Multilevel Analysis of Java Virtual Machine

Progress Report Meeting

May 10, 2018

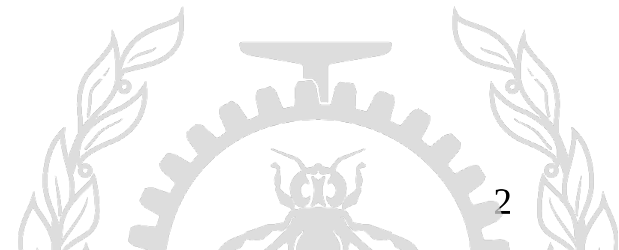
Houssem Daoud **Michel Dagenais**

École Polytechnique de Montréal

Laboratoire **DORSAL**

Agenda

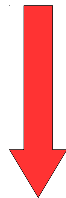
- **Introduction**
- Hotspot Virtual Machine
- Proposed solution
- Demo
- Conclusion



Introduction

The complexity of modern applications is constantly increasing:

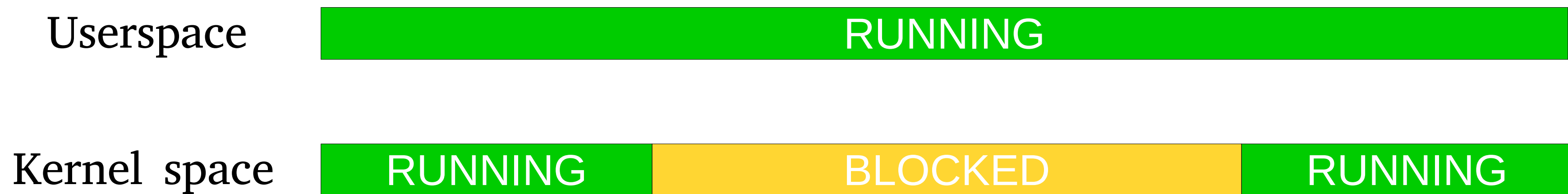
- Multithreading
- Interdependent components
- Virtual runtime environment, containers, etc.



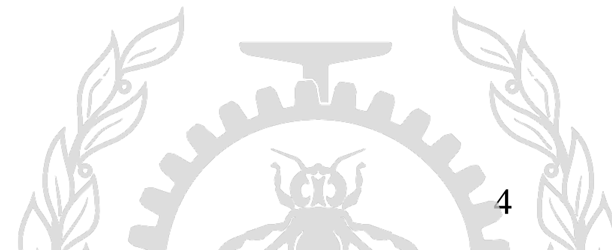
- Analyzing the performance of such applications is very challenging.
- A performance degradation can be caused by the application itself or by the environment on which it is being executed.

Introduction

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.

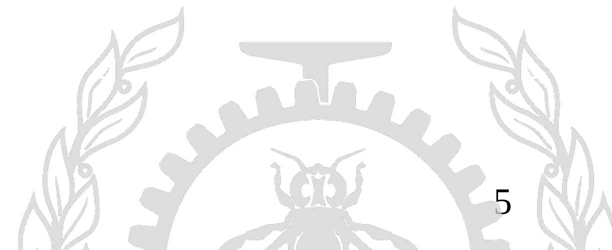


Introduction

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.

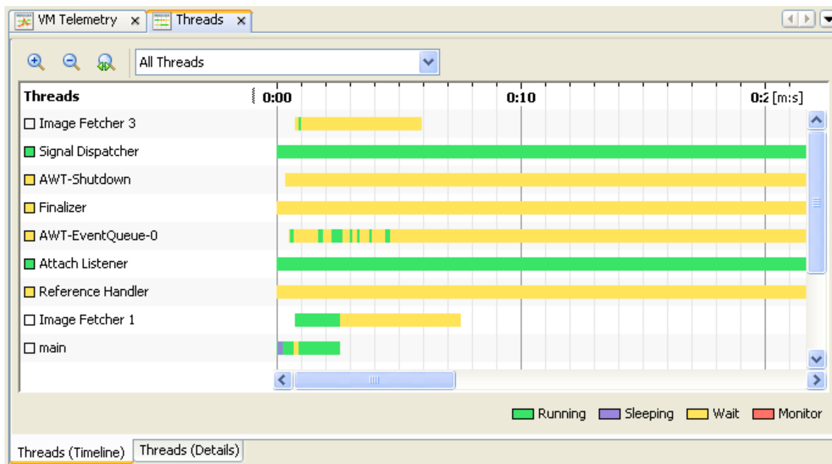
We can achieve that by:

