approach not only records the call chains of threads that are blocked, but also accurately reports the call chains of threads that block other threads by holding a requested lock. We are capable of efficiently collecting this information because we implemented our approach directly in the OpenJDK HotSpot Virtual Machine, a popular high-performance Java Virtual Machine (VM).

The main contributions of this paper are:

1. We describe an approach for efficiently tracing fine-grained lock contention events in a Java VM, and aspects of our implementation in the HotSpot VM. Unlike other approaches, the collected traces show not just where contention occurs, but also where it is caused.

2. We describe a versatile approach for analyzing and visualizing the collected events that enables users to recognize locking bottlenecks and their characteristics in an effective way.

3. We provide an extensive evaluation of our implementation on a server-class machine, studying its overhead, the amount of generated data, the composition of traces, and the effectiveness of specific optimizations. We show that our approach has such low overhead that it is typically feasible to use it in a production environment.

The rest of this paper is organized as follows: Section 2 provides an introduction to locking in Java. Section 3 describes which events we trace and how we record them efficiently. Section 4 characterizes the analysis of our traces and the versatile aggregation and visualization of lock contentions. Section 5 evaluates the runtime overhead and other characteristics of our approach. Section 6 examines related work, and Section 7 concludes this paper.

2. LOCKING IN JAVA

Java has intrinsic support for locking, but it also provides the *java.util.concurrent* package that contains explicit locks and advanced synchronization mechanisms.

2.1 Intrinsic Locks (Monitors)

Each Java object has an intrinsic mutual-exclusion lock associated with it, which is also called the object's *monitor*. Developers can insert *synchronized blocks* in their code and specify an object to use for locking. That object's lock is then acquired before entering the block and executing its statements, and released when exiting the block. When threads contend for a lock, that is, when one thread has acquired a lock by entering a synchronized block, and other threads are trying to enter a synchronized block using the same lock, those threads are blocked until the owner exits its synchronized block and releases the lock. Developers can also declare methods as synchronized, which then acquire the lock of the *this* object when called, and release it again when returning. A notable property of synchronized blocks and synchronized methods is that a lock is guaranteed to be released in the same method and scope in which it was acquired, even when an exception is thrown.

Intrinsic locks further support conditional waiting with the *wait, notifyAll* and *notify* methods. These methods may only be called on an object while holding its lock. The *wait* method releases the lock and suspends the calling thread.

```
class BlockingQueue {
  private final List<Object> q = new LinkedList<>();
  void enqueue(Object item) {
    synchronized(q) {
      q.add(item);
      q.notifyAll();
    }
  }
  Object dequeue() {
    synchronized(q) {
      awaitNotEmpty();
      return q.remove(0);
    }
  }
  void awaitNotEmpty() {
    while (q.isEmpty()) {
      try {
        q.wait();
      } catch (InterruptedException e) { }
    }
  }
}
```

Figure 1: Blocking queue with Java intrinsic locks

When another thread calls *notifyAll* on the same object, all threads that are waiting on its lock are resumed. Each resumed thread then attempts to acquire the lock, and once successful, it continues execution. In contrast to *notifyAll,* the *notify* method wakes up only a single waiting thread. This mechanism is typically used in producer-consumer scenarios, such as threads in a thread pool that wait for tasks to execute.

Figure 1 shows an example of using intrinsic locks to implement a thread-safe blocking queue. The field *q* holds the list of queue items, and the list object's intrinsic lock is used to ensure mutually exclusive queue accesses. The *enqueue* method has a synchronized block to acquire the lock of *q* before it appends an item and calls *notifyAll* to resume any threads waiting on *q*. The *dequeue* method also uses a synchronized block and calls *awaitNotEmpty,* which invokes *wait* on *q* as long as the queue is empty. The *wait* method releases the lock of *q* and suspends the calling thread until another thread resumes it by calling *notifyAll* from *enqueue*. Alternatively, when the thread is interrupted while waiting, *wait* throws an *InterruptedException,* which we catch. In either case, *wait* attempts to reacquire the lock of *q* and once successful, execution continues in *awaitNotEmpty,* which checks again whether the list is not empty. When *awaitNotEmpty* finally returns, the list is guaranteed to be not empty, and *dequeue* can remove and return an item.

The semantics of intrinsic locks are implemented entirely in the Java VM, usually in a very efficient way so their use only incurs significant overhead when threads actually contend for locks [1, 15]. Implementations are typically *non-fair* and allow threads to acquire a recently released lock even when there are queued threads that requested that lock earlier. This increases the throughput by reducing the overhead from suspending threads, and by better utilizing the time periods between when one thread releases a lock and when a queued thread is scheduled and can acquire the lock [4].

2.2 The java.util.concurrent package

Java 5 introduced the *java.util.concurrent* package with classes that provide useful synchronization mechanisms, such as concurrent collections and read-write locks [10]. Most of

these classes do not use Java's intrinsic locks, but rather rely on the newly introduced *LockSupport* facility, which provides a *park* method that a thread can call to park (suspend) itself, and an *unpark* method which resumes a parked thread. Using these two methods as well as compare-and-set operations, the semantics of java.util.concurrent classes can be implemented entirely in Java. The *AbstractQueuedSynchronizer* class further provides a convenient basis for implementing synchronization mechanisms with wait queues. Because these classes are public, application developers can also implement custom synchronization mechanisms on top of them.

ReentrantLock is an example of a mutual exclusion lock in java.util.concurrent that is semantically similar to intrinsic locks, but is implemented entirely in Java on top of AbstractQueuedSynchronizer. Code that uses ReentrantLock must explicitly call its *lock* and *unlock* methods to acquire and release the lock. ReentrantLock also supports conditional waiting by calling *await, signalAll* and *signal* on an associated *Condition* object. Unlike intrinsic locks, an arbitrary number of such condition objects can be created for each lock. Moreover, ReentrantLock has a fair mode which guarantees first-come-first-serve ordering of lock acquisitions and conditional wake-ups. This mode reduces the variance of lock acquisition times, typically at the expense of throughput.

3. LOCK CONTENTION EVENT TRACING

Analyzing lock contention in an application requires observing individual locking events and computing meaningful statistics from them. Maintaining those statistics in the synchronizing threads interferes with the execution of the application, and requires synchronization as well, so it can become a bottleneck itself. Instead, we decided to only record those events in the application threads and to analyze them later. In this section, we describe how we efficiently record events and metadata, which events we record, and how we reconstruct thread interactions from the recorded events.

3.1 Writing and Processing Events

Figure 2 shows an overview of how we write and process trace data in our approach. Naturally, we trace locking events in different application threads, and writing those events to a single shared trace buffer would require synchronization and could become a bottleneck. Therefore, each application thread T_i allocates a *thread-local trace buffer* where it can write events without synchronization. When the trace buffer is full, the thread submits it to the *processing queue* and allocates a new buffer. A *background thread* retrieves the trace buffers from the queue and processes their events. In the depicted scenario, it merges them into a single *trace file*. This trace file can be opened in an *analysis tool* for offline analysis and visualization.

We encode the events in an efficient binary representation to facilitate fast writing, to reduce the amount of generated data and to keep the memory usage of the trace buffers low. Because each buffer is written by just a single thread, we can store that thread once per buffer instead of recording it in each event. Submitting a full buffer to the queue requires synchronization, but we assign a random size between 8 KB and 24 KB to the individual buffers, which is large enough so that this is an infrequent operation. The randomization avoids that threads which perform similar tasks try to submit their buffers at the same time and contend for queue access.

