Synchronization During Java VM Initialization and Termination

Mathias Ricken and Robert Cartwright
Dept. of Computer Science
Rice University
Houston, TX 77005
mgricken@rice.edu, cork@rice.edu

ABSTRACT

The Java language supports concurrent programming using threads. Since thread scheduling is non-deterministic and a thread's time slice may end at any time, access to shared data has to be synchronized to prevent an unintended sequence of reads and writes that might corrupt data. Java virtual machines (VMs) support synchronization using monitors. With a tool developed to record the order of synchronization operations involving monitors, a large number of these costly operations were discovered during VM initialization and termination, even though the application seems strictly single-threaded at these times.

1. INTRODUCTION

As part of a larger project for deterministic unit testing of concurrent programs, we implemented a record-and-replay facility for Java programs that is independent of the VM and transparent to the program. Assuming shared data is protected by a consistent locking protocol and identical input data, an application's execution can be replayed by recording and replaying the schedule of synchronization operations, or "synchronization points" only; the actual thread scheduling on an instruction level is irrelevant.

In Java, these synchronization points mark instances of `monitorenter` and `monitorexit` instructions, as well as several other operations, such as `Object.wait` and `Thread.sleep`. Our application currently only records the order in which `monitorenter` and `monitorexit` instructions are executed; other synchronization points are not yet processed.

During our tests of recording an application's schedule, we were surprised by the large number of synchronization points that were present even in empty or very short programs. In order to understand the relevance of these synchronization points to later phases of the concurrent unit testing project, we began a more detailed analysis.

Section 2 briefly describes our method of transparently recording the sequence of synchronization points in arbitrary programs without modifying the underlying VM.

Section 3 provides an analysis of the abundant synchronization points before the user program begins to execute in its `main` method, and section 4 discusses the synchronization points present during VM termination, after the user program has finished executing.

The conclusion in section 5 considers possible improvements and offers directions for future work.

2. RECORDING SYNCHRONIZATION POINTS

When a unit test is executed, we can assume the input data – and thus the path through the program – is identical during every execution. Thread scheduling therefore is the only other factor that may influence program behavior, and since thread scheduling is non-deterministic, a successful unit test is rendered meaningless. To return to the significant results of single-threaded unit testing, concurrent unit tests have to be executed using all possible schedules and are considered successful if and only if the test was successful with each schedule. Recording the schedule of one execution is a first step towards generating and executing all possible schedules.

If all shared variables are consistently protected by locks, then two threads cannot influence each other except through shared but protected data, which can only be accessed between synchronization points comprised of `monitorenter` and `monitorexit` instructions. Recording the order in which these synchronization points occur therefore is sufficient to capture all significant aspects of the schedule [1]. Situations in which a thread voluntarily relinquishes and regains control of the processor, e.g. by calling and returning from `Object.wait` or `Thread.sleep`, also have to be recorded.

We record these synchronization points by modifying ("instrumenting") the Java bytecode of all classes before it is executed. This method has several advantages over alternative approaches:

- By instrumenting the bytecode, no changes to the actual Java source code are necessary. The application does not even have to be rebuilt to enable or disable schedule recording.
- By using a custom class loader and performing the instrumentation on-the-fly, the schedule can even

```
public class EmptyTest {
   public static void main(String[] args) {
   }
}
```

Program 1: Empty Program

```
public class OneSyncBlockTest {
   public static void main(String[] args) {
      synchronized(OneSyncBlockTest.class) {
      }
   }
}
```

Program 2: Single Synchronized Block

be recorded if classes are loaded dynamically and no source code is available.

- The recording is created independent of the VM executing the program and without requiring any changes to the VM source code.
- The modifications done by the instrumentation are implemented using regular Java or Java bytecode, so the VM may perform a just-in-time (JIT) compilation or other optimizations as it desires. It is not necessary to resort to bytecode interpretation.

