Dynamic Analysis of Java Locks

Steven P. Reiss and Alexander Tarvo
Department of Computer Science
Brown University
Providence, RI. 02912
{spr,alexta}@cs.brown.edu

Abstract

Java provides synchronization primitives in the form of synchronized regions with wait and notify method. Programmers use these regions to implement a variety of higher-level synchronization constructs. Understanding the locking behavior of a Java program requires understanding the high-level locking semantics the programmer intends. We have developed a system that uses dynamic analysis to determine the high-level meaning of different program locks, provides the programmer with information about how locks are used, and visualizes the locks in real time.

1. Introduction

"What is my program doing?" Programmers often ask that question and spend significant amounts of time attempting to answer it using a variety of tools generally not suited to the purpose. The overall goal of our research is to provide appropriate tools that help the programmer address this question effectively and efficiently.

The problem of understanding the program's behavior becomes especially challenging in case of the long-running multithreaded applications. Multithreaded programs result in increased performance on modern multicore and multiprocessor systems. However in order to function correctly, the execution of multiple threads must be carefully synchronized. If the thread synchronization is done improperly, such programs become a source of problems including deadlocks, race conditions, and performance issues. These problems are often hard to detect and reproduce.

Modern programming languages provide a set of primitives for synchronization in multithreaded programs. The basic synchronization primitive in Java is a *monitor* [12], implemented by every Java object using synchronized regions and *wait* and *notify* calls. The monitor maintains a queue of threads waiting on it. When a thread calls the object's *wait*() method, execution of that thread is suspended and it is added to the object's queue. Correspondingly, the *notify*() method fetches one of the waiting threads from the object's queue and resumes its execution when possible; while the *notifyAll()* method resumes execution of all waiting threads. Java also allows *wait()* calls to time out and to return spuriously.

Although monitors and synchronization regions are useful in some simple tasks, more complex programming scenarios require for more sophisticated synchronization mechanisms. Thus programmers use basic synchronization primitives to implement higher-level synchronization mechanisms such as atomic regions, condition variables, semaphores, read-write locks, and barriers. In this paper we call these higher-level abstractions *logical lock types*. While programmers typically write Java code in terms of synchronized regions and *wait()* and *notify()* calls, they design, think about, and maintain their systems in terms of these higher-level logical lock types.

Unfortunately, current tools for understanding and debugging of multithreaded programs provide information in terms of the low-level synchronization primitives rather than the higher-level logical locks that the programmer thinks in. This limits the utility of such tools and makes them difficult to use for analyzing real-world scenarios.

To address this problem, we developed a tool that understands the behavior of the program in terms of high-level synchronization mechanisms and presents this information to the programmer. This tool works in stages.

During the first stage we record all interactions of synchronization primitives in the user program using the standard Java monitoring tool, JVMTI [18]. JVMTI does not report all the information we need, so we infer missing data and create a complete facsimile log of synchronization operations in the user program.

During the second stage, we identify *logical lock classes*, multiple objects in the program that are used as locks and represent different instances of the same logical lock from a design perspective. For example, every Swing document has its own read/write lock. These multiple read/write locks are logically the same in that they share a common behavior and should have the same logical lock type. The second stage of our system finds and groups such logical lock instances.

During the third stage we analyze the complete facsimile log and identify how each logical lock class identified in stage two is used, essentially assigning it to one or more lock types. The lock types we currently identify

include mutexes, barriers, semaphores, read-write locks, producer-consumer locks, general condition variables, and latches. For each type, we identify the relevant synchronization regions appropriate to the type. For example, for a semaphore, we will identify which regions do acquire operations and which regions do release operations. This analysis is flexible enough to detect when a particular lock is used for multiple purposes.

In the fourth stage we present the information on logical locks to the user. This includes a presentation of the logical lock types as well as associated information on lock performance, wait times, and lock nesting. Lock nesting behavior is shown as a graph that can be used to identify potential deadlocks.

The final stage of the system allows the user to select one or more logical locks and then run the program again to visualize the exact behavior of the selected locks. In this case, we instrument the running program by dynamically modifying the Java binary code and provide time-based visualizations of thread and locking behavior. These can be used to identify performance and other lock-related problems in the user program and to get either a high-level overview or a detailed picture of locking behavior in the program.

The tool has been used to analyze and visualize the locking behavior of various Java programs, from the n-body gravitational simulator to a new generation IDE.

The remaining of the paper will be organized as following. After reviewing related work in the next section, we describe stages of our tool in detail. In Section 3. we describe how we reconstruct locking sequences from the information provided by Java's low-overhead lock monitoring. In Section 4. we describe how we identify logical locks. Section 5. then describes how we use the lock sequences to categorize locks by how they are used. The various visualizations and presentations are described in Section 6. We conclude with an analysis of our experience and directions for future work.

2. Related Work

A number of tools exist for finding and describing deadlocks for Java programs. These include JConsole [5], IBM's Multi-Thread Run-time Analysis Tool for Java [6], FindBugs [13], GoodLock [11] and its extensions [1], Jade [19], RacerX [8], and work on library deadlock detection [25]. While these tools are quite effective at what they do, they don't do the right thing. They look only at Java synchronized regions. However, complex, multithreaded systems, even Java systems, typically use a much wider variety of locks. Our experience in writing concurrent systems over the past fifteen years is that only a fraction, less than 50%, of the deadlocks are due to different locking orders of synchronized regions. However, this is all that existing tools check for.

Other tools have concentrated on detecting race conditions. Examples of such tools include FastTrack [10], Goldilocks [7], DataCollider [9], DJIT+ [14], and Eraser [23], and its further development in MultiRace [20]. They find data items that are read by multiple threads and ensure that access to these items is controlled by a lock. These tools again assume that the program controls access to these items through simple locks, for example within synchronized regions. More complex locking schemes such as user-implemented read-write locks are difficult or impossible to detect statically.