Our design supports writing a trace file for offline analysis as well as analyzing the recorded events online. We implemented the event processing in Java, and use a thin native code interface to dequeue the trace buffers from the processing queue and to wrap them in *DirectByteBuffer* objects, which can be read from Java without copying memory. To write trace files, we use the Java NIO subsystem [11], which can use the DirectByteBuffer objects directly. We also support fast compression of the trace data by using a Java implementation of the high-performance LZ77-type compression algorithm *Snappy* [5, 18]. Our online analysis mode currently generates a text file with statistics, but it could be extended to provide an interface for the Java Management Extensions (JMX, [13]) to configure analysis parameters and to access the produced statistics.

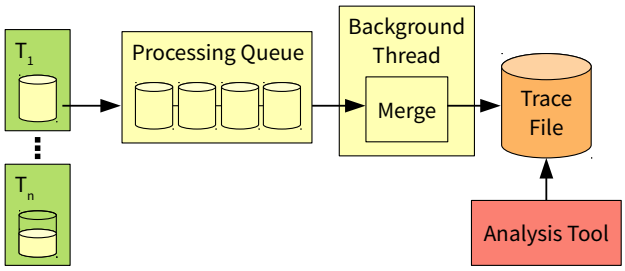

Figure 2: Writing and processing trace events

3.2 Tracing Intrinsic Locks

Locking causes a major performance impact when threads contend for a lock. Threads that fail to acquire a lock are suspended and cannot make progress, and a thread that releases a contended lock must also do extra work to resume a blocked thread as its successor. However, locking itself is not expensive in the HotSpot VM. An available intrinsic lock can be acquired with a single compare-and-set instruction in many cases. When a thread holds a lock only briefly, another thread that requests that lock can often still acquire it through spinning without suspending itself. Therefore, we chose to record only *lock contention* with our approach instead of recording *all* lock operations.

Conceptually, each Java object has an intrinsic lock associated with it. This lock stores its current owner thread, the threads that are blocked trying to acquire it, and the threads that are waiting for notifications on it. However, in a typical application, most Java objects are never used for locking. Therefore, the HotSpot VM assigns a lock to an object only when threads start using that object for locking. Even then, *biased locking* can avoid allocating an actual lock as long as the object is never used by more than one thread.

The act of assigning a lock to an object is called *lock inflation*. Intrinsic locks in HotSpot are data structures in native memory which are never moved by the garbage collector and can therefore be uniquely identified by their address. Whenever lock inflation happens, we record an *inflation event* with the lock's address, the object that the lock is being assigned to, and that object's class. In the trace events that follow, we record only the lock's address. When analyzing the trace, we are thus still able to infer the lock's associated object and its class from the inflation event.

When a thread cannot enter a synchronized block or method because it cannot acquire a lock, we record a *contended enter event* with the lock's address, a timestamp, and the call chain of the thread. Recording the call chain is ex-

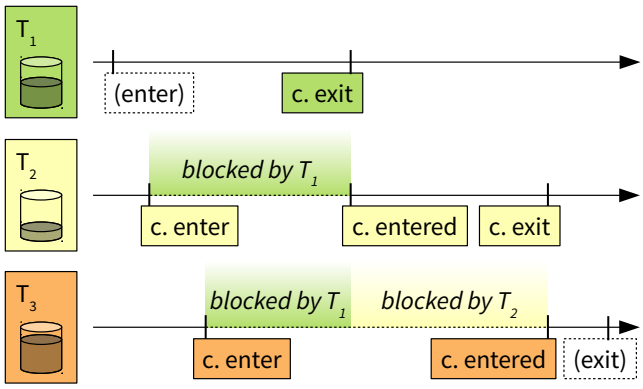

Figure 3: Events in three contending threads

pensive, but the impact on performance is moderate because the thread is unable to make progress either way. When the thread later acquires the lock, we record a *contended entered event* with only a timestamp.

With those two events, we can determine which threads were blocked when trying to acquire a lock, how long they were blocked, and what call chains they were executing. However, this information only reveals the symptoms of locking bottlenecks and not their causes. To determine the causes of contention, we record events not only in those threads that are blocked, but also in those threads which block other threads by holding a contended lock. We modified all code paths in the VM that release a lock when exiting a synchronized block or method so that they check whether any threads are currently blocked on that lock. If so, we write a *contended exit event* with a timestamp and the call chain. We delay recording the call chain and writing the event until after the lock has been released to avoid causing even more contention.

Figure 3 shows an example of events that we trace for three threads T_1, T_2 and T_3 that are executing in parallel and are contending for a single intrinsic lock. First, T_1 acquires the lock without contention, so we do not record an event. Next, T_2 tries to acquire the lock, but the lock is held by T_1, so T_2 writes a *contended enter event* in its trace buffer and suspends itself. T_3 then also fails to acquire the lock and also records a *contended enter event.* When T_1 finally releases the lock, it sees T_2 and T_3 on the lock's queue of blocked threads, so it resumes thread T_2 and writes a *contended exit event.* T_2 acquires the lock and writes a *contended entered event.* When T_2 later releases the lock, it sees T_3 on the lock's queue, resumes T_3 and writes a *contended exit event.* T_3 then acquires the lock and writes a *contended entered event.* When T_3 releases the lock, no threads are queued, and T_3 continues without writing an event. When the trace analysis later examines the events of all threads, it can infer from the *contended exit events* that T_2 and T_3 were being blocked by T_1 holding the lock, and that T_3 was subsequently being blocked by T_2. It can further compute the duration of those periods from the timestamps in the events.

Event ordering.

The trace analysis needs to arrange all events for a specific lock in their correct order to analyze them. For this purpose, we introduced a counter in HotSpot's intrinsic lock structure. When we write an event for a lock, we atomically increment the lock's counter and record its new value in the event as its *per-lock sequence number*. Unlike timestamps, these sequence numbers have no gaps between them, which enables the analysis to determine whether it can already analyze a sequence of parsed events, or whether there are still events in a trace buffer from another thread that it has not parsed yet. This considerably simplifies and speeds up the analysis. However, when a thread records an event and then does not submit its trace buffer for a long time, it still delays the analysis of subsequent events. For this reason, we also reclaim the trace buffers of all threads during garbage collections and add them to the processing queue. The threads subsequently allocate new buffers to write further events.

Sequence numbers are also more reliable than timestamps for correctly ordering events. Although we retrieve the timestamps from a monotonic system clock, this clock typically uses a timer of the CPU or of the CPU core on which the thread is currently executing. When the timers of different CPUs or CPU cores are not perfectly synchronized, the recorded timestamps are not accurate enough to establish a happened-before relationship, while our atomically incremented sequence numbers always guarantee a correct order.

Conditional waiting.

The *wait* method temporarily releases a lock and suspends the calling thread until another thread wakes it up. When the released lock is contended, we also need to write a *contended exit event* with a call chain. However, *wait* may be called in a different method than the one that acquired the lock, as is the case with *awaitNotEmpty* in Figure 1. In this case, recording the current call chain would misrepresent the source of contention. Instead, we generate and record a *forward reference* to the call chain of the method that acquired the lock. We store the reference in the lock structure and when that method releases the lock later, we record a separate event that resolves the reference to that method's call chain.