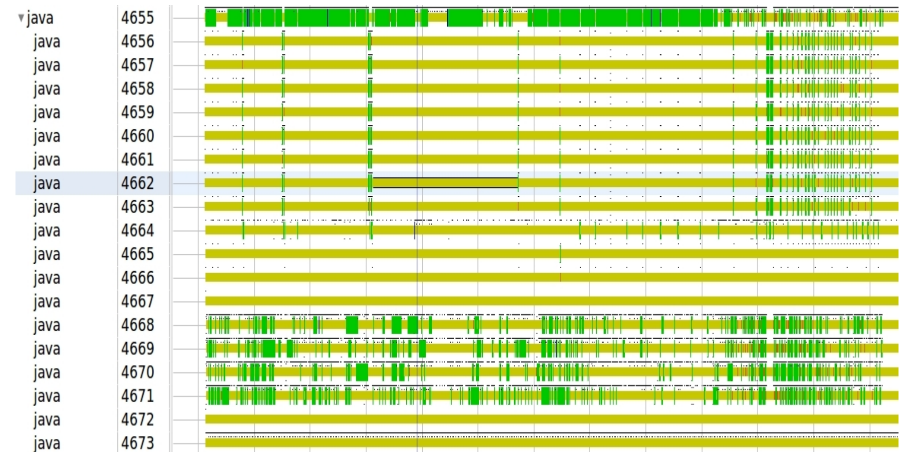
- Tracing the different layers of the system
- Synchronizing the traces using a unified clock
- Analyzing the trace events and providing a comprehensive visualization system



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.

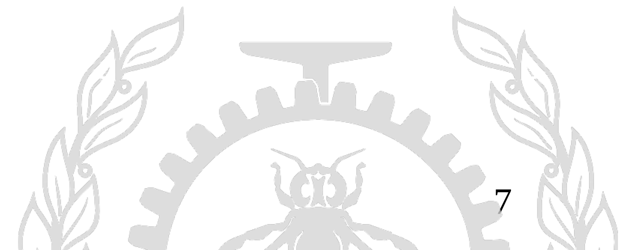




- By bridging the gap between userspace and kernel space traces, we are able to provide very precise information compared to other existing tools

Agenda

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



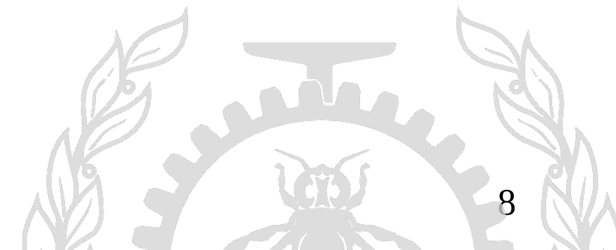
Hotspot Virtual Machine

Java Programming language:

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

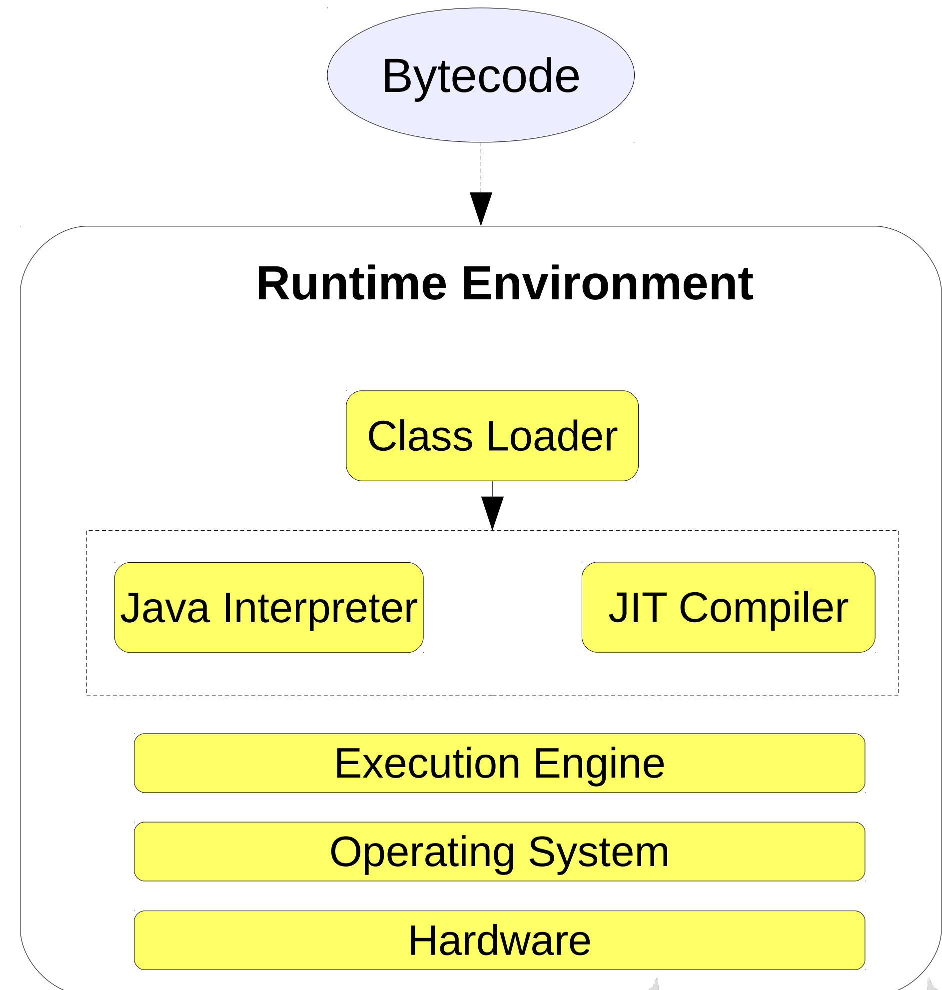
Hotspot VM features:

- 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



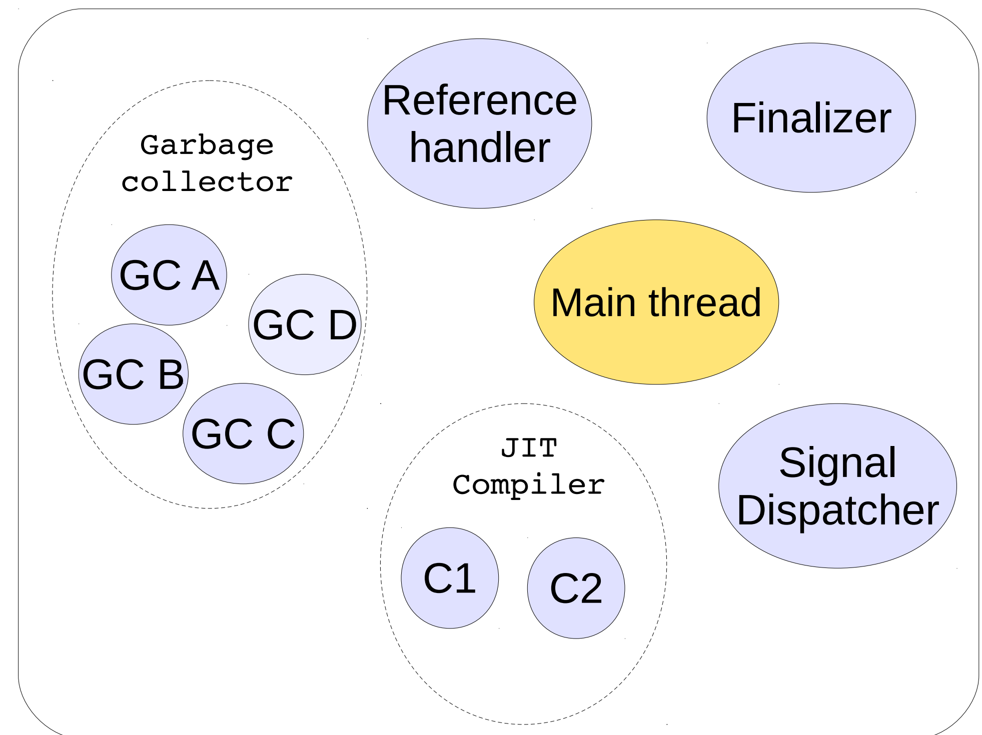
JIT compiler

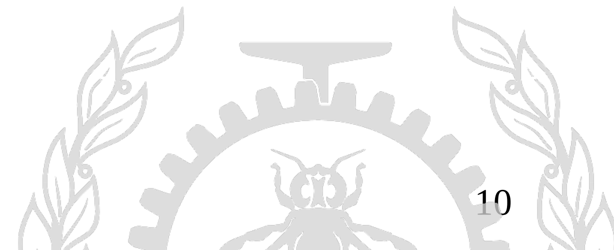
- C1 provides fast compilation.
- C2 is more aggressive and generates more optimized code
- Methods are first compiled by C1; as they become hot, they are recompiled by C2.
  → **Tiered Compilation**

Hotspot Virtual Machine

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

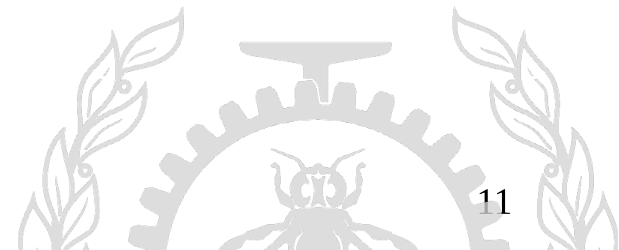
- The main types of threads are:
  - Java threads
  - Compiler threads
  - GC threads
  - VM threads