Several performance analysis tools such as Paje [15], JaVis [17], Threadscope [24], and our JIVE [21] and DYVISE [22] look at locking behavior and try to visualize where and why threads are blocking. They represent interaction of the multiple threads as graphs [3,16,24] or as extended UML diagrams [2,17]. These tools again are dependent on the programmer's use of simple synchronization. For complex locks, they might tell what thread is blocking but it might be difficult to deduce what caused the block and hence what needs to be done to fix the performance problem. For example, the Swing *Document* class provides read/write locks that allow multiple readers and a single writer. The code tracks the current writer thread, but only note the number of readers, not who they are. When such locks are involved in a deadlock with one of the locked threads holding a read lock or when there are performance issues surrounding such locks, it is difficult to use such tools to determine both what thread is responsible for the errant read lock and where and when that lock was acquired.

3. Collecting Lock Traces

In order to determine types and locations of logical locks in the Java program, we must first record the behavior of synchronization primitives in that program. To do this we run the program in a typical configuration, and record all operations performed by the synchronization primitives for further analysis. In particular we need to know when the thread enters the synchronized region, when it exits that region, when it waits on the monitor and when it does not need to wait, and when the thread notifies other threads. From an engineering prospective, this requires identifying entry and exit points for all the synchronized regions; identifying calls to *wait*() and *notify*() and associating them with each other; and distinguishing *wait()* calls used for synchronization from those used for timing.

Recorded traces must be long and detailed to allow an accurate analysis of the locking behavior of the program. But since we are collecting this information on a running system, overhead of the data collection must be minimal.

Locking can occur very often in a program and full monitoring of all synchronized regions would slow the

program down significantly. For performance reasons, Java virtual machines treat locks which are being used by multiple threads (*contended* locks) differently from locks that are being used by a single thread (*uncontended*). The Java JVMTI monitoring tool follows up on this behavior and emits lock trace events only when a synchronized region is contended by multiple threads. Since these events are typically costly in the JVM, tracing them with the JVMTI does not significantly degrade the program's performance.

Using JVMTI allows reducing data collection overhead. However, raw JVMTI data do not contain a complete sequence of synchronization operations in the user program. Restoring this data requires some post-processing.

The first problem is that JVMTI reports all the locks in the application, including locks that occur in standard Java libraries and thus known to be safe. For example, Swing and AWT user interface libraries maintain the event queue guarded by the locks. These locks are constantly being used both by the Swing thread processing the queue and the Swing thread reading input events. These locks are known to be safe and thus are really not of interest to the programmer. To address this problem we maintain a table of "safe" locks. The tracing code ignores safe locks and does not generate output for them.

A more serious problem with the JVMTI traces is that they are incomplete and sometimes inaccurate. JVMTI traces are incomplete since they only include events for attempting to enter a synchronized region, entering a synchronized region, executing a wait, and finishing a wait either by time-out or by notification. They do not include events for exiting a synchronized region or for doing a *notify()* or *notifyAll()* operation. Moreover, JVMTI events do not indicate whether an attempted entry resulted in blocking the thread or not.

JVMTI traces may be inaccurate in cases when a lock that wasn't initially contended becomes contended because another thread attempted to enter it before the first thread could release the lock. In this case events corresponding to entering the lock might be missing: the trace for the lock will start with a *wait* or *waited* event with no prior *enter* event. Correspondingly, if the lock that was initially contended suddenly becomes uncontended, the trace might include a spurious *enter* and *entered* events, but events corresponding to the exiting from the prior wait might be missing.

All these cases are problematic for analysis of logical (high-level) locks. When a Java lock first becomes contended the state of the lock is essentially unknown. For example, for a read-write lock, the number and depth of readers and writers is not known. Similarly for a semaphore, the semaphore count is not known. When a lock becomes uncontended, the sequence of lock events might be incomplete and hence inaccurate.

We address these problems by reconstructing a facsimile complete lock sequence from the raw trace data. Reconstruction involves several steps:

First, we determine when in the trace threads exit a synchronized region and introduce corresponding events into the trace sequence. This is done by tracking what thread currently holds the lock and noting when the region is then entered by another thread. To handle nested locks by the same thread, our tracing code uses JVMTI calls to compute the nesting level of the current lock for the thread and includes that as part of the trace. We insert *exit* events into the trace in the appropriate temporal location.

Second, we need to introduce *notify* events. We do this by first doing a static analysis of the code to identify calls to *Object.notify()* and *Object.notifyAll()* and determining which synchronized regions in the program may make such calls. This involves looking at the synchronized regions themselves as well as the transitive closure of the routines called from those regions. Because this can result in false positives, we limit the depth of the exploration and don't assume every such call does a notify. Instead, we only insert *notify* events in the trace if a synchronized region could call *notify()* or *notifyAll()* and subsequently a call to *wait()* woke up on the same lock. We do a similar analysis to find routines that might call *Object.wait()*.

Third, we determine the scenarios for which we cannot obtain reliable information about the state of a certain lock. As mentioned earlier, the JVMTI monitors the state of the lock only when it is contended. When the lock becomes not contended, JVMTI stops monitoring it. Both of these result in missing or inaccurate data in the trace. To avoid confusion during the future analysis of the trace data, we introduce a notion of a *reset* event. *Reset* events do not correspond to a synchronization event in the program; they only signal that the exact state of the lock is unknown at this point.

Reset events are generated when we notice an inconsistent trace (for example one containing a *wait* event without a corresponding *enter* event), or when the time between events for a given lock is greater than a fixed threshold (currently 0.25 second). The latter is needed because lock sequences might look valid but, if the JVM marked the lock as not contended during that interval and then it later became contended, the overall lock state might not be correct. The time threshold here was determined experimentally and might need to be changed for other JVM implementations. Since *reset* events only affect latter analysis, this does not invalidate the trace.