When a thread is unable to reacquire a lock after waking up from waiting, we also record a *contended enter event,* but this form of contention is not necessarily critical. Often, only the first of several notified threads can make progress, while the other threads will find that the condition that they have been waiting on is again not met, and thus continue waiting. Therefore, we use an extra flag in the event to indicate when contention was preceded by conditional waiting. This allows us to classify this type of contention differently in analysis.

3.3 Tracing Park/Unpark Synchronization

Most synchronization mechanisms in java.util.concurrent are implemented entirely in Java. To trace them, we could instrument each class individually and generate custom trace events that are specific to the semantics of the class in question. However, what these classes have in common is that they rely on the *park* and *unpark* methods that the VM provides through the *LockSupport* class.

Figure 4 shows an outline of the *LockSupport* class. The *park* method parks (suspends) the calling thread. A parked thread remains suspended until another thread calls *unpark* on it. The methods *parkNanos* and *parkUntil* suspend the calling thread until the thread is unparked, or until a timeout has elapsed or a deadline is passed, whichever comes first. The callers of the park methods pass a *blocker object* which represents the entity that caused the thread to park. The blocker object is typically the synchronization object itself

```
class LockSupport {
  static void park(Object blocker);
  static void unpark(Thread thread);
  static void parkNanos(Object blocker, long timeout);
  static void parkUntil(Object blocker, long deadline);
  // ... variants of park without blocker argument ...
}
```

Figure 4: Methods of the LockSupport class

or an object associated with it. For example, *ReentrantLock* passes an instance of its inner class *NonfairSync* (or *FairSync*). Although passing a blocker object is optional, implementers are strongly encouraged to do so for diagnostic purposes.

When a thread cannot acquire a lock, it calls *park* to suspend itself and passes a blocker object that represents the lock. When a thread releases a contended lock, it calls *unpark* to resume a parked thread that has requested the lock. Therefore, we decided to trace the individual *park* and *unpark* calls in all threads. The class of the blocker object reveals the used type of lock (or other synchronization mechanism) and enables us to infer the exact semantic meaning of the park and unpark calls, so we can correlate them with each other and determine which threads blocked which other threads.

We also need to arrange the events for the park and unpark calls in their correct order to analyze them. Ideally, we would also generate separate sequence numbers for each lock so that the events from different locks can be ordered and analyzed independently. However, the lock is represented by the blocker object, which is only passed to *park,* not to *unpark,* and is therefore unknown when writing an unpark event. Therefore, we assign a *global sequence number* to each event, which establishes a definitive happened-before relationship between all park and unpark calls in all threads. We further use the global sequence number of an event to refer to that event from other events.

When a thread calls *park,* we record a *park begin event* with a sequence number, a timestamp, the identity of the blocker object, the object's class, and the call chain. Moreover, we include whether a timeout or deadline was specified upon which the thread would resume even without being unparked.

When a thread calls *unpark* to resume a parked thread, we record an *unpark event* with a sequence number, a timestamp, the identifier of the unparked thread, and the call chain. Moreover, we store the unpark event's sequence number to a thread-local structure of the unparked thread.

As soon as the unparked thread resumes its execution, we write a *park end event* with a sequence number and a timestamp. We retrieve the sequence number of the corresponding *unpark event* from the thread-local structure and also include it in this event, so the analysis can easily match the two events. Because a call to *park* can also end due to a timeout, we also record in the event whether this was the case.

Figure 5 shows an example of events that we record in four contending threads that use a non-fair *ReentrantLock.* Initially, T_1 is able to acquire the lock without contention. Next, T_2 tries to acquire the lock and fails, so it enters the queue of the lock and parks, and thus, we record a *park begin event.* T_4 then also fails to acquire the lock, enters the queue and parks, so we record another *park begin event.* When T_1 releases the lock, it unparks T_2 as its successor and we write an *unpark event.* However, T_3 is able to acquire the lock before T_2 resumes its execution. T_2 writes a *park end event,*

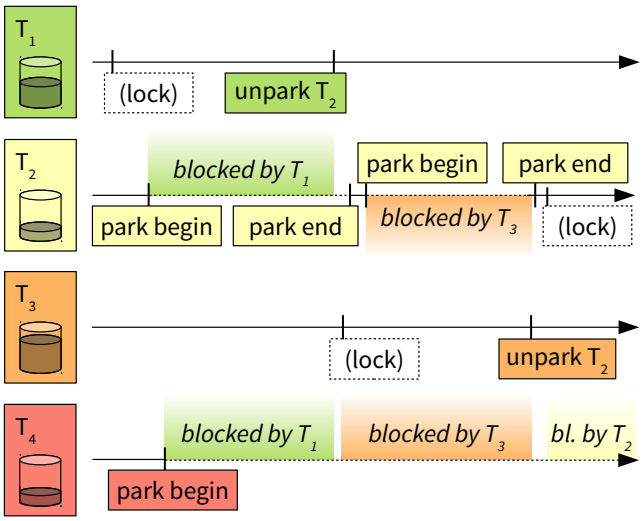

Figure 5: Park/unpark events in contending threads

but finds that the lock is still unavailable, so T_2 parks again, and we write another *park begin event.* Finally, T_3 releases the lock and unparks T_2 as its successor. When T_2 resumes its execution, we record a *park end event,* and T_2 is finally able to acquire the lock. T_4 remains parked.

During the analysis of the trace, we examine the first *park end event* of T_2, which leads us to the *unpark event* of T_1. Because a blocker object of class *ReentrantLock.NonfairSync* was recorded, we can infer that the unpark call was the consequence of an *unlock* operation, and that T_1 held the lock before the *unpark event.* The same applies to the the second unpark call, where T_3 held the lock. We can then account for the contentions in T_2 as being caused by T_1 and T_3 and their recorded call chains. Thread T_4 was also blocked by T_1 and T_3, but unpark was called only on T_2. However, because T_4 specified the same blocker object as T_2 when parking, we can infer that it was blocked by the same lock owners and also account for its contention as being caused by T_1 and T_3. Because the time between an *unpark event* and a *park end event* cannot be precisely attributed to the previous or to the next lock owner, we simply attribute such typically very short time periods to an unknown lock owner.

3.4 Metadata

Our traces contain a significant amount of repetitive data, such as the identities of threads, classes and objects, as well as call chains. Therefore, we want to collect and encode such data as efficiently as possible. However, for the data to be valuable for a user, we need to provide a meaningful representation, such as the name of a thread instead of just its numeric identifier. We decided to address this issue with *metadata events.* When we encounter an entity (such as a thread or a class) for the first time, we record a metadata event with a unique identifier for the entity and include information that is meaningful to a user. In the events that follow, we refer to that entity with only its identifier.

When the application launches a new thread, we record a *thread start event* with the thread's name and the numeric identifier that the Java runtime assigned to the thread. In future events, we refer to the thread only with that identifier. When the name of a thread is changed later, we record a *thread name change event* with the new name.

When we encounter a specific Java class for the first time, we write a *class metadata event* with the fully-qualified name of the class. HotSpot stores the metadata of a class in a data structure with a constant address, so we use that address as the unique identifier of the class. We introduced an additional field in the class metadata structures that we use to mark a class as known. Because two threads might race to write a class metadata event, we atomically update that field before writing an event, and the thread that succeeds in updating the field then writes the event.

