer

[1] http://mevss.jku.at/?page_id=1839

Abstract

Concurrent programming can lead to considerable performance gains on modern, parallel hardware when compared to single-core systems. When dealing with parallelization, however, developers have to explicitly address synchronization to safely access shared resources, which is typically achieved with 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 in such decisions.

We present a novel sampling-based approach for collecting detailed information on lock contention in Java applications. Our approach enables us to detect potential concurrency issues where threads have to wait unnecessarily. Our implementation only relies on the Java Virtual Machine Tool Interface (JVMTI) and on 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%, a low amount of data output and a high accuracy, we consider our approach suitable for use in production environments.

Kurzfassung

Auf moderner, paralleler Hardware können nebenläufige Programme oft erhebliche Performancegewinne gegenüber Single-Core-Systemen erzielen. Die Parallelisierung von Programmen macht es jedoch nötig, Zugriffe auf gemeinsame Daten zu synchronisieren, was normalerweise mittels Locks erreicht wird. Dabei muss man sich oft zwischen einfacheren, aber weniger skalierbaren, und ausgefeilteren, aber fehleranfälligen Lock-Mechanismen entscheiden, was oft schwer ist. In solchen Fällen ist die Laufzeitanalyse von Lock-Belegungscharakteristiken (lock contention analysis) äußerst hilfreich.

Wir präsentieren einen neuartigen, stichprobenbasierten Ansatz, um detaillierte Informationen über Lock Contention in Java-Applikationen zu sammeln. Die Analyse erlaubt uns, potentielle Nebenläufigkeitsprobleme zu erkennen, bei denen Threads unnötigerweise warten müssen. Unsere Implementierung basiert auf dem Java Virtual Machine Tool Interface (JVMTI) und auf Bytecode-Instrumentierung. Wir unterstützen sowohl Intrinsic Locks als auch `java.util.concurrent` Locks. Außerdem können wir nicht nur feststellen, wo Contention *auftritt*, sondern auch, wo sie *verursacht* wird. In Anbetracht eines durchschnittlichen Laufzeitmehraufwandes von etwa 5%, geringer Output-Datenmenge und einer hohen Genauigkeit ist unser Verfahren für den Einsatz in Produktionsumgebungen geeignet.

Acknowledgements

First, I want to thank my advisor Hanspeter Mössenböck for introducing me to this topic and offering me to work on it at the Christian Doppler Laboratory on Monitoring and Evolution of Very-Large-Scale Software Systems at the Johannes Kepler University Linz. Without his effort and encouragements, I would not be working there in the first place.

My utmost gratitude goes to my former colleague Peter Hofer, who supported me throughout the entire time we were working together. Thank you for your guidance, your invaluable insights and your willingness to help with all the problems and questions that arose along the way. I would not have been able to complete this thesis without your assistance.

I would also like to extend my thanks to the Christian Doppler Forschungsgesellschaft and to Dynatrace Austria for funding this work.

Finally, I want to thank my friends and especially my family, who supported me during my studies at the Johannes Kepler University. I could not have accomplished all this without you. Thank you very much!

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Goals	2
1.3	Publication	2
1.4	Outline	2
2	Background	3
2.1	Java	3
2.2	Java Native Interface	4
2.3	Java Virtual Machine Tool Interface	4
2.4	Java Agents	5
2.5	Locking	5
2.6	Locking in Java	6
2.6.1	Intrinsic Locks	6
2.6.2	java.util.concurrent Locks	7
2.7	Application Performance Management	8
3	Approach	11
3.1	Tracing Approach	11
3.2	Sampling-based Approach	13
3.2.1	Overview	14
3.2.2	Intrinsic Locks	15
3.2.3	java.util.concurrent Locks	17
3.2.4	Contention Processing	18
3.2.5	Metadata	18
3.2.6	Important Implementation Details	19
3.2.7	Processing Events in Java	21
4	Analysis	23
4.1	Lock Contention Aggregation	23

4.2	Lock Contention Visualizer Tool	26
5	Evaluation	31
5.1	Test Setup	31
5.2	Synthetic Benchmarks	31
5.3	Real-World Benchmarks	32
5.3.1	Performance Overhead	32
5.3.2	Data Output	34
5.4	Comparison with the VM-Internal Approach	34
5.4.1	Intrinsic Locks	37
5.4.2	java.util.concurrent Locks	42
5.4.3	Unknown Lock Contention	49
6	Related Work	57
7	Conclusion	61
7.1	Summary	61
7.2	Future Work	62
	Bibliography	63

Chapter 1

Introduction

1.1 Motivation

Today's processors have multiple cores or operate in large clusters, thereby providing a considerable performance improvement over single-core systems. To make use of this possible performance improvement, developers have to explicitly write concurrent code, utilizing multiple threads to allow for parallelization of their applications. Concurrent software, however, introduces new problems and challenges, one of which is synchronization to safely access shared resources. This is typically achieved with locks. When several threads try to access a shared resource, they have to compete for a lock. Only one thread at a time can actually hold the lock, while the others have to wait until the lock has been released before they can proceed. Since locking is complicated and error-prone, developers often resort to coarse-grained locking, which is simpler to understand but keeps waiting threads contended for a longer time. Fine-grained locking, on the other hand, reduces the contention time but suffers from a higher complexity and increases the probability of errors. Choosing the appropriate locking mechanism is not an easy task. Developers have a hard time deciding whether coarse-grained locking is sufficient or fine-grained locking should be used instead to improve performance. Therefore, lock contention analysis at run-time is essential to reveal any performance bottlenecks. Such bottlenecks can then be resolved by opting for a more scalable solution.

There are already various lock contention profilers for different languages and scenarios (cf. Chapter 6), but most of them do not capture all necessary information to actually fix any potential performance bottlenecks (e.g., where contention was caused). Those that do cannot universally be applied but rather are restricted to specific environments (e.g., bound to a specific virtual machine implementation) which may be problematic for the use in production systems.

1.2 Goals

The main goal of this master’s thesis is to develop a portable and efficient profiler for Java applications that captures lock contention data at run-time, which can then be analyzed for possible lock contention bottlenecks. The profiler should be capable of recording similar information as the one that was implemented directly in the OpenJDK HotSpot Virtual Machine (VM) [14]. However, it must not rely on a modification of the VM but remain portable while still incurring as low performance and data overhead as possible. The goals and contributions of this thesis therefore are:

- Implementing a novel approach for collecting lock contention information at run time by only using capabilities provided by the commonly available Java VM Tool Interface (JVMTI) and by bytecode instrumentation. This facilitates a VM-independent profiler.
- Collecting lock contention data from Java intrinsic locks as well as from locks of the `java.util.concurrent` framework. In both cases, the data not only shows where lock contention occurs but also where it is caused.
- Visualizing and analyzing the collected data in an external tool, which allows users to identify and locate possible lock contention bottlenecks easily.
- Evaluating the implementation regarding performance overhead, amount of generated data and accuracy. The results will show that the new profiler is reasonably accurate despite causing only a small run-time overhead and low data output.

1.3 Publication

This work was accepted for publication at the International Conference on Performance Engineering (ICPE) 2017 in L’Aquila, Italy, as a work in progress paper with the title *Efficient Sampling-based Lock Contention Profiling in Java* [36].

1.4 Outline

The rest of this thesis is organized as follows: Chapter 2 provides general information on the programming language Java, on all used interfaces, on locking in Java and on application performance management. In Chapter 3, the approach is described in detail, covering a first attempt and then continuing with our sampling-based profiling approach. Chapter 4 gives an overview of how the collected lock contention data is analyzed and visualized. In Chapter 5, the implementation of the profiler is evaluated by measuring its performance overhead and the amount of generated data as well as by determining its accuracy by comparing it to the VM-internal approach by Hofer et al. [14]. Chapter 6 covers other lock contention profilers, and Chapter 7 finally summarizes and concludes this thesis.

Chapter 2

Background

2.1 Java

Java is a high-level programming language that was originally developed by Sun Microsystems in 1995 and was then taken over by Oracle in 2010. As of the official *Java Language Specification* [10], Java is a general-purpose, concurrent, class-based and object-oriented language. Its main design goals are simplicity, portability and security. Java combines several features of C and C++ as well as other programming languages such as Smalltalk and C#. Memory management is handled completely automatically by the Java runtime, normally by means of a garbage collector inside the Java Virtual Machine. Moreover, unsafe constructs such as accessing arrays outside their boundaries and direct memory accesses (like in C with pointers) are prohibited.

Java code is typically not directly compiled to machine code but to the so-called bytecode instruction set and the Java class file format, which are defined in the *Java Virtual Machine Specification* [21]. The Java Virtual Machine (VM) is the main execution and memory management environment where Java bytecode programs are executed. As defined in the specification, the Java VM is an abstract machine which is not tailored to any particular hardware, technology or operating system. Therefore, a concrete implementation has to be provided for actually running Java programs. There are several implementations for Java VMs, such as the widely used HotSpot VM from Oracle [26] (part of the OpenJDK [31]), Oracle's jRockit VM [32] or the J9 VM from IBM [17]. It is the responsibility of these implementations to provide the platform independence for which Java is well-known.

The Java VM is the first of two parts of the *Java Runtime Environment* (JRE). The JRE also includes a large class library, which contains many useful classes and frameworks for dealing with threading, IO, networking, security, visualization, functional programming and many other features.

2.2 Java Native Interface

Java provides the capability of calling native methods via the *Java Native Interface* (JNI) which is a standard application programming interface (API) for communicating with native code in C, C++ or assembly [25]. This is helpful when having to access native libraries that cannot otherwise be used via Java directly or when it is required to manipulate memory. For instance, many IO methods of the class library rely on JNI to call the respective native IO functions. With JNI, developers can access Java objects (creation, inspection, modification) and can even call Java methods from native code. It must be noted that JNI might break Java's platform independence, namely when code is executed that is *not* platform independent (e.g., by using a library which is only available for Windows platforms).

2.3 Java Virtual Machine Tool Interface

The *Java Virtual Machine Tool Interface* (JVMTI) is a native API (written in C or C++) for profiling and monitoring purposes [29]. It allows developers to inspect the current state of the VM and also to control the execution of programs running in the Java VM. A native program that uses the functionality of JVMTI is called an *agent*. The agent runs together (i.e., in the same process) with the VM that executes the currently running application. For accessing and managing Java data types, JVMTI uses JNI. JVMTI provides multiple methods for inspecting the VM state of the currently running application. This includes, for example, the state of the heap or various information on threads. Additionally, Java classes and objects can be inspected in detail as well. JVMTI also offers capabilities for controlling the application. Most notably, these are thread-related functions, e.g., for suspending or resuming Java threads.

JVMTI is deeply intertwined with the VM and, thus, provides many direct hooks in the form of event callbacks which originate from the VM. Examples thereof are events signaling the startup and the termination of the VM, the start and the end of threads, various garbage-collection-related events, and many more. Similar to the Java VM, JVMTI is also only an abstract interface which must be provided by a concrete VM implementation. Since not all VMs offer the same amount of functionality, the available JVMTI features depend on the VM implementation as well. To cope with this problem, JVMTI includes so-called *capabilities* which indicate whether certain functions (for state inspection and execution control) or events are supported by the particular implementation of the VM. Thus, agents that rely on such features can only be run on VMs that support these features.

Since JVMTI agents are programmed in C or C++, platform independence is, of course, not guaranteed and must be handled accordingly.

2.4 Java Agents

Agents cannot only be written with JVMTI but also with the API provided by the package `java.lang.instrument` [27], which was introduced in Java 5. These agents are written in Java, and there are no programming restrictions, which means that Java agents may perform arbitrary tasks. However, the main use of Java agents is instrumentation, that is, a modification of the bytecode of Java classes. For every new class definition or redefinition, developers may transform and change the bytecode of this class before it is loaded. Changes may include introducing new fields or new methods or modifying existing methods down to the expression level. Such bytecode transformations are typically accomplished with frameworks such as ASM [3].

2.5 Locking

Locking is a common way of synchronization to safely access shared resources. Otherwise, data inconsistencies could arise when multiple threads read and/or write the same data concurrently. A classic example of such an inconsistency would be a simple counter that is incremented by two threads:

- Setup: shared counter variable $x = 0$, concurrently running threads T_1 and T_2 .
- T_1 tries to increment x by 1 and reads the current value of x which is 0 (x_1).
- T_2 also tries to increment x by 1 and reads the current value of x which is still 0 (x_2).
- T_1 stores the increment result of $x_1 + 1 = 0 + 1 = 1$ in x .
- T_2 overwrites $x = 1$ with the increment result of $x_2 + 1 = 0 + 1 = 1$.
- x is now 1 and not 2 (as expected) because T_2 read an inconsistent state of x . T_2 must not have read the value of x before T_1 finished its increment.

This particular concurrency problem can easily be solved by using a lock that protects the increment of x (this would result in an atomic increment of x). A lock can only be acquired and owned by exactly one thread, all other threads which want to acquire the same lock have to wait. This thread that obtained the lock can now safely access all shared data without any inconsistency hazards. However, using locks also introduces lock contention, which is the waiting of threads that cannot acquire a lock. While waiting, threads cannot proceed with task execution. This problem is intensified the longer a thread owns a lock, i.e., the more code the lock protects or the longer the thread needs to execute the synchronized statements before releasing the lock again. Consequently, coarse-grained locking tends to suffer from higher lock contention compared to fine-grained locking. On the other hand, fine-grained locking is more error-prone, and concurrency related bugs and anomalies are typically more difficult to detect, to locate and to fix.

2.6 Locking in Java

Java provides two lock implementations: intrinsic locks and locks from the `java.util.concurrent` package.

2.6.1 Intrinsic Locks

Every object in Java has exactly one intrinsic lock which is also known as the *monitor* of the object. A monitor can be used for mutual exclusion of threads in *synchronized* code blocks. The *synchronized* keyword can also be used for entire methods. In this case, the lock of the *this* object or of the class object (for static methods) is used. 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. In early versions, the VM would delegate locking directly to expensive locking primitives of the operating system, but nowadays they are directly implemented in the VM and typically impose a significant overhead only when there is actually lock contention [1, 16, 20, 39]. Intrinsic locks are *unfair*. This means that waiting threads do *not* necessarily acquire the lock in the same chronological order in which they first requested it. After the owner thread releases the lock, the Java VM may choose any of the waiting threads as next owner or even a new thread that just requested the lock at this moment. Unfair locking typically increases throughput [8].

