JOHANNES KEPLER
UNIVERSITY LINZ

Master's Thesis

Portable Tracing of Lock Contention in Java Applications

Student: Andreas Schörgenhumer

Institute for System Software

Dipl.-Ing. Peter Hofer

Phone.: +43 732 2468-4369

Fax: +43 732 2468-4345

peter.hofer@jku.at

Linz, 8/3/2016

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

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

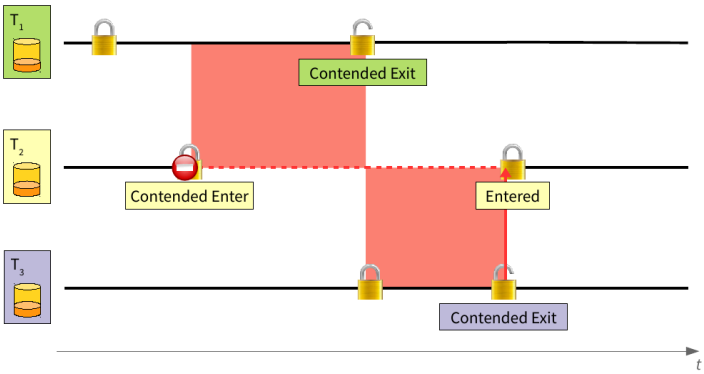

Figure 1: Example of three contending threads, where T_2 is first blocked by T_1 and then by T_3

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

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

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

Peter Hofer

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