Finding a unique identifier for Java objects is difficult. Because objects are moved by the garbage collector, their address is not suitable as an identifier. Instead, we refer to an object by recording its *identity hash code* and its class in our events. In HotSpot, the identity hash code of an object is a 31-bit integer that is randomly generated and stored with the object. In rare cases, two different objects of the same class can be assigned the same identity hash code, so that the two objects would be indistinguishable during analysis. We consider this to be an acceptable tradeoff compared to a more complex approach that involves tracking objects.

When recording call chains, we also refer to individual Java methods. Like classes, HotSpot stores the metadata of Java methods in data structures with constant addresses, which we use as unique method identifiers. When we encounter a method for the first time, we write a *method metadata event* with its identifier, the identifier of the method's class, the method's name, and the method's signature. We also use a newly introduced field to mark known methods.

Although the thread which first encounters an entity records a metadata event for it, some other thread may submit its trace buffer before that thread does. The trace analysis must be able to handle situations where events refer to an entity whose metadata event has not been processed yet, and must be able to resolve such references later.

Call chains.

We consider call chains to be vital for understanding locking bottlenecks, but walking the stack and storing them is expensive. Therefore, we have devised several techniques to record call chains more efficiently.

In a typical application, the number of call chains which use locks is limited, and many of the events that we record share identical call chains. To reduce the amount of data, we maintain a hash set of known call chains. When we record a call chain for an event, we look it up in that set. If it does not exist, we assign a unique identifier to it, insert it into the set, and write a *call chain metadata event* with its identifier and its methods. If the call chain already exists in the set, we can just record its identifier. We compute the hash code of a call chain from the instruction pointers from its methods.

Because multiple threads can access the set of call chains concurrently when recording events, those accesses require synchronization. We minimized the risk that the hash set becomes a bottleneck by implementing its operations in a *lock-free* way: when we record a call chain, we first walk the collision chain for its hash code without using any synchronization. If the call chain is found, we simply use its identifier. Otherwise, we generate a unique identifier for the call chain, and attempt to insert it into the set using a compare-and-set operation. If that operation succeeds, we record a call chain metadata event. If the compare-and-set fails, some other thread has inserted a call trace in the same collision chain, so we start over and check if the collision chain now contains the call chain in question. Therefore, we keep the overhead of insertion to a minimum, and looking up an existing call chain incurs no synchronization overhead at all.

We also optimized the stack walk itself. A JIT-compiled Java method usually has several of its callees inlined into its compiled code, and walking the stack frame of such a method typically entails resolving which methods are inlined at the current execution position. In HotSpot, this requires decoding compressed debugging information from the compiler. Instead, we perform light-weight stack walks which do not resolve the inlined methods, and also store call chains without the inlined methods in our hash set. We resolve the inlined methods only when we encounter a new call chain for which we write a call chain metadata event.

Finally, we devised a technique to reuse parts of a call chain that were recorded earlier in the same thread. We derived this technique from our earlier research on incremental stack tracing [8]. When we walk the stack of a thread to construct its call chain, we cache the resulting call chain in a thread-local structure. We also mark the stack frame that is below the top frame by replacing its return address with the address of a code snippet, and retain the original return address in a thread-local structure. When the marked frame returns, the code snippet is executed and simply jumps to the original return address, and the marking is gone. However, as long as the marked frame does not return, we can be certain that the frames below it have not changed. When we walk the stack again later and encounter a marked frame, we can stop the stack walk and complete the call chain using the frames of the cached call chain. This technique is intended to reduce the overhead of stack walks when recording multiple events in the same method, such as a contended enter event and a contended exit event, or multiple park begin events.

Unloading of classes and compiled methods.

We refer to classes and methods by using the constant addresses of their metadata structures as identifiers. However, when HotSpot's garbage collector detects that a class loader has become unreachable, it unloads all classes loaded by that class loader and reclaims the memory occupied by their metadata. When other metadata is loaded into the same memory, the addresses that we used as identifiers in earlier events become ambiguous during the analysis of the trace.

Therefore, we need to record when identifiers become invalid. We extended the class metadata event to include the class loader, which we also identify by the address of its metadata structure. When a class loader is unloaded during garbage collection, the application is at a *safepoint,* which means that all application threads are suspended and cannot write trace events. At this point, we first reclaim the trace buffers of all threads, which can still contain references to classes that are about to be unloaded, and add them to the processing queue to ensure that they are processed first. Then we acquire a new buffer, write a single *class loader unload event* with the identifier of the class loader, and immediately submit the buffer to the processing queue. When the trace analysis processes this event, it forgets all class, method and call chain metadata that refers to the unloaded classes. Finally, we let HotSpot unload the classes.

Compiled methods are also frequently unloaded, for example when assumptions that were made during their compilation turned out to be wrong. Other code can then be

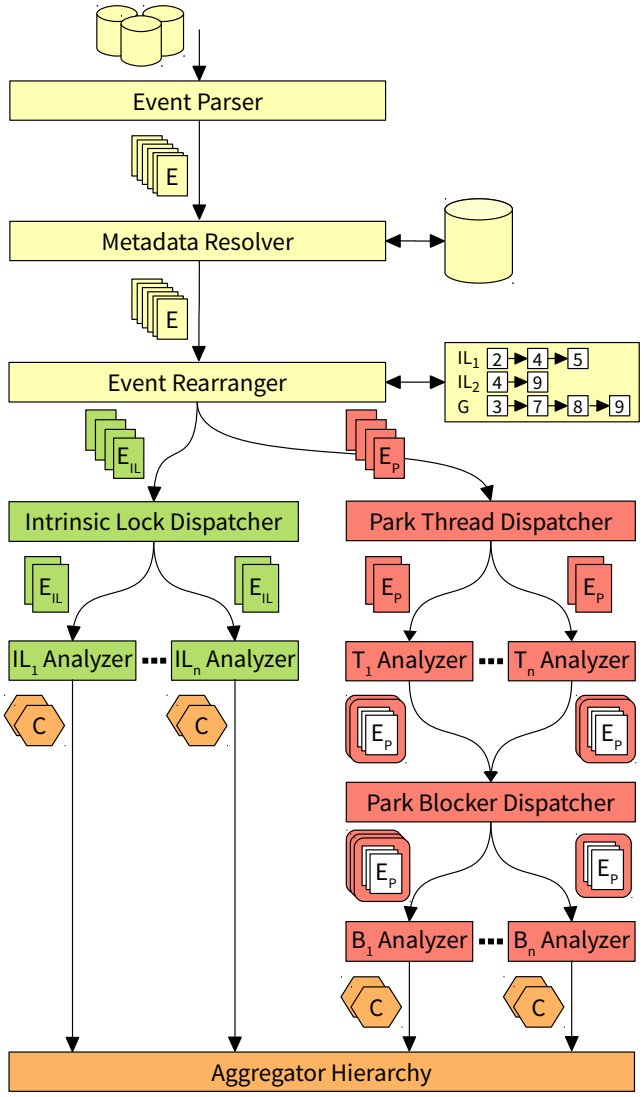

Figure 6: Processing events to identify contentions

loaded into the same memory. Because we store addresses of compiled methods in our call chains, these addresses can then become ambiguous, so we purge call chains with unloaded methods from our set of known call chains.

4. TRACE ANALYSIS

In order to identify synchronization bottlenecks effectively, we need to compute meaningful statistics from the recorded events. We accomplish this in two phases: first, we correlate events from different threads with each other to identify individual lock contentions. In the second phase, we aggregate these contentions by user-defined criteria.