Java also provides the possibility of *conditional waiting*. Every time a thread owns a monitor, it can call the methods *wait*, *notify* or *notifyAll*. *wait* causes the thread to release the monitor and to switch to a waiting state until another thread wakes it up again by calling *notify* or *notifyAll*. In contrast to the waiting time when threads cannot acquire a lock, this waiting time is deliberate and expected and is thus not considered as lock contention. Typical applications of conditional waiting are producer-consumer systems where consumers need to wait for data in order to continue working.

Figure 2.1 presents a small example of how intrinsic locks can be used. The code shows a simple implementation of a thread-safe list. The two methods *add* and *get* simply delegate their call to the internal list but wrap the calls in synchronized blocks. Both blocks use the intrinsic lock (monitor) of the *list* object. Now, multiple threads can safely use this data structure without having to fear any data inconsistencies. As mentioned above, threads automatically release the currently owned monitor upon exiting the synchronized block, also in case of any exceptions that might occur (for instance, invoking *get* with a negative index).

```
class ConcurrentList<E> {  
    private final List<E> list = new ArrayList<>();  
  
    void add(E item) {  
        synchronized (list) {  
            list.add(item);  
        }  
    }  
  
    E get(int index) {  
        synchronized (list) {  
            return list.get(index);  
        }  
    }  
}
```

Figure 2.1: Implementation of a thread-safe list with intrinsic locks.

2.6.2 java.util.concurrent Locks

The `java.util.concurrent` package of Java 5 offers another way of implementing synchronization. The `LockSupport` class provides the methods *park* (suspend thread) and *unpark* (resume thread) for the current thread and a given blocker object. The blocker object is useful for monitoring purposes because it can be used to determine the reasons why threads are suspended. 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 the performance overhead, most notably under contention and especially in heavily multi-threaded applications where the overhead of intrinsic locks would be high [19]. An example for an implementation based on AQS is the `ReentrantLock` class that provides similar semantics as intrinsic locks. The `CountDownLatch` class or the `Semaphore` class are further instances based on AQS.

The `java.util.concurrent` package additionally comes with support for conditional waiting by factoring out the respective methods for intrinsic locks into so-called *Condition* objects. Analogously to intrinsic locks, the methods *await*, *signal* and *signalAll* are provided for conditional waiting when using `java.util.concurrent` locks. However, in contrast to intrinsic locks, multiple *Condition* objects can be associated with a single lock, which enables a more fine-grained control.

Figure 2.2 shows the same example as Figure 2.1, but here, the locking is achieved with the `java.util.concurrent` framework. In addition to the internal list, a `ReentrantLock` is created. In the two methods *add* and *get*, this lock is now explicitly accessed by invoking *lock* to acquire the lock and *unlock* to release it again. Since locks are no longer automatically released in case of errors, developers have to manually address exceptional behavior. In the example, this

is done by enclosing the code after *lock* into a try-finally block to ensure that *unlock* is always called, regardless of possible exceptions.

```
class ConcurrentList<E> {
    private final List<E> list = new ArrayList<>();
    private final ReentrantLock rl = new ReentrantLock();

    void add(E item) {
        rl.lock();
        try {
            list.add(item);
        } finally {
            rl.unlock();
        }
    }

    E get(int index) {
        rl.lock();
        try {
            return list.get(index);
        } finally {
            rl.unlock();
        }
    }
}
```

Figure 2.2: Implementation of a thread-safe list with `ReentrantLock` from `java.util.concurrent`.

2.7 Application Performance Management

Application Performance Management is a general term that covers monitoring and profiling in various areas such as memory, network traffic, threads (for example, measuring lock contention), transactions and more. Profiling approaches should provide as much and as accurate information as possible while imposing minimal overhead, ideally without any major restrictions. Generally, not all aspects can be fulfilled, and developers have to opt for a trade-off between detail of information, accuracy, performance and universal application of their profiler.

There are many ways to profile an application. A common approach is to implement sampling-based profilers. Sampling gathers data periodically in contrast to collecting it continuously. It comes with a few essential advantages such as its scalability and its lower overhead. Reducing the sampling rate typically results in decreased overhead, which can also be done dynamically to adapt to the current situation. Moreover, sampling-based approaches often require less intrusive techniques than extensive instrumentation or direct modification of existing source code. Of course, there are disadvantages as well. Incompleteness (e.g., calls of a specific method are completely missed) and inaccuracy (e.g., not all method calls are caught) are the most problematic issues. Since sampling only gathers parts of the whole

information, sampling profilers cannot make any guarantees but only provide results that are statistically correct to a certain degree.

There exists another problem which is inherent to all profiling approaches collecting time-related or time-sensitive information. Whenever additional code has to be executed, this necessarily affects the execution of the inspected application as well, which is described by Mytkowicz et al. as the *observer effect* [23]. This can lead to small changes regarding memory management (for instance, writing profiling data into memory can cause cache misses and additional memory requests [15]) or thread scheduling, which can, in turn, impact things such as garbage collection, execution times of threads, lock contention or thread scheduling in general. This also applies to continuous and, in theory, to 100% accurate profiling approaches. Therefore, the collected data actually may not accurately represent the *real* state and behavior of an application. However, the lower the overall impact of the profiler is, the more this problem should be alleviated.

Chapter 3

Approach

3.1 Tracing Approach

Our first idea was to adapt the Java VM-internal approach described by Hofer et al. in [14]. That is, we wanted to create a profiler that accurately captures all occurrences of lock contention, including where contention occurs and where it is caused. Hofer et al. implemented a tracing approach where they record events for lock contention. They then aggregate and visualize those events so the users can inspect the locking behavior of their applications.

Figure 3.1 shows a small and simplified example how their event tracing works for Java intrinsic locks. There are three concurrently running threads T_1 , T_2 and T_3 and a single lock visualized by a padlock. First, T_1 acquires the lock without any problems because the lock is currently not owned by another thread. Then, T_2 also wants to acquire the same lock but fails to do so because T_1 currently holds this lock. Therefore, T_2 cannot enter the synchronized block and has to wait, causing a *Contended Enter* event (time stamp x_1) to be written. After some time, T_1 releases the lock again. Since the lock is contended (T_2 is waiting), a *Contended Exit* event (time stamp x_2) is written. By chance, T_3 quickly acquires the lock before T_2 (example of unfairness as mentioned in Section 2.6.1). Thus, T_2 has to keep waiting until T_3 releases the lock again and writes the *Contended Exit* event. T_2 can finally acquire the lock and writes a *Contended Entered* event (time stamp x_3) to signal the end of the waiting state. Based on these events, the exact locking behavior for this particular lock and the participating three threads can be extracted. As indicated by the red areas, T_2 had to wait for T_1 for the duration of $x_2 - x_1$ and for T_3 for the duration of $x_3 - x_2$.

We wanted to perform the same steps but with techniques that do not require a modification of the VM. JVMTI provides two essential callbacks that are analogous to the similarly named events in Figure 3.1. The first one is the *MonitorContendedEnter* (MCE) callback which is invoked whenever a thread cannot acquire an object's lock and must wait because another thread currently holds this lock. The second callback is *MonitorContendedEntered* (MCEd), which signals when a waiting thread finally acquires the lock. Unfortunately, JVMTI

Figure 3.1: Tracing monitors in Java.

does not offer a *MonitorContendedExit* callback. Therefore, we had to use bytecode instrumentation.

When Java code is compiled to bytecode, a synchronized block will always be transformed into a *monitorenter* instruction (thread tries to acquire the lock), followed by the bytecode of the enclosed code part, and into multiple *monitorexit* instructions (thread releases the lock; one instruction for each possible exit point: synchronized block simply ends, return statements in the block, uncaught exceptions in the block). For every *monitorexit*, we added a native call into our JVMTI agent and checked whether the monitor is actually contended or not. The *synchronized* keyword can also be applied to methods which does not result in any *monitorenter* or *monitorexit* instructions in the bytecode. In this case, we simply added the same native call whenever the method returns (normally or exceptionally). This way we could simulate the missing *Contended Exit* in Figure 3.1.

Figure 3.2 gives an overview of how lock contention data of monitors is collected. There are two agents, a native JVMTI agent and a Java agent which is used for instrumentation. The native agent utilizes the two above mentioned JVMTI callbacks MCE and MCED. Every time we receive a MCE callback, we append an additional data structure to the object whose monitor is contended. This data structure is added via JVMTI's object tagging capability where we can assign a single value (the tag – usually a pointer) to each object. In the data structure, we store information of the object itself and a list of all threads that are currently waiting for the object's monitor. If the object already has a data structure (some other thread or threads invoked the MCE callback earlier), we just add the current thread to the thread list. As a result, we get a collection of tagged objects, which are all objects whose monitors are currently contended (indicated via the *A*, *B* and *C* circles in the figure). In the MCED callback, we perform the opposite: Whenever a thread invokes this callback, we get

the data structure of the object whose monitor was just acquired and remove this thread from the data structure's thread list. The Java agent inserts a native call into our JVMTI agent whenever a monitor is released again (called *Monitor Exit* in the figure), regardless of actual contention. Hence, we have to check in native code whether this monitor is actually contended (a *contended* monitor exit). If the corresponding object has a data structure assigned and the thread list is not empty, then there is indeed contention which we collect.

Figure 3.2: Overview of the tracing approach for collecting lock contention data of monitors. The grey background represents actions performed in the JVMTI agent, the white background actions in Java.

After having implemented a functional prototype, we decided to test it before integrating further features such as recording stack traces and adding support for `java.util.concurrent` locks. Since we had to inspect *every* monitor exit, regardless of whether such an exit is actually contended or, in fact, irrelevant, we expected the overhead to be rather significant. Figure 3.3 shows the performance overhead of our tracing approach on a set of standard real-world benchmarks (cf. Chapter 5 for a more detailed explanation of the test setup and the benchmarks). *No Tracing* represents the baseline without using our profiling agents, *Tracing Enabled* are the results when executing the benchmarks with our agents. As it turned out, the overhead was simply too high to continue with this approach and we chose to implement a sampling-based approach instead.

3.2 Sampling-based Approach

As pointed out in Section 2.7, sampling-based profiling is ideal to reduce the performance overhead from which our previous approach was heavily suffering. We hazarded the consequences sampling entails and were able to successfully implement a similarly powerful sampling-based

Figure 3.3: Preliminary performance overhead of the tracing approach.

approach for recording lock contention. Section 3.2.1 will provide an overview, which will then be described in more detail in the following sections. For conveniently addressing objects whose monitors suffer from contention or blocker objects used in `java.util.concurrent` locks, we introduce the phrase *contended objects*.

3.2.1 Overview

Figure 3.4 shows a coarse 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 for this object and the stack traces of these threads. The collected data is stored in *contention events* that are written into an event buffer. This buffer is asynchronously processed by a separate Java program, and the results are written to a file which can then be analyzed with the same visualization tool as described in [14]. The processing includes an optional data reduction (data that is not needed and can thus be discarded immediately) and the option to stream the data on a socket connection rather than writing it to a file. The stream can also be viewed live in the visualization tool at run time. Both the file output and the socket connection output can optionally be compressed.

While taking the sample, we suspend all threads in order to guarantee that the recorded stack traces correspond to the locations where contention occurred. This ensures a consistent snapshot of all thread states. After recording contention data, we resume all threads again and let the sampling thread sleep for a randomized duration based on the specified sampling interval. More formally, the thread sleeps for a randomized duration in the range of $[0; 2s)$, where s is the specified sampling interval. For instance, if $s = 10$ ms, then the sleep duration would be in the range of $[0; 20)$ ms. If we sampled strictly every s seconds, the profiler would interfere with program activities that are performed also at fixed intervals. Therefore, this randomization is necessary to avoid getting biased data [23].

We additionally collect metadata which we use to efficiently store lock contention information in the events created in the sampling loop. Metadata is also stored in events, which are put into the same event buffer as the contention events.

Figure 3.4: Coarse overview of the sampling-based approach.

3.2.2 Intrinsic Locks

Like in the tracing implementation, we get the currently contended objects for intrinsic locks from the JVMTI callback `MonitorContendedEnter` (MCE). As previously explained, 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 (ID; cf. Section 3.2.5) and a contention flag which indicates whether the object is currently contended. The flag is initially set, i.e., it is set after the first MCE callback for this object. Figure 3.5 shows an example of how an object is assigned a new data structure and is added to the list of contended objects.

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

Figure 3.5: Example of how intrinsic locks are handled via MCE and the list of contended objects.

all threads that are currently waiting for this object's lock, including conditionally waiting threads. 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. Figure 3.6 shows an example of this case. We also perform the same actions when the only waiting threads that GOMU returns are conditionally waiting threads, because we do not consider this as lock contention (cf. Section 2.6.1). The data structure must only be deleted when the object itself is deleted. Otherwise, we would lose the unique identifier. In case the object is contended again at some point, we would then assign a new identifier and, thus, wrongly handle the same object as a distinct one. If the MCE callback is invoked again for an object which was removed from the list, we simply set the contention flag again and reinsert it into the list.

Figure 3.6: Example of GOMU returning no waiting threads for an object which is no longer contended.

We could also track uncontended objects with the `MonitorContendedEntered` callback that 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 would affect our analysis results.

3.2.3 java.util.concurrent Locks

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 2.6.2, the `AbstractQueuedSynchronizer` (AQS) class provides a basis for locking implementations with wait queues. In the AQS class, there is a convenience method `parkAndCheckInterrupt` (displayed in Figure 3.7) which invokes the `LockSupport.park` method and is used throughout the AQS-code. Hence, this method is the perfect spot for applying bytecode instrumentation to intercept lock contention information of `java.util.concurrent` locks.

```
private final boolean parkAndCheckInterrupt() {
    LockSupport.park(this);
    return Thread.interrupted();
}
```

Figure 3.7: The original implementation of `parkAndCheckInterrupt` of AQS.

In this method, we introduce additional bytecode that adds a method call which inserts the blocker object into a list of contended objects (separate from the list used for intrinsic locks). Figure 3.8 shows the modified `parkAndCheckInterrupt` method with the newly inserted `beforeAQSPark` method. The blocker object is always the `this` object, that is, an instance of AQS. This is essential since we can now use the interface of AQS on all blocker objects.

```
private final boolean parkAndCheckInterrupt() {
    at.jku.mevss.lockcontention.agent.runtime.JavaUtilConcurrent.beforeAQSPark(this);
    LockSupport.park(this);
    return Thread.interrupted();
}
```

Figure 3.8: The implementation of `parkAndCheckInterrupt` of AQS after bytecode instrumentation.