When classes are loaded, their bytecode is scanned for occurrences of `monitorenter` and `monitorexit` instructions. If such an instruction is found, a call to a method inside the recorder is added before the `monitorenter` or after the `monitorexit` instruction. As parameters, the type of the synchronization point and the identity of the thread executing it are passed to the recorder, as well as additional information about the location, if desired. All of these pieces of information are represented using primitive data (`longs`) so as to not create additional object instances and complicate garbage collection.

It is important to note that the method performing the recording has to be synchronized itself; otherwise, several threads were allowed to record events simultaneously and potentially corrupt the data. However, the synchronization of this method may *not* be recorded in the schedule, as the method is not present in the original, unmodified program. The same holds true for any code added during the instrumentation that requires synchronization.

The recorder application executes in a second VM (master VM) that is connected to the user program's VM (slave VM) using the Java Debug Interface (JDI). Therefore, nothing in the recorder application can influence the behavior and the thread scheduling of the user program. Unfortunately, program execution in the slave VM has to be suspended whenever the two VMs communicate, and relaying the information about every synchronization point to the master VM immediately severely degrades performance. The information about the synchronization points is therefore stored in a buffer (`long[]`) in the slave VM and transferred to the master VM as one large piece whenever the buffer is full.

The schedule of a user program is not solely controlled by the synchronization points in the user program, synchronization done in API code called by the user program affects the schedule as well. Therefore, synchronization points in the Java API – most of which is found in the `rt.jar` file -- have to be recorded as well. To avoid instrumenting the same classes time and again, we instrument the `rt.jar` file and other parts of the Java API using an off-line tool and create an alternate version of the Java API that already includes all the necessary modifications. This alternate version is then placed on the Java boot classpath instead of the original API.

When we performed first tests on very small programs, such as Program 1 and Program 2, we were surprised that over 1,200 synchronization points were recorded, regardless of the small size of the programs. We decided that additional information about the location of these synchronization points would be useful in determining whether our recorder functioned properly, and maybe even at later points in the project. However, all information about synchronization points was stored in a buffer of `longs`, so it was necessary to uniquely represent both class and method names using just `longs`.

Our instrumentation tool achieves this by maintaining a method database while it is instrumenting classes and assigning each class a serial number. The method is specified by its position in the class file. Whenever a synchronization point is recorded, a class-method pair of `longs` is recorded along with the synchronization point type and thread. Given the method database and these pairs, it is possible to retrieve the names of the class and method in which a synchronization point occurred. If the location needs to be specified even more precisely, the program counter (PC) could be added as fifth piece of information.

When the schedule of a program is to be recorded, the user starts the recorder application in the master VM and specifies the user program using command line arguments. Using JDI, the master VM then creates the slave VM, establishes a set of events the master VM is interested in, and initiates the execution of the user program. These events are used to temporarily suspend execution in the slave VM and process data at the start of the slave VM and its death, when the `main` method is entered and exited, and when the buffer is full and needs to be transferred and emptied. The variety of JDI events is actually sufficient to perform schedule recording without instrumentation and

just with JDI, but the performance loss of such an approach is just unacceptable. An instrumented program with a buffer and limited communication executes an order of magnitude faster, as Figure 1 and Table 1 indicate.

3. SYNCHRONIZATION POINTS BEFORE THE `main` METHOD

The recorded schedules [2] for all programs begin with three threads in operation: "main" (0), "Reference Handler" (1), and "Finalizer" (2). At least threads 1 and 2 appear to belong to the first 12 synchronization points recorded, which create three new threads: There are three occurrences of a pair of `monitorenter` and `monitorexit` instructions in the constructor of the `Thread` class, during which we assign a unique ID number (given in parentheses above) to each thread, and another pair in `Thread.nextThreadID`, in which the Java API performs a similar task. The ID assignment has to be synchronized to ensure that a thread will always be assigned the same ID under any given schedule. This, in essence, makes thread creation a synchronization point in its own right. While it may be possible to use Java's existing IDs, we decided to insert our own ID to maintain complete control. It is likely that these 12 synchronization points actually create the threads with IDs 1, 2, and 3, and that the "main" thread with ID 0 exists without being explicitly created. We currently do not know the name of the thread with ID 3.

