The Journal of Systems and Software 167 (2020) 110589

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Multilevel analysis of the java virtual machine based on kernel and userspace traces

Houssem Daoud*, Michel Dagenais

École Polytechnique de Montréal, 2900 Boulevard Édouard-Montpetit, Montréal, Québec H3T 1J4, Canada

ARTICLE INFO

Article history:
Received 30 September 2018
Revised 15 February 2020
Accepted 27 March 2020
Available online 6 April 2020

Keywords:
Java
JVM
Tracing
Profiling
Performance

ABSTRACT

Performance analysis of Java applications requires a deep understanding of the Java virtual machine and the system on which it is running. An unexpected latency can be caused by a bug in the source code, a misconfiguration or an external factor like CPU or disk contention. Existing tools have difficulties finding the root cause of some latencies because they do not efficiently collect performance data from the different layers of the system. In this paper, we propose a multilevel analysis framework that uses Kernel and userspace tracing to help developers understand and evaluate the performance of their applications. Kernel tracing is used to gather information about thread scheduling, system calls, I/O operations, etc. and userspace tracing is used to monitor the internal components of the JVM such as the garbage collectors and the JIT compilers. By bridging the gap between kernel and userspace traces, our tool provides full visibility to developers and helps them diagnose difficult performance issues. We show the usefulness of our approach by using it to detect problems in different Java applications.

1. Introduction

Computer programs are becoming increasingly complex. Modern applications involve different interdependent components interacting together and running on optimized multi-core processors. In recent years, there has been an increasing interest in virtual machine based programming languages, because of their flexibility and ease of deployment. A well known example of these languages is Java. Java has been considered as one of the most widely used programming languages in industry (Cass, 2017). It is platform independent and it offers a great amount of advanced technologies like automatic memory management and Just-in-Time compilation.

However, the advanced mechanisms provided by Java made performance debugging more challenging. The developer is usually not fully aware of the internal behavior of the Java virtual machine (JVM) and, as a result, he is unable to pinpoint issues that may happen at runtime. For example, a misconfigured option can cause unnecessary garbage collection cycles that can be easily avoided once detected.

Unexpected behaviors are very hard to detect without proper tools. A wide range of analysis tools are available for developers to evaluate the performance of Java applications. Those tools may be classified into two main classes: monitoring tools and profilers. Monitoring tools provide high level information about the current state of the virtual machine. They present statistics about object allocation, garbage collection, active threads, etc. Profilers give additional information about the logic of the application. By periodically collecting the callstacks of the different threads, profilers can detect slow functions and the logic of the code behind them.

Existing performance analysis tools provide valuable information about the overall performance of Java applications. However, those tools have some limitations. First, the tracing mechanisms used to collect data are not always very efficient and introduce a significant overhead. Second, those tools do not provide a fully integrated analysis that correlates the data gathered from the different layers of the system.

In this paper, we propose a unified Java performance analysis framework that covers the whole software stack, from the user application down to the operating system. We achieve that by using a hybrid approach based on kernel and userspace tracing (Desnoyers, 2009) (Goulet, 2012). We use Kernel tracing to collect low-level information from the operating system (CPU scheduler, Block devices, network interfaces, etc) and userspace tracing to collect information from the different components of the Java virtual machine, such as the garbage collector and the JIT compiler. The traces are collected, synchronized and analyzed using a correlation model. By collecting data from multiple sources and layers, our tool offers full visibility of the system and helps in detecting problems that are very difficult to see otherwise.

* Corresponding author.
E-mail addresses: houssem.daoud@polymtl.ca (H. Daoud), michel.dagenais@polymtl.ca (M. Dagenais).

https://doi.org/10.1016/j.jss.2020.110589
0164-1212/

We presented different use cases where our tool was able to detect low-level latencies that are difficult to analyze using traditional tools such as I/O and CPU contention. We also showed that our tool does not introduce a big overhead and can be used without altering the normal behavior of applications.

The main contributions are as follows:

- Instrumentation of the Hotspot(Griswold, 1998) virtual machine using LTTng(Desnoyers and Dagenais, 2006), a low-overhead tracer available in Linux.
- Multilevel performance analysis model that correlates and analyses trace data from different sources.
- A Visualization system that helps users understand and identify performance degradations.

The rest of the paper is organized as follows: in Section 2, we introduce the different components of the JVM and we discuss the existing Java performance analysis tools. Then, we describe the architecture of the proposed solution in sections 3 and 4. We present three use cases in Section 5 and we evaluate the efficiency of the framework in Section 6. The conclusion and future work are then presented.

2. Background

In this section, we provide a description of the different components of the Java Virtual Machine and we list the different studies related to the performance analysis of each. Then, we list some existing Java performance analysis tools and we discuss their limitations. Finally, we define tracing and we justify the choice of LTTng as the main tracer for our framework.

2.1. Java virtual machine architecture

The Java programming language was designed with the purpose of writing platform independent programs that can be run on any hardware architecture. To meet this goal, the idea was to have an intermediate software layer able to interpret and execute Java programs, the Java Virtual Machine (JVM).

There are many implementations of the JVM developed by different organisations. The most popular ones are Oracle Hotspot(Oracle Hotspot, 2018), Oracle JRockit(Oracle jrockit, 2018), and Eclipse OpenJ9(Eclipse openj9, 2018). Those JVMs satisfy the specification published by Oracle in order to ensure interportablility(Java, 2018). This abstract specification does not provide implementation details. Developers have the freedom to introduce different optimization mechanisms into their virtual machines. Hotspot is an open source JVM and is distributed with the OpenJDK package. It is considered as the reference implementation, and will be used throughout this paper.

In the next sections, we will describe the general architecture of Hotspot. We focus on the performance aspect of memory management, JIT compilation and thread management.

2.1.1. Memory management: Garbage collection

In more traditional programming languages like C and C++, memory management operations, like allocating and releasing objects, have to be called explicitly by the programmer. The advantage of this approach is that it gives the developer full control over the behavior of the program. However, manually managing memory references is a delicate task and may be the source of different programming errors. The most common problems are dangling pointers and memory leaks. Many studies have focused on helping programmers to detect and avoid those problems in languages with explicit memory management (Akritidis, 2010; Caballero et al., 2012; Lee et al., 2015; Hastings and Joyce, 1991). However, those techniques require a lot of care and rigor from programmers. To free developers from the burden of memory management, many programming languages like Java and C# introduced automated mechanisms based on garbage collectors(Jones and Lins, 1996). The main tasks of a garbage collector are allocating memory objects and automatically releasing them when they are no longer needed. The collection process starts when a certain condition is met, usually when the heap occupancy reaches a certain threshold.

Designing a garbage collector implies different trade-offs. The first trade-off concerns collection frequency. Garbage collection should be executed frequently enough to ensure that unreachable objects do not stay in memory for a long time, but on the other hand infrequently to avoid slowing down the application. The second trade-off is between response time and space efficiency. When the program allocates an object, the garbage collector must be able to quickly find space for the new object, and at the same time avoid memory fragmentation (Johnstone and Wilson, 1998) by finding the best fit for the object.

Java virtual machines propose different mechanisms to improve the performance of garbage collectors. Compacting collectors reduce fragmentation by periodically scanning the heap and moving the used blocks to an adjacent place (Morris, 1978). Parallel collectors use multiple threads to execute garbage collection tasks in a concurrent manner (Boehm et al., 1991; Jones et al., 2016). Generational Garbage collectors divide memory objects into different generations based on their age and set different collection parameters for each (Lieberman and Hewitt, 1983).

Analyzing the impact of garbage collectors on applications behavior has always been a subject of interest among researchers. Studies (Attanasio et al., 2001; Blackburn et al., 2004; Hansen, 2000) used benchmarking to compare different garbage collector implementations. Many memory allocation workloads are executed with different GC configurations in order to compare their impact on execution time. Lengauer et al. (2017) provided a comprehensive study of the different benchmarking suites in order to help the user choose a convenient workload for their tests.

The major drawback of benchmarking is that synthetic workloads are limited and cannot exactly imitate real world applications. Many studies proposed to compute objects lifetimes based on garbage collector traces (Hertz et al., 2006; 2002; Lengauer et al., 2015). Collecting garbage collection events solely does not provide enough context about the application. To deal with this problem, Ricci et al. proposed *Elephant tracks* (Ricci et al., 2013), a tracing tool that traces the entry and exit of functions in addition to GC events. The advantage of this tool is that it gives more context about the application when a garbage collection happens. However, tracing all functions introduces a big overhead. The same result can be achieved by recovering the call stack when a garbage collection is triggered, as we describe in details in paragraph 3.1.3.

2.1.2. Just-In-Time (JIT) compilation

The Java virtual machine uses the Runtime environment to interpret the bytecode generated by the Java compiler. If a certain method is called multiple times, the JVM may decide to compile it to more optimized native code, in order to improve the execution time. This process is called JIT compilation.