Each object in the list is inspected in the sampling loop. A unique identifier (ID) is created for the object if it has not already been assigned one. We attach the same data structure using JVMTI's object tagging mechanism as we do for intrinsic locks because it could theoretically happen that the `java.util.concurrent` blocker object itself is used as an intrinsic lock. However, as long as this is not the case, the (intrinsic lock) contention flag of the data structure is unused for blocker objects. Afterwards, the object's owner thread and all waiting threads are retrieved via the AQS interface methods `getExclusiveOwnerThread` and `getQueuedThreads` (this can be seen as an analogous step to calling GOMU for intrinsic locks). Again, it might happen that an object in the list is no longer contended because we do not know when a contention ends. If so, `getQueuedThreads` returns no waiting threads, and we remove the object from the list. Figure 3.9 shows an example of this case.

(a) Invoking *getExclusiveOwnerThread* and *getQueuedThreads* on the AQS-object *X* yield the owner thread and all waiting threads (*waiters*).

(b) The number of waiting threads was 0, so *X* is removed from the list of contended objects. The contention flag is unused.

Figure 3.9: Example of the AQS-Interface returning no waiting threads for a blocker object which is no longer contended.

3.2.4 Contention Processing

For each actually contended object in both the list for intrinsic locks and `java.util.concurrent` locks, we record stack traces of the owner thread and every waiting thread using JVMTI and pack all information into a contention event. This event stores the contention type (either intrinsic lock or `java.util.concurrent` lock), the object data, the owner thread and its current stack trace, all waiting threads and their current stack traces, a time stamp and the duration of the contention. The duration is an approximation and is set to the specified sampling interval (e.g., if the sampling interval is 10 ms, then the approximated contention duration is also 10 ms). The contention event is put into a global event buffer which is processed by an asynchronous Java thread (cf. Section 3.2.7). After all contended objects of both lists have been processed, the sampling iteration ends, and the sampling thread sleeps for the specified interval.

It must be noted that a contention event does not necessarily represent a *complete* contention. It may, of course, happen that an object is contended far longer than a single sampling interval. In this case, we only record a *part* of the total contention time. The object is still contended in the next sampling iteration, in which we collect the next portion of the contention. When the contention is finally over (e.g., after 15 sampling intervals), we simply merge all these equivalent contention events (this is done by our aggregation framework; cf. Section 4.1), which yields the final *complete* contention (e.g., $15 \cdot 10 \text{ ms} = 150 \text{ ms}$).

3.2.5 Metadata

In order to efficiently create the contention event mentioned above, we do not include information such as the thread name, the class signature or the stack trace data in this event

itself but rather use unique identifiers (thread ID, object ID, stack trace ID) 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 create a *thread start event* with the name of the thread (e.g., “Thread-7”) and with a unique ID. 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 (e.g., F14CC30A) and the class signature (e.g., “java.lang.Object”) of the object, and of a unique ID. The *new stack event* is created whenever we get a stack trace from JVMTI’s *GetStackTrace* which we have not yet encountered. It stores the method signature (e.g., “addNewUser(java.lang.String name, int age)”) as well as the class signature (e.g., “data.access.Database”) and the bytecode index (e.g., 17) of every stack frame, and a unique ID as well. All these metadata events are written to the same global buffer, where also the contention events are put.

Figure 3.10 shows all metadata events and an example of how a contention event is composed of the IDs stored in those metadata events. In the example, we receive a thread start via JVMTI’s *ThreadStart* callback at some point in time and record a thread start event with its name. We assign the unique identifier 25 to this event. At another point in time, the MCE callback is invoked and we record a new object event including all relevant data and map it to the unique identifier 213. While taking a sample, we inspect the object with the ID 25 and record the stack traces of its owner thread and all waiting threads. When calling the JVMTI method *GetStackTrace* for each of those threads, we create a new stack event (if we have not encountered it before) consisting of all required metadata information and assign it again a unique identifier. For instance, the stack trace’s ID of the owner thread of object 213 is set to 40. After all stack traces have been recorded, the contention event is assembled. The time stamp (in milliseconds) and the duration (in nanoseconds) are stored directly in the event, all other information is only referenced with the appropriate IDs (thread ID, object ID, stack trace ID – color coded with blue, red, cyan). That is, the object data is referred to by ID 213, the owner thread by ID 25, the owner’s stack trace by ID 40 and analogously all waiting threads with their stack traces.

Once a metadata event (thread start event, new object event, new stack trace event) has been created, all further contention events can reference it via its ID for the lifetime of the profiled Java application. Since no metadata is stored twice (including stack traces, cf. Section 3.2.6), this ensures a substantial reduction of the amount of generated data.

3.2.6 Important Implementation Details

There are a few important remarks to be made on the implementation itself. Firstly, we tried to minimize locking in our sampling profiler to avoid additional lock contention and delays that would distort our results. To accomplish this goal, we tried to minimize code that is executing concurrently, and for handling the event and object lists, we used lock-free data structures. A lock-free data structure is not implemented with locks but rather utilizes

Figure 3.10: All metadata events with their origins and an example of how a contention event is assembled. The creation of new object events can be caused whenever we encounter a new object in JVMTI's MCE callback as well as when handling locks from `java.util.concurrent`.

compare-and-set operations. Such operations try to update a value if and only if it has not changed (e.g., by some other thread) compared to an expected value. This comparison and conditional update are performed atomically to ensure a consistent, thread-safe state.

Another implementation feature is our stack trace cache. This is crucial in keeping the amount of generated data as low as possible because stack traces are our biggest data entities (each stack trace consists of multiple frames, which again store method and class signatures). Since stack traces do not have an execution state like threads or objects, we cannot simply assign IDs to them. Instead, we store the data structure returned by JVMTI's `GetStackTrace` under a unique ID in a custom hash set and create a new stack trace event with the complete metadata and with this ID. Whenever an equal stack trace is encountered, we simply return the ID of the entry found in the cache. This cache is particularly effective in lock contention scenarios because a lock is often used in the same context, resulting in many identical stack traces.

It may happen that JVMTI's `GetObjectMonitorUsage` returns no owner thread but nevertheless waiting threads. The same is true for `getExclusiveOwnerThread` and `getQueuedThreads` from AQS. This can occur when sampling in a "transition state" where the lock is in process of changing ownership. For instance, thread T_1 has released the lock and wakes up the waiting thread T_2 . This takes some time in which it could happen that we take a sample. The contended object would then be without an owner but with the waiting thread T_2 . In such a case, we attribute the contention to *unknown*. Section 5.4.3 covers this topic in more detail.

3.2.7 Processing Events in Java

All events that are put into the global event buffer are processed in an asynchronous Java program that is started together with the profiling agents. This program periodically retrieves events from the buffer and processes them in a specifiable way. The events are efficiently accessed via *DirectByteBuffer* objects from JNI in conjunction with the Java NIO framework [28], where memory can be directly accessed without having to copy data (i.e., the Java VM tries to perform native IO operations directly on the native memory rather than working on an intermediate buffer on the Java heap). We decided to handle events in Java rather than in native code because it is much more convenient and Java already provides a large class library for IO and network operations.

Before creating any output, events can optionally be preprocessed. This includes filtering out parts of or even entire events in order to reduce the amount of output data. For example, users can choose to ignore the identity hash code of the *new object event* or they can select to discard all *thread start events* if they are not interested in threads. Events that should be filtered or ignored can be specified in a separate configuration file.

The last step of the event processing is writing the output. There are two possibilities: One can either write the events to a file or transmit them on a socket connection. The first option creates a custom binary format file that can be opened with the visualization tool presented in Section 4.2. The file can only be analyzed offline, that is, after the profiled Java application has finished. To view the lock contention online, the user can also choose not to write the events to a file but to send them to a specified network socket connection instead. If the visualization tool is open and awaiting incoming events from the same socket connection, the results will be displayed during the run time of the profiled Java application. Both the file output and the socket output can optionally be compressed with the efficient, very high-speed compression algorithm *Snappy* [9, 38] to further decrease the size of the output data.

Chapter 4

Analysis

4.1 Lock Contention Aggregation

To analyze the collected contention data described in Chapter 3, we used the same aggregation framework as in [12] and extended it to match the capabilities and needs of our sampling-based profiler. This framework allows us to inspect lock contention in a user-defined way by grouping and aggregating the data by a set of existing *aspects*. The following aspects are available:

- *Group*: Lock contention data can be grouped by either “monitors” (intrinsic locks) or “java.util.concurrent” locks.
- *Object class*: Lock contention data can be grouped by the different classes of the contended objects.
- *Object*: Lock contention data can be grouped by the individual objects. The identity hash code is used as a distinguishing factor.
- *Owner thread*: Lock contention data can be grouped by the different lock owner threads. The thread name is used as a distinguishing factor.
- *Contending thread*: Lock contention data can be grouped by the different threads that waited to acquire a lock.
- *Owner current stack trace*: Lock contention data can be grouped by the different stack traces of the lock owner threads. The method’s and the class’s signature of each stack trace frame are used as a distinguishing factor. Stack traces are considered equal if they are composed of the same stack frames in the same order.
- *Owner stack trace*: Same as before, but the stack trace is cut off at the frame where the owner thread first acquired the lock. In other words, this is the stack trace up to the method in which the lock was acquired by the owner thread for the first time.

- *Owner method*: Lock contention data can be grouped by the different methods of the lock owner threads where the lock was first acquired. That is, it can be grouped by the topmost frame of the *owner stack trace* aspect. The method's and the class's signature are used as a distinguishing factor.
- *Owner current top method*: Like before, but the grouping is done by the topmost frame of the *owner current stack trace* aspect.
- *Contending stack trace*: Lock contention data can be grouped by the different stack traces of the threads that waited to acquire a lock.
- *Contending method*: Lock contention data can be grouped by the different methods of the threads that waited to acquire a lock. That is, the topmost frame of the *contending stack trace* aspect.

Users may choose arbitrary and even multiple aspects in any order to group the lock contention data. The selected aspects in the specified order will then be used to compute an *aggregator* tree which is composed of aggregators that match those aspects. Each aggregator corresponds to exactly one aspect and consists of a type and of aggregation nodes. The type specifies how lock contention data is distributed among the different nodes, which can be described as “instances” of that type. The aggregation nodes collect and group all incoming lock contention data. For each type, there can be multiple nodes, depending on how many different instances of the type exist.

The following example shows how such an aggregator tree is built from the aspects the user selected. Given two aspects and their ordering *object class* $\rightarrow$ *owner thread*, an aggregator tree with two levels is created, which is shown in Figure 4.1. The first level (Aggregator₁) consists of the *object class* type and its n aggregation nodes (abbreviated to Σ -Node), the second level (Aggregator₂) consists of the *owner thread* type and its aggregation nodes. The figure shows the raw structure of the tree. Now, lock contention data can be streamed into this aggregator tree which will be grouped, distributed and aggregated according to this structure.

Figure 4.1: Example of an aggregator tree based on the aspect ordering *object class* $\rightarrow$ *owner thread*.

The input to an aggregator tree are contention objects, which simply contain all data from a *single* lock contention incident. Lock contention incidents were collected in contention events and metadata events (cf. Section 3.2) during profiling the Java application. A contention object contains the same data as a contention event, with the exception that the contention object stores only *one* of the waiting threads and its stack trace. Thus, the resulting contention object consists of the contention type (either intrinsic lock or java.util.concurrent lock), the object data, the owner thread and its stack trace, one waiting thread and its stack trace, a time stamp and the contention duration. As a consequence, one contention event may result in multiple contention objects. For instance, an event that contains five waiting threads will result in five contention objects, one for each waiting thread. Another difference is that the contention object does no longer reference other metadata events with IDs but resolves these references and stores all metadata directly.

Figure 4.2 shows an example of how two contention objects are created from a contention event. The event stores two lock contention incidents where two threads with IDs 3 and 26 (and their stack traces 7 and 32) had to wait. Therefore, two contention objects are created. Both share some common data, that is, the object data, the owner thread and its stack trace, the time stamp and the contention duration, but they differ in the contending thread and its stack trace (indicated with the colors red and blue). Before composing the contention object, all references with IDs are resolved by a lookup in a *Metadata Store*, which simply stores all metadata events (thread start events, new object events, new stack events) that were created during profiling time.

Figure 4.2: Example of how two contention objects are created from a contention event. For reasons of readability, the stack traces are displayed in a shortened form.

Another example now shows how these contention objects are streamed into an aggregator tree. Figure 4.3 shows the same aggregator tree structure as in Figure 4.1 and four contention objects that will be the input. Each of the contention objects is color-coded and has a path attached which shows where it is delegated to in the tree.

The first one is the red contention object. Aggregator₁ groups contention by *object class*. Since the red object is the first object, a new specific aggregation node with the red object's class *java.lang.Object* is created. The contention object is then passed further down the tree to Aggregator₂, which groups by *owner thread*. Again, red is the first object and thus a new specific aggregation node with red's owner thread *Thread-7* is created. Every aggregation node in this path will add red's lock contention duration to its current value (in this example, the current value of the two aggregation nodes is 0 since they have just been created). The total duration of a node is displayed in the white box in the figure. After red has been handled, the next contention object is selected.

In the same way as before, purple is first grouped by *object class* which causes the creation of a new specific aggregation node with the purple object's class *java.util.List*. Then, purple is grouped by *owner thread*. A new aggregation node must be created with purple's owner thread *Thread-7*.

The next contention object is the orange one. Orange is first grouped by *object class*. Since orange's object class *java.lang.Object* already exists, the same node can be reused. Thereby, the node's current duration value of 10 is updated by orange's lock contention duration which results in a total value of 20. However, grouping by *owner thread* requires a new specific aggregation node with orange's owner thread *worker-54*.

The last contention object is the blue one. Again, the *object class* node already exists and can be reused (the node's duration value is updated to 30). When grouping by *owner thread*, also blue's owner thread *Thread-7* already exists and can be reused as well (the node's duration value is updated to 20). After all contention objects have been processed, the aggregator tree is complete and is ready for analysis. In this example, the tree shows exactly which object classes (and to what extent) caused contention and who was responsible for this contention (owner threads).

Of course, users may specify much deeper and more detailed aggregator trees by merely selecting more aspects that they would like to inspect. The aggregation framework allows for a simple and powerful analysis, which addresses the user's needs. To further enhance the user experience, we developed a lock contention visualization tool that displays such trees and will be presented in the next section.