Once all the post-processing steps are defined, the overall data collection process is simple. The event trace generated by the JVMTI is saved into the csv file and then split into individual sequences according to the lock identifiers. As a result, events related to different locks form separate sequences. Then each sequence

Event	Thread	Time	Synchronized Region
ENTER	11	96803438443	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTERED	11	96803571761	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
WAIT	11	96804810391	Ljava/awt/MediaTracker;@waitForID(IJ)Z@76
ENTER	25	96810704083	Ljava/awt/MediaTracker;@setDone()V@0
ENTERED	25	96810819386	Ljava/awt/MediaTracker;@setDone()V@0
WAITED	11	96810860111	Ljava/awt/MediaTracker;@waitForID(IJ)Z@76
ENTER	11	96811028249	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTERED	11	96811186800	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTER	11	96811326567	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTERED	11	96811464982	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTER	11	96811603310	Ljava/awt/MediaTracker;@removeImage(Ljava/awt/Image;I)V@0
ENTERED	11	96811740357	Ljava/awt/MediaTracker;@removeImage(Ljava/awt/Image;I)V@0
ENTER	11	98704830638	Ljavax/swing/ImageIcon;@loadImage(Ljava/awt/Image;)V@8
ENTERED	11	98704949691	Ljavax/swing/ImageIcon;@loadImage(Ljava/awt/Image;)V@8
ENTER	11	96803438443	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTERED	11	96803571761	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
UNLOCK	11	0	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
WAIT	11	96804810391	Ljava/awt/MediaTracker;@waitForID(IJ)Z@76
ENTER	25	96810704083	Ljava/awt/MediaTracker;@setDone()V@0
ENTERED	25	96810819386	Ljava/awt/MediaTracker;@setDone()V@0
NOTIFY	25	0	Ljava/awt/MediaTracker;@setDone()V@0
UNLOCK	25	0	Ljava/awt/MediaTracker;@setDone()V@0
WAITED	11	96810860111	Ljava/awt/MediaTracker;@waitForID(IJ)Z@76
ENTER	11	96811028249	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTERED	11	96811186800	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
UNLOCK	11	0	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
UNLOCK	11	0	Ljava/awt/MediaTracker;@waitForID(IJ)Z@0
ENTER	11	96811326567	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTERED	11	96811464982	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
UNLOCK	11	0	Ljava/awt/MediaTracker;@statusID(IZZ)I@0
ENTER	11	96811603310	Ljava/awt/MediaTracker;@removeImage(Ljava/awt/Image;I)V@0
ENTERED	11	96811740357	Ljava/awt/MediaTracker;@removeImage(Ljava/awt/Image;I)V@0
RESET	11	0	Ljavax/swing/ImageIcon;@loadImage(Ljava/awt/Image;)V@8
ENTER	11	98704830638	Ljavax/swing/ImageIcon;@loadImage(Ljava/awt/Image;)V@8
ENTERED	11	98704949691	Ljavax/swing/ImageIcon;@loadImage(Ljava/awt/Image;)V@8

FIGURE 1. Lock trace events before and after initial processing.

is sorted by the time of the event, and the post-processing steps, described above, are applied to it.

The final result is a sequence of events for each lock that includes the event type, the time (which is order-correct but might not be otherwise accurate for inserted events), the thread id, and the lock id. Example of such sequence is depicted in the Figure 1 where the top set shows the events reported by the JVMTI and the bottom set shows the events as we reconstructed them. This new sequence is what we use for the analysis of logical lock types.

4. Identifying Lock Classes

In terms of the Java program the logical lock can be seen as an object and the synchronized regions associated with that object. However, the program can create multiple instances of such objects and use them in exactly the same fashion to provide a uniform locking behavior. In this case we can think of creating multiple *instances* of the same logical lock.

For example, consider the read-write lock implemented by the Java *Document* class. This class along with methods that are using it constitute a single logical lock, and individual instances of that *Document* class are just instances of that logical lock. Providing the parallel to the object-oriented programming, one can think of this logical lock as of a “lock class” that can have many instances.

For the purpose of lock type analysis we want to study these logical lock classes, not the individual lock instances. Unfortunately, the lock trace we collected contains data on instances of the logical lock, where every such instance is represented by a separate set of records in the trace. Thus in order to do lock type analysis we first need to identify lock classes from the trace, before we can analyze the logical locks.

To identify a particular lock class we use the class name of the object being locked and the set of locations in the program’s code (method name and byte offset) comprising the synchronized regions used by that object for locking.

Neither the class of the object nor the set of synchronized regions alone are sufficient to uniquely identify lock classes. A class might extend *Hashtable* (a

class that does its own locking) and implement its own higher-level lock on top of it. In this case, lock locations of this new class will overlap with those that belong to the *Hashtable* class. Inheritance can also be irrelevant. If the program defines its own subclass of *Hashtable* and does not do any additional locking, the new object should be treated as a *Hashtable* for locking purposes. It is also relatively common to create objects of the *Object* class to be used as locks for different purposes. Here the class of the lock provides no useful information.

We rely on simple heuristics to identify logical locks in the trace. Two instances of the lock are considered to belong to the same logical lock if:

- Their objects have the same class and they share at least one code location; or
- Their objects have different classes and they share 75% or more of their code locations.

This cut off was determined experimentally by looking at how locks were used in a suite of sample applications.

Once lock instances are merged, the resultant lock classes are viewed as having the combined set of associated locations and classes. We iterate the process attempting to merge locks into the new classes. In this case of comparing lock classes with individual locks or other lock classes, we consider to locks to be of the same class if one contains all the classes used by the other.

5. Identifying Logical Lock Types

The next step in our lock analysis is identifying the logical lock type of each lock class. Our approach is to define the set of lock types that can be identified in the program. For each type we implemented a *checker* algorithm, which takes a lock trace as an input and attempts to match it to the expected behavior of the corresponding lock type.