4.1 Correlation and Contention Analysis

Figure 6 shows the process of extracting contentions from a trace. The *event parser* processes one trace buffer at a time. It parses the events in the buffer and forwards each event to the metadata resolver. The *metadata resolver* extracts all metadata from metadata events and keeps them in data structures. It replaces the metadata identifiers in all types of events with references to those data structures so that later phases can access those data structures directly. The metadata resolver then passes the events to the *event rearranger,* which reorders them according to their sequence numbers. The event rearranger maintains one queue per intrinsic lock and passes each intrinsic lock's events in their correct order to the *intrinsic lock dispatcher.* For park/unpark events with a global sequence number, the event rearranger uses a single queue and passes the events to the *park thread dispatcher.*

The *intrinsic lock dispatcher* creates a *lock analyzer* for each intrinsic lock that it encounters in the events, and passes the events of that lock to its analyzer. The analyzer replays the events and keeps track of which threads were blocked and which thread held the lock, and finally generates *contention objects.* These contention objects store the duration of the contention, the thread that was blocked, the thread's call chain, the lock's associated object, and that object's class. Most importantly, the contention objects also store the cause of the contention, that is, the thread that held the lock and that thread's call chain. These contention objects are then submitted to aggregators that compute statistics, which we describe in Section 4.2.

For the park/unpark mechanism, the analysis is more complex. The *park thread dispatcher* creates a *thread analyzer* for each thread that parked or was unparked, and forwards the events of that thread to its analyzer. The thread analyzer replays the events and creates bundles of related *park begin, unpark* and *park end* events. It submits those bundles to the *park blocker dispatcher,* which creates a *blocker analyzer* for each blocker object that occurs in those bundles. We implemented different types of blocker analyzers to handle the different synchronization semantics of java.util.concurrent classes. The park blocker dispatcher chooses which type of blocker analyzer to create based on the blocker object's class. The blocker analyzer examines the event bundles that are passed to it, tracks the state of the blocker, and creates contention objects that it submits to the aggregator hierarchy. We have implemented blocker analyzers for *ReentrantLock* and for *ReentrantReadWriteLock.* ReentrantLock is very similar to intrinsic locks, and its analyzer processes events as described in the discussion of Figure 5. With a ReentrantReadWriteLock, multiple readers can share the lock at the same time without calling park or unpark, so we do not record events for those readers. Only the last reader that releases the lock calls unpark on a blocked writer and records an event. Therefore, our analyzer for ReentrantReadWriteLock cannot determine all readers which blocked a writer, but it still accurately determines which writers blocked which readers, and which writers blocked each other.

4.2 Aggregation of Contentions

To enable users to find and examine locking bottlenecks in an effective way, we devised a versatile method to aggregate the individual contentions by different aspects. These aspects of contentions are the contending thread, the contending call chain (or method), the lock owner thread, the lock owner's call chain (or method), the lock object (an intrinsic lock's associated object, or a park blocker object), and that object's class. As another aspect for aggregation, we categorize contentions into groups for intrinsic locks and for park/unpark synchronization. The user selects one or more of these aspects in a specific order. We then build a hierarchy of selectors and aggregators that break down all contentions by those aspects. An *aggregator* computes the

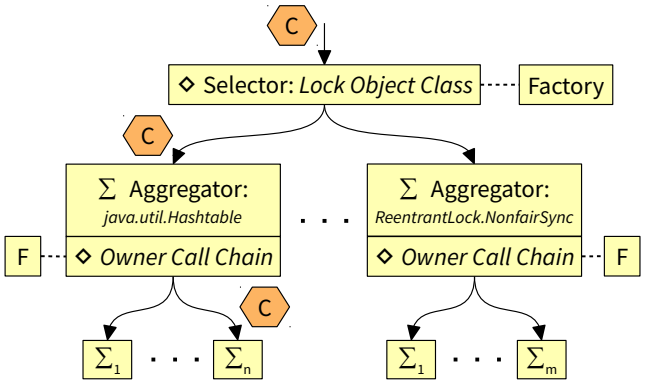

Figure 7: Aggregating events

total duration of all contentions that it processes. A *selector* distinguishes contentions by a specific aspect and forwards them to a specific aggregator according to their values.

Figure 7 shows an example in which contentions are first aggregated by the *lock object's class,* and then by the *lock owner's call chain.* When the trace analysis produces a contention object, it submits it to the hierarchy's *root selector,* which distinguishes contentions by the lock object's class. The selector has a factory that it uses to create an aggregator for each distinct lock object class that it encounters. Assuming that the submitted contention has *java.util.Hashtable* as its lock object class, the selector forwards the contention to the aggregator for that class. The aggregator then adds the contention's duration to its total duration. Because we aggregate by the lock owner's call chain next, each aggregator on this hierarchy level is coupled with a selector that distinguishes a contention by the lock owner's call chain after the aggregator has processed it. The selector also has a factory (denoted by F), which it uses to create aggregators for each encountered call chain and then forwards contentions to them. Assuming that the contention matches the call chain of aggregator Σ_n, the selector forwards the contention to that aggregator, which then adds the contention's duration to its total duration. Because there are no more aspects to aggregate by, the aggregators on that level are not coupled with selectors. The final result of aggregating multiple contentions is the tree of aggregators and the total contention times that they computed. In this example, a user might recognize from that tree that there is significant contention with *Hashtable* object locks, and that there are two call chains that cause most of these contentions. The user could then choose to optimize the code that these two call chains execute, or also select a different data structure or implementation, such as ConcurrentHashMap.

4.3 Interactive Visualization

In order to enable users to perform a comprehensive offline analysis of the generated traces, we built an interactive visualization tool. Figure 8 shows a screen capture of this tool displaying a trace file from the *avrora* benchmark, which simulates a network of microcontrollers. The main window is divided into three parts: a drill-down selection panel at the top, an aggregation tree in the center, and a detail view for the selected entry in the aggregation tree at the bottom.

In this example, the drill-down panel is configured to aggregate contentions first by group, then by the lock object's class, next by individual lock objects, then by the lock owner's method, and finally by the lock owner's call chain. Therefore, the root level of the tree displays the different groups that we categorize events in. The first entry represents contentions from intrinsic locks, and the *Total Duration* column displays that it makes up 100% of all contentions. It also shows an absolute value of 37 seconds, which is the total time that all threads spent being blocked on intrinsic locks. The entries on the next two tree levels break down that time by lock object classes and further by individual objects, and show that 99.78% of the contention comes from locking *java.lang.Class* objects, and that all of that contention involves a single *Class* instance with identity hash code *49abf544*. The two tree levels below that display owner methods and owner call chains. With 99.32%, *SimPrinter.printBuffer* is almost always the owner of the lock when a contention occurs. The multiple owner call chains show that this method is called from more than one location, and that the amount of caused contention varies significantly by call chain. Call chains are typically too long to fit into a single line, so we show *[+n]* to denote that *n* calls have been omitted. However, a user can select a specific entry to view the entire call chain in the detail view below the tree.

We examined the source code of avrora and found that *SimPrinter.printBuffer* is used to log simulation events and that it calls certain static methods of class *Terminal*. To avoid that the output from different threads is interleaved, printBuffer acquires the intrinsic lock of the *Class* object of Terminal, which poses a locking bottleneck. In Figure 8, we see two call chains that cause 72.75% and 11.41% of all contention, and in both of them, printBuffer is called by *fireAfterReceiveEnd*. This method logs when a simulation node receives a network packet, which is the most frequently logged event, and its output is very large because it includes a hexadecimal representation of the packet's data. To mitigate this bottleneck, contention could be reduced by logging fewer details or fewer events, or also by queueing log events and asynchronously writing them to a file in a background thread.