4.2 Lock Contention Visualizer Tool

The *Lock Contention Visualizer* (LCV) is a powerful, versatile and user-friendly tool for displaying lock contention in multiple views. We introduced it in earlier work for the VM-internal

Figure 4.3: Example of a concrete aggregation of contention objects that are input to a specific aggregator tree. The contention objects are in a shortened form that shows only the relevant information in the ordering: object class signature, owner thread name and lock contention duration (in milliseconds).

profiler [14] and have extended it to support the output generated by our new sampling-based profiler.

After having opened a file that was created with the sampling-based profiler, the user can choose three different views to analyze the lock contention data in this file. The most important view is the main *drill down* view where the user can select any of the aspects introduced in Section 4.1 in arbitrary order (the so-called drill down). This will create an aggregator tree which is displayed in the main window and automatically adapts to changes in the drill down.

Figure 4.4 shows a screenshot of the visualizer when applied to a small example application. In this application, a simple hash map which contains products is queried concurrently by four threads. There are two ways of accessing this map: either by querying by article number (fast hash lookup) or by querying by name (slow sequential lookup until the matching name is found). Both accesses are synchronized with the same lock by using the monitor of the hash map object. How a thread accesses the map is randomized. In 90% of the cases, the thread will query by article number, and in 10% of the cases, it will query by name.

After LCV has loaded the file, the drill down (or aspects) can be specified. The example shows the following selection: *group (G)* $\rightarrow$ *object class (OC)* $\rightarrow$ *contending stack trace (CST)* $\rightarrow$ *owner stack trace (OST)*. This results in the aggregation tree that is shown in the center part of Figure 4.4. The tree can be expanded at will. In the example, we simply follow the

path which caused the highest lock contention. Certain nodes in the tree provide additional information. When selecting such a node, this information is displayed in the lower part of the window. In the screenshot, a stack trace node is selected and the complete, detailed stack trace is shown.

The total execution time of the application was about 8.5 seconds which could theoretically lead to a maximum lock contention time of $4 \cdot 8.5 = 34$ seconds if all four threads would block all the time and the program would then just terminate. As depicted in Figure 4.4, the total lock contention time was 22 seconds (a rather significant amount which indicates a bottleneck) and was caused entirely (100%) by the monitors of the object class *java.util.HashMap*. At the next level in the tree (CST), contention is split into two nodes: the contending stack trace with the topmost method *queryByArticleNo* with 78.41% and the contending stack trace with the topmost method *queryByName* with 21.59% (the “(+1)” simply indicates that one method of the stack trace is not directly displayed in the tree view but has to be viewed in the detail view below). Without looking further down the tree, a user might conclude that *queryByArticleNo* should be fixed or improved since more than three quarters of lock contention was recorded there. However, our profiler allows us to also inspect where the contention is actually caused. With the *owner stack trace* level in the tree (OST), we can clearly see that, in fact, *queryByName* was almost always (77.89%) the reason why threads had to wait at *queryByArticleNo*. This is also true when threads had to wait at *queryByName* itself: 21.45% of the total waiting time was also caused by *queryByName*. Thus, the real issue can now be fixed by improving the problematic *queryByName* method.

Figure 4.4 also shows a tiny fraction of lock contention that could not be attributed to any specific owner. As explained in Section 3.2.6, such lock contention is handled as unknown.

Besides the offline analysis of the contention files, LCV also supports an online aggregation of lock contention by opening a socket connection to which the sampling-based profiler can connect. The user can then see the recorded lock contention while the application is running and can inspect how it changes over time.

LCV is publicly available for download¹ and can currently be used in conjunction with the VM-internal profiler by Hofer et al. [14], which can be downloaded for free as well².

¹http://mevss.jku.at/?page_id=1839 under *Downloads* → *Trace Analysis Tool*

²http://mevss.jku.at/?page_id=1839 under *Downloads* → *Java Development Kit*

Figure 4.4: Visualization of an aggregator tree in LCV.

Chapter 5

Evaluation

5.1 Test Setup

Our setup consisted of a hyper-threaded quad-core Intel Core i7-4770 with 16 GB of main memory, which operated under Ubuntu Linux 15.10. We disabled dynamic frequency scaling and turbo boost for more stable and reliable results. All benchmarks and other Java applications that we inspected were executed using OpenJDK version 8u45. Although our new sampling-based profiler is not restricted to this particular version, we nevertheless decided to use it as basis since this would allow a direct comparison with the VM-internal approach (cf. Section 5.4) by Hofer et al. [14] whose implementation is based on exact measurements.

5.2 Synthetic Benchmarks

Before running real-world benchmarks, we wanted to check whether our profiler worked as expected. Therefore, we created a small test suite using the Java Microbench Harness (JMH [30]). JMH is a powerful framework for writing nano, micro, milli and macro benchmarks. It addresses common issues and complications such as warm-up times or reliable, constant processing resource consumption. JMH enables developers to precisely specify the behavior of their benchmarks and, moreover, provides detailed data for further analysis.

Our test suite creates lock contention for intrinsic locks as well as for `java.util.concurrent` locks in a predefined way by varying the number of threads, lock sites (the places in the code where locks are acquired) and the duration of lock holding times. This means that we can predict which threads are waiting for how long at what lock site. We found that our sampling-based approach yielded the predicted results. Section 5.4 covers `java.util.concurrent` locks in more detail when comparing the results with those produced by the VM-internal profiler.

5.3 Real-World Benchmarks

For simulating real-world tasks, we decided to use both the DaCapo 9.12 [2] benchmark suite and the Scala Benchmarking Project 0.1.0 [37]. We had to exclude the DaCapo benchmarks *batik* and *eclipse* because they do not work on OpenJDK 8. We also had to exclude *tradesoap*, which constantly timed out on our test system. Each benchmark was run with 45 iterations (running in the same instance of the VM) of which we only used the last ten iterations for our evaluation to exclude the VM’s startup phase. We repeated this process 10 times (running in different VM instances) to avoid other biases in the results. The benchmark suite was started while no processes other than essential system services were running. The graphical user interface was disabled as well. 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.

5.3.1 Performance Overhead

Figure 5.1 shows the median execution times for every benchmark including compression, grouped by multi-threaded and single-threaded benchmarks. The first and third quartiles are represented by the error bars. The two *G.Mean* columns show the geometric mean over all benchmarks of their respective group, with a 50% confidence interval displayed as error bars. For the multi-threaded benchmarks, the mean run-time overhead is 1.6% when sampling with 20 samples/s and 4.9% when sampling with 100 samples/s. For the single-threaded benchmarks, the mean run-time overhead is under 0.1% when sampling with either 20 samples/s or 100 samples/s, respectively.

As expected, the higher sampling frequency has a greater impact on performance. The highest overhead is caused by *actors* with about 32% at 100 samples/s. This is because the number of possibly contended objects (cf. Figure 3.4 in Section 3.2.1) to inspect in the sampling loop is 4 to 20 times larger than in all other benchmarks. On the other hand, some benchmarks such as *apparat* or *factorie* even slightly gain performance. This can be attributed to small influences on thread scheduling and on garbage collection.

Another expected finding is the low overhead when running our profiler on single-threaded applications because expensive operations are only performed when lock contention occurs. We do record some information such as thread start events or garbage-collection-related lock contention (the garbage collector is running concurrently (cf. threading and synchronization in the Java system classes *Finalizer* and *Reference*) which might cause contention), but, as can be seen in the figure, the impact is negligible.

Figure 5.2 shows further run-time overhead results. In this case, recording data from `java.util.concurrent` locks was disabled; only lock contention originating from intrinsic locks was captured. We also executed the benchmarks with and without compression to determine the influence of the compression algorithm. As it turned out, disabling the `java.util.concur-`

Figure 5.1: Run-time overhead when sampling with 20 samples/s and 100 samples/s with compression enabled compared to using no profiler.

rent feature mostly impacts the overhead of *actors* (a reduction by about 5% at 100 samples/s) since this is the only benchmark that causes significant lock contention *and* utilizes `java.util.concurrent` locks. The difference between enabling and disabling compression is negligible. In some cases, compression even increases performance but by less than 0.1% and in other cases, it incurs an additional overhead of again less than 0.1%.

5.3.2 Data Output

Another interesting question is how much data is generated by our profiler. Figure 5.3 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 1.9 MB/s, followed by *scalatest* with about 500 KB/s. For 20 samples/s the same benchmarks drop to 460 KB/s and 115 KB/s, respectively. All other benchmarks produce significantly less output, especially those that are not part of Figure 5.3 for which we record less than 50 KB/s. Enabling compression substantially decreases the amount of generated data by typically 70% to 90% and reduces it to under 170 KB/s in all cases.

5.4 Comparison with the VM-Internal Approach

Running our custom JMH benchmark suite (cf. Section 5.2) was only the first step towards checking whether the sampling-based profiler yields correct results. Therefore, we decided to also evaluate the accuracy of the results of the real-world benchmarks (cf. Section 5.3). In contrast to the synthetic benchmarks, we did not know the expected lock contention of those benchmarks. To overcome this issue, we executed the same benchmarks with the VM-internal profiler [14] and used its results as the “ground truth” or accuracy baseline, respectively. As mentioned in Section 2.7, this is problematic since the modified VM may potentially yield incorrect measurements itself, resulting in an invalid ground truth. On the other hand, this is the case for all such profilers and the VM-internal profiler was the most practical choice for comparing the accuracy because it uses the same aggregation framework and causes only very low overhead while still recording *all* lock contention information.

The idea behind comparing the results of both profilers is to compare aggregator trees (cf. Section 4.1) with equal structure by computing their overlap. The overlap of two trees determines how similar they are in terms of edge weights and has often been used in other research (e.g., for calling context trees like in [7, 12, 22, 41]). Equation 5.1 shows how the overlap o of two aggregator trees AT_1 and AT_2 is calculated. e is an edge that must be part of both trees, i.e., the parent node and the child node of the edge must be equal in both trees. $rw(e, AT_x)$ is the relative weight of edge e in aggregator tree AT_x , i.e., the weight of the edge divided by the total edge weight of the tree (the sum of all relative edge weights must be 1). The overlap o is the sum of the relative weights of all edges that are part of both aggregator trees where for every edge e that is both in AT_1 and in AT_2 the minimum relative weight is

Figure 5.2: Run-time overhead without recording `java.util.concurrent` locks when sampling with 20 samples/s and 100 samples/s with compression enabled and disabled compared to using no profiler.

Figure 5.3: Output data written per second. The subscripts show the sampling rate in samples/s.

selected. This results in a value ranging from 0 (no overlap at all) to 1 (the two trees are identical in nodes and edge weights).

$$o(AT_1, AT_2) = \sum_{e \in AT_1 \cap AT_2} \min(rw(e, AT_1), rw(e, AT_2)) \quad (5.1)$$

As mentioned before, each real-world benchmark was executed with 45 iterations where we only used the last ten for the actual evaluation. For computing the overlap, we also kept this in mind. Hence, we created an aggregator tree for each single iteration of a benchmark, that is, lock contention is collected separately for each iteration. Again, we only selected the last ten for further processing. We merged these ten remaining single iteration aggregator trees into a so-called combined iteration aggregator tree. When computing the overlap, we always calculate two different values: the *intra*-overlap and the *inter*-overlap. The intra-overlap (or self-overlap) is the overlap of the individual iteration aggregator trees and the combined iteration aggregator tree. This can be used to determine how stable the results of one particular profiler or of a single benchmark itself are. On the other hand, the inter-overlap is the overlap of the combined iteration aggregator trees of two different profilers.

The last question was how the aggregator trees should look like, that is, which aspects should be selected to group lock contention data. We decided to use the following aspect ordering: *object class* $\rightarrow$ *contending stack trace* $\rightarrow$ *owner stack trace*. This is a typical approach for quickly analyzing lock contention problems because it shows which objects are responsible for contention in general, where exactly contention occurs and finally where it is caused.

However, grouping by *owner stack trace* introduces an additional problem specific to our new approach. Because we use sampling, we cannot determine the owner stack trace directly but only the owner *current* stack trace (both terms are as defined in Section 4.1). While taking a sample, we only have access to the current state of the application. Fortunately, JVMTI

provides the method *GetOwnedMonitorStackDepthInfo* (GOMSDI) which provides exactly the information where an intrinsic lock was acquired so we can extract the owner stack trace from the current stack trace. Unfortunately, the JVMTI implementation of OpenJDK's HotSpot VM does not return where the lock was acquired for the *first* time but rather where it was acquired *most recently*. This is a problem if the same lock is acquired again, e.g., in recursive scenarios or if another method is called which is protected by the same lock.

Figure 5.4 shows such a problematic case. In Figure 5.4a, some thread processes an object of class *ObjectProcessor*. First, the thread acquires the lock *myLock* in the method *process*. This method calls the *contains* method which is also protected by the same lock *myLock*. The thread acquires the same lock twice and starts to execute the *java.util.Arrays.equals* method. At this point, we happen to take a sample in our profiler and thus suspend all threads. Naturally, the recorded stack trace of our example owner thread looks like the one in Figure 5.4b, which is the *current* stack trace. We know that the owner stack trace is the same as the *current* one, except that it is cut off at the frame where the lock was *first* acquired, that is, at the frame *ObjectProcessor.process* at depth 2. Therefore, the expected owner stack trace is the one shown in Figure 5.4c. However, JVMTI's GOMSDI returns the depth where the lock was acquired *most recently*, which is at depth 1 at the frame *ObjectProcessor.contains*. This results in the stack trace in Figure 5.4d.

Albeit not wrong per se, we had to find a solution to correctly compare the sampling-based profiler with the VM-internal profiler. Thus, we decided to modify the JVMTI implementation of the HotSpot VM for the sake of comparison. We simply changed GOMSDI to return the *first* lock acquisition instead of the *most recent* one.

This modification allows us to correctly compare lock contention from intrinsic locks, but for *java.util.concurrent* locks, there is unfortunately no quick solution to determine the owner stack trace. We do not have a method or interface that is equivalent to JVMTI's GOMSDI, and we cannot determine where the lock was first acquired using the available data without redesigning our entire profiling implementation regarding *java.util.concurrent* locks. Thus, the above described overlap computation cannot be applied to these locks. The only way to compare our new profiler to the VM-internal one is to manually inspect and analyze the lock contention output of both in the Lock Contention Visualizer. Therefore, the comparison is split into two different sections. Section 5.4.1 covers intrinsic locks with the overlap computation as introduced above, and Section 5.4.2 describes the manual comparison of *java.util.concurrent* lock contention.