Compilers optimize the code using techniques like inlining, loop unrolling, etc. Two types of compilers are available. Client compilers (C1) are designed for client-side applications. Those compilers are fast and generate decently optimized code. C1 compilers are designed to work on low resource machines and do not slow down the startup time of the application. On the other hand, servers compilers (C2) are designed for server applications and provide highly optimized code. C2 compilation is slow and using it during startup introduces a significant overhead. Recent JVMs provide

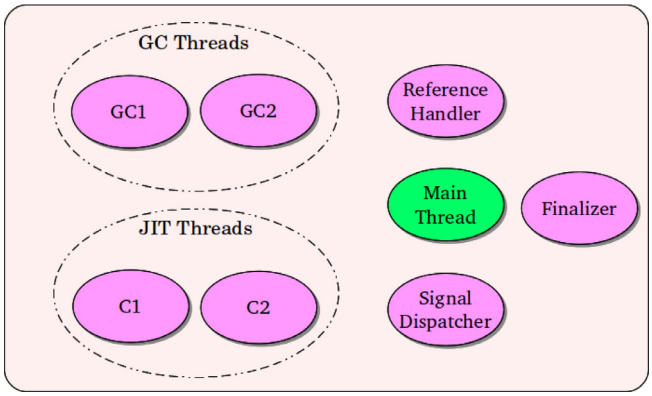

Fig. 1. Overview of the different types of threads.

Table 1
Thread states.

State	Defintion
thread_new	the thread has been created, but not yet started
thread_in_Java	the thread is currently running Java code
thread_in_native	the thread is running native code
thread_in_vm	the thread is executing virtual machine operations
thread_blocked	the thread is blocked, because of a lock, a disk operation, etc.

a technique called *tiered compilation* (Jantz and Kulkarni, 2013), which combines both compilers. A hot method is first compiled by the C1 compiler and then recompiled by C2 if it is executed more often than a certain threshold. The advantage of tiered compilation is that it does not slow down the startup time of an application, but at the same time provides highly optimized code for very hot methods.

Many studies have been conducted to evaluate the effects of compilation on the overall program performance. Selective compilation is mainly based on the observation that the program spends most of the time executing the same functions, which we call *hot functions* (Arnold et al., 2005; Knuth, 1971). The biggest challenge is to set the right threshold for selecting those functions. A high threshold makes the compiler very conservative, declining many optimization chances. But setting a low threshold can trigger a large number of unnecessary compilations. Finding the ideal number is usually based on predictions based on runtime tracing, usually using performance counters (Kotzmann et al., 2008).

2.1.3. Thread management

Thread management is a very important task of the Java virtual machine. Different types of threads are involved. Some are created by the Java application itself and other threads are needed by the JVM to execute low-level operations like signal handling and garbage collection. The Hotpot virtual machine uses a 1:1 threading model. Every JVM thread is associated to a native operating system thread.

The JVM defines different tread types. The most important ones are Java threads, GC threads, compiler threads and VM threads (Fig. 1).

The possible states of a thread are presented in Table 1.

2.2. The LTTng tracing framework

Tracing is a mechanism that collects execution data from an application at runtime. In order to avoid the observer effect, tracers use advanced techniques to minimize the overhead. Therefore, tracing is one of the most used performance analysis techniques for production systems. Instead of stopping the application, the tracer writes the different events into a file, and the analysis is executed offline. A trace event is described by a name, a timestamp, and a payload containing multiple parameters. It is possible to trace system calls, interrupts, function calls, etc.

LTTng is an open source tracing tool developed at the DORSAL laboratory, as an improvement to the LTT tracer. It offers low-overhead tracing for the Linux operating system. The Linux kernel can be instrumented statically using TRACE_EVENT macros, or dynamically using Kprobes(Krishnakumar, 2005). LTTng uses per-CPU circular buffers and lock-free data structures like RCU to support highly parallel applications. Read-Copy-Update (RCU) is a synchronization mechanism that allows reads to happen concurrently with updates, while at the same time avoiding cache coherence overhead between CPU cores. Trace events are written in circular buffers using The Common Trace Format (CTF), a binary format designed for trace files, more efficient than text formats used by loggers (The common trace format CTF, 2018).

LTTng UST is designed to offer low-cost userspace tracing. It shares the same circular buffer mechanism as the Kernel tracer and it uses URCU(Desnoyers et al., 2011) (Userspace Read-Copy-Update), a userspace version of the RCU algorithm. The design of LTTng guarantees reentrency, interrupt-safety and signal-safety. Gebai and Dagenais (2018) compared LTTng UST with other userspace tracers and showed that the TRACEPOINT infrastructure used by LTTng outperforms other callbacks mechanisms. LTTng UST handles the tracepoints fully in userspace and does not require a context switch to the Kernel space, unlike DTrace(Gregg and Mauro, 2011) and SystemTap(Eigler and Hat, 2006).

2.3. Java performance analysis tools

Analyzing the performance of Java applications is one of the biggest challenges due to the complexity of the JVM and the different components involved. Many technologies are provided to help programmers detect performance issues which can be caused by inefficient programming or a non optimal configuration.

Java Management Extension (JMX) (Sullins and Whipple, 2002) is a performance monitoring technology that helps developers manage the different resources of an application locally or over the network. Every monitored resource is described by an MBean such as ThreadMXBean, ClassLoadingMXBean, MemoryMXBean, etc. Internally, an MBean is a Java object and it can be registered dynamically at runtime. For example, a Memory management MBean contains attributes like *total_memory* and *free_memory* and methods like *getHeapMemoryUsage()*. MBeans are managed by an MBeanServer which can be controlled using different connectors like RMI, JMXMP, or any other user-defined protocol. JMX Beans are developed in Java and, as a result, they usually introduce a large overhead and can alter the normal behavior of the program due to heap pollution.

The Java Virtual Machine Tool Interface (JVMTI) (JVM tool interface, 2018) is a programming interface to inspect and to control the execution of Java applications. JVMTI agents are developed in C/C++ and are able to inspect the state and to control the execution of Java applications. They are generally used by diagnosis and monitoring programs like debuggers, profilers, memory analyzers, etc. The clients of the JVMTI are called *agents*. An agent can send requests to the JVMTI to execute control functions. For example, it is possible to force a garbage collection, access and modify a Java class attribute, etc. The agent can also receive a notification from the JVMTI if a certain event is triggered, such as thread creation, class loading or a JIT compilation. JVMTI also allows *bytecode instrumentation*, the ability to alter the normal behavior of the appli-

cation by injecting a new code at runtime. This can be useful for instrumentation or to update execution counters.

Many monitoring tools have been developed using the technologies described above to help the developer gather valuable information at runtime. JConsole (The Java monitoring and management console, 2018) is a graphical tool that uses JMX to provide basic information about threads, memory, classes etc. It also allows data collection from user-defined MBeans, which provides a more personalized analysis. VisualVM (Visualvm, 2018) is another monitoring tool similar to JConsole, but with more advanced views. It offers the possibility to track the state of each thread during the execution, evaluate memory usage and provide heap dumps.

Wang et al. (2015) instrumented the J9 Virtual machine with LTTng tracepoints and used them to collect performance data. The limitation of their work is that it only focus on the data collection and does not provide any algorithm or tool for data analysis. Hotspot JVM contains DTrace(Gregg and Mauro, 2011) probes that can be used to collect performance data from the virtual machine. Those tracing probes are convenient because they are dynamic and can be activated at runtime but they are very costly compared to LTTng(Brosseau, 2020). DTrace uses a callback mechanism based on system calls. Every time the application reaches a tracepoint, a system call is generated by the tracer and the probe handler is executed in Kernel space. Switching the context from user space to Kernel space for each tracepoint introduces a significant overhead compared to pure userspace tracing. LTTng alternatively uses the TRACEPOINT infrastructure. Probe handlers are fully executed in userspace and do not involve the Kernel. DTrace is not officially supported on Linux, and the current alternatives, such as dtrace4linux(Dtrace4linux, 2016), are not stable.

When it comes to profiling, existing tools like jstack, JProfiler and HPROF (Hprof: A heap/cpu profiling tool, 2018) periodically call the function *GetAllStackTraces* provided by JVMTI, to recover the call stacks, and use the collected data to evaluate function durations. The problem with this technique is that the function *GetAllStackTraces* only returns when all the threads reach a safepoint, which affects the sampling period. This limitation is known as *safepoint-bias*. HonestProfiler (Honest profiler, 2018) solves this problem by using an undocumented API function *AsyncGetCallTrace* to sample the Java application. *AsyncGetCallTrace* captures exactly what a JVM is doing, without waiting for a safepoint. The only drawback of this technique is that it only captures the call stacks of Java threads and other virtual machine threads like GC or JIT threads.