Agenda

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



Proposed solution

Architecture

Profiling Agent

Java Application

Userspace Tracer

JVM

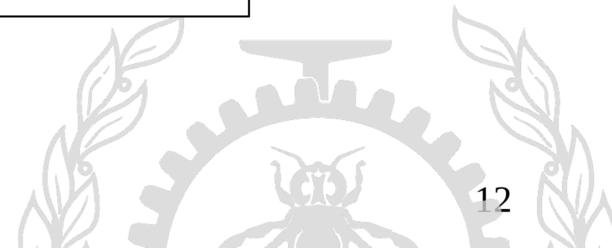
Trace

Kernel Tracer

Operating System

Trace Analyzer

Visualization system



Proposed solution

Tracing

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

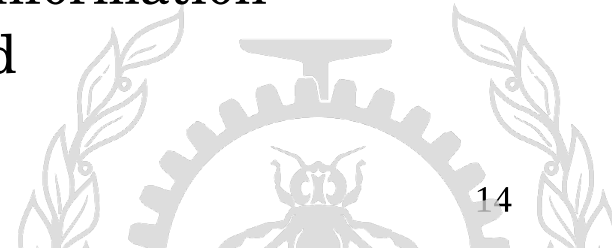
Java Thread	Thread_start Thread_stop Thread_sleep_start Thread_sleep_stop Thread_status
Compiler threads	method_compile_begin method_compile_end
GC threads	gctaskthread_start gctaskthread_end gctask_start gctask_end report_gc_start report_gc_end
VM threads	vmthread_start vmthread_stop vmops_begin vmops_end

Proposed solution

Tracing

Memory usage	object_alloc alloc_new_tlab alloc_outside_tlab alloc_requiring_gc
Thread contention	monitor_wait monitor_waited monitor_notify monitor_notifyAll contended_enter contended_entered contended_exit

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

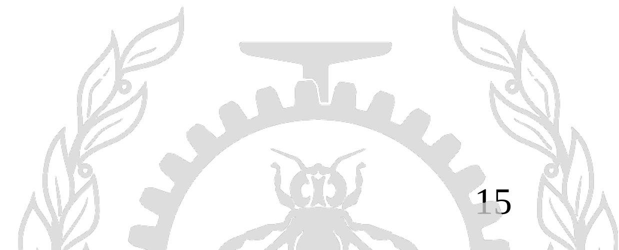


Proposed solution

Profiling

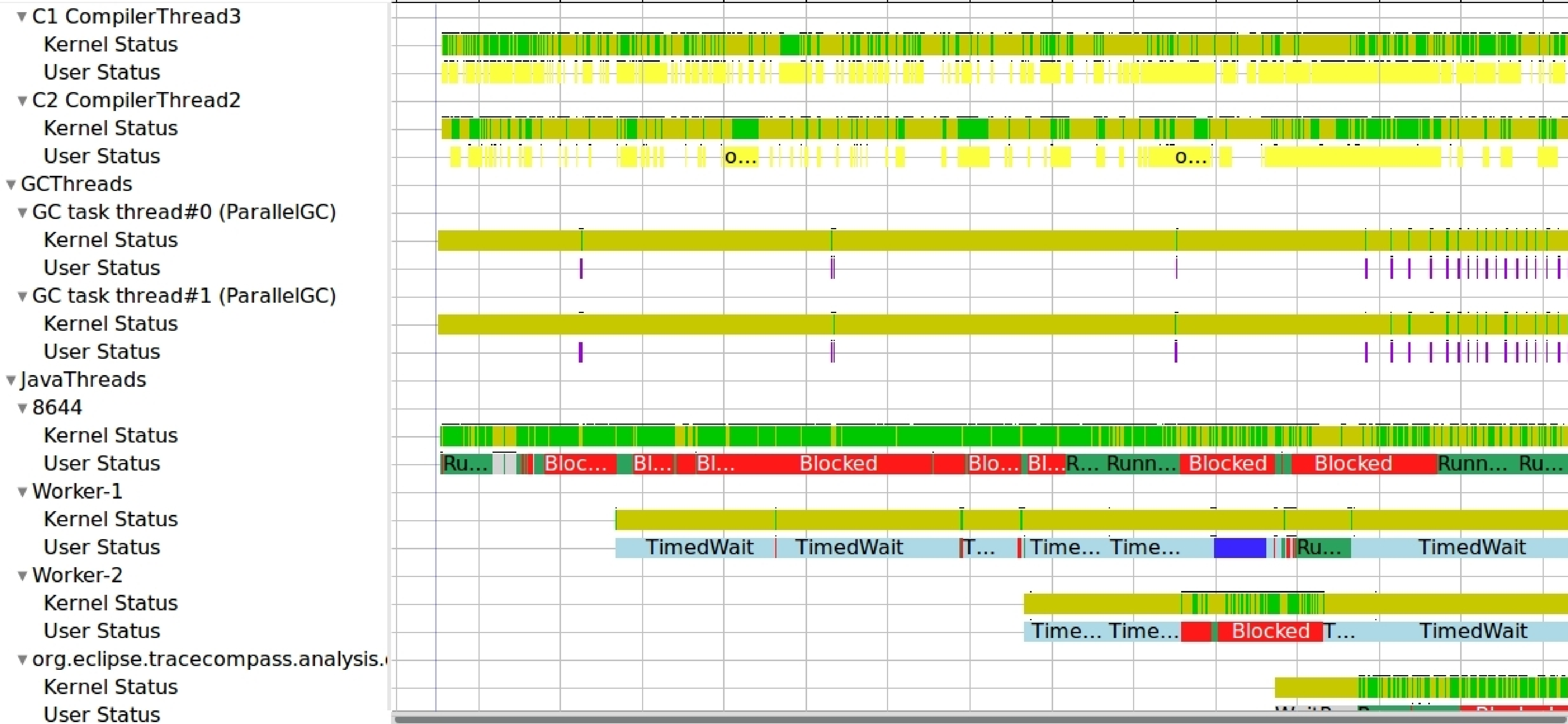
- We developed a JVMTI agent that generates periodic profiling events.
- The callstack is captured by integrating libunwind into Lttng-UST
- **`PreserveFramePointer`** option is needed to walk Java stacks