5.4.1 Intrinsic Locks

Figure 5.6 shows the inter-overlap of the VM-internal profiler and the sampling-based profiler for the real-world, multi-threaded benchmarks (single-threaded benchmarks cannot cause lock contention, which would result in an unnecessary comparison). We used various sampling rates ranging from 10 samples/s up to 10000 samples/s. In general, there is a slight tendency towards higher accuracy (that is, higher overlap) when sampling with higher rates.

```

class Program {
    void main() {
        ObjectProcessor myOP;
        ...
        myOP.process(testObject);
    }
}

class ObjectProcessor {
    ...
    void process(Object item) {
        synchronized (myLock) {
            if (contains(item)) {
                // process "item"
            }
        }
    }
}

boolean contains(Object item) {
    synchronized (myLock) {
        // check if "item" exists
        if (java.util.Arrays.equals(...)) ...
        // owner thread is currently in
        // "java.util.Arrays.equals"
    }
}

```

(a) When calling the method *process*, the same lock is acquired twice. The dots ... represent omitted code. The owner thread is currently in the method *java.util.Arrays.equals*.

depth	frame
0	java.util.Arrays.equals(...)
1	ObjectProcessor.contains(...)
2	ObjectProcessor.process(...)
3	Program.main()
...	further stack frames ...

(b) Owner *current* stack trace.

depth	frame
2	ObjectProcessor.process(...)
3	Program.main()
...	further stack frames ...

(c) Owner stack trace *first* acquisition.

depth	frame
1	ObjectProcessor.contains(...)
2	ObjectProcessor.process(...)
3	Program.main()
...	further stack frames ...

(d) Owner stack trace *most recent* acquisition.

Figure 5.4: Example of a thread acquiring the same lock twice and the resulting stack traces. The most recent stack trace frames are displayed at the top.

For the benchmarks *aurora*, *h2*, *sunflow*, *tmt*, *tomcat* and *tradebeans* the sampling-based profiler often reaches an accuracy of about 85% up to 95% (even with a sampling rate of 1000 or less), which are good results.

In the benchmarks *jython*, *luindex*, *scalac* and *scaladoc*, there is no lock contention, except for the contention caused by the Java garbage collector. Garbage-collection-related lock contention adds up to only a tiny absolute contention time of less than 0.1 ms compared to the execution time of one benchmark iteration. This is the reason why the sampling-based profiler does not record any data, which results in 0% overlap. Both the average execution times and the average lock contention times are listed in Figure 5.8.

The next two interesting cases are the benchmarks *apparat* and *lusearch*. In contrast to the other four benchmarks mentioned before, these two do cause lock contention but very little thereof. The average lock contention time is only 1.8 ms for *apparat* and 0.6 ms for *lusearch* (cf. Figure 5.8). Moreover, the intra-overlap of the VM-internal profiler, which is shown in Figure 5.7, is only about 28% for both benchmarks. That is, the results of the individual benchmark iterations can hardly be considered stable. These two observations explain the low overlap.

The lock contention of the benchmarks *pmd* and *scalatest* is special in the sense that there is only one main object class (that causes over 95% of total contention) but there are many very similar (though not equal) stack traces with about equal contention distribution, which results in many different and short contention units. The effect of this is also visible in the low intra-overlap of the VM-internal profiler in Figure 5.7. Since we compute the overlap with aggregator trees that use stack traces (of contending threads and owner threads), chances are that we get a lower overlap. Figure 5.5 shows the problem of short contention units (for simplicity reasons, the sampling interval is set to the fixed value s rather than the actual randomized value).

In Figure 5.5a, contention A stretches over 20 sampling iterations, which results in a decent duration estimation: $actual(A) \approx 20 \cdot s = estimation(A)$. In Figure 5.5b, the total duration of contention A is much shorter and barely stretches over 2 iterations (the contention starts just before the first sampling iteration at time stamp x_1 and ends right after the next iteration at time stamp x_2). In this case, we overestimate the actual duration by roughly 50%: $actual(A) \approx 1 \cdot s < 2 \cdot s = estimation(A)$. The problem of overestimation increases with shorter actual contention durations, which is shown in Figure 5.5c. Contention A is very short, and we see it only in one sampling iteration (it starts just before the first iteration at time stamp x_1 and ends shortly after). This results in an overestimation of about 70%: $actual(A) \approx 0.3 \cdot s < s = estimation(A)$. Of course, this might only happen from time to time, but since there are so many different, short contention incidents, and we have to inspect all the different stack traces, the error accumulates.

Therefore, we changed the aggregator tree structure to *methods* instead of stack traces: *object class* $\rightarrow$ *contending method* $\rightarrow$ *owner method*. This results in merging many similar stack traces (those that have the same method in the topmost stack frame), which leads to fewer estimation errors. With the new aspect ordering, the overlap significantly increases

(a) Contention *A* stretches over 20 iterations. The estimated duration is sufficiently accurate.

(b) Contention *A* barely stretches over 2 iterations. The estimated duration is off by about 50%.

(c) The duration of contention *A* is very short (only seen in one iteration). The estimated duration is off by about 70%.

Figure 5.5: Overestimation of short contention durations.

to about 85% for *pmd* and 65% for *scalatest* (the intra-overlap of the VM-internal profiler increases also to over 99% in both cases). The reason why *scalatest*'s accuracy does not improve further is because of approximately 30% unknown contention (contention which is not attributed to any owner thread; cf. Section 3.2.6) that leads to strongly diverging aggregator trees. The issue of unknown lock contention is covered in more detail in Section 5.4.3, in which we try to explain reasons why it occurs.

Unknown contention is even more problematic for the benchmark *xalan* where our sampling-based profiler cannot determine the lock owner in over 60% of the cases. Additionally, the intra-overlap of the VM-internal profiler is only at about 63% (Figure 5.7), which indicates unstable results in the individual benchmark iterations. Both of these observations explain the low overlap.

The most problematic benchmark is *actors*. We also record unknown contention of approximately 30%, which would theoretically still allow an overlap of about 70%. However, we cannot achieve a higher overlap than the 30% shown in Figure 5.6 because the recorded lock contention data itself partially does not match the results produced by our accurate VM-internal profiler. A possible explanation is the fact that *actors* uses a high number of threads which all cause very little lock contention (down to nanoseconds). The shorter the individual lock contention occurrences are, the higher is the chance of an overestimation of the actual lock contention duration by our sampling-based profiler (as already described above for the benchmarks *pmd* and *scalatest*). Furthermore, one difference here is that we do no longer have only one main object class, to which most of the total lock contention is attributed, but multiple ones. Secondly, we do not encounter many similar but many different stack traces. Thus, using the aggregator tree structure which replaces the stack trace aspects with method aspects does not lead to any improvement. Increasing the sampling rate helps to some extent, but even this has its limits. Figure 5.9 shows these limitations.

Figure 5.9a presents a simplified code structure of the sampling loop. The point is that every iteration of the sampling loop does not just suspend all threads at once, then record the data, resume all threads at once and then immediately sleep for the duration of the sampling interval before repeating the whole process again in the next sampling iteration. In fact, when starting to take a sample, some code *B* has to be executed *before* we even come to the part where threads are suspended. This includes checks whether we actually need the suspension (the contended objects lists could be empty). Only after running the *B* code, we can suspend the threads. This, however, cannot be done *at once* but only *sequentially*, which results in some threads still running until the suspension is fully complete. After the data is recorded, the threads are resumed but again *sequentially*, that is, it takes some time until all threads are running again. We are still not done yet because we have to calculate the randomized sleep duration based on the specified sampling interval (cf. Section 3.2.1). This is done in the *A* code *after* all threads are resumed. Upon completion of *A*, the sampling thread can finally sleep until the next iteration.

Figure 5.9b depicts why this is problematic, especially for higher sampling rates, that is, shorter sampling intervals. The capital letters refer to the steps of Figure 5.9a. Suppose we

take samples at time stamps x and $x + 1$. In the naive point of view, where every operation is performed instantaneously, Δx would represent the entire time in which threads are inactive and data is recorded. Afterwards, all threads are immediately resumed and the sampling is paused for the duration of the sampling interval s (actually, the randomized duration, but for simplicity reasons, this is neglected here; moreover, the average sleep duration will converge to s since the randomization is evenly distributed). At $x + 1$, the threads are suspended again and the next sample is taken. The recorded lock contention duration is guessed to be s , which is sufficiently correct (we might still overestimate as described above for the benchmarks *pmd* and *scalatest*). However, this scenario is different in reality. The actual duration where all threads are suspended is only ΔD because threads are still running during ΔB , ΔS , ΔR and ΔA . Thus, the lock contention duration guess of s is not really correct any more. The application threads are running longer than s , namely for the duration of $r = \Delta R + \Delta A + s + \Delta B + \Delta S$. Therefore, we suddenly underestimate our duration guess. Of course, ΔB , ΔS , ΔR and ΔA are very short, which might not have a noticeable impact when s is a large value. On the other hand, the effect becomes significant when s is small (worst case: $s \approx 0$). Thus, increasing the sampling rate, that is, decreasing the interval s , makes the distortion significant.

All these findings (unknown contention, underestimation, high number of threads with short individual contention units that lead to overestimation) are reasons for the poor accuracy of our sampling-based profiler regarding the *actors* benchmark.

5.4.2 java.util.concurrent Locks

With `java.util.concurrent` locks, we cannot automatically determine the owner stack trace with our sampling-based approach, so we have to manually compare the results of both profilers using LCV. For the VM-internal profiler, we can use the same aggregator tree as before, but for the sampling-based profiler, we have to replace the *owner stack trace* aspect with the *owner current stack trace* aspect. All of the following sampling-based results were collected when profiling with 100 samples/s.

Of the real-world benchmarks, only *actors*, *apparat*, *pmd* and *tomcat* cause actual contention from `java.util.concurrent` locks, although only the first two create noticeable contention. Hence, we show the results of these two benchmarks.

In contrast to intrinsic locks, `java.util.concurrent` lock contention in the *actors* benchmark is indeed distributed over many similar stack traces. Therefore, we decided to use the method-aggregator tree structure instead (analogous to before with intrinsic locks with the benchmarks *pmd* and *scalatest*). Figure 5.10 shows the expanded aggregator trees of both profilers for *actors*. As one can see, the amount and the distribution of lock contention shown by the sampling-based profiler strongly match the results of the VM-internal profiler. At the contending method `sat..LinkedBlockingQueue.take`, 75.6% of the total contention time is spent, compared to the sampling result of 75.89%. Approximately the same accuracy holds for the next most problematic method `juc..LinkedBlockingQueue.take` where threads wait for

Figure 5.6: Inter-overlap of the VM-internal profiler *VMInternalProf* and the sampling-based profiler *SamplingProf*. The subscripts show the sampling rate in samples/s.

Figure 5.7: Median intra-overlap of the VM-internal profiler. The error bars indicate the first and third quartile, respectively.

Benchmark	AET [ms]	ALCT [ms]
actors	2769	167.8
apparatus	6069	1.8
avrora	2063	925.3
h2	3322	3270.6
jython	1730	0
luindex	405	> 0.1
lusearch	453	0.6
pmd	877	1921.4
scalac	1048	> 0.1
scaladoc	858	> 0.1
scalatest	708	623.6
sunflow	1413	64.6
tmt	3775	255.5
tomcat	1005	2823.9
tradebeans	3092	16243.9
xalan	394	31.5

Figure 5.8: Real-world, multi-threaded benchmarks: average execution time (AET) and average lock contention time (ALCT) of one iteration when running with the VM-internal profiler.

Figure 5.9: Increasing the sampling rate (lowering the interval) has its limits.

15.24% of the total contention time, compared to our result with 12.21%. The last interesting contending method is *sat..LinkedBlockingQueue.signalNotEmpty* at which 7.01% of the total contention time is spent, compared to our result of 9.71%. The absolute duration is not correct, but this is due to the many little contention units created by multiple concurrent threads. As discussed before, this leads to a wrong estimation. However, our sampling-based profiler heavily struggles to determine the owner thread. Apparently, the *AbstractQueuedSynchronizer* interface does not return an owner thread despite having waiting threads in its queue. On the other hand, also the VM-internal profiler seems to have some problems with this benchmark since it returns a noticeable amount of unknown lock contention as well.

We also analyzed the *apparat* benchmark using the stack trace-aggregator tree structure. Figure 5.11 shows the expanded aggregator trees of both profilers. Similar to *actors*, the overall contention overlap is sufficiently high: The stack trace (+6) *LinkedBlockingQueue.take()*, ... at which the most lock contention occurred (90.89%) was also recorded as the hottest one by our sampling-based profiler (92.59%). But again, unknown contention is a troublesome issue. Possible reasons why this phenomenon is particularly manifest in *java.util.concurrent* will be described in Section 5.4.3.

Since it looked as if our profiler might not work correctly for *java.util.concurrent* locks, we used our synthetic JMH-based benchmark suite to verify whether we encountered unknown contention also in this scenario. As comparison basis, we used the results produced by the two profilers for the following custom benchmarks, which were executed for 20 s each:

▼ OC java.util.concurrent.locks.ReentrantLock\$NonfairSync	100.00% (17 s)
▼ CM sat.: LinkedBlockingQueue.take()	75.60% (12 s)
OM sat.: LinkedBlockingQueue.take()	69.06% (11 s)
OM (unknown)	4.32% (735 ms)
OM sat.: LinkedBlockingQueue.signalNotEmpty()	1.89% (321 ms)
OM sat.: LinkedBlockingQueue.offer(...)	0.34% (57 ms)
OM j.l.Thread.interrupt()	0.00% (663 µs)
▼ CM juc.: LinkedBlockingQueue.take()	15.24% (2.595 ms)
OM juc.: LinkedBlockingQueue.take()	13.27% (2.260 ms)
OM juc.: LinkedBlockingQueue.offer(...)	0.90% (153 ms)
OM (unknown)	0.86% (146 ms)
OM juc.: LinkedBlockingQueue.signalNotEmpty()	0.20% (33 ms)
OM ad.DefaultCompletableFuture.completeWithResult(...)	0.00% (739 µs)
▼ CM sat.: LinkedBlockingQueue.signalNotEmpty()	7.01% (1.194 ms)
OM sat.: LinkedBlockingQueue.take()	6.33% (1.078 ms)
OM (unknown)	0.44% (75 ms)
OM sat.: LinkedBlockingQueue.signalNotEmpty()	0.24% (40 ms)
OM sat.: LinkedBlockingQueue.offer(...)	0.01% (908 µs)
▼ CM juc.: LinkedBlockingQueue.signalNotEmpty()	1.86% (316 ms)
▼ CM sat.: LinkedBlockingQueue.offer(...)	0.20% (734 ms)

(a) Expanded aggregator tree snippet of the VM-internal profiler.