Branden Gregg used Perf to profile Java applications (Gregg and Spier, 2020). A Kernel timer is used to periodically extract the callstack from the target application. Unlike other profilers, Perf is able to collect Kernel and Java callstacks at the same time, providing better visibility into the application. However, profiling does not offer the same level of details as a tracer. For example, a tracer is able to identify exactly when a garbage collection cycle started and finished, the garbage collector parameters, and the number of object released in each garbage collection cycle, whereas Perf can only estimate the overall garbage collection time.

In summary, the existing monitoring tools provide a good view of what is happening in the application, but have important limitations. Firstly, the data gathering mechanisms used by some tools are not very efficient and introduce a significant overhead. Secondly, most tools do not provide a comprehensive analysis that integrates data collected from the different layers of the system.

In the rest of the paper, we propose a performance analysis framework that uses a low-overhead tracer to collect data, simultaneously from the user and the kernel space. Advanced algorithms and data models are implemented to correlate the gathered information and to provide a full visibility of the system.

3. Proposed solution

Analyzing the runtime behavior of a Java application requires a deep understanding of the JVM and the environment on which it is running. To have full visibility of the application, our approach is based on collecting data from multiple layers: the user application, the Java virtual machine and the operating system. The collected traces are then synchronized and analyzed by inspecting the occurrence and the causality of the different events. The general architecture of the proposed analysis framework is described in Fig. 2.

3.1. Data collection

3.1.1. Userspace tracing

LTTng-UST is able to collect data from userspace applications. The data collection is achieved by inserting probes in the source code of the application and receiving an event every time the execution hits that point. The event payload contains the data required by the analysis. In order to understand the runtime behavior of Java applications, we instrumented the different components of the Java Virtual Machine. Fig. 3, summarizes the different userspace tracepoints used for the analysis

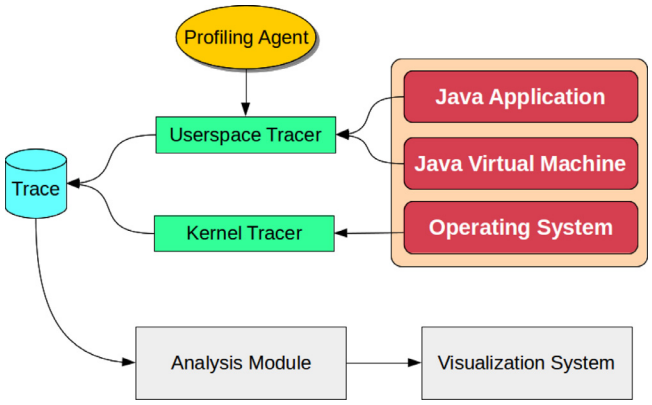

Fig. 2. General Architecture.

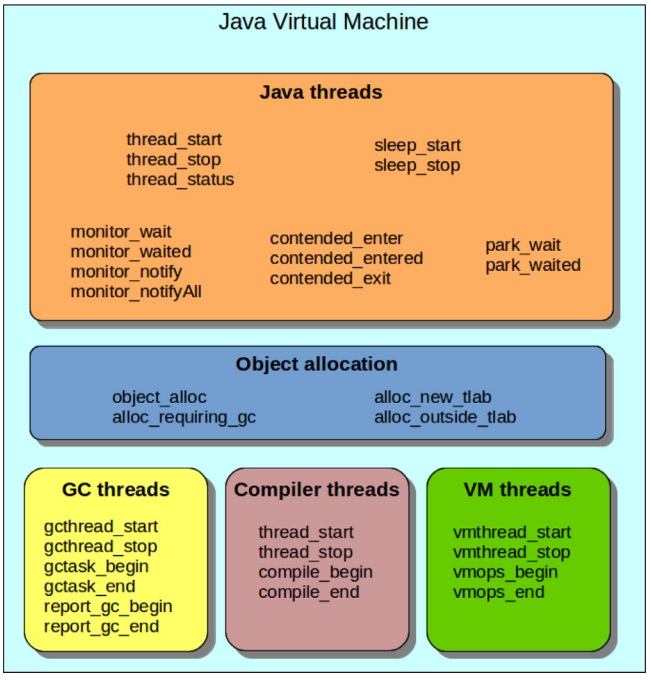

Fig. 3. Userspace Tracepoints.

Table 2
Java threads events.

Tracepoint	Definition
thread_start / thread_stop	A Java thread is started.
thread_sleep_start / thread_sleep_stop	The thread is sleeping
monitor_wait / monitor_waited	The thread is waiting for a monitor
monitor_notify / monitor_notifyAll	The current thread notifies other threads that the monitor is released
contended_enter / contended_entered / contended_exit	Those events show when a thread is blocked on a lock and when it enters and exits a critical section. The lock is defined by a name and an address
park_wait / park_waited	The events indicate when a thread is parked and unparked.

Table 3
GC threads events.

Tracepoint	Definition
gcthread_start / gcthread_stop	A GC thread is started or stopped
gc_begin/gc_end	Those events give when the name and the id of the garbage collection algorithm exectued by the JVM
gctask_begin / gctask_end	These events show the specific task currently processed by the current GC thread

Table 4
JIT compiler events.

Tracepoint	Definition
thread_start / thread_stop	A compiler thread is started or stopped
compile_begin / compile_end	A method is being compiled by the thread

Table 5
VM threads events.

Tracepoint	Definition
vmthread_start / vmthread_stop	A VM thread is started or stopped
vmops_begin/ vmops_end	The VM operation is called concurrent if it can be executed without blocking Java threads

Table 6
Object allocation events.

Tracepoint	Definition
object_alloc	A new object is allocated by the current thread
alloc_new_tlab	The object is stored in a newly created TLAB.The object is stored in a newly created TLAB.
alloc_outside_tlab	The object is created in the shared memory region.
alloc_requiring_gc	The last allocation caused the execution of the garbage collector

The trace events used can be organized into different categories:

- Java threads tracepoints (Table 2)
- Garbage collector tracepoints (Table 3)
- JIT compiler tracepoints (Table 4)
- VM threads tracepoints (Table 5)
- Object allocation tracepoints (Table 6)

3.1.2. Kernel tracing

Kernel tracing provides detailed information about the operating system. The extracted data is very important to understand exactly what is happening in the system during the execution of the application. For example, the application can be scheduled out by the OS scheduler, or blocked for an I/O operation. By tracing the operating system, we can have a better picture of what is running on the same computer.

In this paper, we trace the OS scheduler in order to know which thread is running on which CPU core. This information is important to detect CPU contention and preemption between different applications. The *sched_switch* event is called whenever a thread is scheduled in or out of the CPU, and *sched_waking* when a blocked application is ready to be executed again.

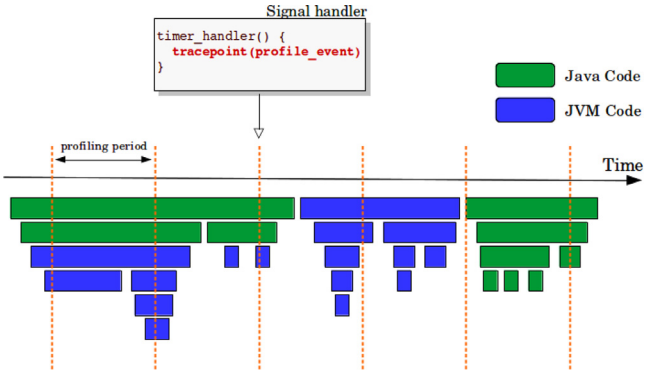

Fig. 4. Profiling.

We also trace the system calls issued by the target application. The *read* and *write* systems calls are used for file and sockets operations. The *futex* system calls are used to track lock contention between the different threads etc.

On a deeper level, we are using the block layer instrumentation to collect data about disk usage, and the network layer to track the packets sent over the network.

3.1.3. Profiling agent

In addition to kernel and userspace events, we want our tool to provide profiling data so that we can inspect hot functions. Unlike other tools, we decided not to use the *AsyncGetCallTraces* function, because it is unable to get the callstack of JVM internal threads. Instead, we created a patch that integrates Libunwind (The libunwind project, 2018) into LTTng-UST. When a userspace event is fired, the tracer walks the callstack and includes it in the payload. Libunwind recover both, native and Java stacks and resolves the symbols using the [address-symbol] map kept by the JIT compiler. *Libunwind* uses optimized caching mechanisms to accelerate the stack walking. If a frame has been unwound before, it is likely to find it in the cache and thus no further processing is required. This techniques avoids the safepoint-bias and makes the profiling precise. The JIT compiler may decide to inline some methods to optimize the execution time. In this case, the method name does no appear in the call stack and is not handled by our tool.

Our profiling agent is basically a JVMTI agent that uses a timer to periodically generate a special UST event. The callstack is walked everytime the profiling event is fired, as shown in Fig. 4.