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



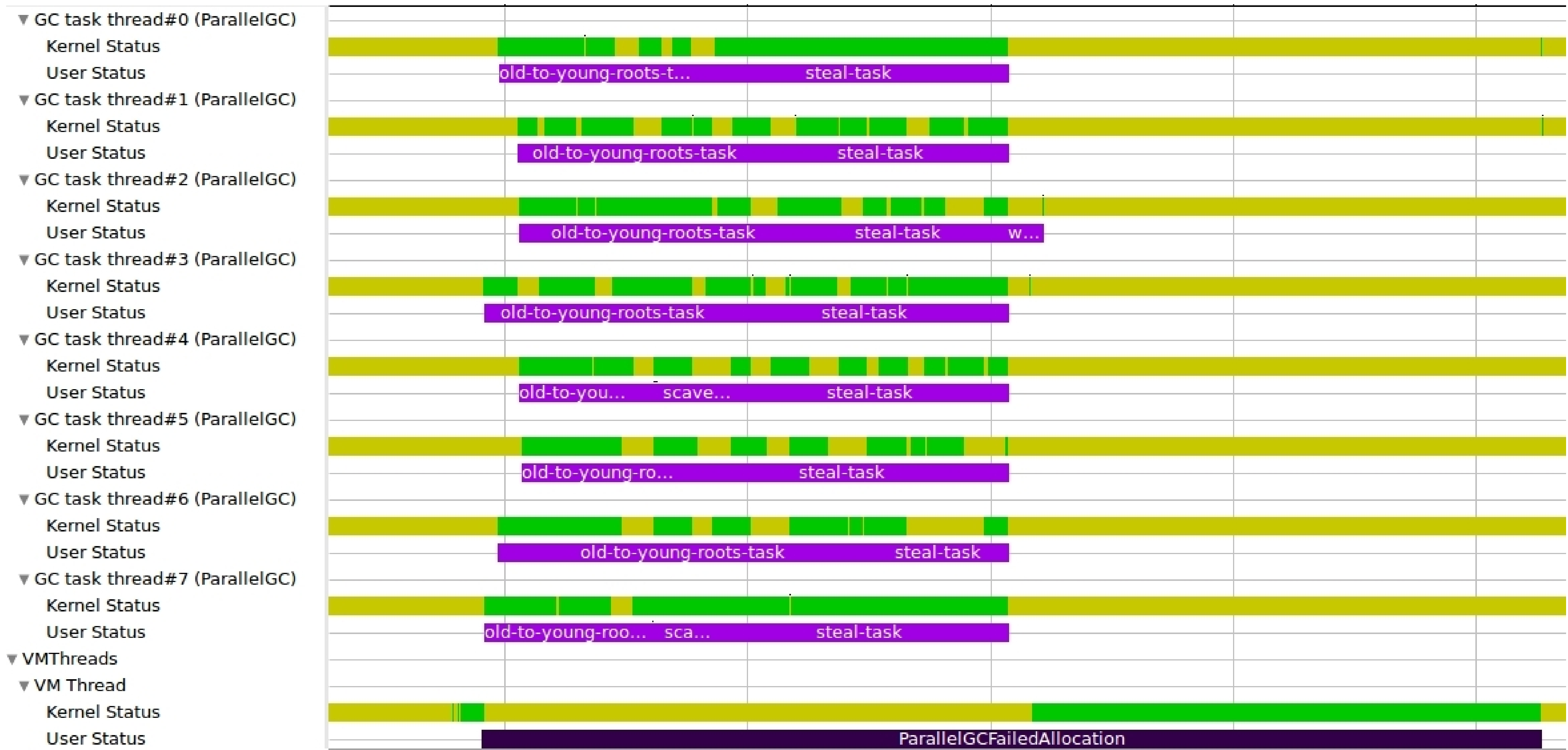
Proposed solution

Threads View



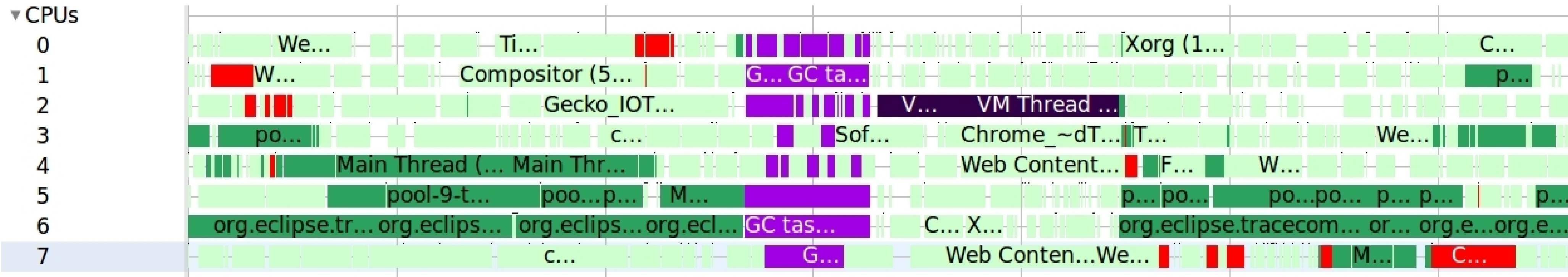
Proposed solution

Threads View

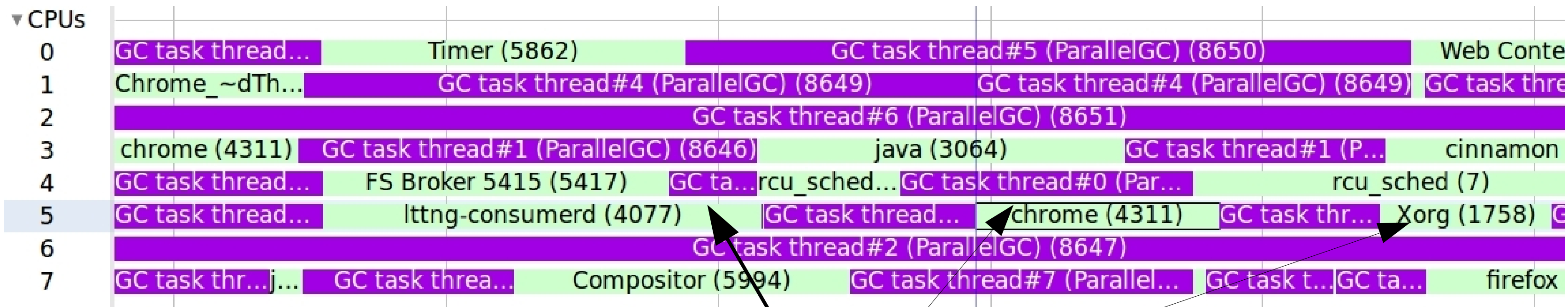


Proposed solution

CPU View



