

Impact on performance of using JVMTI events for tracking method calls in place of native methods generated via bytecode instrumentation

Filip Kliber

JVMTI¹ is a programming interface, which provides ways to inspect and control the behaviour of application running in Java virtual machine. Common use case of JVMTI is to have an Agent, which is a library written in native language, and target program running in JVM. Preferably during early stages of initialization of the VM, the agent is loaded into the VM and can register notification handlers for certain events happening in the VM.

One of the events present in JVMTI is **MethodEntry** (and corresponding **MethodExit**), which is triggered upon entry of any method in targeted Java program (including Java Class Library). The specification of JVMTI states the following:

Enabling method entry or exit events will significantly degrade performance on many platforms and is thus not advised for performance critical usage (such as profiling). Bytecode instrumentation should be used in these cases.

This experiment tries to measure the impact on performance of using **MethodEntry** and **MethodExit** events as opposed to bytecode instrumentation.

Terminology (simplified)

ClassPrepare(class) event is an event in the VM, that occurs when a **class** is fully loaded and prepared, but no code of that class has executed yet.

VMDeath() event is second important event in the VM, which happens when the VM terminates the execution and is the last event that will occur during the execution of the program.

ClassFileLoadHook(name, bytecode, new_bytecode) event is generated when the VM tries to load class file for class with given **name**. The Agent can set **new_bytecode** and the VM will load different bytecode for given class.

MethodEntry(method) and MethodExit(method, ret_value) events are events that occur when given the execution of **method** begins (or ends).

Difficulties

The main issue is with the bytecode instrumentation itself. The running VM informs the attached Agent of when it wants to load the class (**.class**) and provides an option to interchange the bytecode of class, that is being loaded. The problem is, that there exists no library for bytecode instrumentation in C or C++. There are many libraries in Java, that allow easy modification of existing bytecode. In order to resolve this problem, the Agent starts a different process, written in java, that acts as *instrumentation server*, and delegates the instrumentation process to that server.

¹Java Virtual Machine™ Tool Interface — <https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html>

Agents architectures

The experiment uses four different agents (libraries) written in C++. The structure of all agents is very similar.

When the agent is loaded, it performs needed initialization of itself and registers handlers to important events generated by the VM.

Agent `jvmti_none`

This is the core of all agents, the only thing it does is measuring the duration of “initialization of VM” (duration between attachment of the Agent and `ClassPrepare` event for class containing the `main` method) and the duration of the execution of the program (duration between `ClassPrepare` event for class containing the `main` method and `VMDeath` event).

Agent `jvmti_events`

Also registers both `MethodEntry` and `MethodExit` events (after the main class is prepared) and counts how many such events happen.

Agent `jvmti_bci`

Instead of registering `MethodEntry` and `MethodExit` events, it registers `ClassFileLoadHook` event and on occurrence of that event requests the BCI² server to add declaration of following methods:

```
private static native void __MethodEntry__();  
private static native void __MethodExit__(Object ret_value);
```

and inserts the call to `__MethodEntry__` method at the beginning of each method and call to `__MethodExit__` before every `return` statement.

Agent `jvmti_bci_info`

Is exactly the same as `jvmti_bci`, but handlers for `__MethodEntry__` and `__MethodExit__` also asks the JVM TI for the representation of currently executing method (retrieved via stack trace). This comparison is done because the JVM TI events supply this information by default, and it isn't cheap operation, as it will be seen later.

Both `jvmti_bci` and `jvmti_bci_info` have two variants — Instrumenting only classes corresponding to given benchmark, and instrumenting all classes.

It is important to note, that this description is *very* simplified and there are lot of technical details, that has to be treated very carefully (for example invocation of native method can call `ClassLoader.findNative`, which is method, that again invokes native call, which results with `StackOverflowError`).

Also, these two approaches (events and bytecode instrumentation) are not equivalent and both have their pros and cons. Here are few of them:

- With instrumentation, it is easy to specify which method entries and exists should be tracked. This is not possible via events.
- Method exit via instrumentation is not capable of covering method exit that happens due to exception being thrown. On the other hand, Method exit event is fired even in this case and the agent is noted accordingly.

²stands for ByteCode Instrumentation

Benchmarks

In order to test the difference between these agents, following benchmarks were prepared:

VirtualCalls

Is a simple class, that specifies **Base** interface with **int method()** method and provides 10 different implementations. Once the execution starts, pool of size of 100 is populated with random implementations of **Base**. Then a implementation is selected at random and the method is invoked. This is repeated million times (10^6). Then the pool is reseeded and the process repeats, 10 times.

Zip

Is a simple application, that extracts the content of file compressed via *zip* and then repacks it back. It uses *ZeroTurnaround ZIP library*³. The archive used for testing is ~20MiB large archive containing the source code of my bachelor thesis.