Figure 8 demonstrates that our tool enables users to see the methods and call chains that caused contentions, unlike common approaches that show only which methods and call chains were blocked. However, by choosing different aspects for aggregation, users can also extract a wealth of other information, such as whether some threads caused or suffered more contention than other threads, which call chains held the lock when a specific call chain was blocked and for how long, or which contended locks were used by a specific thread, method, or call chain.

5. EVALUATION

We implemented our approach for OpenJDK 8u45 and evaluated it with synthetic workloads from a purpose-built test suite, and with real-world benchmarks.

5.1 Synthetic Workloads and Correctness

In order to verify that our approach accurately depicts the locking behavior of an application, we devised a test suite that generates predictable synthetic locking workloads. We built this test suite using the Java Microbenchmark Harness [12]. In our tests, we vary the number of threads and the number of call chains. Most importantly, we vary how long the different threads or call chains hold a lock, which changes how much contention each of them causes. We implemented those

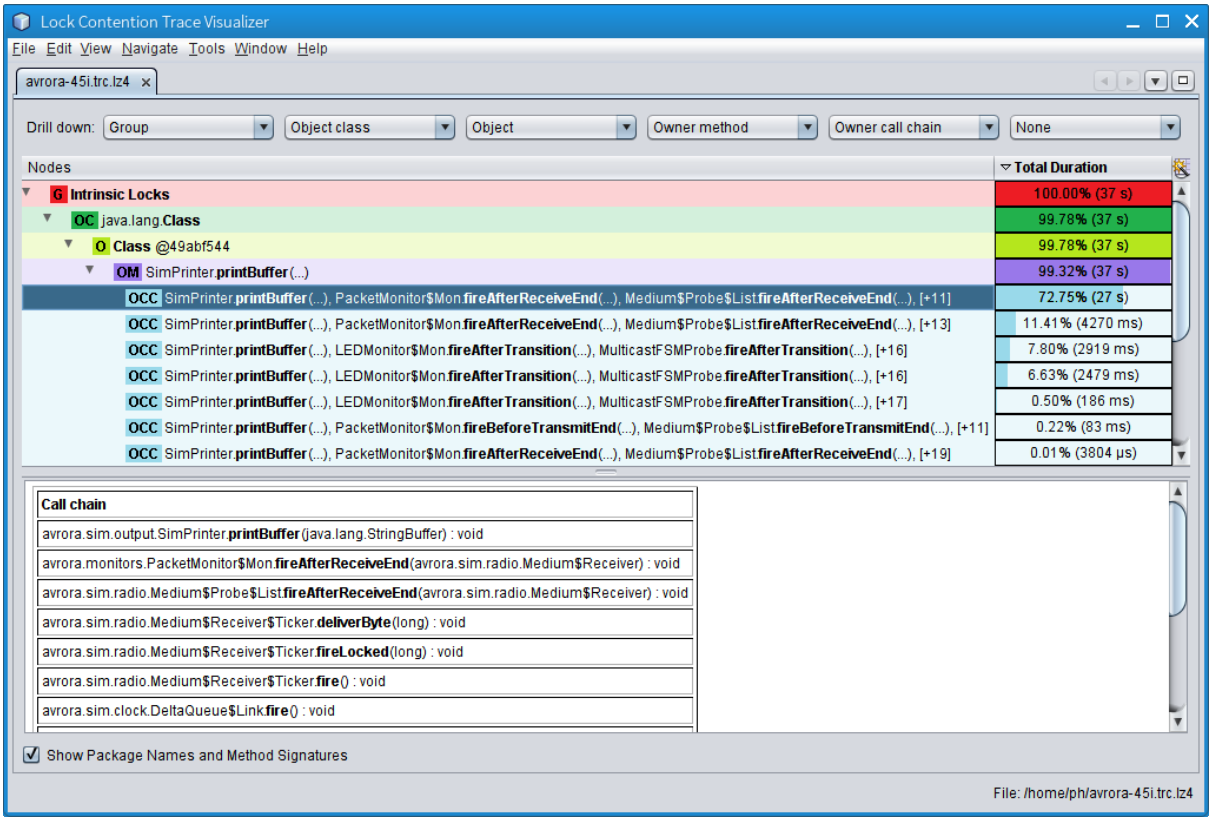

Figure 8: Visualization of a bottleneck in the *avrora* benchmark

tests for intrinsic locks and for java.util.concurrent locks and found that our generated traces match the expected amounts of contention for each test. Moreover, we verified that the recorded call chains are correct when the code throws exceptions, when the compiler inlines code, and when the code uses *wait* and *notify*.

5.2 Benchmarking

We evaluated our approach with the DaCapo 9.12 benchmark suite [2] and with the benchmarks of the Scala Benchmarking Project 0.1.0 [16]. Both suites consist of open source, real-world applications with pre-defined, non-trivial workloads.[1] We chose to execute 45 successive *iterations* of each benchmark in a single VM instance, and to discard the data from the first 35 iterations to compensate for the VM's startup phase. We tracked the start and the end of the benchmark iterations to extract the execution time and other metrics per iteration, and reinitialized event tracing at the start of each iteration. We further executed more than 10 *rounds* of each benchmark (with 45 iterations each) to ensure that the results are not biased by optimization decisions which the VM makes in its warm-up phase.

We performed all tests on a server-class system with two Intel Xeon E5-2670v2 processors with ten cores each and with hyperthreading, and thus, 40 hardware threads. The system has a total of 32 GB of memory and runs Oracle Linux 7. To get more reproducible results, we disabled dynamic frequency scaling and turbo boost, and we used a fixed Java heap size of 16 GB. With the exception of system services, no other processes were running during our measurements.

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

5.3 Runtime Overhead

We measured the benchmark execution times with tracing when writing an uncompressed output file, when writing a compressed output file, and when analyzing the events online. The online analysis executes in parallel to the benchmark and aggregates the contentions by lock object class, then by contending call chain, and then by the lock owner's call chain. We compare these execution times to those of an unmodified OpenJDK 8u45 without tracing.

Figure 9 shows the median execution times of each benchmark, normalized to the median execution times without tracing. The error bars indicate the first and third quartiles. We categorized the benchmarks into multi-threaded and single-threaded benchmarks. The *G.Mean* bars show the geometric means of each category, and their error bars indicate a 50% confidence interval. For the multi-threaded benchmarks, the mean overhead of generating a trace file is 7.8%, both with and without compression. With online analysis, the mean overhead is 9.4%. For the single-threaded benchmarks, the mean overhead of generating a compressed trace file is 0.8%, and with online analysis, it is 1.2%. The overhead for single-threaded benchmarks is caused in part by the trace buffer management, and in part because the JDK itself uses multi-threading and synchronization, which we trace as well.