Our current set of checkers includes mutexes, delays, barriers, semaphores, producer-consumer locks, read-write locks, latches, and conditional locks.

The trace for every logical lock in the program is passed to all the checkers. The checker assigns the lock to the given type ("validates" it) if two conditions are met:

- The sequence of events for that lock is consistent with the behavior of a lock of the given type (absence of negative evidence); and
- The sequence of events shows positive evidence of being a lock of the given type.

The consistency check is obviously necessary, but is not sufficient to identify lock types accurately. Any simple mutex that never does a wait would be consistent with most lock types. Similarly, a mutex that waits at some point could be viewed as a read-write lock that never does a read.

```
have_wait = false              // wait has been seen
have_notify = false            // notify has been seen
initial_count = -1             // initial value of the semaphore
count = 0                      // current semaphore count

foreach Event<type,location>
 if type is RESET then
    count = 0
 else if location is a P-location then// lock
    switch (type)
       case WAIT :
          have_wait = true;
          if (initial_count < 0) initial_count = -count;
          if (initial_count + count > 0) MARK INVALID
       case UNLOCK :
          --count;
          if (initial_count >= 0 && initial_count + count < 0)
             MARK INVALID
       case NOTIFY :
          MARK INVALID
 else if location is a V-location then// unlock
    switch (type)
       case ENTERED :
          ++count;
       case WAIT or WAITED
          MARK INVALID
       case NOTIFY :
          have_nofify = true;

Valid if not marked invalid && have_wait && have_notify &&
    initial_count >= 0
```

FIGURE 2. Semaphore checking algorithm

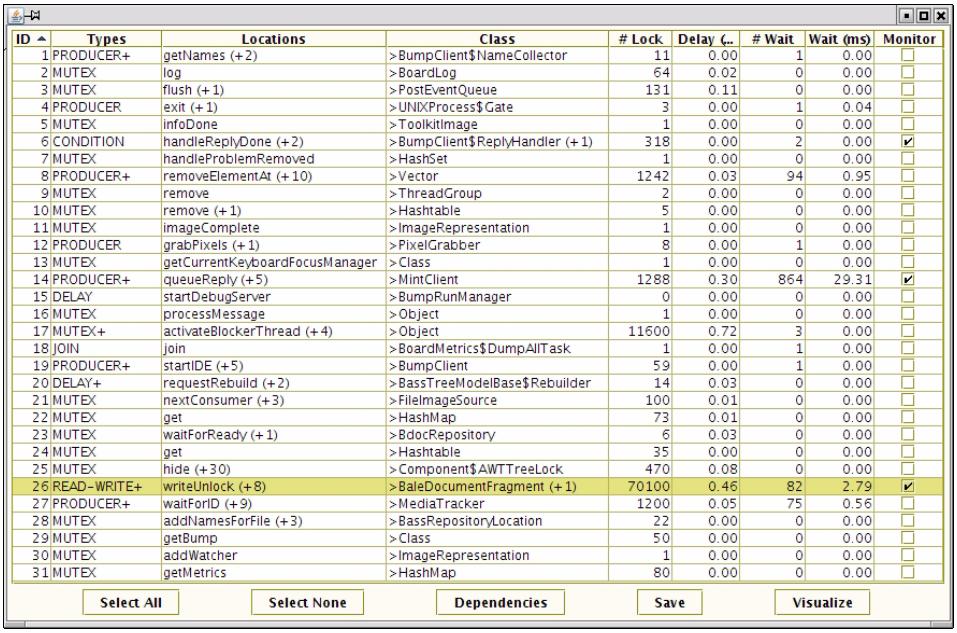
To further improve accuracy of lock identification, checkers analyze traces separately for each instance of the lock class. If any of these traces contradicts the expected behavior, the checker invalidates the assignment of the lock to a given type. However, positive evidence only has to come from one of the considered traces. Figure 2 shows the internal checking algorithm for semaphores.

Although the high level view of our approach looks simple, implementation of both the checkers and the supporting code is complicated by a set of challenges.

Tracking Lock States. Most locks maintain some internal state that determines their behavior. For example, the state of the semaphore is determined by the semaphore count, while the state of the read-write lock is determined by the number of reader and writer threads.

Knowing the state of the lock is essential to recognize the positive evidence for that lock. Consider an example when the thread trying to acquire the semaphore enters the wait state, and then is released by the next unlock on the semaphore. The corresponding sequence of events would be a positive evidence for the semaphore whose count is zero, but it will be inconsistent with the semaphore whose count is different.

Unfortunately, upon beginning of the trace and after each reset event the initial state of the lock may be unknown and the checker must deduce it from the trace. Until the state of the lock becomes known again, the checker cannot reliably find positive evidence for