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

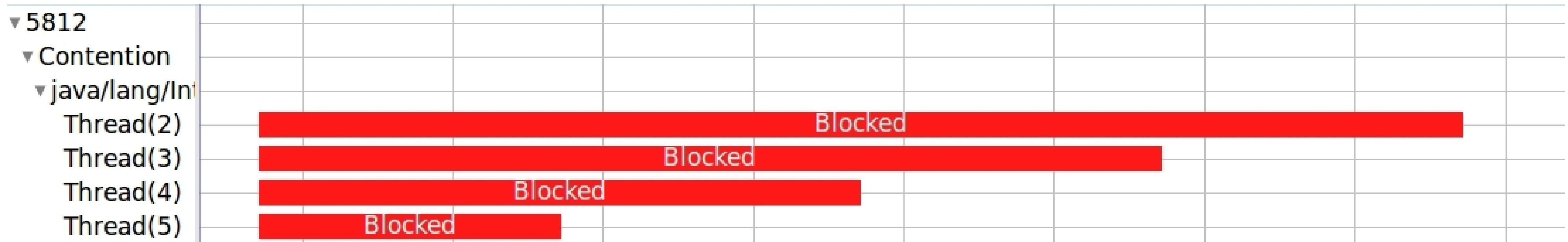


The view shows that a GC thread is being preempted by other processes

Proposed solution

Lock Contention

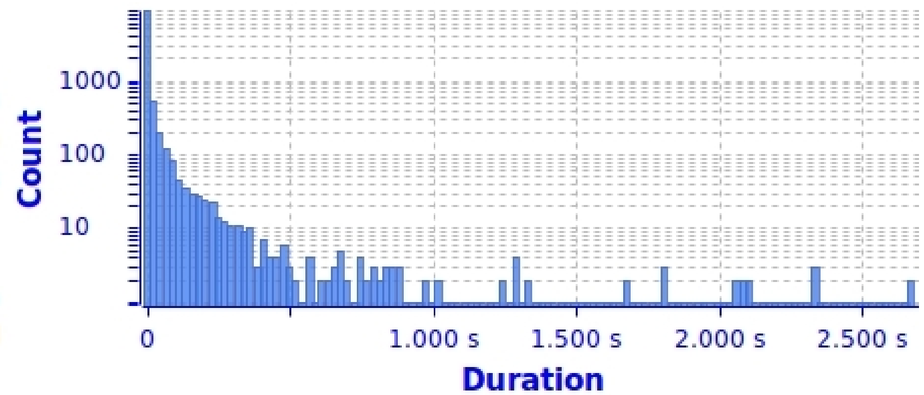
```
Integer monitor = new Integer(0);

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