In addition to profiling, the provided patch is also useful to know which parts of the code are triggering certain behaviors. For instance, by recovering the callstack of object allocation events, we

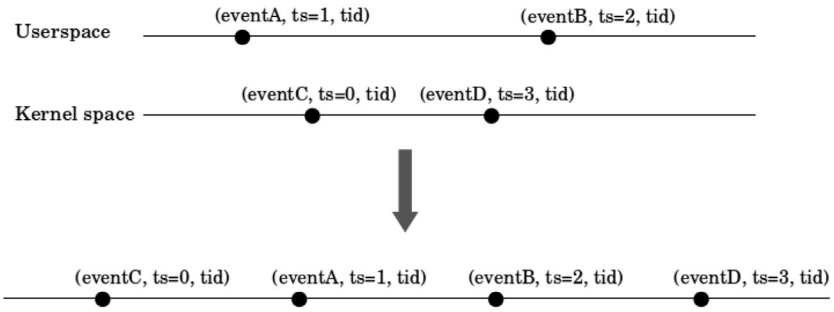

Fig. 5. Trace synchronization.

can detect the functions that are generating most of the allocations.

3.2. Trace synchronization

In our analysis, we use LTTng to collect data from different data sources. We are getting information from the Java application, the JVM and from the operating system. We defined three separate channels to collect the events.

- Kernel channel: All kernel events are registered in this channel: system calls, network events, CPU scheduler events, disk events, etc.
- User channel: Userspace events are registered in this channel: thread management, memory allocations, garbage collection etc. The callstack context is activated for this channel.
- Profiling channel: Profiling events are written in this channel. The callstack context is activated for this channel.

The synchronization of the different layers is done based on the monotonic clock of the operating system, which insures total ordering. Using timestamps and thread TIDs, we are able to associate events between different layers and define causality relationships between them (Fig. 5).

4. Data analysis and visualization

4.1. Data model

One of the biggest challenges in tracing is the huge number of events generated, at a very high frequency. Problem detection has to be automated or at least semi-automated. Many approaches have been proposed to simplify data analysis. Ezzati-Jivan and Dagenais (2012) proposed a technique based on data abstraction to reduce trace size. Related low-level events are grouped into more meaningful higher level compound events. Another approach is to use machine learning techniques to automatically detect anomalies.

In this paper, we use a combination of metric-based abstraction and visual abstraction to present the data in a convenient way. Metric-based abstraction consists of reading the trace events and summarizing them by defining a set of metrics. Just by watching the metrics, the user should be able to have a fairly precise idea about the application performance. Visual abstraction consists in graphically representing the data and showing it on the screen display. We used Trace Compass (Toupin, 2011), a trace analysis tool, to create views that represent the different components of the system

Diagnosing a performance issue requires to navigate the trace horizontally, choosing different time ranges, and vertically, zooming in and out on different threads. Instead of reading the trace events every time a new time range is selected, we decided to use

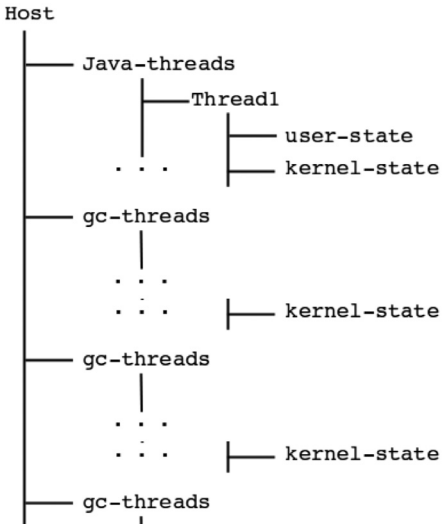

Fig. 6. Proposed attribute tree.

a stateful approach that consists in saving the state of the system in an incrementally built database. When the user navigates the trace, the states are queried directly from the database, without the need to reread the trace. Many memory-based data structures are available for time interval based data, such as Segment-tree or R-tree, but using them in our context is not possible due to the huge size of trace files. It is not possible to keep all the necessary information in memory. Traditional relational databases can be used, but previous research has shown that their query time is not optimal for trace analysis. *The Modeled State System* (Montplaisir et al., 2013b; 2013a) has proven to be a very efficient data structure for holding the state intervals of the system in a tree-like fashion. It is composed of an *attribute tree* and a *state history tree*. The attribute tree describes the different components of the system, and the state history tree keeps the state of each component throughout time.

We designed an attribute tree that contains all the information necessary to analyze Java applications. A sample of this tree is presented in Fig. 6. For every thread created by the JVM, we keep two attributes. The user-state attribute describes the state of the thread from the userspace level. A thread can be marked as running, sleeping, waiting for a monitor etc. The kernel-state attribute contains lower-level kernel information about the thread: running on CPU, waiting for futex, executing a system call, etc.

4.2. Finite state machines

In order to efficiently track the state of each system component, we modeled state transitions as finite state machines. In the

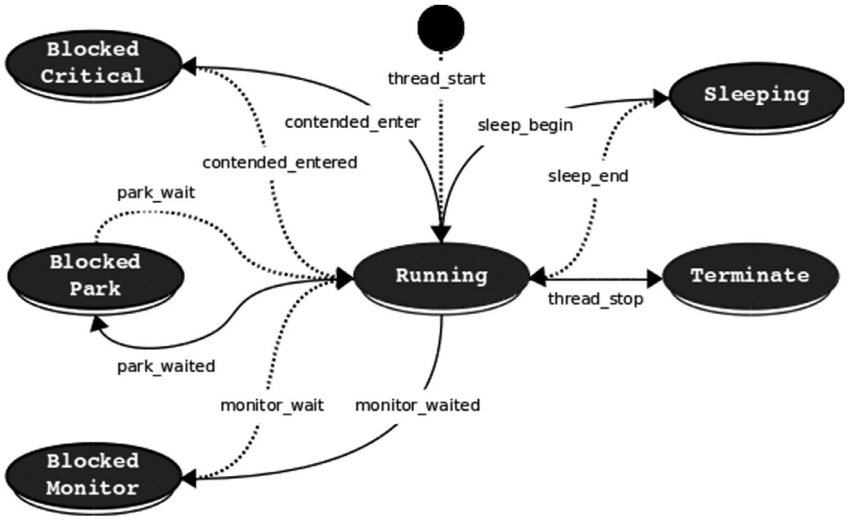

Fig. 7. Java thread life cycle.

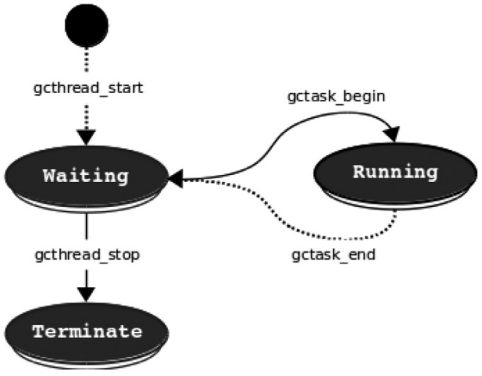

Fig. 8. Garbage collection thread life cycle.

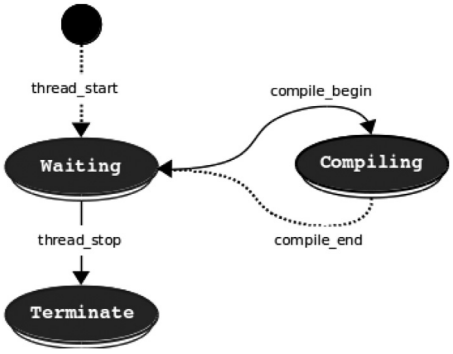

Fig. 9. Compiler thread life cycle.

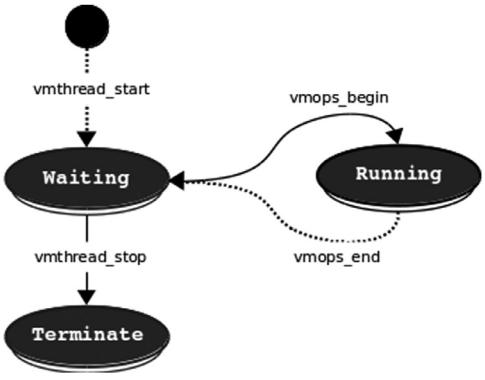

Fig. 10. VM thread life cycle.

next subsections, we present the state machines that we defined for each type of thread created by the application.

4.2.1. Java threads life cycle

It is important to know the state of each Java thread during execution. Having a clear picture about all the executing threads, and if they are running or not, helps in detecting performance bottlenecks. We inserted many tracepoints into the task manager of the Hotspot JVM in order to get this information. Based on the collected events, we are able to create a model that defines the life cycle of each thread. The different states defined are:

- Running: the thread is executing Java code
- Sleeping: the thread is waiting for a timer
- Blocked on critical section: the thread is waiting to enter a critical section
- Blocked on monitor: the thread is waiting on a monitor. It is waiting for a signal from another thread to be able to execute. It is also possible to define the maximum waiting time before going back to the running state
- Blocked on park: the thread is waiting for a permission to execute. It is also possible to define the maximum waiting time before going back to the running state

The life cycle of Java threads is shown in Fig. 7

4.2.2. GC Threads life cycle

GC Thread are created by the JVM to execute garbage collection operations. Instead of collecting the heap with a single thread, the memory space is divided into different regions, and each is collected with a separate thread. Although some critical GC operations cannot be performed in parallel, using multiple threads greatly increases the speed of garbage collection. The number of GC threads can be manually set using the *-XX:ParallelGCThreads* option.