ID	Types	Locations	Class	# Lock	Delay (...	# Wait	Wait (ms)	Monitor
1	PRODUCER+	getNames (+2)	>BumpClient$NameCollector	11	0.00	1	0.00	☐
2	MUTEX	log	>BoardLog	64	0.02	0	0.00	☐
3	MUTEX	flush (+1)	>PostEventQueue	131	0.11	0	0.00	☐
4	PRODUCER	exit (+1)	>UNIXProcess$Gate	3	0.00	1	0.04	☐
5	MUTEX	infoDone	>ToolkitImage	1	0.00	0	0.00	☐
6	CONDITION	handleReplyDone (+2)	>BumpClient$ReplyHandler (+1)	318	0.00	2	0.00	☑
7	MUTEX	handleProblemRemoved	>HashSet	1	0.00	0	0.00	☐
8	PRODUCER+	removeElementAt (+10)	>Vector	1242	0.03	94	0.95	☐
9	MUTEX	remove	>ThreadGroup	2	0.00	0	0.00	☐
10	MUTEX	remove (+1)	>Hashtable	5	0.00	0	0.00	☐
11	MUTEX	imageComplete	>ImageRepresentation	1	0.00	0	0.00	☐
12	PRODUCER	grabPixels (+1)	>PixelGrabber	8	0.00	1	0.00	☐
13	MUTEX	getCurrentKeyboardFocusManager	>Class	1	0.00	0	0.00	☐
14	PRODUCER+	queueReply (+5)	>MintClient	1288	0.30	864	29.31	☑
15	DELAY	startDebugServer	>BumpRunManager	0	0.00	0	0.00	☐
16	MUTEX	processMessage	>Object	1	0.00	0	0.00	☐
17	MUTEX+	activateBlockerThread (+4)	>Object	11600	0.72	3	0.00	☐
18	JOIN	join	>BoardMetrics$DumpAllTask	1	0.00	1	0.00	☐
19	PRODUCER+	startIDE (+5)	>BumpClient	59	0.00	1	0.00	☐
20	DELAY+	requestRebuild (+2)	>BassTreeModelBase$Rebuilder	14	0.03	0	0.00	☐
21	MUTEX	nextConsumer (+3)	>FileImageSource	100	0.01	0	0.00	☐
22	MUTEX	get	>HashMap	73	0.01	0	0.00	☐
23	MUTEX	waitForReady (+1)	>BdocRepository	6	0.03	0	0.00	☐
24	MUTEX	get	>Hashtable	35	0.00	0	0.00	☐
25	MUTEX	hide (+30)	>Component$AWTTreeLock	470	0.08	0	0.00	☐
26	READ-WRITE+	writeUnlock (+8)	>BaleDocumentFragment (+1)	70100	0.46	82	2.79	☑
27	PRODUCER+	waitForID (+9)	>MediaTracker	1200	0.05	75	0.56	☐
28	MUTEX	addNamesForFile (+3)	>BassRepositoryLocation	22	0.00	0	0.00	☐
29	MUTEX	getBump	>Class	50	0.00	0	0.00	☐
30	MUTEX	addWatcher	>ImageRepresentation	1	0.00	0	0.00	☐
31	MUTEX	getMetrics	>HashMap	80	0.00	0	0.00	☐

Select All | Select None | Dependencies | Save | Visualize

FIGURE 3. Table showing the result of lock analysis.

the given lock type. Moreover, when the lock is in an unknown state, the set of consistent event sequences is broader.

Tracking Locking Regions. To validate certain types of the locks, e.g. semaphores and read-write locks, the checker needs to establish the purpose of the various code locations that operate that lock, namely – what locking operation they perform. For example, a semaphore will have one set of code regions that are locking the semaphore and another set that are unlocking. Similarly a read-write lock will have different regions for doing a read-lock, a write-lock, a read-unlock, and a write-unlock operations. Generally these sets should be disjoint (although we let the read and write unlock sets overlap).

To determine what operations different regions perform, checkers evaluate all possible assignments of different lock operations to code regions. We somewhat reduce the number of possible assignments by knowing which regions actually do *wait* and *notify* operations and then applying appropriate heuristics.

In some cases there can be multiple overlapping valid assignments for the lock. This generally occurs when there is insufficient information on the lock because either the event trace was too short or the lock state could not be reliably deduced for all locations and regions that are superfluous to the lock are not explicitly excluded. In this case we prefer the assignment which involves the smallest number of locking regions for the lock.

Merging Checker Outputs. Since the sequence of events for the lock is passed to all the checkers, multiple checkers can "validate" the sequence. This can occur in two ways. First, the same logical lock can be used in different roles in the program. For example, one set of code locations use the read-write lock of a Document object for its designated purpose, while other routines synchronize on the same object to affect simple mutual exclusion. In another example, we use the same object for two control two separate queues. Second, the functionality of certain lock types can overlap. For example, a semaphore is a special instance of a general conditional lock.

To generate the final lock type we call the checkers in such an order that more specific checkers are looked at first. Moreover, each checker analyzes the set of synchronized regions for the lock and marks those regions that are consistent with the given lock type. We relate the lock to the corresponding type if:

- The checker validates the trace for the lock; and
- The checker only uses synchronized regions of the lock that were not used by previous checkers.

6. Displaying the Result

Once we have analyzed the locks, the next step is to present this information to the programmer. We visualize collected information in two phases. During the first phase a list of logical locks and their types is displayed at a high level. For the second phase, the programmer can select a subset of these locks to be visualized in more detail in later runs of the system.

High-Level Lock Visualization. Results of the lock analysis are shown in Figure 3. This and the latter figures depict results of the analysis of Code Bubbles, an interactive front end for programming [4]. Code

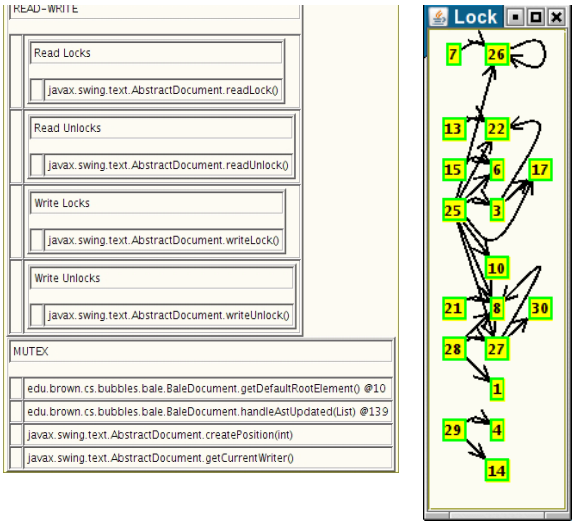


FIGURE 4. Tool tip showing lock types and the lock nesting display.

Bubbles uses synchronization extensively to control access to editors, to handle messages, and to provide background computations where appropriate.