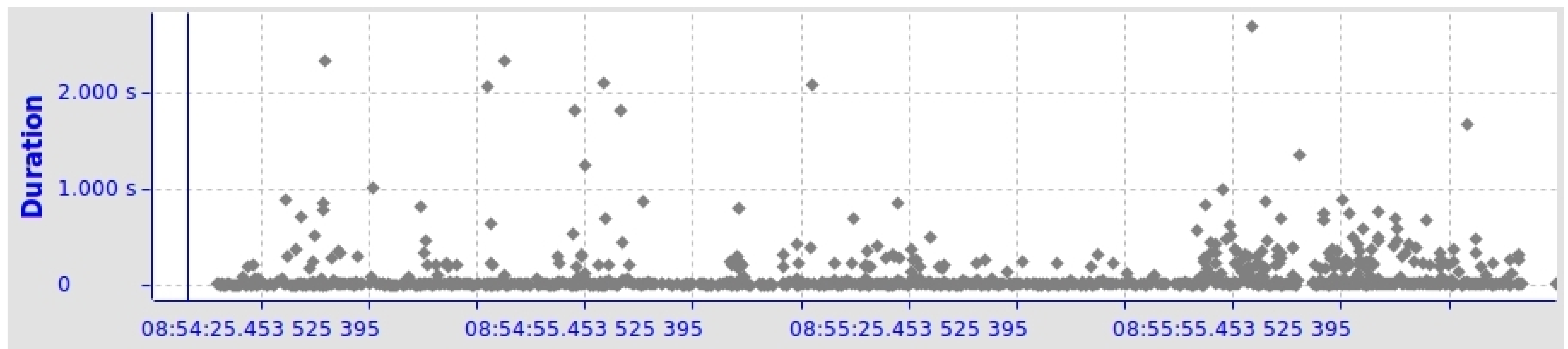


Proposed solution

JIT Statistics

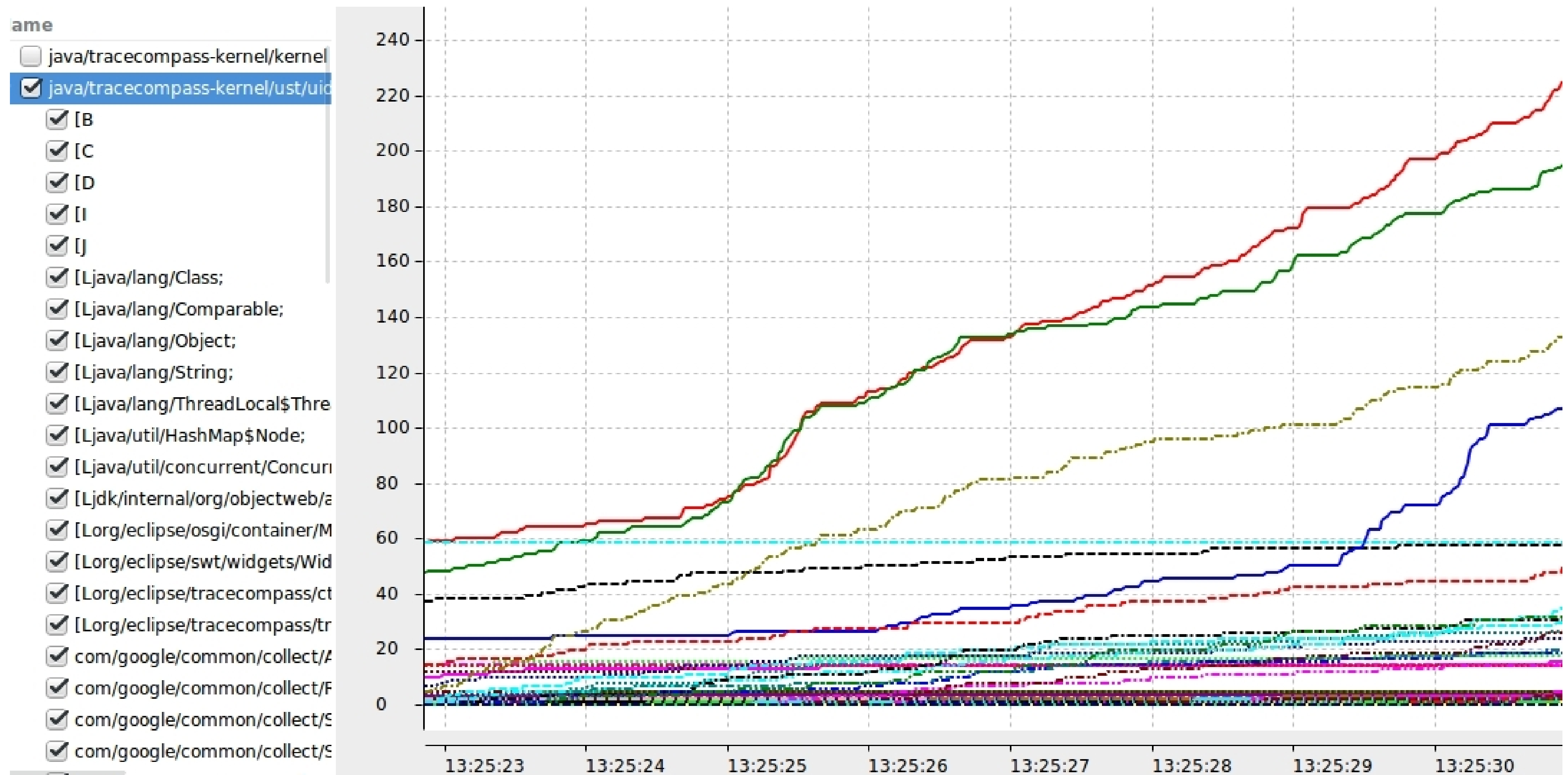
Duration	Name	Content
135,730	getDefaultPort	name=java/net/URLStreamHandler
137,736	isProcessClassRecursio	name=org/eclipse/osgi/internal/hooks