The events *report_gc_start* and *report_gc_stop* are generated when a GC operation is started and finished. The payload of those events indicates the type of the garbage collector in use. Many GC threads can be used by the JVM to execute garbage collection operations. The events *gctaskthread_start* and *gctaskthread_stop* indicate the lifetime of a GC thread. A GC operation requires different GC

tasks. Those tasks can potentially be executed in parallel by different threads. The *gctask_start/stop* events are used to know which task is being executed by which thread.

4.2.3. JIT Threads life cycle

Compiler threads are created at the startup of the application and they continue running until the end. A JIT thread can be a C1 or C2 compiler and the number of threads depends on the configuration. JIT threads are basically daemons that stay inactive until a compilation task is required. The tracepoints *compile_begin* and *compile_end* mark the beginning and the end of a compilation task, and their payload contains the name and the class of the compiled function.

4.2.4. VM Threads life cycle

VM threads are used to execute internal virtual machine operations. The events *vmthread_start* and *vmthread_stop* indicate when a VM thread is created and killed. VM threads are daemons that are waiting continuously for virtual machine tasks. When a new task appears in the VMOperationQueue, the VM thread executes it and then goes back to sleep. The execution is marked by the *vmops_begin* and *vmops_end* events.

4.3. Analysis algorithm

Detecting the root cause of a performance problem requires an advanced analysis that covers all the involved components. In this work, we developed an algorithm to help users detect problems in their Java applications. The algorithm is based on a top-down approach that consists of analyzing higher level data to detect irregularities and then go deeper to provide more precision.

The root cause analysis algorithm is presented in Algorithm 1. If the thread spends a lot of time sleeping or blocked, we mark that as a potential problem and we return the related call stacks. If the thread is marked by the JVM as running but it is not executing on a CPU core, we investigate problems related to GC, JIT compilation and resource contention. If the thread is running on the CPU but it takes a lot of time to execute, we use the profiler to detect slow functions and we return this information to the user for further analysis.

Algorithm 1: Analysis algorithm.

```
Input: thread
Output: root_causes
root_causes ⟵ [ ];
[ running, sleeping, blocked ] ← compute_user_stats(thread);
if sleeping > threshold then
    sleep_callstacks ← getCallstacks(sleep);
    root_causes+ = (sleeping, sleep_callstacks);
end
if blocked > threshold then
    monitor_callstacks ← getCallstacks(monitor);
    park_callstacks ← getCallstacks(park);
    root_causes+ = (monitor, monitor_callstacks);
    root_causes+ = (park, park_callstacks);
end
if the thread is running then
    if GC detected then
        gc_callstacks ← getCallstacks(gc);
        root_causes+ = (gc, gc_callstacks);
    end
    if JIT detected then
        jit_callstacks ← getCallstacks(jit);
        root_causes+ = (jit, jit_callstacks);
    end
    [ on_cpu, syscalls ] ← compute_kernel_stats(thread);
    if syscalls > threshold then
        Get the kernel events involved;
        if Kernel latency (disk/network/cpu contention) then
            root_causes+ = (resource_contention);
        end
    else
        # The thread is running on the cpu;
        profiler_callstacks ← getProfiler();
        root_causes+ = (program_logic, profiler_callstacks);
    end
end
```

4.4. Visualization

As mentioned above, trace files are generally very big and contain a tremendous amount of information. The trace events are low-level and very difficult to analyze manually. In this paper, we provide a visualization framework that gathers the data from the state system and presents it in a graphical way. Our goal is to have flexible views that offer full visibility of the application, and guide the user in performance troubleshooting. To achieve that, we implemented our views as a part of Trace Compass (Toupin, 2011).

Trace Compass is an open source trace visualization tool which offers different views designed for specific aspects of the system. Many analysis have been proposed for Trace Compass, such as the critical path analysis (Giraldeau and Dagenais, 2016), virtual machines analysis (Gebai and Dagenais, 2014) (Nemati and Dagenais, 2018), etc. The data structures and the algorithms proposed in this paper resulted in many additional views for Java applications analysis. The views are synchronized: when the user selects a time range, all the views are updated to show details about that specific region of the trace.

The *Threads view* (Fig. 11) shows the state of all the threads created by the application, including regular Java threads, JIT threads, GC threads and VM threads. This provides an overview of the general behavior of the application and how the different threads interact with each other. The view shows by default the kernel and the userspace states side by side, but the user can still decide to hide unneeded information. Another interesting feature provided by this view is its ability to interact with the *Profiler view*. For example, in Fig. 11, the user selected a time range to investigate a garbage collection operation, and the profiler view (Fig. 12) is updated to show the call stacks of the GC thread during the selected range.

The *CPU view* (Fig. 13) shows the threads running on each core of the CPU. This view is very useful to detect preemption. A thread can be seen as running from the userspace perspective but it is not actually executed because it has been preempted by the operating system scheduler, due to CPU contention.

The *lock contention view* shows the list of monitors and the different threads acquiring and releasing them. In Fig. 14, we see that threads named Thread-*n* are competing to acquire the monitor BufferedImage.

The *memory view* shows the class and the number of objects allocated by the different threads (Fig. 15). By recovering the call stack of object allocation events, we can detect which functions are consuming most of the memory.

5. Use cases

The main contribution of our tool is being able to provide Kernel and userspace information at the same time, providing low-level details about the behavior of Java applications. In this sec-

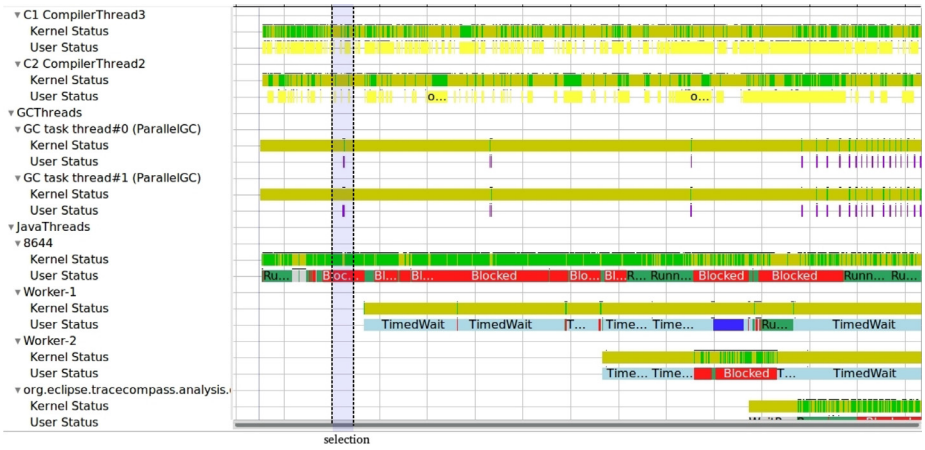

Fig. 11. Threads view.

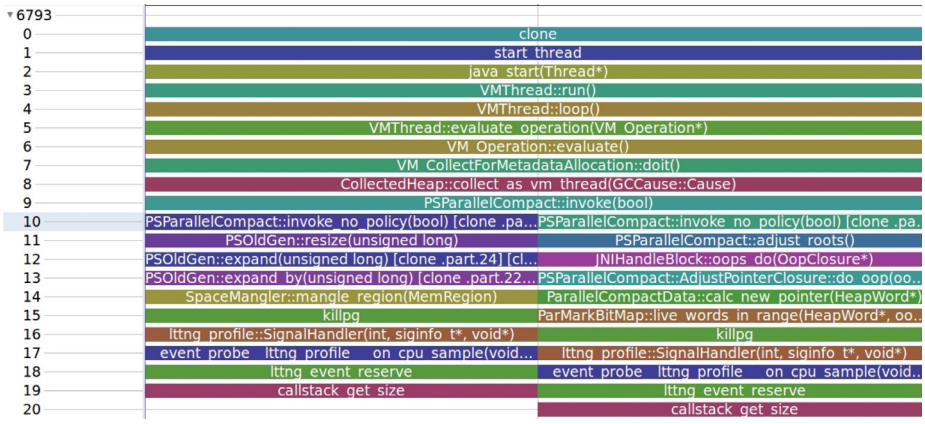

Fig. 12. Profiler view.