A number of `monitorenter` and `monitorexit` pairs in `Hashtable.get` follow, as well as synchronization points in

- `sun.misc.Launcher$AppClassLoader.loadClass`
- `sun.misc.URLClassPath.getLoader`
- `java.util.Stack.pop and peek`
- `sun.misc.URLClassPath.getLoader`
- `java.net.URL.hashCode`
- `java.net.URLStreamHandler.getHostAddress`

The next 64 synchronization points are all `monitorenter` and `monitorexit` pairs in `StringBuffer.append(char)`, and this will be a reoccurring pattern during VM initialization. While we do not know the value of the character passed as parameter, we do know that one or more strings are being constructed here by appending characters, and each character creates two synchronization points.

After these `monitorenter` and `monitorexit` pairs in `StringBuffer.append(char)` there is another section of 12 synchronization points that create three threads, probably with IDs 4, 5, and 6. At this time we do not know the names of the threads with IDs 4 and 5; they seem to be rather short-lived. Thread 6, however, remains alive for the rest of the program execution and is named "Signal Dispatcher".

After these new threads are created, there are several more long sequences in `StringBuffer.append(char)`, and additional synchronization points in the aforementioned classes, as well as in

- `java.lang.Runtime.loadLibrary0`
- `java.io.ExpiringCache.get`
- `java.lang.ref.Finalizer.add`
- `java.util.zip.ZipFile.getEntry`
- `java.util.zip.ZipFile.getInputStream`
- `java.util.zip.ZipFile.getInputStream`
- `java.util.zip.Inflater.inflate`
- `java.util.zip.Inflater.finished`
- `java.util.zip.Inflater.needsDictionary`
- `java.util.zip.ZipFile$ZipFileInputStream.read`
- `java.util.zip.ZipFile$ZipFileInputStream.close`
- `java.util.zip.Inflater.setInput`
- `java.util.zip.ZipFile.releaseInflater`
- `java.net.URL.set`
- `sun.misc.Resource.cachedInputStream`
- `sun.net.www.protocol.file.Handler.openConnection`
- `java.io.ExpiringCache.put`
- `java.security.Permissions.add`
- `java.io.FilePermissionCollection.add`
- `java.security.BasicPermissionCollection.add`

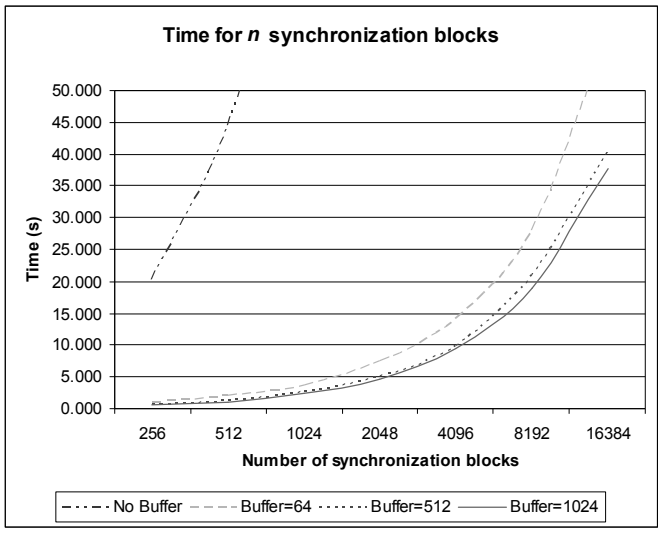

Figure 1: Performance With and Without Buffer

Number Sync. Blocks	Time (s) No Buffer	64 Elements	512 Elements	1024 Elements
256	20.422	1.000	0.625	0.532
512	44.547	1.938	1.250	1.079
1024	85.219	3.657	2.547	2.360
2048	166.062	7.359	5.016	4.593
4096	334.718	14.094	9.953	9.391
8192	741.606	28.015	21.032	19.031
16384	1473.516	58.516	40.297	37.766

Table 1: Performance With and Without Buffer

It is safe to assume that classes are loaded from jar files, which are just renamed zip files. The calls to `StringBuffer` probably generate the names of jar files by concatenating string constants with environment variables. Java's security model is being created and checked in the background to ensure file and class accesses do not violate security policies.