141,065	isProcessClassRecursio	name=org/eclipse/osgi/internal/weavi
141,936	isProcessClassRecursio	name=org/eclipse/osgi/internal/hooks
142,264	removeEldestEntry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





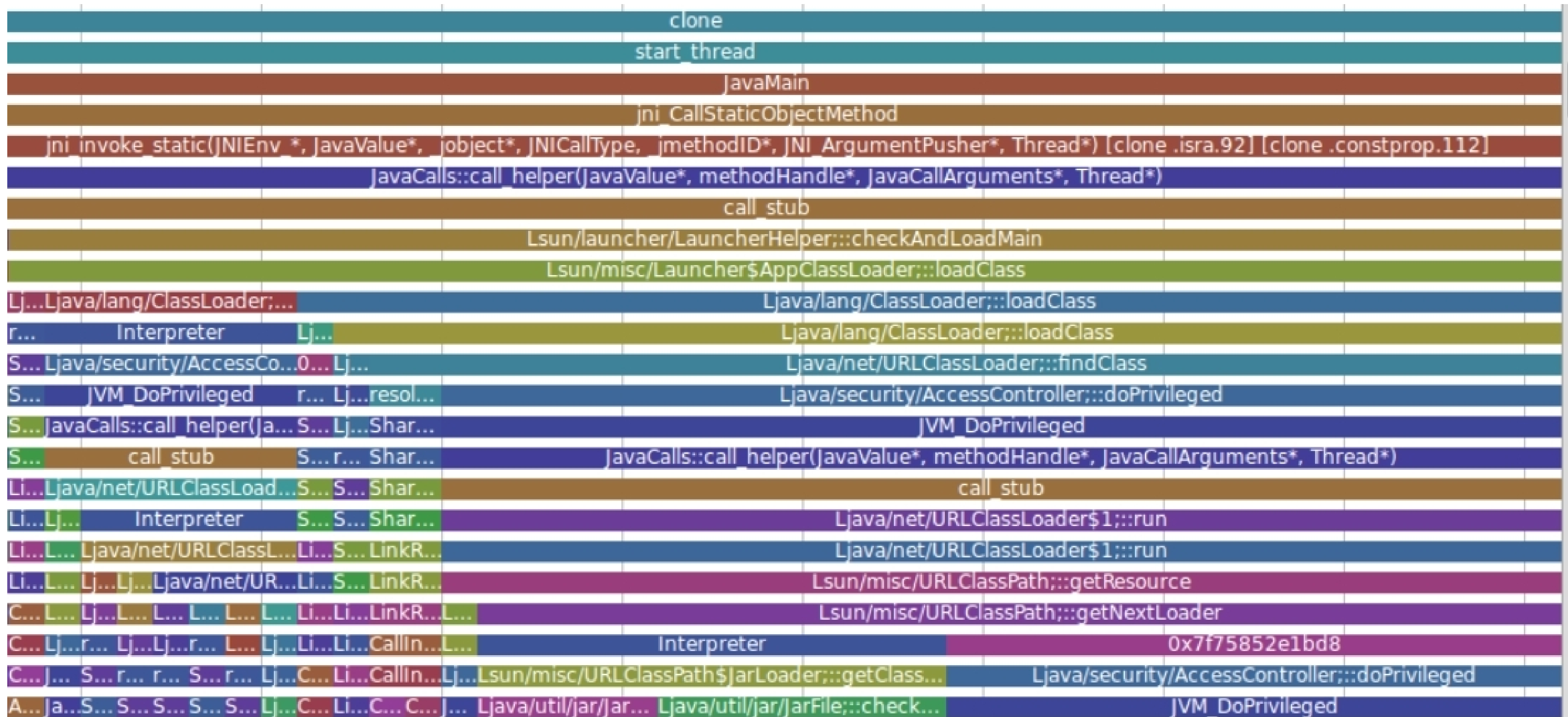
Proposed solution

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