▼ OC java.util.concurrent.locks.ReentrantLock\$NonfairSync	100.00% (13 s)
▼ CM sat.: LinkedBlockingQueue.take()	75.89% (10 s)
OCST (unknown)	75.89% (10 s)
▼ CM juc.: LinkedBlockingQueue.take()	12.21% (1.610 ms)
OCST (unknown)	12.21% (1.610 ms)
▼ CM sat.: LinkedBlockingQueue.signalNotEmpty()	9.17% (1.210 ms)
OCST (unknown)	9.17% (1.210 ms)
▼ CM juc.: LinkedBlockingQueue.signalNotEmpty()	2.50% (330 ms)
▼ CM sat.: LinkedBlockingQueue.offer(...)	0.23% (30 ms)

(b) Expanded aggregator tree of the sampling-based profiler.

Figure 5.10: Aggregator trees in LCV for one iteration of the actors benchmark.

▼ OC java.util.concurrent.locks.ReentrantLock\$NonfairSync	100,00% (314 ms)
▼ CST (+6), LinkedBlockingQueue.take(), ThreadPoolExecutor.getTask(), ThreadPoolExecutor.runWorker(...), (+2)	90,99% (286 ms)
OST (+6), LinkedBlockingQueue.take(), ThreadPoolExecutor.getTask(), ThreadPoolExecutor.runWorker(...), (+2)	57,17% (179 ms)
OST (+5), LinkedBlockingQueue.take(), ThreadPoolExecutor.getTask(), ThreadPoolExecutor.runWorker(...), (+2)	26,85% (84 ms)
OST (unknown)	6,57% (20 ms)
OST (+5), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+33)	0,35% (1,090 µs)
OST (+5), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+15)	0,05% (147 µs)
OST (+5), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+45)	0,00% (12 µs)
► CST (+7), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), ResizableThreadPoolScheduler.execute(...), (+14)	5,01% (15 ms)
► CST (+7), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+15)	1,70% (5,350 µs)
► CST (+5), LinkedBlockingQueue.take(), ThreadPoolExecutor.getTask(), ThreadPoolExecutor.runWorker(...), (+2)	1,44% (4,534 µs)
► CST (+7), LinkedBlockingQueue.offer(...), ResizableThreadPoolScheduler.execute(...), (+32)	0,60% (1,874 µs)
► CST (+7), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+33)	0,13% (396 µs)
► CST (+7), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+40)	0,11% (357 µs)
► CST (+7), ThreadPoolExecutor.awaitTermination(...), EventRequester.lambda\$start\$0(...), EventRequester.lambda\$start\$1.run(), (+1)	0,01% (34 µs)
► CST (+7), ThreadPoolExecutor.processWorkerExit(...), ThreadPoolExecutor.runWorker(...), ThreadPoolExecutor.run(), (+1)	0,01% (22 µs)
► CST (+7), LinkedBlockingQueue.signalNotEmpty(), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), (+41)	0,00% (10 µs)

(a) Expanded aggregator tree of the VM-internal profiler.

▼ OC java.util.concurrent.locks.ReentrantLock\$NonfairSync	100,00% (270 ms)
▼ CST (+6), LinkedBlockingQueue.take(), ThreadPoolExecutor.getTask(), ThreadPoolExecutor.runWorker(...), (+2)	92,59% (250 ms)
OCST (unknown)	92,59% (250 ms)
► CST (+7), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), ResizableThreadPoolScheduler.execute(...), (+32)	3,70% (10 ms)
► CST (+7), LinkedBlockingQueue.offer(...), ThreadPoolExecutor.execute(...), ResizableThreadPoolScheduler.execute(...), (+14)	3,70% (10 ms)

(b) Expanded aggregator tree of the sampling-based profiler.

Figure 5.11: Aggregator trees in LCV for one iteration of the *apparatus* benchmark.

- **Benchmark 16-8-equal:** In this benchmark, 16 threads are running concurrently. Each one tries to acquire a lock from one of the eight possible lock sites (eight different methods) which are all protected by the same `ReentrantLock` from `java.util.concurrent`. When holding the lock, the owner thread has to execute parameterizable JMH-code that consumes CPU time (a specified number of tokens are processed, which always takes about the same time on a fixed test system and, hence, determines how long the lock should be owned). Every lock site consumes an equal amount of CPU time, that is, it causes an equal amount of lock contention.

Figure 5.12 shows the results in LCV. For comparing the correctness, we first have to check if the *contending stack trace* nodes match. In both aggregator trees, there are eight main stack traces (one for each lock site). At each of them, approximately the same amount of time had to be waited (between 12% and 13% of the total lock contention time). Then, we examined the *owner current stack trace* more closely. As expected, the sampling-based profiler yields almost always the *Blackhole.consume* method as the top frame of the stack trace since this is the method that consumes the CPU time and does all the work in the benchmark. However, we can easily extract the *owner stack trace* by simply discarding this particular stack frame which leaves the *Contention\$Base.contend* method as the new top frame – the same as that of the VM-internal profiler. Now we can compare the results of both profilers and see that the aggregator trees are very similar. Thus, our sampling-based profiler has a high accuracy.

- **Benchmark 16-8-equal-short:** This is the same as the benchmark *16-8-equal*, but the lock holding time (the amount of consumed CPU time) is significantly shorter. This results in very short lock contention units.

Figure 5.13 shows the results in LCV. In contrast to before, however, the amount of unknown contention has increased drastically, which, in turn, has a negative impact on the accuracy of our sampling-based profiler. The *contending stack trace* overlap is still high (both profilers yield similar values between 12% and 13%) but, for example, 23 s of the 35 s of the contending stack trace (+7), *Contention\$Base.contend(...)*, *Contention\$EightSite.site6(...)*, ... (in the following abbreviated to *site6*) could not be attributed to any owner.

- **Benchmark 4-4-vary:** In this benchmark, four concurrent threads try to acquire a lock from four possible lock sites. The CPU consumption when holding a lock varies for each lock site. Lock *site3* causes twice the amount of lock contention of *site2* which causes twice the contention of *site1* which again causes twice the contention of *site0*. This will result in an exponential decrease of the caused lock contention at the owner stack trace level. To ensure that threads do not immediately re-acquire the lock and distort this lock site variation, we added a small pause duration at the end of the method after releasing the lock.

Figure 5.14 shows the results in LCV. Similar to the benchmark *16-8-equal*, the sampling-based profiler produces the expected outcome. The contention variation correctly applies

to the owner lock sites. For example, the VM-internal profiler reports that at *site0* (a contending stack trace), threads have to wait 32.1% of the total contention time which is caused by the remaining three owner lock sites (owner stack traces with the exponential decrease of the caused lock contention), i.e., by *site3* with 18.27%, by *site2* with 9.16% (half of *site3*) and by *site1* with 4.67% (half of *site2*). In comparison to that, our sampling-based approach returns 32.09% waiting time, caused by 18.46%, 9.17% and 4.46% for the same lock sites. These are accurate results.

- **Benchmark *4-4-vary-short*:** This benchmark is the same as *4-4-vary*, but the lock holding time (the amount of consumed CPU time) is significantly shorter and there is also no pause any more after releasing the lock. This results in very short lock contention units.

Figure 5.15 shows the results in LCV. Again, short lock contention units seem to be problematic for the sampling-based profiler. As it was the case in the *16-8-equal-short* benchmark, the *contending stack trace* nodes match those of the VM-internal aggregator tree (e.g., *site1* with 25.49% compared to the sampling-based result of 25.69%). But the high amount of unknown lock contention prevents a high accuracy when also taking the owner stack traces into account.

Overall, we could show that our sampling-based profiler is capable of successfully and accurately recording java.util.concurrent lock contention. However, short contention units seem to cause issues with unknown contention which we will discuss in more detail in the following section.

5.4.3 Unknown Lock Contention

Unknown lock contention is recorded when either JVMTI's `GetObjectMonitorUsage` or AQS's `getExclusiveOwnerThread` and `getQueuedThreads` do not return an owner thread but nevertheless waiting threads. As the results from the previous sections indicate, unknown contention is particularly relevant when dealing with short lock contention units (e.g., in the *actors* benchmark or in the synthetic *short* benchmarks).

Figure 5.16 shows where unknown lock contention originates from. Assume that there are two threads T_1 and T_2 which run concurrently. T_1 acquires the lock first and begins to execute the protected code. Then, at time x_1 , T_2 wants to acquire the same lock but cannot do so because T_1 is still the owner. Therefore, T_2 is suspended until T_1 releases the lock at x_2 and wakes up the waiting thread. However, T_2 cannot *instantly* be woken up and acquire the lock since this procedure requires some time. Code has to be executed to let other threads know that the lock is now free. The scheduler has to select the next thread to run, and, finally, T_2 acquires the lock at x_3 .

If we take a sample between x_1 and x_2 , that is, during Δy , we do not encounter any problems because T_1 is currently the owner thread and T_2 a waiting thread. However, if it happens that we take the sample between x_2 and x_3 instead, that is, during Δz , we do not have an owner thread any longer because T_1 has already released the lock at that time.

▼ OC java.util.concurrent.locks.ReentrantLock\$NonFairSync		100,00% (322 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site7 (...), Contention_EightSite_z: site7_thrpt_jmhStub (...) (+11)		13,59% (43 s)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site4 (...), Contention_EightSite_z: site4_thrpt_jmhStub (...) (+11)		8,40% (27 s)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site6 (...), Contention_EightSite_z: site6_thrpt_jmhStub (...) (+11)		2,28% (7,344 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site2 (...), Contention_EightSite_z: site2_thrpt_jmhStub (...) (+11)		1,52% (4,905 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site1 (...), Contention_EightSite_z: site1_thrpt_jmhStub (...) (+11)		0,33% (1,069 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site7 (...), Contention_EightSite_z: site7_thrpt_jmhStub (...) (+11)		0,27% (855 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site0 (...), Contention_EightSite_z: site0_thrpt_jmhStub (...) (+11)		0,26% (851 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site5 (...), Contention_EightSite_z: site5_thrpt_jmhStub (...) (+11)		0,26% (851 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site3 (...), Contention_EightSite_z: site3_thrpt_jmhStub (...) (+11)		0,26% (851 ms)
OST (unknown)		0,00% (61 µs)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site1 (...), Contention_EightSite_z: site1_thrpt_jmhStub (...) (+11)		13,53% (43 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site0 (...), Contention_EightSite_z: site0_thrpt_jmhStub (...) (+11)		13,14% (42 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site2 (...), Contention_EightSite_z: site2_thrpt_jmhStub (...) (+11)		12,97% (41 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site5 (...), Contention_EightSite_z: site5_thrpt_jmhStub (...) (+11)		12,81% (41 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site3 (...), Contention_EightSite_z: site3_thrpt_jmhStub (...) (+11)		12,67% (40 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site6 (...), Contention_EightSite_z: site6_thrpt_jmhStub (...) (+11)		12,32% (39 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site4 (...), Contention_EightSite_z: site4_thrpt_jmhStub (...) (+11)		8,97% (28 s)
▼ CST (+7), ThreadPooExecutor: processWorkerExit (...), ThreadPooExecutor: runWorker (...) (+1)		0,00% (3,853 µs)
▼ CST (+5), LinkedBlockingQueue: take (), ThreadPooExecutor: getTask (), ThreadPooExecutor: runWorker (...) (+2)		0,00% (1,747 µs)
▼ CST (+7), ThreadPooExecutor: interruptIdleWorkers (...), ThreadPooExecutor: tryTerminate (), ThreadPooExecutor: processWorkerExit (...)		0,00% (379 µs)

(a) Expanded aggregator tree of the VM-internal profiler.

Figure 5.12: Aggregator trees in LCV for the sythetic benchmark *16-8-equal*.

▼ OC java.util.concurrent.locks.ReentrantLock\$NonfairSync		100.00% (301 s)
▼ CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site7 (...), Contention_EightSite. z_site7_thrpt_jmhStub (...), (+11)		13.61% (40 s)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site4 (...), (+12)		8.27% (24 s)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site6 (...), (+12)		2.32% (7.000 ms)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site2 (...), (+12)		1.53% (4.620 ms)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site1 (...), (+12)		0.34% (1.030 ms)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site0 (...), (+12)		0.31% (920 ms)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site5 (...), (+12)		0.29% (860 ms)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site3 (...), (+12)		0.27% (820 ms)
OCST Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site7 (...), (+12)		0.27% (820 ms)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site1 (...), Contention_EightSite. z_site1_thrpt_jmhStub (...), (+11)		13.54% (40 s)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site0 (...), Contention_EightSite. z_site0_thrpt_jmhStub (...), (+11)		13.12% (39 s)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site2 (...), Contention_EightSite. z_site2_thrpt_jmhStub (...), (+11)		12.96% (39 s)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site5 (...), Contention_EightSite. z_site5_thrpt_jmhStub (...), (+11)		12.78% (38 s)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site3 (...), Contention_EightSite. z_site3_thrpt_jmhStub (...), (+11)		12.66% (38 s)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site6 (...), Contention_EightSite. z_site6_thrpt_jmhStub (...), (+11)		12.31% (37 s)
► CST (+7), Contention\$Base. contend (...), Contention\$EightSite. site4 (...), Contention_EightSite. z_site4_thrpt_jmhStub (...), (+11)		9.03% (27 s)
► CST Thread. currentThread () (+6), Contention\$Base. contend (...), Contention\$EightSite. site2 (...), (+12)		0.00% (10 ms)

(b) Expanded aggregator tree of the sampling-based profiler.

Figure 5.12: Aggregator trees in LCV for the synthetic benchmark *16-8-equal* (cont.).