Interestingly, this is where the schedules for the empty program (Program 1) and the non-empty program (Program 2) begin to differ: After 1,280 recorded synchronization points, the empty program begins with VM termination. It does not even enter the `main` method. Apparently, the VM completely optimizes calls to empty methods away.

The non-empty program features several additional calls to `sun.misc.Launcher$AppClassLoader.loadClass`, probably to load the local classes of the user program, and then runs through the synchronization points in the `main` method exactly as expected: Program 2 has exactly one pair of `monitorenter` and `monitorexit` instructions.

It is conceivable that the names and contents of command line parameters change the number and sequence of synchronization points slightly. We have not investigated this issue yet.

In evaluating the existence of these synchronization points it is critical to remember that all of them were executed by the "main" thread. Several other threads were created and even started, but none of them performed any synchronization events that we were able to capture. If we assume we are aware of all synchronization, and that the initialization code of the VM is written in a correct and safe way that is free of possible race conditions, then all these synchronization points are unnecessary: In a single-threaded environment – or a multi-threaded environment without any shared data – no synchronization is necessary.

4. SYNCHRONIZATION POINTS AFTER THE `main` METHOD

After the user program terminates, which – although we have not experimented with this situation – should happen when all non-daemon threads have died, a thread with ID 9 and the fitting name "DestroyJavaVM" begins to run through a sequence of eight synchronization points.

Since the buffer only gets transmitted when it is full, we cannot read the contents of the last buffer before the VM dies unless the buffer coincidentally becomes full at the same time. The master VM can also query the buffer when a JDI event suspends the slave VM, but when the "VM Death" event is received, the slave VM has already died and the buffer contents are not available anymore. We therefore do not have very good information about the synchronization in "DestroyJavaVM" yet. We do know that it appears to create another thread and then perform some synchronization in `java.lang.Shutdown.shutdown` and `java.lang.Shutdown.sequence`.

Fortunately, there are ways to receive detailed information nonetheless: Once a synchronization point for thread 9 is encountered, the current contents of the buffer are transmitted, and then the buffer is deactivated. All remaining synchronization points are reported to the master VM immediately. That degrades performance, but during shutdown that is not important anymore.

The small programs we have used for this study do not show the effects of garbage collection, finalizers, and daemon threads, either. In addition to the implementation of the aforementioned improvements, further investigation in these areas is needed.

When additional synchronization operations such as `Object.wait` and `Thread.sleep`, but also thread creation and thread death, are recorded – if possible with extended information such as location and parameters – we will be able to get a better understanding of the threads that VM initialization and termination create internally.

5. CONCLUSION

The recording application developed for the concurrent unit testing project has shown that on-the-fly instrumentation of Java class files is a transparent and efficient method of gathering runtime information independently of the VM.

The first results gathered with this tool show a surprisingly large number of synchronization points even for empty or very small programs – over 1,200 just in the VM initialization and termination. These portions of program execution seem to be single-threaded; therefore, it should be possible to disable or remove synchronization and improve program performance.

Assuming VM initialization and termination truly are single-threaded, the unsynchronized `StringBuilder` class should be used as an alternative to `StringBuffer`. Generally, providing an unsynchronized version of common data structures has the potential to enhance runtime performance and simplify the concurrency aspects of a program.

The hundreds of synchronization points incurred by the unnecessary use of `StringBuffer` also underscore the need for the ability to ignore certain uncritical synchronization points when generating schedules for the concurrent unit testing project.

6. REFERENCES

[1] D.L. Bruening. Systematic testing of multithreaded Java programs. Master's thesis, MIT, 1999.

[2] Available from the author.