Results are organized as a table. The first column identifies the lock, and the second column identifies the logical type of the lock. A plus at the end of the type indicates that multiple types were found for this lock. Details of all the lock types associated with this lock, for example, which synchronized regions comprised the lock, are provided through tool tips as seen in Figure 4.

The third and fourth columns of the table identify the lock more precisely by providing the set of synchronized regions (locations) and the classes of the objects that are locked for this logical lock. Cases where there are multiple locations or classes are also indicated by a plus sign. Again, tool tips provide details, giving the full method name, the full set of locations, and the full class name for all associated classes.

The next four columns provide information about how the lock was used in the sample run, helping the programmer identify which locks might be interesting to analyze further or might be problematic. The first two of these columns show the number of times the lock was used and the total delay encountered on entering the lock's synchronized regions. The next two columns show the number of times wait was called for the lock and the total amount of time the lock actually waited for notification.

The final column lets the user select locks to be monitored in a future run. Below the table are buttons to help with this selection, to save it, and to start a detailed analysis of selected locks (see below).

The Dependencies button on the bottom pops up another window showing how locks were nested during the sample run as seen in Figure 4. The boxes correspond to the different logical locks, identified by the ID from the initial table. An arc from one node to another indicates that a synchronized region for the second node was entered from within a synchronized region of the first. This information can be useful in identifying potential deadlocks caused by lock ordering conflicts, which will result in a cycle in the graph.

Detailed Lock Visualization. High-level visualization shows summary information on locks, but it does not provide all the details of locking behavior. In particular, users are often interested in the timeline of interactions between certain locks and the program's threads and between locks and other program objects. Unfortunately, the trace collected using the JVMTI is not sufficiently accurate and informative for collecting correct timeline information. Instead, we must instrument the program's code to closely monitor the selected locks.

To instrument and re-run the program, the user can select one or more locks for more detailed analysis and press the "Visualize" button. The instrumentation, generated using the ASM byte code package, generates a call at the pre-enter, enter, and exit of each synchronized region for the particular type of lock as well as for all calls to *wait()*, *notify()*, and *notifyAll()* associated with each lock of that type.

While instrumentation might seem more involved than the simpler tracing using the JVMTI, the fact that it is only being used on a selected set of locks and that the instrumentation is in Java and is relatively lightweight let us run the instrumented program with minimal overhead. (JVMTI instrumentation is relatively expensive.) However, there are several other problems related to instrumentation that require solution.

The first problem is that synchronized methods are handled by the JVM rather in the Java byte code. As such, it is difficult to instrument either the pre-enter (which can occur on any call of the method, even those that involve virtual calls or reflection), or the exit (which might occur because of an exception). To handle these situations, we rewrite synchronized methods as unsynchronized methods with a synchronized region containing their body. This region is synchronized either on *this* or on the class object as appropriate.

The second problem is that calls to *wait*() and *notify()* or *notifyAll()* related to the selected locks are not always obvious. While they generally occur inside the relevant synchronized region, they also can occur in methods called from within that synchronized region. One solution here would be to instrument all calls to these methods in the program, but this would result in the unreasonable overhead. Instead, we instrument all *wait()*, *notify()* and *notifyAll()* calls originating from classes that manipulate the lock of the given type

which we determine from our previous analysis. However, even in this case some of these calls can be still irrelevant to the lock. Filtering these irrelevant calls is done during the further analysis of the resulting trace.

The instrumentation generates a series of events that are parameterized by the thread, event type, and a unique id for the lock object. In addition, events that mark the attempted entry into a synchronized region include a unique identifier for the region and the class name of the lock object. Each thread maintains its own set of events.

The instrumentation monitor periodically (currently every 10 milliseconds) wakes up and sends recorded events to the analyzer. To minimize impact on the running program this is done in a separate thread. Moreover, each of the per-thread event sets is triple buffered, with one buffer is the current target for new events, one that might be the target for events that were in progress when the buffer switch occurred, and one from which events are sent to the analyzer.

The trace analyzer first takes the instrumentation output and generates the series of relevant events for the selected threads. It sorts the set of events by time, discards events that don't correspond to locks selected by the user (for example extraneous waits and notifies), and adds the region id and class name for the lock to each event.

This sequence of events is then passed to several processors that generate visualizations for lock and thread behavior using tools developed in [22]. In particular, two types of visualization are produced.

The first type of visualization is a *thread-centric diagram* that shows the timeline of interactions between threads and locks. Rows of the diagram correspond to threads, while horizontal axis represents time. Note that even if the trace data is accurate within one microsecond, each pixel in the diagram corresponds to a much larger time interval because of effects of scale. For example in the diagram depicting results of five minute run each pixel corresponds to the time interval of about 0.5 milliseconds.

Each thread in the diagram is depicted as a pipe. The inside of the pipe is colored to represent the thread state (one of *Locked*, *Blocked*, or *Waiting*), with the fill being proportional to the time that the thread spent in that state during the particular interval. The outside of the pipe is colored to reflect the lock object the thread interacts with. Notify-wait calls are shown as lines going from the source thread to the target thread, colored from red (notify end) to green (wait end). Detailed information about the diagram is given through tool tips.

This display is maintained dynamically as the program runs. The scroll bar on the bottom lets the user zoom in to any particularly time during the execution, while the bar on the right lets the user to traverse across a set of threads.

