Multilevel Analysis of Java Virtual Machine

Progress Report Meeting

May 10, 2018

Houssem Daoud       Michel Dagenais

École Polytechnique de Montréal
Laboratoire DORSAL
Agenda

- Introduction
- Hotspot Virtual Machine
- Proposed solution
- Demo
- Conclusion
Introduction

The complexity of modern applications is constantly increasing:

- Multithreading
- Interdependent components
- Virtual runtime environment, containers, etc.

- Analyzing the performance of such applications is very challenging.
- A performance degradation can be caused by the application itself or by the environment on which it is being executed.
Existing performance analysis tools usually collect runtime information from the userspace, which offers a very limited visibility of the system.

- Disk operation
- Network operation
- Waiting for CPU, etc.
Our goal is to provide a performance analysis framework that offers a full visibility of the system, from the application down to the operating system.

We can achieve that by:

- Tracing the different layers of the system
- Synchronizing the traces using a unified clock
- Analyzing the trace events and providing a comprehensive visualization system
Introduction

- We used that methodology to create an advanced performance analysis tool for Java Applications.
- The proposed tool covers the Java application, the runtime environment and the operating system.
- By bridging the gap between userspace and kernel space traces, we are able to provide very precise information compared to other existing tools.
Agenda

- Introduction
- Hotspot Virtual Machine
- Proposed solution
- Demo
- Conclusion
Hotspot Virtual Machine

Java Programming language:

- Platform independent
- Object Oriented
- Dynamic
- Interpreted/Compiled

Hotspot VM features:

- Class loader
- Java bytecode interpreter
- JIT compiler
- Automatic memory management mechanisms
- Several garbage collectors
Hotspot Virtual Machine

- Java programs are first executed by the Java Interpreter
- Hot methods are sent to the JIT compiler for further optimization

JIT compiler
- C1 provides fast compilation.
- C2 is more aggressive and generates more optimized code
- Methods are first compiled by C1; as they become hot, they are recompiled by C2.
  → Tiered Compilation
Hotspot Virtual Machine

- Each Java thread (Thread) is mapped to a native operating system thread (OSThread)

- The main types of threads are:
  - Java threads
  - Compiler threads
  - GC threads
  - VM threads
Agenda

- Introduction
- Hotspot Virtual Machine
- Proposed solution
- Demo
- Conclusion
Proposed solution

Architecture

- Profiling Agent
- Java Application
- Userspace Tracer
- JVM
- Kernel Tracer
- Operating System
- Trace Analyzer
- Visualization system

Trace
Proposed solution

Tracing

We use Lttng-UST to collect information from Hotspot virtual machine

<table>
<thead>
<tr>
<th>Java Thread</th>
<th>Thread_start</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Thread_stop</td>
</tr>
<tr>
<td></td>
<td>Thread_sleep_start</td>
</tr>
<tr>
<td></td>
<td>Thread_sleep_stop</td>
</tr>
<tr>
<td></td>
<td>Thread_status</td>
</tr>
<tr>
<td>Compiler threads</td>
<td>method_compile_begin</td>
</tr>
<tr>
<td></td>
<td>method_compile_end</td>
</tr>
<tr>
<td>GC threads</td>
<td>gctaskthread_start</td>
</tr>
<tr>
<td></td>
<td>gctaskthread_end</td>
</tr>
<tr>
<td></td>
<td>gctask_start</td>
</tr>
<tr>
<td></td>
<td>gctask_end</td>
</tr>
<tr>
<td></td>
<td>report_gc_start</td>
</tr>
<tr>
<td></td>
<td>report_gc_end</td>
</tr>
<tr>
<td>VM threads</td>
<td>vmthread_start</td>
</tr>
<tr>
<td></td>
<td>vmthread_stop</td>
</tr>
<tr>
<td></td>
<td>vmops_begin</td>
</tr>
<tr>
<td></td>
<td>vmops_end</td>
</tr>
</tbody>
</table>
**Proposed solution**

**Tracing**

<table>
<thead>
<tr>
<th>Memory usage</th>
<th>object_alloc</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>alloc_new_tlab</td>
</tr>
<tr>
<td></td>
<td>alloc_outside_tlab</td>
</tr>
<tr>
<td></td>
<td>alloc_requiring_gc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Thread contention</th>
<th>monitor_wait</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>monitor_waited</td>
</tr>
<tr>
<td></td>
<td>monitor_notify</td>
</tr>
<tr>
<td></td>
<td>monitor_notifyAll</td>
</tr>
<tr>
<td></td>
<td>contended_enter</td>
</tr>
<tr>
<td></td>
<td>contended_entered</td>
</tr>
<tr>
<td></td>
<td>contended_exit</td>
</tr>
</tbody>
</table>

Kernel events are also collected to provide low-level information about the system on which the application is executed.
Proposed solution

Profiling

- We developed a JVMTI agent that generates periodic profiling events.
- The callstack is captured by integrating libunwind into Lttng-UST
- `PreserveFramePointer` option is needed to walk Java stacks

```
java -agentpath:liblttng-profile.so -XX:+PreserveFramePointer Main
```
Proposed solution

Threads View
Proposed solution

Threads View
Proposed solution

**CPU View**

- This view shows on which CPU each Java thread is running.

The view shows that a GC thread is being preempted by other processes.
Proposed solution

Lock Contention

```java
Integer monitor = new Integer(0);

synchronized(monitor) {
    //Critical section
}
```
Proposed solution

JIT Statistics

<table>
<thead>
<tr>
<th>Duration</th>
<th>Name</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>135,730</td>
<td>getDefaultPort</td>
<td>name=java/net/URLStreamHandler</td>
</tr>
<tr>
<td>137,736</td>
<td>isProcessClassRecursioni</td>
<td>name=org/eclipse/osgi/internal/hooks</td>
</tr>
<tr>
<td>141,065</td>
<td>isProcessClassRecursioni</td>
<td>name=org/eclipse/osgi/internal/weavi</td>
</tr>
<tr>
<td>141,936</td>
<td>isProcessClassRecursioni</td>
<td>name=org/eclipse/osgi/internal/hooks</td>
</tr>
<tr>
<td>142,264</td>
<td>removeEldestEntry</td>
<td>name=java/util/LinkedHashMap</td>
</tr>
<tr>
<td>144,766</td>
<td>get</td>
<td>name=java/lang/ref/Reference</td>
</tr>
<tr>
<td>146,830</td>
<td>isNullSource</td>
<td>name=org/eclipse/osgi/internal/loader</td>
</tr>
<tr>
<td>151,881</td>
<td>isLegalReplacement</td>
<td>name=sun/nio/cs/ISO_8859_1$Encod</td>
</tr>
<tr>
<td>151,938</td>
<td>if_R</td>
<td>name=sun/security/provider/SHA2</td>
</tr>
</tbody>
</table>

![Graph showing JIT statistics](image-url)
Proposed solution

Memory usage
Proposed solution

Profiler
Demo
Conclusion

- By tracing the different layers of the Java virtual machine and the Linux Kernel, we were able to provide an advanced Java performance analysis framework that covers the whole software stack.

- The same methodology can be used to analyze other complex applications.

- As a next step, we are planning to use dynamic tracing to collect the data instead of inserting static tracepoints.

houssem.daoud@polymtl.ca