▼ OC java.util.concurrent.locks.ReentrantLock\$NonFairSync		100,00% (291 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site6 (...), Contention_EightSite_z: site6 _thrpC_jmhStub(...), (+11)		12,55% (36 s)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site0 (...), Contention_EightSite_z: site0 _thrpC_jmhStub(...), (+11)		1,71% (4.980 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site4 (...), Contention_EightSite_z: site4 _thrpC_jmhStub(...), (+11)		1,68% (4.893 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site1 (...), Contention_EightSite_z: site1 _thrpC_jmhStub(...), (+11)		1,62% (4.729 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site2 (...), Contention_EightSite_z: site2 _thrpC_jmhStub(...), (+11)		1,59% (4.627 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site3 (...), Contention_EightSite_z: site3 _thrpC_jmhStub(...), (+11)		1,59% (4.625 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site7 (...), Contention_EightSite_z: site7 _thrpC_jmhStub(...), (+11)		1,56% (4.536 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site5 (...), Contention_EightSite_z: site5 _thrpC_jmhStub(...), (+11)		1,55% (4.502 ms)
OST (+5), Contention\$Base: contentd (...), Contention\$EightSite: site6 (...), Contention_EightSite_z: site6 _thrpC_jmhStub(...), (+11)		0,77% (2.245 ms)
OST (unknown)		0,49% (1.435 ms)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site5 (...), Contention_EightSite_z: site5 _thrpC_jmhStub(...), (+11)		12,55% (36 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site3 (...), Contention_EightSite_z: site3 _thrpC_jmhStub(...), (+11)		12,54% (36 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site7 (...), Contention_EightSite_z: site7 _thrpC_jmhStub(...), (+11)		12,53% (36 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site2 (...), Contention_EightSite_z: site2 _thrpC_jmhStub(...), (+11)		12,53% (36 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site1 (...), Contention_EightSite_z: site1 _thrpC_jmhStub(...), (+11)		12,45% (36 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site4 (...), Contention_EightSite_z: site4 _thrpC_jmhStub(...), (+11)		12,42% (36 s)
▼ CST (+7), Contention\$Base: contentd (...), Contention\$EightSite: site0 (...), Contention_EightSite_z: site0 _thrpC_jmhStub(...), (+11)		12,42% (36 s)
▼ CST (+5), LinkedBlockingQueue: take ()		0,01% (38 ms)
▼ CST (+7), ThreadPooledExecutor: getTask ()		0,00% (5.706 µs)
▼ CST (+7), ThreadPooledExecutor: processWorkerExit (...), ThreadPooledExecutor: runWorker (...), (+2)		0,00% (19 µs)

(a) Expanded aggregator tree of the VM-internal profiler.

Figure 5.13: Aggregator trees in LCV for the synthetic benchmark *16-8-equal-short*.

▼ od	java.util.concurrent.locks.ReentrantLock\$NonFairSync	100,00% (284 s)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site5 (...), Contention_EightSite_z. site5_thrpt_jmhStub (...), (+11)	12,60% (35 s)
▼ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site6 (...), Contention_EightSite_z. site6_thrpt_jmhStub (...), (+11)	12,55% (35 s)
OCST	(unknown)	8,34% (23 s)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site0 (...), (+12)	0,61% (1.730 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site2 (...), (+12)	0,59% (1.680 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site4 (...), (+12)	0,58% (1.650 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site3 (...), (+12)	0,55% (1.550 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site7 (...), (+12)	0,51% (1.450 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site5 (...), (+12)	0,48% (1.360 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site1 (...), (+12)	0,47% (1.350 ms)
OCST	Blackhole. consumeCPU (...), Contention\$Base. contend (...), Contention\$EightSite. site6 (...), (+12)	0,28% (810 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site0 (...), Contention_EightSite_z. site0_thrpt_jmhStub (...), (+11)	0,04% (120 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site4 (...), Contention_EightSite_z. site4_thrpt_jmhStub (...), (+11)	0,02% (60 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site2 (...), Contention_EightSite_z. site2_thrpt_jmhStub (...), (+11)	0,02% (60 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site7 (...), Contention_EightSite_z. site7_thrpt_jmhStub (...), (+11)	0,02% (60 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site6 (...), Contention_EightSite_z. site6_thrpt_jmhStub (...), (+11)	0,02% (50 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site3 (...), Contention_EightSite_z. site3_thrpt_jmhStub (...), (+11)	0,01% (40 ms)
OCST	Contention\$Base. contend (...), Contention\$EightSite. site1 (...), Contention_EightSite_z. site1_thrpt_jmhStub (...), (+11)	0,01% (20 ms)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site3 (...), Contention_EightSite_z. site3_thrpt_jmhStub (...), (+11)	12,54% (35 s)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site2 (...), Contention_EightSite_z. site2_thrpt_jmhStub (...), (+11)	12,53% (35 s)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site7 (...), Contention_EightSite_z. site7_thrpt_jmhStub (...), (+11)	12,51% (35 s)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site1 (...), Contention_EightSite_z. site1_thrpt_jmhStub (...), (+11)	12,49% (35 s)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site4 (...), Contention_EightSite_z. site4_thrpt_jmhStub (...), (+11)	12,40% (35 s)
▶ CST (+7)	Contention\$Base. contend (...), Contention\$EightSite. site0 (...), Contention_EightSite_z. site0_thrpt_jmhStub (...), (+11)	12,38% (35 s)
▶ CST	Thread. currentThread 0, (+6), Contention\$Base. contend (...), Contention\$EightSite. site0 (...), (+12)	0,00% (10 ms)

(b) Expanded aggregator tree of the sampling-based profiler.

Figure 5.13: Aggregator trees in LCV for the synthetic benchmark *16-8-equal-short* (cont.).

OC	java.util.concurrent.locks.ReentrantLock\$NonFairSync	100.00% (64 s)
▼ CST	(+7), Contention\$Base contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)	32.10% (20 s)
OST	(+5), Contention\$Base contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)	18,27% (11 s)
OST	(+5), Contention\$Base contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)	9,16% (5,944 ms)
OST	(+5), Contention\$Base contend(...), Contention\$FourSite.site1(...), Contention_FourSite_z.site1_thrpt_jmhStub(...), (+11)	4,67% (3.032 ms)
OST (unknown)		0,00% (75 µs)
▼ CST	(+7), Contention\$Base contend(...), Contention\$FourSite.site1(...), Contention_FourSite_z.site1_thrpt_jmhStub(...), (+11)	28,44% (18 s)
OST	(+5), Contention\$Base contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)	18,27% (11 s)
OST	(+5), Contention\$Base contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)	7,86% (5.098 ms)
OST	(+5), Contention\$Base contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)	2,31% (1.501 ms)
OST (unknown)		0,00% (54 µs)
▼ CST	(+7), Contention\$Base contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)	25,26% (16 s)
▼ CST	(+7), Contention\$Base contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)	14,20% (9.212 ms)
▼ CST	(+7), Thread\$Dnlf\$vari iter nrmace\$Wnkar\$Vitr \ Thread\$Dnlf\$vari iter rin\$Wnkar \ Thread\$Dnlf\$vari iter\$Wnkar rin \	0.00% (166 ns)

(a) Expanded aggregator tree snippet of the VM-internal profiler.

OC java.util.concurrent.locks.ReentrantLock\$NonFairSync	100.00% (62 s)
▼ CST (+7), Contention\$Base contend(...), Contention\$FourSite_z.site0_thrpt_jmhStub(...), (+11)	32.09% (20 s)
OCST Blackhole: consumeCPU(...), Contention\$Base contend(...), Contention\$FourSite.site3(...), (+12)	18.46% (11 s)
OCST Blackhole: consumeCPU(...), Contention\$Base contend(...), Contention\$FourSite.site2(...), (+12)	9.17% (5.730 ms)
OCST Blackhole: consumeCPU(...), Contention\$Base contend(...), Contention\$FourSite.site1(...), (+12)	4.46% (2.790 ms)
▼ CST (+7), Contention\$Base contend(...), Contention\$FourSite_z.site1_thrpt_jmhStub(...), (+11)	28.64% (17 s)
OCST Blackhole: consumeCPU(...), Contention\$Base contend(...), Contention\$FourSite.site3(...), (+12)	18.46% (11 s)
OCST Blackhole: consumeCPU(...), Contention\$Base contend(...), Contention\$FourSite.site2(...), (+12)	7.85% (4.910 ms)
OCST Blackhole: consumeCPU(...), Contention\$Base contend(...), Contention\$FourSite.site0(...), (+12)	2.32% (1.450 ms)
▼ CST (+7), Contention\$Base contend(...), Contention\$FourSite_z.site2_thrpt_jmhStub(...), (+11)	25.24% (15 s)
▼ CST (+7), Contention\$Base contend(...), Contention\$FourSite_z.site3_thrpt_jmhStub(...), (+11)	14.03% (8.770 ms)

(b) Expanded aggregator tree of the sampling-based profiler.

Figure 5.14: Aggregator trees in LCV for the synthetic benchmark 4-4-vary

▼ 0d java.util.concurrent.locks.ReentrantLock\$NonFairSync		100.00% (45 s)
▼ CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site1(...), Contention_FourSite_z.site1_thrpt_jmhStub(...), (+11)		25.49% (11 s)
OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)		6.95% (3.167 ms)
OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)		6.80% (3.097 ms)
OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)		6.69% (3.047 ms)
OST (unknown)		5.05% (2.299 ms)
▼ CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)		25.44% (11 s)
OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)		7.12% (3.243 ms)
OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)		6.84% (3.116 ms)
OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site1(...), Contention_FourSite_z.site1_thrpt_jmhStub(...), (+11)		6.58% (2.999 ms)
OST (unknown)		4.89% (2.227 ms)
► CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)		24.80% (11 s)
► CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)		24.27% (11 s)
► OST (+5), Contention\$Base.contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)		0.00% (0.000 ms)

(a) Expanded aggregator tree snippet of the VM-internal profiler.

▼ 0d java.util.concurrent.locks.ReentrantLock\$NonFairSync		100.00% (50 s)
▼ CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site1(...), Contention_FourSite_z.site1_thrpt_jmhStub(...), (+11)		25.69% (12 s)
OCST (unknown)		16.71% (8.450 ms)
OCST Blackhole.consumeCPU(...), Contention\$Base.contend(...), Contention\$FourSite.site3(...), (+12)		3.50% (1.770 ms)
OCST Blackhole.consumeCPU(...), Contention\$Base.contend(...), Contention\$FourSite.site2(...), (+12)		2.75% (1.390 ms)
OCST Blackhole.consumeCPU(...), Contention\$Base.contend(...), Contention\$FourSite.site0(...), (+12)		2.51% (1.270 ms)
OCST Contention\$Base.contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)		0.12% (60 ms)
OCST Contention\$Base.contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)		0.06% (30 ms)
OCST Contention\$Base.contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)		0.04% (20 ms)
▼ CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site0(...), Contention_FourSite_z.site0_thrpt_jmhStub(...), (+11)		25.34% (12 s)
OCST (unknown)		16.14% (8.160 ms)
OCST Blackhole.consumeCPU(...), Contention\$Base.contend(...), Contention\$FourSite.site3(...), (+12)		3.62% (1.830 ms)
OCST Blackhole.consumeCPU(...), Contention\$Base.contend(...), Contention\$FourSite.site1(...), (+12)		2.71% (1.370 ms)
OCST Blackhole.consumeCPU(...), Contention\$Base.contend(...), Contention\$FourSite.site2(...), (+12)		2.61% (1.320 ms)
OCST Contention\$Base.contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)		0.12% (60 ms)
OCST Contention\$Base.contend(...), Contention\$FourSite.site1(...), Contention_FourSite_z.site1_thrpt_jmhStub(...), (+11)		0.08% (40 ms)
OCST Contention\$Base.contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)		0.06% (30 ms)
► CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site3(...), Contention_FourSite_z.site3_thrpt_jmhStub(...), (+11)		24.64% (12 s)
► CST (+7), Contention\$Base.contend(...), Contention\$FourSite.site2(...), Contention_FourSite_z.site2_thrpt_jmhStub(...), (+11)		24.33% (12 s)

(b) Expanded aggregator tree of the sampling-based profiler.

Figure 5.15: Aggregator trees in LCV for the synthetic benchmark 4-4-vary-short.

Nevertheless, T_2 is still a waiting thread since the “wake up and acquire lock”-process is not yet complete. Hence, taking samples in the time span Δz is exactly the cause of unknown lock contention.

The reason why the amount of unknown lock contention increases when the duration of owning a lock decreases is simply a statistical problem. Δz should more or less always take about the same amount of time. On the other hand, Δy heavily depends on the code that is protected by the lock. The shorter Δy becomes, the lower the chances are that we take a sample with an owner thread (i.e., during Δy). However, the chances of sampling during Δz are still the same, which increases the chance that we take a sample without an owner thread (i.e., during Δz) relative to the total execution time. The short contention times of, for instance, the *actors* benchmark as well as the results of the *short* benchmarks in Figures 5.13b and 5.15b show this effect: The amount of unknown contention increases with decreasing contention times.

One last point to discuss is the question why `java.util.concurrent` locks seem to suffer even more from this problem. For intrinsic locks, a thread does not immediately have to suspend itself when failing to acquire the lock. Instead, it is first *spinning*, that is, repeatedly trying for a short period of time to get the lock after all. If the spinning succeeds, the thread can acquire the lock without having to suspend itself and, in turn, without having to be woken up again. In this case, our sampling-based profiler does not record unknown lock contention. With `java.util.concurrent` locks, however, there is no lock spinning but just a simple compare-and-set (CAS) operation. The contending thread only tries to acquire the lock once (i.e., it tries to set itself as the owner via CAS) and, upon failure (i.e., CAS was not successful, because there is already an owner), immediately suspends itself, which exactly results in the scenario explained above and illustrated in Figure 5.16. Hence, the likelihood of unknown lock contention is higher when dealing with locks from `java.util.concurrent`.

Figure 5.16: Illustration of how a thread releases a lock and wakes up a waiting thread which previously wanted to acquire the same lock. The red line indicates the suspension state of T_2 .

Unfortunately, there is nothing we can do about this problem in our sampling-based approach without introducing major changes or even opting for an alternative approach entirely. This is a drawback we have to accept.