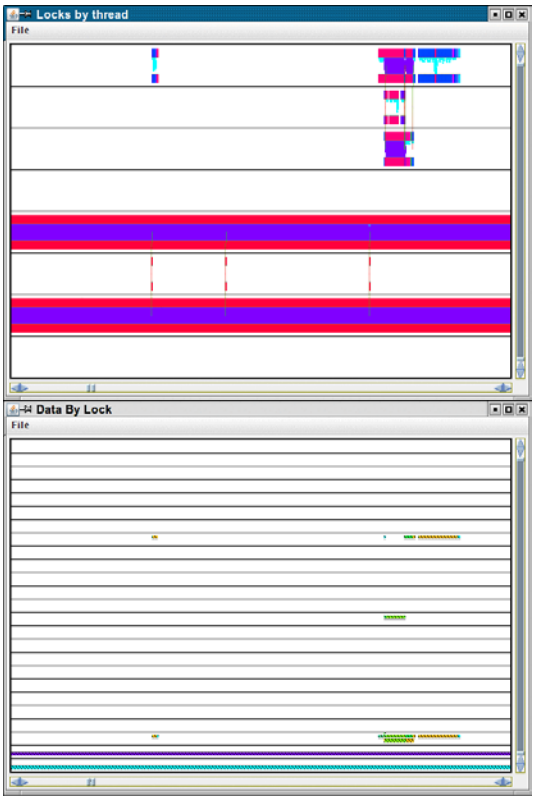


FIGURE 5. Visualization of locks by thread over time (top view) and view of individual lock behavior over time (bottom view).

A thread-centric visualization of the Code Bubbles IDE is shown at the top of Figure 5. The three threads toward the bottom represent a dual producer-consumer subsystem that operates two message queues (requests and replies) with a single reader. The tubes with purple insides are the consumer threads for the request and reply queues. These spend almost all of their time waiting. The thread in between is the reader thread. When a message or reply arrives, the producer puts it on the appropriate queue and wakes up the consumer threads. The threads at the top of the diagram represent read-write locks for editor regions that are being created by the program. The top thread is the Swing thread that handles the UI work. The other two threads are background processing threads that are charged with setting up the formatting for the various editors.

The second visualization is a *lock-centric diagram* that shows the timeline of locking activities for each lock. It can be seen at the bottom of Figure 5. Rows of the diagram correspond to locks, while horizontal axis represents time. Currently the diagram supports visualizing only two lock types: producer-consumer and read-write locks.

Each row of the diagram is divided by a gray line into top and bottom regions. In the moments when the

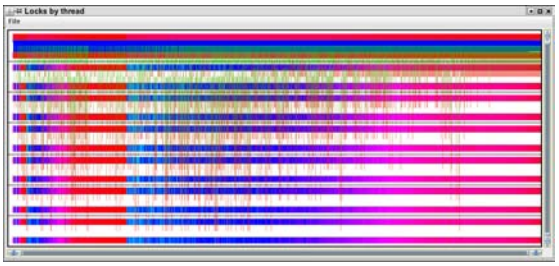


FIGURE 6. Thread-based visualization of n-body simulation

lock is in use, bars are drawn in these regions. The bottom bar represents threads waiting on the lock. The color of the bar denotes the thread waiting on the lock, while the texture of the bar represents the type of the lock (read-write or producer-consumer). The top bar displays lock-specific information. For the producer-consumer locks the top bar represents queued tasks, and for the read-write lock it shows if the lock was held for reading or writing.

Thread-centric and lock-centric visualizations are kept synchronized in time. The user can use the horizontal (time) scroll bar in either window to control both diagrams. This feature allows him to easily correlate thread and lock behavior by comparing diagrams.

7. Conclusion and Future Work

To assess the practical value of our tool we will have to make it more robust and widely available so others can use it on a wide variety of their own systems. In the meantime we have used it to analyze several of our own complex Java systems. Our experiments have shown the practicality and utility of the tool but also have pointed out a number of weaknesses and concerns. Addressing these concerns remains our main direction for the future work discussed below.

Reducing overhead. One particular concern was the amount of slowdown caused by the analysis and instrumentation of the user program. To determine the slow down, we experimented with the multithreaded program performing n-body gravity simulation with 10,000 objects. As it can be seen from the trace shown in Figure 6, the program does extensive locking: there are 24,000 contended locks in the program (groups of object are also considered to be a special type of object). Considering the large number and high frequency of locking, this program can be seen as a particularly bad scenario for the purpose of lock analysis.

Without the instrumentation, the running time of the program is 355 +/- 5 seconds (averaged over 5 runs). JVMTI monitoring of all contended locks increased the running time by about 69% to 600 +/- 10 seconds (averaged over 5 runs). The slowdown of the detailed trace, concentrating on the locks of interest (again 24,000 locks) to get the above figure was hardly noticeable, with the total time begin about 360 +/- 4 seconds (averaged over 5 runs). The lower overhead here shows the efficiency of our instrumentation code.

While some of these slowdowns are significant, we feel that improvements could be made by making the tracing more efficient. Currently the JVMTI uses text format for tracing and the trace size for the above run is about 55M. It could be significantly reduced by using a more efficient binary format for trace storage.

Accuracy of lock analysis. To evaluate the accuracy of lock analysis, we examined the lock types reported for a variety of systems we have implemented and matched the known lock types to the reported types. The result was that, for a long enough initial run, the program was able to correctly identify the logical types of all the locks in the systems. The analysis accuracy is heavily dependent on having a long enough initial run to ensure that all relevant locks are exercised. Locks that were not used or that were rarely used during the initial run are not reported or reported improperly. For example, complex locks that were not used extensively were sometimes reported as mutexes.

To some extent this problem is alleviated by merging of lock instances. Merging provides more examples of the lock usage and hence more opportunities to find both positive and negative examples of a particular logical lock type. But using a longer initial run remains our mainstay to ensure accuracy.

Improving visualizations. Current graphical visualizations provide either a high-level overview of the system's locking behavior, or a low-level analysis, but not both. The principle problem is one of scale. Locking typically occurs at the microsecond intervals, and even a relatively short run of a long-running system, say 10 minutes, produces tremendous amount of information to be visualized.