The overheads of the individual benchmarks correlate directly with the amount of contention that they exhibit. The benchmarks with the highest overhead are actors and

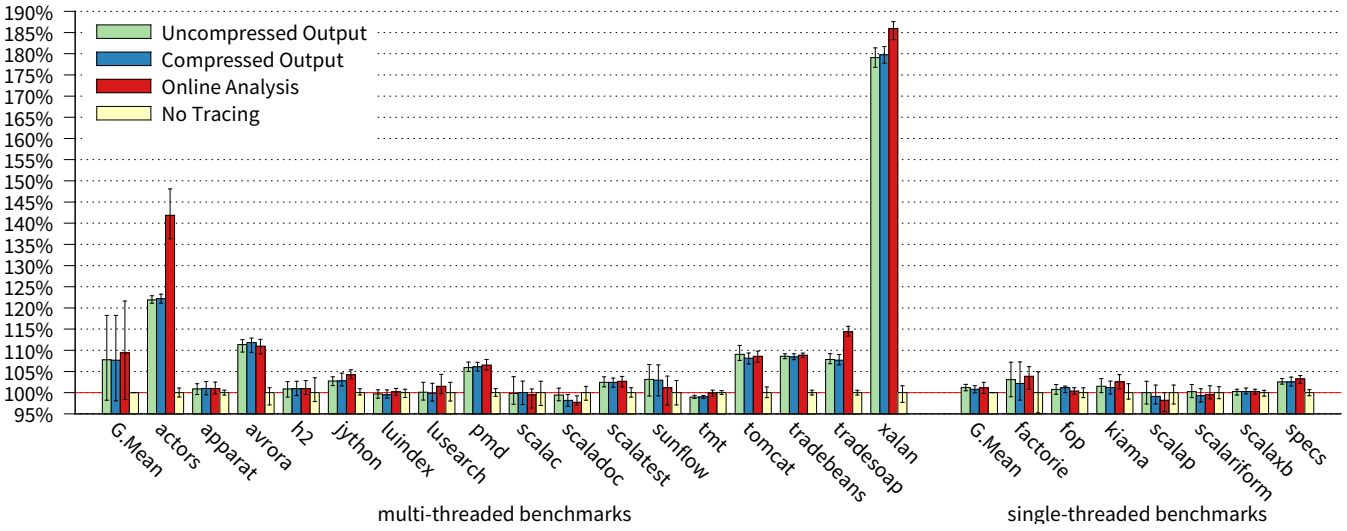

Figure 9: Overhead of tracing with uncompressed output, with compressed output, and with online analysis, relative to no tracing

xalan. *actors* is a concurrency benchmark with many fine-grained interactions between threads, and tracing them results in an overhead of 22%. Online analysis increases it to 42%, which is caused by the benchmark's extensive use of java.util.concurrent synchronization, for which the analysis is more complex and thus slower. The *xalan* benchmark transforms XML documents. It distributes work to as many threads as there are hardware threads, but all threads access a single Hashtable instance. This causes a substantial number of short contentions on our machine with 40 hardware threads, and tracing them incurs an overhead of around 80%.

For some benchmarks such as *scaladoc* or *tmt,* tracing even slightly improves the benchmark's performance. We attribute this to the delays that are introduced by recording events and call chains. David et al. found that HotSpot's locks saturate the memory bus, and that delays in lock acquisition reduce memory bus contention, which can increase performance [3]. Also, after we write a contended enter event, HotSpot's implementation of intrinsic locks retries to spin-acquire the lock while it adds the thread to the queue of blocked threads. A delay before that can increase its chances of being successful instead of suspending the thread. Additionally, the activity of the background thread in which we write the trace file or analyze the trace data influences the behavior of the garbage collector and of the thread scheduler, which can also have small beneficial effects.

The runtime overhead of our tracing is below 10% for all but three benchmarks. We consider this to be feasible for monitoring a production system. On a quad-core workstation, we measured even lower mean overheads below 3%. In future work, we intend to address cases in which the overhead is higher, in particular when tracing a substantial number of short contentions. We plan to support enabling and disabling tracing at runtime, which would allow users to analyze lock contention on a production system on demand, while causing little to no overhead when tracing is not active.

5.4 Generated Amount of Data

The amount of generated trace data is also an important factor for production use. Figure 10 shows the mean amount

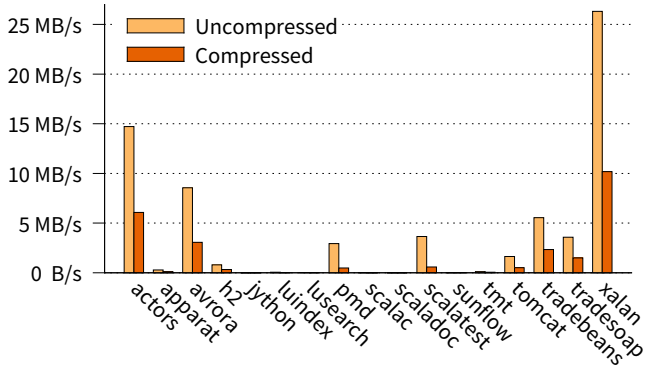

Figure 10: Trace data generated per second

of generated trace data per second for the multi-threaded benchmarks, with and without compression. Tracing *xalan, actors* and *avrora* generates the most uncompressed trace data at 26.3 MB/s, 14.7 MB/s, and 8.6 MB/s, respectively. For the other benchmarks, our approach generates less than 6 MB of uncompressed data per second. For those benchmarks that do not exhibit significant contention, we record less than 50 KB per second. Our on-the-fly compression typically reduces the amount of data by between 60% and 70%, and decreases the data rate of *xalan* to around 10 MB/s. Therefore, 60 minutes of trace data from a *xalan*-type application require less than 40 GB of disk space and should be more than sufficient to analyze performance problems.

We also inspected the memory footprint of the trace buffers, which have a mean capacity of 16 KB. We found that we typically use fewer than 100 buffers at any time, and hence occupy less than 2 MB of memory with trace buffers.

5.5 Trace Composition

We further examined the composition of the generated traces. Figure 11 shows the relative frequencies of individual events for the multi-threaded benchmarks. We grouped all types of metadata events for brevity. The *actors* and *apparat* benchmarks are the only ones that predominantly

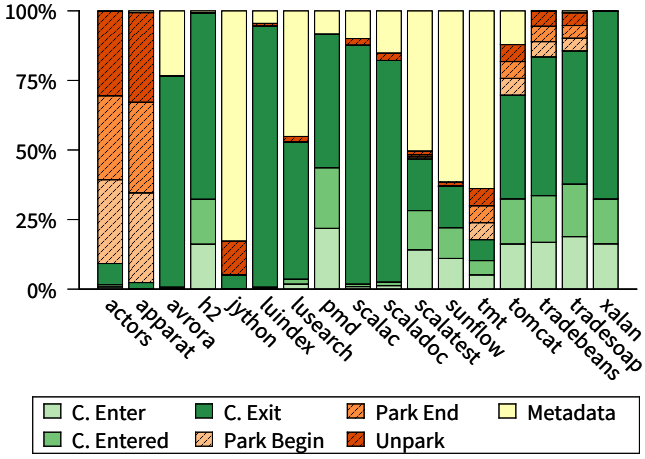

Figure 11: Frequency of trace events

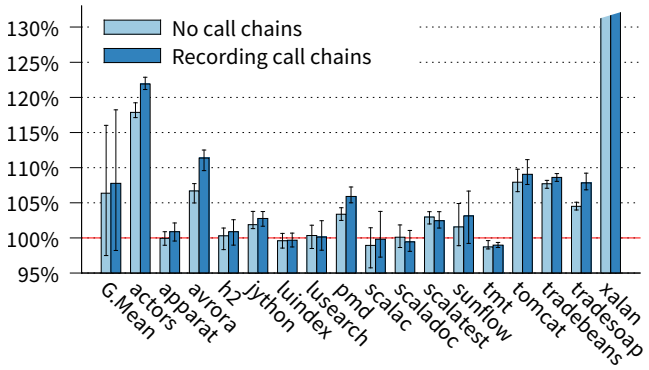

Figure 12: Overhead of recording call chains

rely on java.util.concurrent synchronization, with *tmt, tomcat, tradebeans* and *tradesoap* using it to some extent. As would be expected, there are typically equal numbers of park begin events, park end events, and unpark events. Surprisingly, we record only unpark events for some benchmarks, such as *jython.* This is because these benchmarks call the *Thread.interrupt* method, which always implicitly performs an unpark operation, regardless of whether the interrupted thread is currently parked.