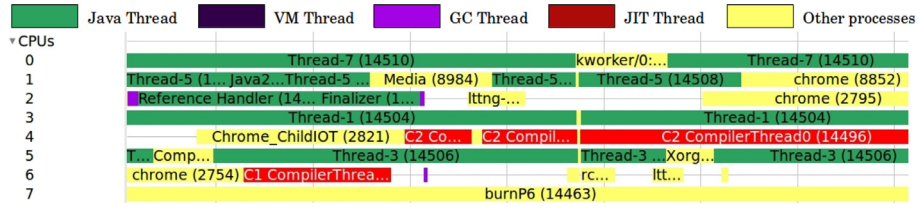

Fig. 13. CPU view.

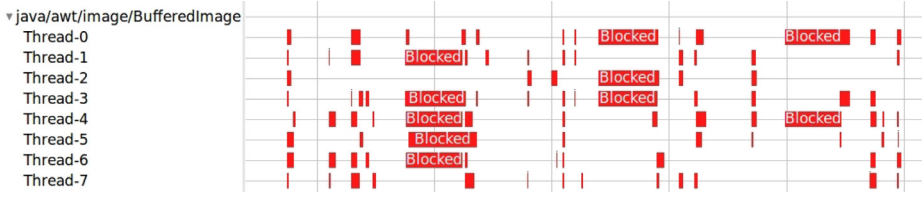

Fig. 14. Lock contention view.

tion, we present different use cases where our tool is able to solve challenging problems.

5.1. Use case 1: CPU contention of processes

Many existing Java analysis tools are able to show the state of the different Java threads at runtime. The problem with their approach is that it is fully based on userspace information. Consequently, the views presented are misleading in a way. When a thread is marked in userspace as running, it does not mean that it is actually being executed on a CPU core. The operating system can decide at any time to schedule out the thread and allocate the CPU core to another application. Unlike other tools, by combining kernel and userspace events, we are able to provide precise information about the CPU cores and which thread is physically running on each core.

The software analyzed in this use case is a photo editing application written in Java. It is implemented as a web service which receives a picture from the network, applies a filter and sends it back over the network. The filter algorithm is fully parallel and can

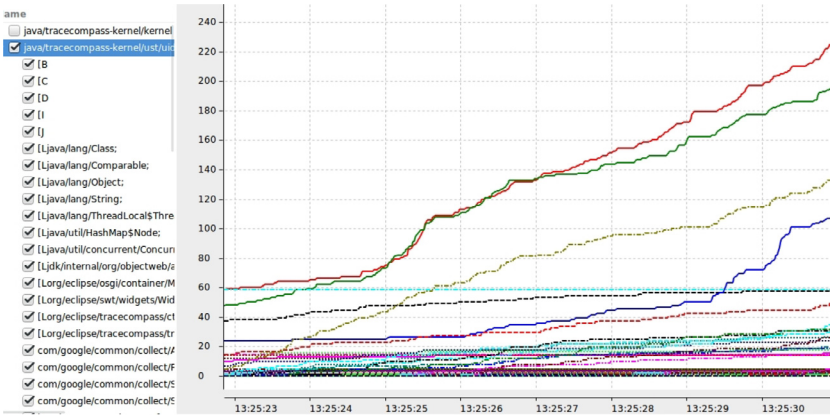

Fig. 15. Memory usage view.

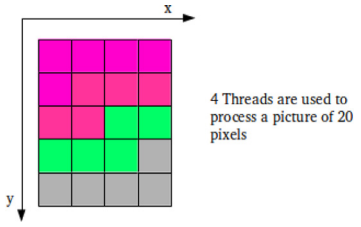

Fig. 16. Parallel processing of a picture.

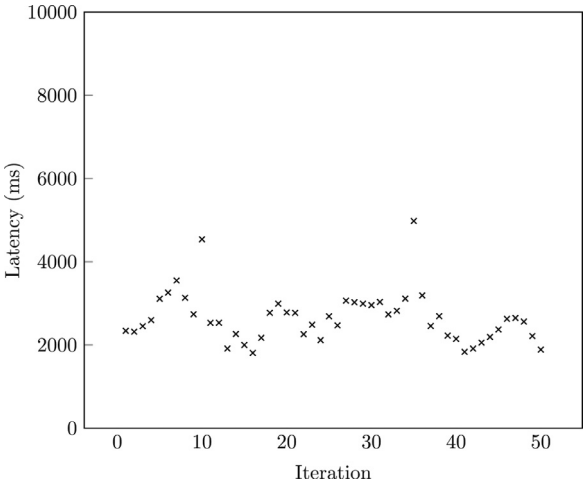

Fig. 17. Execution time.

be executed by multiple threads. Fig. 16 shows how tasks are distributed: the task size is computed by dividing the total number of pixels by the number of threads. The remainder is then distributed among threads to insure maximum fairness.

We conducted some experiments with the goal of evaluating response time of the application. Interestingly, after applying the same filter multiple times on the same picture, we noticed that the execution time is not stable. It fluctuates between 2 and 4 seconds, as shown in Fig. 17.

To dig more, we used our tool to see what is happening exactly in the system during the execution. We activated the userspace tracepoints, as well as the kernel events *sched_switch* and *sched_waiting*, which show when a process is scheduled in and out by the operating system. Fig. 18 shows the different processes running on each of the 8 cores in the system. We can see that Java threads are sharing the CPU cores with other system processes colored in yellow. Java threads are sometimes interrupted by other processes. For example, Thread-2 (TID 16683) was interrupted by Chrome (TID 2836) in Fig. 19.

This execution time variability is not acceptable, since the response time has to be more predictable for the users. The response time should ideally not depend on the other applications running on the same system. To solve this problem, we decided to use the CPU isolation techniques provided by modern systems. We created two *cgroups*: the first contains the Java threads and the other contains the rest of the system processes. Using the *cpuset* command, we associated 4 exclusive cores to the Java application so that they are not disturbed by other processes.

We ran the same experiment again with the new configuration and we used our analysis tool to visualize the behavior of the system. We can see in Fig. 20 that the Java threads are running on cores 0–3, without any interruption by other system processes. The other processes are running on the other CPU cores, which was the desired behavior. Fig. 21 shows that the execution time is much more stable than before.

Solving low-level performance problems like this is not possible just by looking at the source code or analyzing JVM events. The only way to see those details is to have full visibility of the system, including operating system internals.

5.2. Use case 2: File reading latency analysis

Input/Output (I/O) operations are fully managed at the operating system level. For instance, when an application wants to read data from the disk, it uses a read system call to notify the operat-

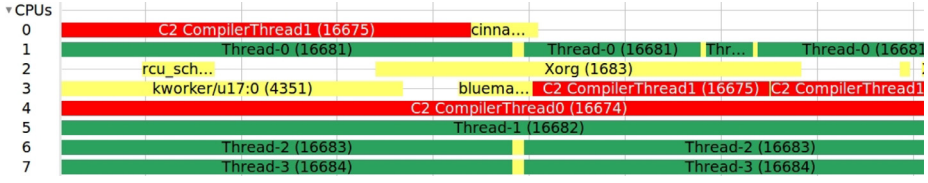

Fig. 18. Java threads are sharing the CPU cores with other applications.

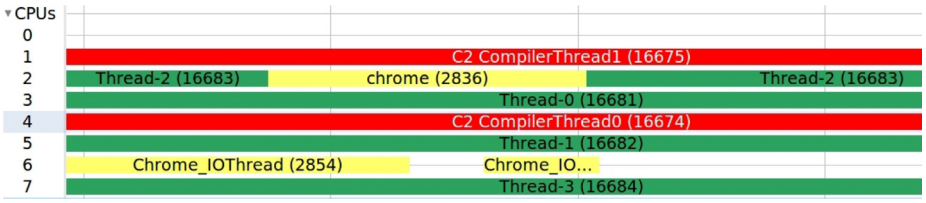

Fig. 19. A Java thread is interrupted by another application.

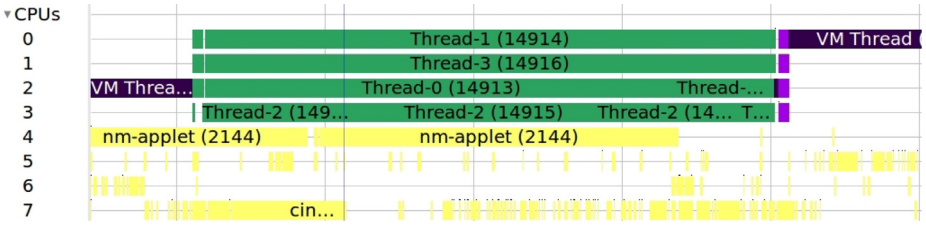

Fig. 20. Java threads are exclusively running on cores 0,1,2,3.