Chapter 6

Related Work

As already mentioned, Hofer et al. [12, 13, 14] proposed a modification of the OpenJDK HotSpot VM (i.e., a VM-internal profiler) to record lock contention for both intrinsic locks and for `java.util.concurrent` locks by tracing contention events. They 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 stack traces. Their approach utilizes the aggregation framework presented in Section 4.1, and the created output can also be viewed in the Lock Contention Visualizer tool. In contrast to our sampling-based approach, the VM-internal profiler records all contention with high accuracy and still causes an acceptable mean overhead of just 7.8% for multi-threaded applications (this was tested on 40 hardware threads rather than only on eight threads like in our test setup). 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. [5, 6]. Their profiler *Free Lunch* monitors the so-called *critical section pressure* (CSP) for each lock over a specified measurement interval of 1 second (this is used to separate the application execution into phases which are 1 s long, and for each phase, the CSP is calculated). This metric represents the ratio of the time threads have to wait for lock acquisition and the time threads are running (thread progress metric). If the CSP of a lock reaches a threshold, this means that thread progress is impeded and that there seems to be a lock contention bottleneck. In this case, information about this lock as well as a stack trace from one of the blocked threads are collected. The authors report that their modifications of the VM incur a worst-case overhead of 6%.

Tallent et al. [40] proposed a sampling-based lock contention profiler for C programs which associates a counter with each lock. The profiler periodically inspects all threads. 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 stack trace. While this approach has an overhead of only 5%, it does neither record which threads had to wait nor their stack traces.

Using hardware performance counters can result in even less overhead. The sampling profiler of Inoue and Nakatani [18] utilizes such counters in the IBM Java VM to collect information on where locks are acquired and where blocking occurs. They insert a special instruction at lock acquisition sites and combine it with a technique named *CallerChaining* to efficiently record stack traces using the call stack depth. Their profiler achieves an overhead of typically less than 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) [11], a commercial tool built into the Oracle JDK. In most cases, it imposes only about 1% run-time overhead and provides data on the objects used in locking, which threads were blocked and their stack traces. 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 stack trace, however, is not recorded and threads that held the same lock before this last thread are not taken into account either. Moreover, `java.util.concurrent` locks are not supported.

Research investigating `java.util.concurrent` locks was conducted by Patros et al. [33]. They modified the IBM Java VM to collect park contention data whenever threads are parked. For each park blocker object, a record is created which is updated throughout the execution of the profiled application. The records include class data of the blocker object, time and park duration information, 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 an overhead below 0.5%. However, they collect no information about where contention is caused.

Qi et al. [35] also created a profiler called *jucprofiler* for detecting `java.util.concurrent` lock contention, but they worked with an instrumentation-based approach. They instrumented multiple `java.util.concurrent` classes such as `AbstractQueuedSynchronizer`, `ReentrantLock` or `ConcurrentHashMap` to keep track of all occurring lock contention as well as conditional waiting. The results provide a detailed view on where lock contention occurs, including contending threads with their stack traces, duration information and lock contention counters (how often a certain lock was contended). They do not collect the owner thread, however. The created trace file can be visualized in the development environment Eclipse via special views. *jucprofiler* is part of the *Multicore Software Development Kit* [4], which is a debugging and testing tool set for the IBM Java VM.

HaLock is a hardware-assisted lock profiler for C developed by Huang et al. [15]. It makes use of a special data recording hardware called hardware memory tracing tool (HMTT), which is used for reducing the memory impact of conventional profiling approaches in order to avoid accuracy distortions. For collecting lock contention information such as thread ID and lock address, they instrumented the Pthread library. Time-related information is directly taken from HMTT when the data is written to avoid time inconsistencies which could become problematic when dealing with different clocks of multiple CPUs. Their tests reveal that *HaLock* substantially decreases cache misses and additional memory requests compared to other profilers, which results in accuracy discrepancies among the tested profilers. Again, the cause of contention cannot be detected.

The Pthread library was also instrumented with *mutrace* by Poettering [34] but without the need for additional hardware. It collects lock contention data which includes the stack trace of the contended lock, how often it was used, how often the owner thread changed, how often the lock was contended as well as duration information. However, detailed data such as the actual threads and further lock owner information is not part of the profiler.

The Java 2 Platform Standard Edition also comes with a profiler that, beside other things, can monitor intrinsic locks. It is called *HPROF* [24] and was initially developed by Sun Microsystems and presented by Liang and Viswanathan [20]. HPROF is a native agent which uses JVMTI and reports lock contention information, for example, which threads had to wait for how long for what locks as well as a summary of all contended objects including stack traces, sorted by the amount of lock contention which was recorded there. However, determining the lock ownership or profiling locks from `java.util.concurrent` is not supported.

Chapter 7

Conclusion

7.1 Summary

We introduced a novel sampling-based approach for recording lock contention in Java applications. The implementation of the profiler does not require modifications of the VM but solely relies on JVMTI and bytecode instrumentation, which is beneficial when dealing with different Java or VM versions in diverse environments. We can collect detailed and accurate lock contention information for both intrinsic locks as well as for locks from the `java.util.concurrent` framework. Moreover, the recorded data not only shows where contention occurs but also where it is caused.

For analyzing and evaluating our approach, we presented a powerful aggregation framework, which can be used in combination with the Lock Contention Visualizer to enable a versatile and user-friendly analysis. Users can arbitrarily group and aggregate lock contention data provided by our profiler according to various aspects in order to view and analyze only the data that is of actual interest.

We evaluated our novel approach on a set of real-world benchmarks and showed that our implementation incurs a run-time overhead of only 1.6% when profiling with 20 samples/s and 4.9% with 100 samples/s. Furthermore, the amount of generated data is low. The profiler generates less than 1.9 MB/s when sampling with 100 samples/s and only 460 KB/s when sampling with 20 samples/s. Compressing the data reduces the amount of data even more without incurring a noticeable additional overhead. We also evaluated the accuracy of the results of our sampling-based profiler by comparing it to the VM-internal profiler by Hofer et al. [14] which we used as the accuracy baseline. Based on real-world benchmarks and on synthetic benchmarks, we showed the high accuracy of our measurements and extensively discussed all problems that occurred. In conclusion, we consider our new sampling-based profiler suitable for the use in production systems.

7.2 Future Work

As with most prototype projects, there is still much to be done to improve and extend our sampling-based lock contention profiler. Future work includes extracting the *owner stack trace* rather than only the owner *current* stack trace when handling `java.util.concurrent` locks. Furthermore, there are a few possibilities for optimizations and enhancements regarding the current code. Another interesting but still missing aspect is to determine the impact of our profiler when the inspected application is fully utilizing the CPU. Finally, one last important matter is to test our profiler on a server-class machine to see whether it also scales with a high number of processors and cores, both with regard to performance and to data.

Bibliography

- [1] David F. Bacon, Ravi Konuru, Chet Murthy, and Mauricio Serrano, *Thin Locks: Featherweight Synchronization for Java*, Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '98, ACM, 1998, pp. 258–268.
- [2] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann, *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*, Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications (New York, NY, USA), OOPSLA '06, ACM, 2006, pp. 169–190.
- [3] Eric Bruneton, Pierre Cregut, Rémi Forax, Eugene Kuleshov, and Andrei Loskutov, *ASM: Java Bytecode Manipulation and Analysis Framework*, <http://asm.ow2.org/index.html>, Accessed: December 19, 2016.
- [4] Raja Das, *Multicore Software Development Kit*, <https://www.ibm.com/developerworks/community/groups/service/html/communityoverview?communityUuid=9a29d9f0-11b1-4d29-9359-a6fd9678a2e8>, 2011, Accessed: December 7, 2016.
- [5] Florian David, *Continuous and Efficient Lock Profiling for Java on Multicore Architectures*, Ph.D. thesis, Université Pierre et Marie Curie, 2015.
- [6] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller, *Continuously Measuring Critical Section Pressure with the Free-lunch Profiler*, Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (New York, NY, USA), OOPSLA '14, ACM, 2014, pp. 291–307.
- [7] Daniele Cono D’Elia, Camil Demetrescu, and Irene Finocchi, *Mining Hot Calling Contexts in Small Space*, Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '11, ACM, 2011, pp. 516–527.

- [8] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea, *Java Concurrency in Practice*, Addison-Wesley Professional, 2006.
- [9] Google, *Snappy: A fast compressor/decompressor*, <http://google.github.io/snappy/>, Accessed: December 27, 2016.
- [10] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley, *The Java® Language Specification: Java SE 8 Edition*, <https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf>, 2015.
- [11] Marcus Hirt and Marcus Lagergren, *Oracle JRockit: The Definitive Guide*, Packt Publishing Ltd, 2010.
- [12] Peter Hofer, *Method Profiling and Lock Contention Profiling on the Java Virtual Machine Level*, Ph.D. thesis, Johannes Kepler University Linz, 2016.
- [13] Peter Hofer, David Gnedt, and Hanspeter Mössenböck, *Efficient Dynamic Analysis of the Synchronization Performance of Java Applications*, Proceedings of the 13th International Workshop on Dynamic Analysis (New York, NY, USA), WODA 2015, ACM, 2015, pp. 14–18.
- [14] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck, *Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level*, Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering (New York, NY, USA), ICPE '16, ACM, 2016, pp. 263–274.
- [15] Yongbing Huang, Zehan Cui, Licheng Chen, Wenli Zhang, Yungang Bao, and Mingyu Chen, *HaLock: Hardware-assisted Lock Contention Detection in Multithreaded Applications*, Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (New York, NY, USA), PACT '12, ACM, 2012, pp. 253–262.
- [16] Charlie Hunt and Binu John, *Java Performance*, Addison-Wesley Professional, 2011.
- [17] IBM, *J9 Virtual Machine (JVM)*, https://www.ibm.com/support/knowledgecenter/SSYKE2_8.0.0/com.ibm.java.win.80.doc/user/java_jvm.html, Accessed: January 30, 2017.
- [18] Hiroshi Inoue and Toshio Nakatani, *How a Java VM Can Get More from a Hardware Performance Monitor*, Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (New York, NY, USA), OOPSLA '09, ACM, 2009, pp. 137–154.
- [19] Doug Lea, *The java.util.concurrent synchronizer framework*, Science of Computer Programming 58 (2005), no. 3, 293–309.
- [20] Sheng Liang and Deepa Viswanathan, *Comprehensive Profiling Support in the Java Virtual Machine*, COOTS, 1999, pp. 229–242.

- [21] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley, *The Java® Virtual Machine Specification: Java SE 8 Edition*, <https://docs.oracle.com/javase/specs/jvms/se8/jvms8.pdf>, 2015.
- [22] Philippe Moret, Walter Binder, and Alex Villazon, *CCCP: Complete Calling Context Profiling in Virtual Execution Environments*, Proceedings of the 2009 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (New York, NY, USA), PEPM '09, ACM, 2009, pp. 151–160.
- [23] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney, *Evaluating the Accuracy of Java Profilers*, Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '10, ACM, 2010, pp. 187–197.
- [24] Oracle, *HPROF: A Heap/CPU Profiling Tool*, <https://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html>, Accessed: January 11, 2017.
- [25] Oracle, *Java Native Interface 6.0 Specification (JNI)*, <https://docs.oracle.com/javase/8/docs/technotes/guides/jni/>, Accessed: December 8, 2016.
- [26] Oracle, *Java SE HotSpot at a Glance*, <http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html>, Accessed: January 30, 2017.
- [27] Oracle, *java.lang.instrument*, <https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html>, Accessed: December 8, 2016.
- [28] Oracle, *java.nio*, <https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html>, Accessed: December 27, 2016.
- [29] Oracle, *JVM™ Tool Interface Version 1.2.3 (JVMTI)*, <https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>, Accessed: December 8, 2016.
- [30] Oracle, *OpenJDK Code Tools: Java Microbench Harness (JMH)*, <http://openjdk.java.net/projects/code-tools/jmh/>, Accessed: December 8, 2016.
- [31] Oracle, *OpenJDK: The HotSpot Group*, <http://openjdk.java.net/groups/hotspot/>, Accessed: January 30, 2017.
- [32] Oracle, *Oracle® JRockit Introduction Release R28*, https://docs.oracle.com/cd/E15289_01/doc.40/e15058/toc.htm, Accessed: January 30, 2017.
- [33] Panagiotis Patros, Eric Aubanel, David Bremner, and Michael Dawson, *A Java Util Concurrent Park Contention Tool*, Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores (New York, NY, USA), PMAM '15, ACM, 2015, pp. 106–111.
- [34] Lennart Poettering, *Mutrace*, <http://0pointer.de/blog/projects/mutrace.html>, 2009.

-
- [35] Yao Qi, Raja Das, and Zhi Da Luo, *Profiling java.util.concurrent locks*, <https://www.infoq.com/articles/jucprofiler>, 2010.
 - [36] Andreas Schörgenhumer, Peter Hofer, David Gnedt, and Hanspeter Mössenböck, *Efficient Sampling-based Lock Contention Profiling in Java*, ICPE '17, ACM, 2017.
 - [37] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder, *Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine*, Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (New York, NY, USA), OOPSLA '11, ACM, 2011, pp. 657–676.
 - [38] Dain Sundstrom, *Snappy in Java*, <https://github.com/dain/snappy>, Accessed: December 27, 2016.
 - [39] Pool Tõnis, *Lock optimizations on the HotSpot VM*, Tech. report, University of Tartu, 2014.
 - [40] Nathan R. Tallent, John M. Mellor-Crummey, and Allan Porterfield, *Analyzing Lock Contention in Multithreaded Applications*, Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (New York, NY, USA), PPOPP '10, ACM, 2010, pp. 269–280.
 - [41] Xiaotong Zhuang, Mauricio J. Serrano, Harold W. Cain, and Jong-Deok Choi, *Accurate, Efficient, and Adaptive Calling Context Profiling*, Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (New York, NY, USA), PLDI '06, ACM, 2006, pp. 263–271.

Curriculum Vitae

Personal Information

Name	Andreas Schörgenhumer, BSc
Address	Fadingerstraße 29 4730 Waizenkirchen Austria
Telephone	+43 664 88583231
E-Mail	andischoe@gmail.com
Date of Birth	15.05.1993
Place of Birth	4600 Wels Austria

Education

09.2003 – 07.2007	Elementary school in Gymnasium Dachsberg in Prambachkirchen
09.2007 – 06.2011	Upper school with Matura in Bundes-Oberstufen-Realgymnasium Grieskirchen with specialization in artistic education and voluntary immersion in informatics
10.2011 – 10.2015	Bachelor of Science (BSc) in Computer Science at Johannes Kepler University Linz
10.2015 – 02.2017	Master of Science (Dipl.-Ing.) in Computer Science at Johannes Kepler University Linz

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated and that all direct and indirect sources are acknowledged as references. This printed thesis is identical with the electronic version submitted.

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Masterarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe. Die vorliegende Masterarbeit ist mit dem elektronisch übermittelten Textdokument identisch.

Linz, February 3, 2017

Andreas Schörgenhumer, BSc