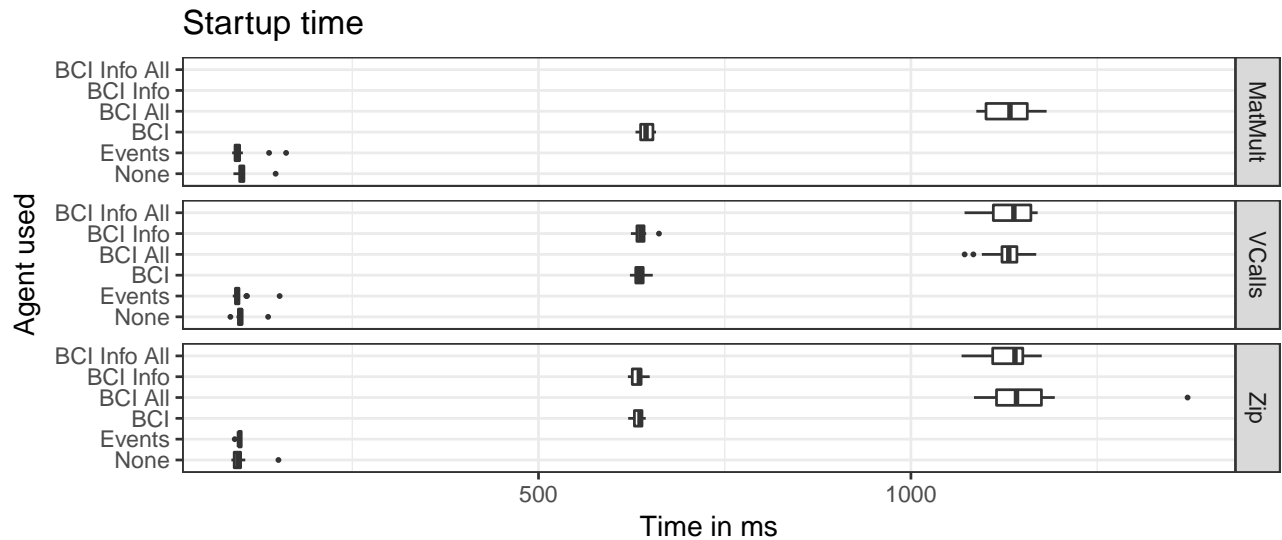
MatrixMultiplication

Uses *EJML library*⁴ to multiply two 256×256 matrices of random double values and computes sum of all elements of the result. Even though the matrices are quite small, performing `jvmti_bci_info` takes very long time (minutes, as opposed to milliseconds without any agent), so they are not performed. Analysis of `jvmti_bci_info` is thus dependant only on previous two benchmarks.

All benchmarks were run with all agents (sequentially) and this was repeated multiple times to minimize random noise during the measurements.

The experiment was conducted on my laptop (Windows 7, Intel Core i7-5500U CPU @ 2.4GHz, 8GB RAM) with no additional programs running.