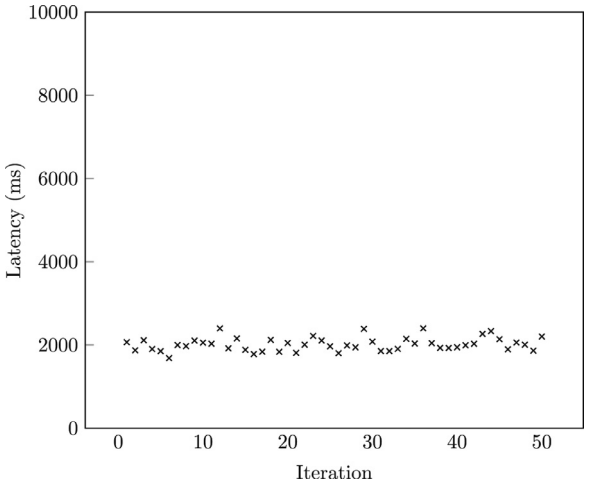

Fig. 21. Execution time after CPU isolation.

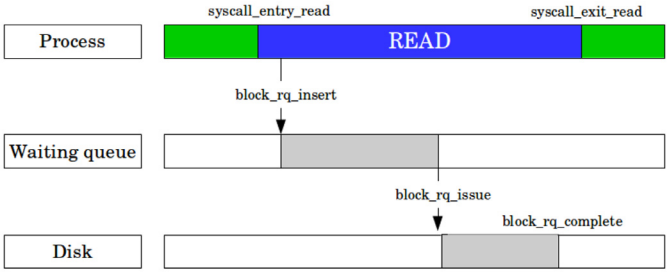

Fig. 22. Disk request events.

ing system. The OS creates a disk request and registers it in the I/O scheduler waiting queue. The disk scheduler then decides which request has to be issued first to the driver for processing. The different trace events involved in a disk operation are presented in Fig. 22.

In this use case, we analyze a Java application that reads multiple files from the hard disk for later processing. This application is I/O bound and the speed of the disk directly affects the performance of the application. We performed a top-down analysis, as explained in the Algorithm 1, to understand how the program works and the factors affecting its performance. The idea is to analyze higher level data to detect latency problems and then go deeper to provide more details.

We started first with the userspace layer to have a clear picture of the application logic. Fig. 23 shows that the program is repetitively calling the *read* function which fills the I/O buffers with data from the disk. We notice that the function sometimes takes a longer time to return.

By tracing the system calls (Fig. 24), we can see that the Java application issues a *read* system call and then is blocked for a certain time until the data becomes available (Fig. 24).

The waiting time is unstable, even for the same buffer size. In order to understand the underlying reason, we decided to perform a lower-level analysis that involves the block layer events: *block_rq_insert, block_rq_issue* and *block_rq_complete*.

Fig. 25 shows the content of the waiting queue of the hard disk throughout time. The requests colored in green correspond to the target Java application and the yellow ones correspond to other processes. We can see that the Java requests are served very quickly when the waiting queue is empty. However, at some point, a process called *update.py* issued a burst of disk requests, which affected the processing time of Java requests. One of the implications that emerge from these findings is that the performance of an application cannot be evaluated independently. It is very important to use a holistic approach that involves the environment on which it is running.

5.3. Use case 3: Analysis of runtime parameters

The JVM offers the possibility to select different runtime options like the collection algorithm, the number of JIT threads, etc. It is sometimes difficult to select the options that fit the application and the system environment. In this section we show how, by visualizing the behavior of the application, the programmer can detect and fix misconfigurations.

5.3.1. Garbage collector options

In the first example, we traced a Java application to see if the garbage collector is working properly. Fig. 26 shows that there are 16 garbage collector threads that are running to perform collection tasks. The blue boxes indicate the name of the GC task executed by each thread. Interestingly, by zooming in, we can see that the threads spend most of their time waiting, they are not all running at the same time (Fig. 27).

The reason for the observed behavior is the fact that the system has only 8 CPU cores, and 16 GC threads were created. Those

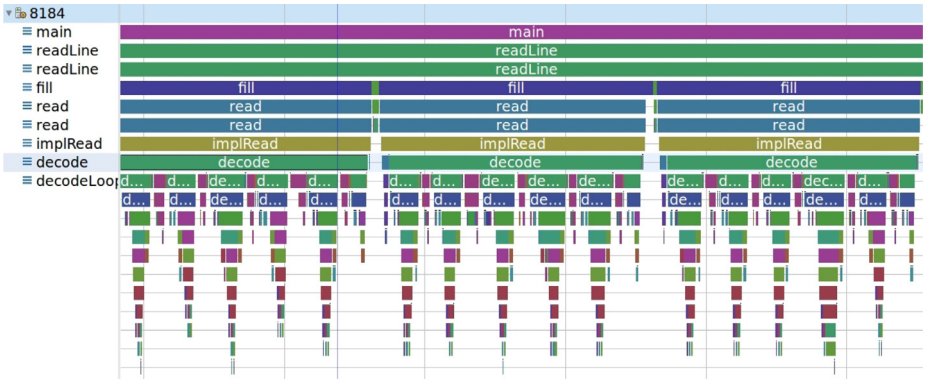

Fig. 23. Userspace reading.

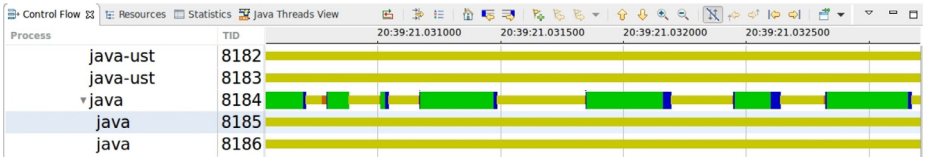

Fig. 24. Reading system calls.

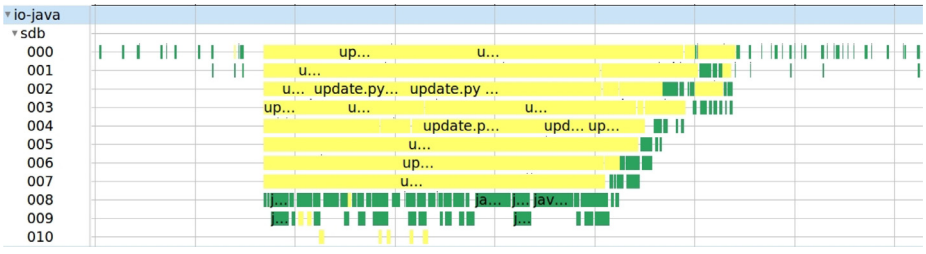

Fig. 25. Disk usage.

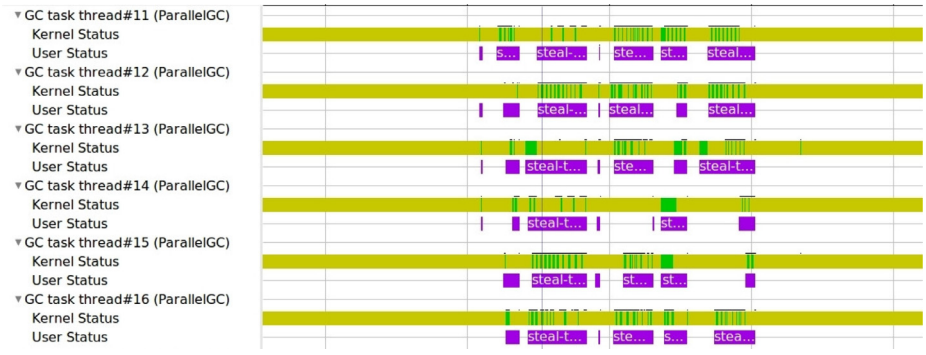

Fig. 26. 16 GC threads are created.

Fig. 27. The GC threads are not always running at the same time.

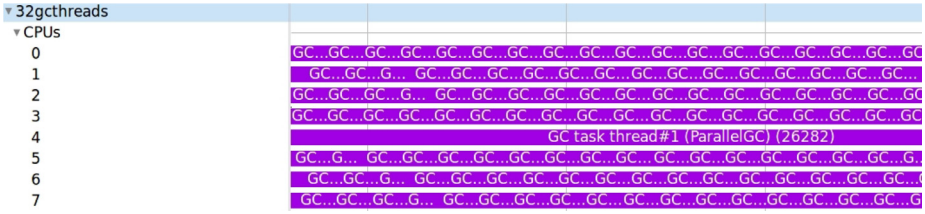

Fig. 28. There is a CPU contention caused by a big number of GC threads.