With intrinsic locks, we record an equal number of contended enter and contended entered events, but always a significantly higher number of contended exit events. The difference is particularly large with benchmarks that acquire locks only very briefly, such as *luindex.* The reason for that is non-fair locking, with which a thread can instantly acquire an available lock even when there are queued threads, but when that thread releases the lock, it must write a contended exit event. When one of the queued threads has already been resumed and cannot acquire the lock, we do not write any additional events in that thread. With park-based synchronization, we would record another park begin event and park end event in that case, which is why the number of the three types of park events is more balanced.

For some benchmarks, metadata constitutes a relatively large portion of the trace data. Most of these benchmarks exhibit low contention, so that the amount of metadata that we record for that contention becomes relevant. In contrast to those benchmarks, *scalatest* and *tomcat* exhibit significant contention, but they generate many Java classes at runtime which use locks, so we record a large number of different call chains. *avrora* heavily uses conditional waiting with intrinsic locks, and to collect correct call chains, we must record an additional metadata event when a lock is finally released.

5.6 Call Chain Optimizations

In order to assess how much of our overhead comes from recording call chains, we also measured the tracing overhead when not recording call chains. Figure 12 compares the overheads of tracing with and without recording call chains for the multi-threaded benchmarks when writing an uncompressed trace file. When not recording call chains, the mean overhead decreases from 7.8% to 6.4%. Hence, our method of recording call chains typically constitutes less than 20% of the tracing overhead. The overhead for tracing *xalan* (which is cut off in the chart) is reduced from 79% to 73%.

We further examined the effectiveness of reusing call chains. In all benchmarks for which we recorded more than 10,000 call chains per second, we could reuse more than 99.5% of all call chains from the set of known call chains (*pmd* is the sole exception at 89%). This reduces the amount of metadata in the trace and requires only light-weight stack walks for the lookups in the set. With our technique of marking stack frames, we further save examining between 10% and 50% of all stack frames for those same benchmarks.

6. RELATED WORK

Tallent et al. [20] describe a sampling profiler which, like our approach, identifies which threads and call chains block other threads and call chains by holding a contended lock. The profiler associates a counter with each lock and periodically takes samples. When it samples a thread that is blocked on a lock, it increases that lock's counter. When a thread releases a lock, the thread inspects the lock's counter, and if it is non-zero, the thread "accepts the blame" and records its own call chain. The profiler was implemented for C programs and is reported to have an overhead of 5%, but determines *only* which threads and call chains blocked other threads. Our approach also records which threads and call chains were blocked by a specific thread or call chain, which we consider to have diagnostic value when reasoning about performance problems that *occur* in a specific part of an application.

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

Inoue et al. [9] describe a sampling profiler in a Java VM which uses hardware performance counters to observe where the application acquires locks and where it blocks. It constructs call chains with a probabilistic method that uses the stack depth. The profiler is claimed to have an overhead of less than 2.2%, but it does not determine which threads and call chains block other threads and call chains.

Java Flight Recorder (JFR, [6]) is a commercial feature of the Oracle JDK that efficiently records performance-related events in an application. It collects information on blocked

threads, their call chains, and the classes of lock objects. To keep the overhead low, JFR records only long contentions (more than 10ms) by default. JFR also records which thread most recently owned a lock before it could be acquired after a contention. However, although more than one thread can own a lock over that time, it considers the entire time that is spent blocking to be caused by that thread. Unlike our approach, JFR does not record call chains for threads that block other threads. It also only provides contention statistics for intrinsic locks and not for java.util.concurrent locks.

The Java VM Tool Interface (JVMTI, [14]) provides functionality to observe contention from intrinsic locks. Profilers can register callbacks for contention events and then examine the thread, the lock's associated object, and the call trace of the blocked thread. However, those events cannot be used to determine the thread which holds the lock or that thread's call chain. JVMTI also does not provide events to observe contention from java.util.concurrent locks.

Stolte et al. [17] describe *Polaris,* a system for analyzing and visualizing data in multidimensional databases. Polaris provides a visual query language for generating a range of graphical presentations, which enables users to rapidly explore the data. We believe that such a system would complement our visualization tool, and we consider implementing an export feature for the trace analyis results to a format that can be used with such systems.

7. CONCLUSIONS AND FUTURE WORK

We presented a novel approach for analyzing locking bottlenecks in Java applications by efficiently tracing lock contention in the Java Virtual Machine. We trace contention from both Java's intrinsic locks and from java.util.concurrent locks. For the analysis of the traces, we devised a versatile approach to aggregate and visualize the recorded contentions. Unlike other methods, our approach shows not only where contention occurs, but also where contention is caused. Nevertheless, our implementation in the HotSpot VM incurs a low mean overhead of 7.8%, so we consider it feasible to use our approach for monitoring production systems.

In future work, we intend to focus even further on the activities of lock owner threads while their locks are contended, for example by taking periodic samples of their call chains. We also plan to extend our analysis approach to identify connections between lock contentions, such as when a thread holds a contended lock and is then blocked when trying to acquire another lock. We further intend to collect more information about lock objects, such as the call chain where a lock object was allocated. This could provide additional information when locks are stored with data objects that are propagated to different parts of an application. Finally, we consider analyzing conditional waiting with both intrinsic locking and java.util.concurrent mechanisms. This could reveal problems such as when threads wait for each other to finish related tasks, but some tasks take significantly longer than others, leaving some threads idle.

8. ACKNOWLEDGEMENTS

This work was supported by the Christian Doppler Forschungsgesellschaft, and by Dynatrace Austria. We thank Thomas Schatzl for supporting us in evaluating our approach.

9. REFERENCES

[1] D. F. Bacon, R. Konuru, C. Murthy, and M. Serrano. Thin locks: featherweight synchronization for Java. In *ACM SIGPLAN Not.*, volume 33, pages 258–268, 1998.

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

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

[4] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea. *Java concurrency in practice*. Pearson Education, 2006.

[5] Google. Snappy: a fast compressor/decompressor. http://google.github.io/snappy/, 2015.

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

[7] P. Hofer, D. Gnedt, and H. Mössenböck. Efficient dynamic analysis of the synchronization performance of Java applications. WODA '15, pages 14–18, 2015.

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

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

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

[11] Oracle. java.nio package in Java SE 8. https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html.

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

[13] Oracle. The Java™ Management Extensions. http://openjdk.java.net/groups/jmx/.

[14] Oracle. JVM™Tool Interface version 1.2. http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html, 2015.

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

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

[17] C. Stolte, D. Tang, and P. Hanrahan. Polaris: A system for query, analysis, and visualization of multidimensional relational databases. *IEEE Trans. on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[18] D. Sundstrom. Snappy in Java. https://github.com/dain/snappy, 2015.

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

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