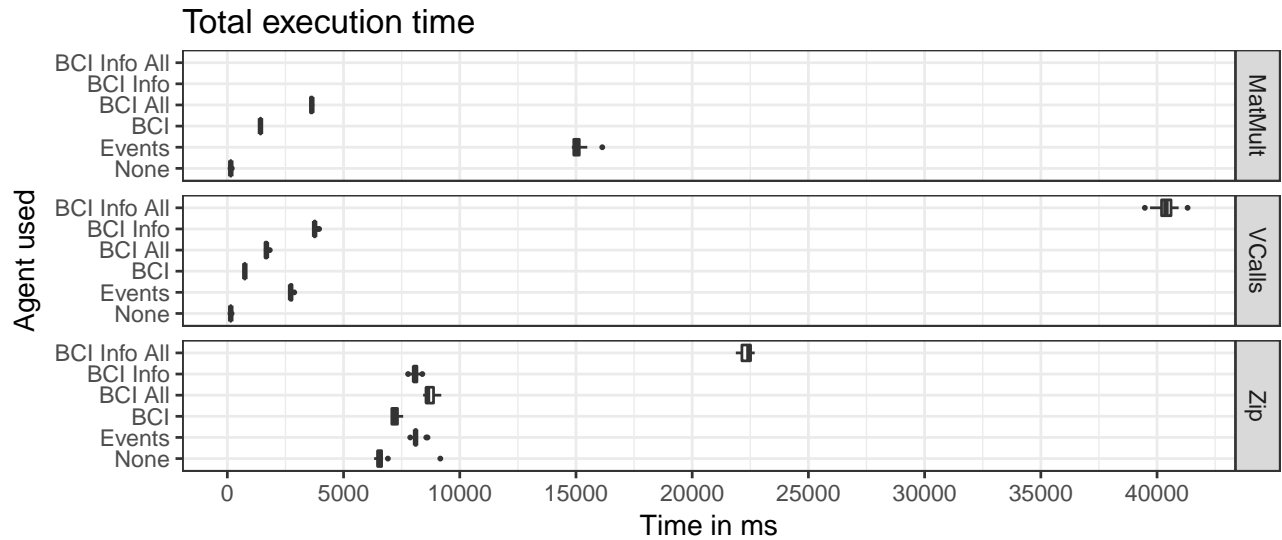
Results



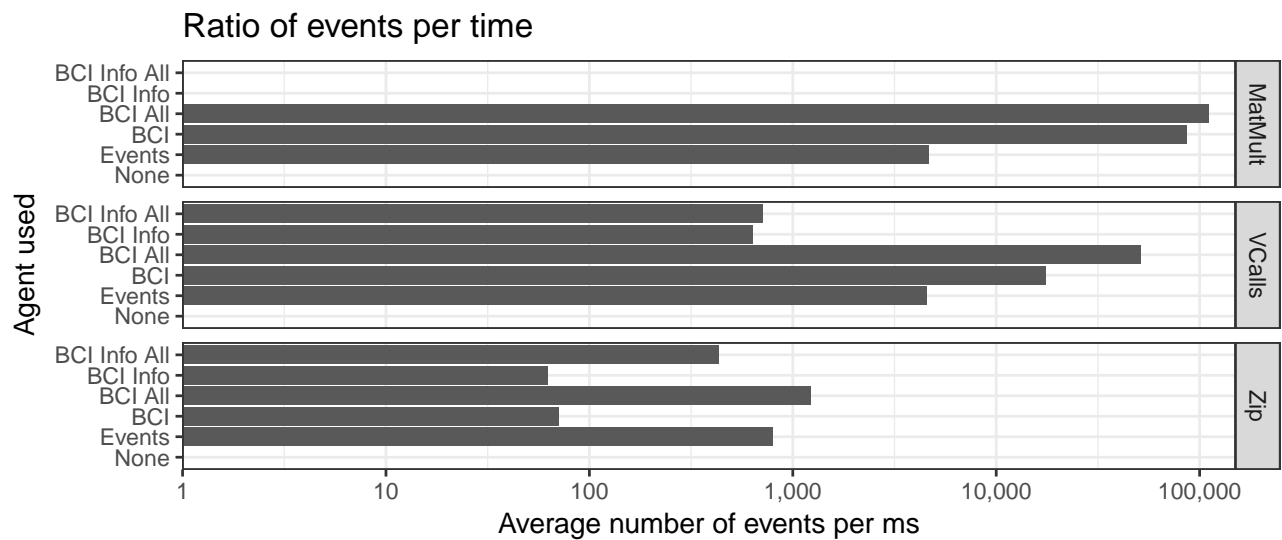
Above is the graph showing startup times of each agent. It proves, that configuration of **events** doesn't add to the startup time. Instrumenting classes however does, as is seen for **bci*** agents, and the more classes are instrumented, the longer it takes.

³<https://github.com/zeroturnaround/zt-zip>

⁴<http://ejml.org/wiki/index.php>



This graph shows total execution time for each agent. This gives us an idea of how expensive `MethodEntry` and `MethodExit` events are and how expensive is it to ask JVMTI for call stack to retrieve internal representation of method, that is being currently executed.



Another interesting question would be how fast are events processed by different agents during the execution. This answer to this can be seen on the graph above. Zip benchmark has significantly lower ratio, because it does a lot of I/O operations. This is also reasoning behind the difference of ratios between BCI and BCI All.

Reasoning

One of the possible *culprits*, that comes to mind when dealing with performance fluctuations in Java is **JIT**. Java VM can be configured to output information about method being JIT compiled⁵. This is part of the output from BCI for `VirtualCalls` benchmark compiled with specified configuration.

⁵-XX:+PrintCompilation

```

...
1199 67    n 0    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl3::__MethodEntryHook__ (native) (static)
1199 68    n 0    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl3::__MethodExitHook__ (native) (static)
1202 69    n 0    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl1::__MethodEntryHook__ (native) (static)
1202 70    n 0    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl1::__MethodExitHook__ (native) (static)
...
1256 100   4    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl3::method (24 bytes)
1257 71    3    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl3::method (24 bytes)    made not entrant
1260 101   4    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl0::method (24 bytes)
1261 80    3    cz.cuni.mff.d3s.benchmarks.VirtualCalls$Impl0::method (24 bytes)    made not entrant
...

```

The output of `BCI Info` is pretty much the same, but agent `Events` is not even attempting to JIT compile anything, which means that the program is being interpreted for the whole time, which is the reasoning behind such lower performance.

It is still unknown why methods provided by JVMTI takes that long to execute, but one possible (partial) solution to that would be to let the instrumentation pass reasonable arguments via the hook (name, class, parameters, modifiers) directly, because at the point of instrumentation, they are compile time constants. This approach is not tested.