When all this data is visualized in the single window, each row of pixel can represents up to 10,000 events. Such visualization provides high-level view of the system's behavior, but finer details might be lost. As a result, potential problems and interesting patterns that occur on the scale of individual events are hidden from the user. The scroll bars let one zoom in, but once the user observes results at the ten-millisecond scale, he tends to lose a high-level context.

One solution would be a multiscale display, possibly one that does some type of fish-eye display showing details while still providing the overview. Another alternative is automatic detection of relevant events and highlighting them in visualization.

The large amount of data also tends to restrict the responsiveness of the displays. The system is still quite usable displaying 10 minutes worth of data. However, attempting to display 10 hours or 10 days is currently infeasible. This can be addressed by retooling our data store to use precomputed groupings.

Supporting different lock types. Our tool analyzes Java code at the level of synchronization primitives. However, there exist other ways to implement locks in Java and we are working on detecting those. First, newer versions of Java include built-in implementations of several types of logical locks in *java.util.concurrent*. These are not reported through JVMTI and would have to be detected using other techniques such as code instrumentation. Second, we are looking at identifying the use of files as locks, where the Java program attempts to create a unique file or directory and waits or sleeps if the create fails. Third, we hope to identify message send/receive pairs where the receive might be delayed.

Finally, we want to extend our tool to provide a better analysis of potential deadlocks, using techniques such as [1] where static dependencies and dynamic information are used to show actual and potential problems.

The code for our system is available as the dylock in ftp://ftp.cs.brown.edu/u/spr/dyvise.tar.gz.

Acknowledgements. This work was done with support from the National Science Foundation through grants CCR-1012056 and support from Microsoft.

8. References

1. Rahul Agarwal and Scott D. Stoller, "Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables," *PADTAD-IV*, pp. 51-59 (July 2006).

2. Cyrille Artho, Klaus Havelund, and Shinichi Honiden, "Visualization of concurrent program executions," *COMPSAC 2007*, pp. 541-546 (2007).

3. Isabelle Attali, Denis Caromel, and Marjorie Russo, "Graphical visualization of Java objects, threads, and locks," *IEEE Distributed Systems online* Vol. **2**(1) pp. 1-35 (2001).

4. Andrew Bragdon, Steven P. Reiss, Robert Zeleznik, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola, Jr., "Code bubbles: rethinking the user interface paradigm of integrated development environments," *ICSE 2010*, pp. 455-464 (2010).

5. Mandy Chung, "Using JConsole to monitor appliocations," *Sun Microsystems*, *http://java.sun.com/developer/techicalArticals/j2se/jconsole.html*, (December 2004).

6. Raja Das, Zhi Gan, Yao Qi, and Zhi Da Luo, "Multi-thread run-time analysis tool for Java," *http://www.alphaworks.ibm.com/tech/mtrat*, (2009).

7. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran, "Goldilocks: a race and transaction-aware Java runtime," *PLDI 2007*, pp. 245-255 (2007).

8. Dawson Engler and Ken Ashcraft, "RacerX: effective, static detection of race conditions and deadlocks," *SOSP 2003*, pp. 237-252 (October 2003).

9. John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk, "Effective data-race detection for the kernel," *OSDI 2010*, (2010).

10. Cormac Flanagan and Stephen Freund, "FastTrack: efficient and precise dynamic race detection," *PLDI 2009*, (2009).

11. Klaus Havelund, "Using runtime analysis to guide model checking of Java programs," *kuragmcjp*, pp. 245-264 (August 2000).

12. C. A. R. Hoare, "Monitors: an operating system structuring concept," *CACM* Vol. **17**(10) pp. 549-557 ().

13. David Hovemeyer and William Pugh, "Finding bugs is easy," *OOPSLA 2004 Companion*, pp. 132-136 (2004).

14. Ayal Itzkovitz, Assaf Schuster, and Oren Zeev-Ben-Mordechai, "Towards integration of data race detection in DSM systems," *Journal of Parallel and Distributed Computing* Vol. **59**(2) pp. 180-203 (1999).

15. Jacques Chassin de Kergommeaux and Benhur de Oliveira Stein, "Paje: an extensible environment for visualizing multi-threaded program executions," *Proc. of 6th International Euro-Par Conference, Lecture Notes in Computer Science* Vol. **1900** pp. 133-140 Springer-Verlag, (2000).

16. Byung-Chul Kim, Sang-Woo Jun, Dae Joon Hwang, and Yong-Kee Jun, "Visualizing potential deadlocks in multithreaded programs," *ICPCT 2009*, pp. 321-330 (2009).

17. Katharina Mehner, "JaVis: a UML-based visualization and debugging environment for concurrent Java programs," *Software Visualization: Lecture Notes in Computer Science* Vol. **2269** pp. 643-64 (2002).

18. Sun Microsystems, "JVM Tool Interface," *http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/jvmti.html*, (2004).

19. Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay, "Effective static deadlock detection," *ICSE 2009*, pp. 386-396 (May 2009).

20. Eli Pozniansky and Assaf Schuster, "MultiRace: efficient on-the-fly data race detection in multithreaded C++ programs," *Concurrency and Computation: Practice and Experience* Vol. **19**(3) pp. 327-340 (2007).

21. Steven P. Reiss, "JIVE: visualizing Java in action," *Proc. ICSE 2003*, pp. 820-821 (May 2003).

22. Steven P. Reiss, "Controlled dynamic performance analysis," *Proc. 2nd Intl. Workshop on Software and Performance*, (June 2008).

23. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson, "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. on Computer Systems* Vol. **15**(4) pp. 391-411 (November 1997).

24. Kyle Wheeler and Douglas Thain, *Journal of Concurrency and Computation: Practice and Experience* Vol. **22**(1) pp. 45-67 (2009).

25. Amy Williams, William Thies, and Michael D. Ernst, "Static deadlock detection for Java libraries," *ECOOP 2005*, pp. 602-629 (2005).