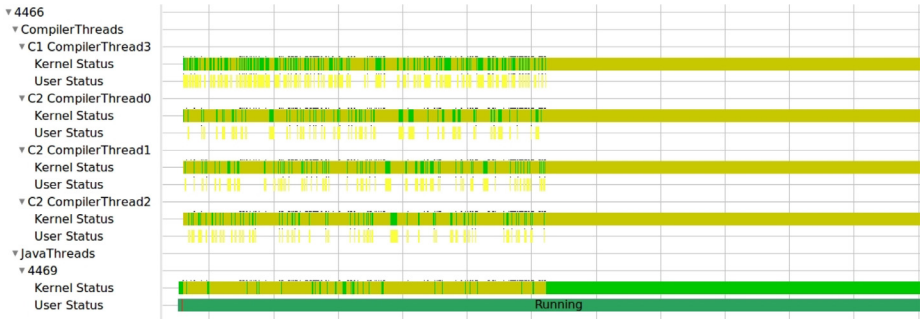

Fig. 29. Execution with *Xcomp* option.

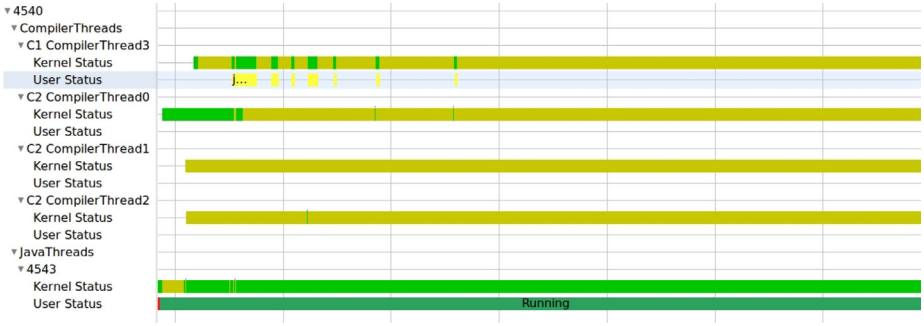

Fig. 30. Execution with regular JIT tiered compilation.

threads cannot all be scheduled at the same time, due to CPU contention. As shown in Fig. 28, GC threads are competing with each other for CPU cores.

We can fix this behavior by using the option *-XX:ParallelGCThreads* to set a more appropriate number of parallel GC threads.

5.3.2. JIT Compiler options

In this subsection, we analyze the startup latency of a Java application. The program takes a very long time to load before running. To detect the problem, we used our tool to observe its behavior from when it is launched until it actually starts running.

Fig. 29 shows that the JIT compiler is very active before the program starts running. This behavior is surprising since the JIT compiler is only supposed to compile some functions after they were called many times, thus minimally disturbing the main program. By looking at the JVM configuration, we discovered that the program was executed with the *XComp* option, which asks the JIT compiler to compile all functions, even if they are not used frequently.

Removing this options dramatically improved the startup time of the application, because the JIT compiler will only compile the frequently executed functions, as shown in Fig. 30.

6. Evaluation

In this section, we evaluate the overhead introduced by our solution. Our goal is to have minimal impact to insure that tracing does not affect the normal behavior of the monitored application. We used DaCapo 2018 (Blackburn et al., 2006), a standard benchmarking suite, to generate different workloads, and we computed for each the tracing overhead and the analysis cost.

6.1. Tracing cost

The tests are executed on an Intel i7-4790 CPU @ 3.60GHz, with 32 GB of main memory and an Intel SSD 530 Series 240 GB hard disk. The system is running Linux Kernel version 4.4 and the traces are gathered using LTTng 2.10.

The experiments are performed with the following configurations:

- *Base:* The tracing is disabled.
- *User* Only userspace tracepoints are enabled
- *Kernel* Only Kernel tracepoints are enabled
- *User+Kernel* Both Kernel and userspace events are enabled.

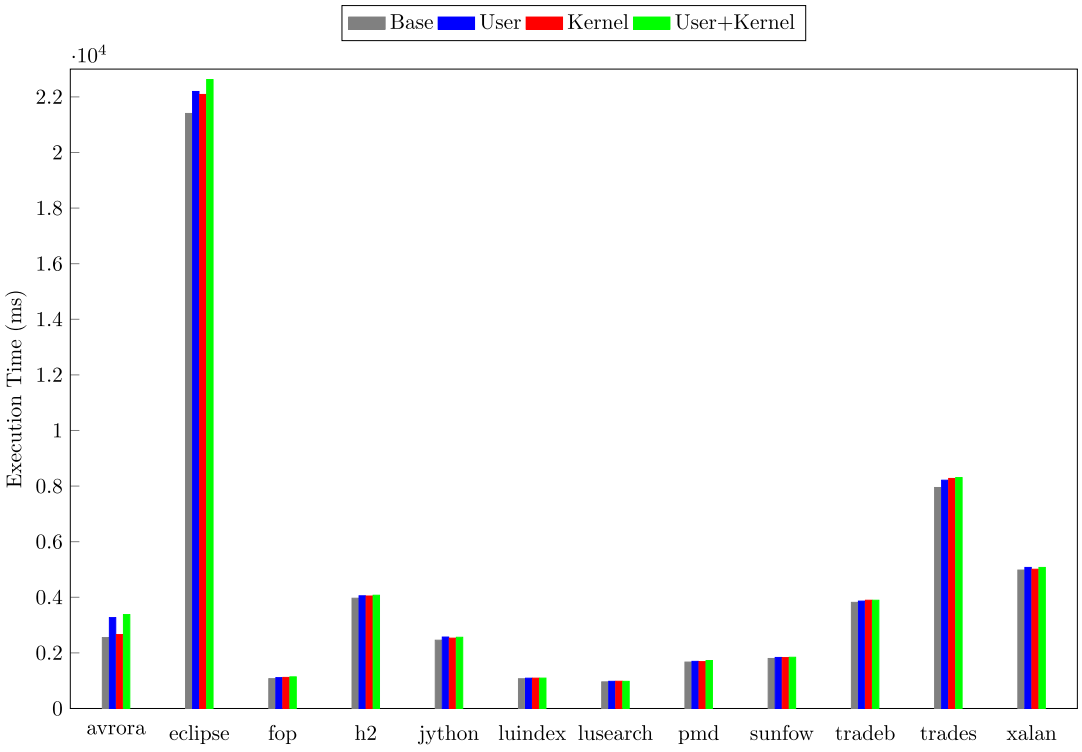

Fig. 31. Impact of tracing on execution time.

Table 7
Impact of tracing on execution time.

Benchmark	Base, ms	Base Std.Dev., ms	User, ms	User Std.Dev., ms	Kernel, ms	Kernel Std.Dev., ms	User+Kernel, ms	User+Kernel Std.Dev., ms	User Overhead,%	Kernel Overhead,%	User+Kernel Overhead, %
avrora	2550	26	3269	17	2659	22	3378	28	28.2	4.3	32.5
eclipse	21,404	207	22,195	210	22,088	183	22,624	192	3.7	3.2	5.7
fop	1073	10	1110	6	1118	10	1138	9	3.5	4.2	6.1
h2	3967	31	4054	37	4046	23	4074	24	2.2	2	2.7
jython	2457	16	2570	24	2530	16	2562	23	4.6	3	4.3
luindex	1070	11	1090	6	1091	8	1093	11	1.9	2	2.2
lusearch	958	6	978	6	980	8	977	8	2.1	2.3	2
pmd	1670	16	1695	9	1686	11	1725	15	1.5	1	3.3
sunflow	1799	16	1836	14	1831	15	1843	13	2.1	1.8	2.5
tradebeans	3818	22	3863	32	3894	33	3894	38	1.2	2	2
tradesoap	7950	51	8212	53	8275	60	8307	63	3.3	4.1	4.5
xalan	4980	48	5074	26	5009	35	5074	45	1.9	0.6	1.9

The results of the experiment are presented in Table 7, Figs. 31 and 32. The graphs show that tracing does not have a big impact on the execution time of the applications. The tracing overhead does not exceed 6.1% for most benchmarks, even when kernel and userspace events are enabled at the same time. This low overhead makes our tool acceptable in production systems.

The only exception happened with the *avrora* workload, where the tracing overhead reached 32.5%. We decided to use our tool to analyze the characteristics of the workload and see the reason for this surprisingly high overhead. Figs. 33 and 34 show the thread states and the event statistics of *avrora*. We can see that many threads (node-0, node-1, etc) are created. The states of those threads are changing at a very high frequency. In fact, those threads are competing to enter critical sections protected by synchronization mechanisms. The event statistics view shows that there are 837,300 *sched_switch* events, most of them being caused by monitors and lock contention events. This very high events frequency is the cause of the considerable tracing overhead. We can consider this kind of workloads as an atypical worst case scenario.

Table 8
Analysis cost.

Benchmark	Trace size (MB)	Analysis time (ms)
avrora	172	88,064
eclipse	13.9	7255
fop	11.7	5990
h2	8.6	4411
jython	7.6	3974
luindex	9.1	4795
lusearch	10.4	5366
pmd	9.2	4820
sunflow	8.8	4558
tradebeans	11.6	6148
tradesoap	8.8	4576
xalan	10.2	5220

6.2. Analysis cost

In this paragraph, we present the disk space and the time required to run the analysis. Table 8 shows the trace size and the

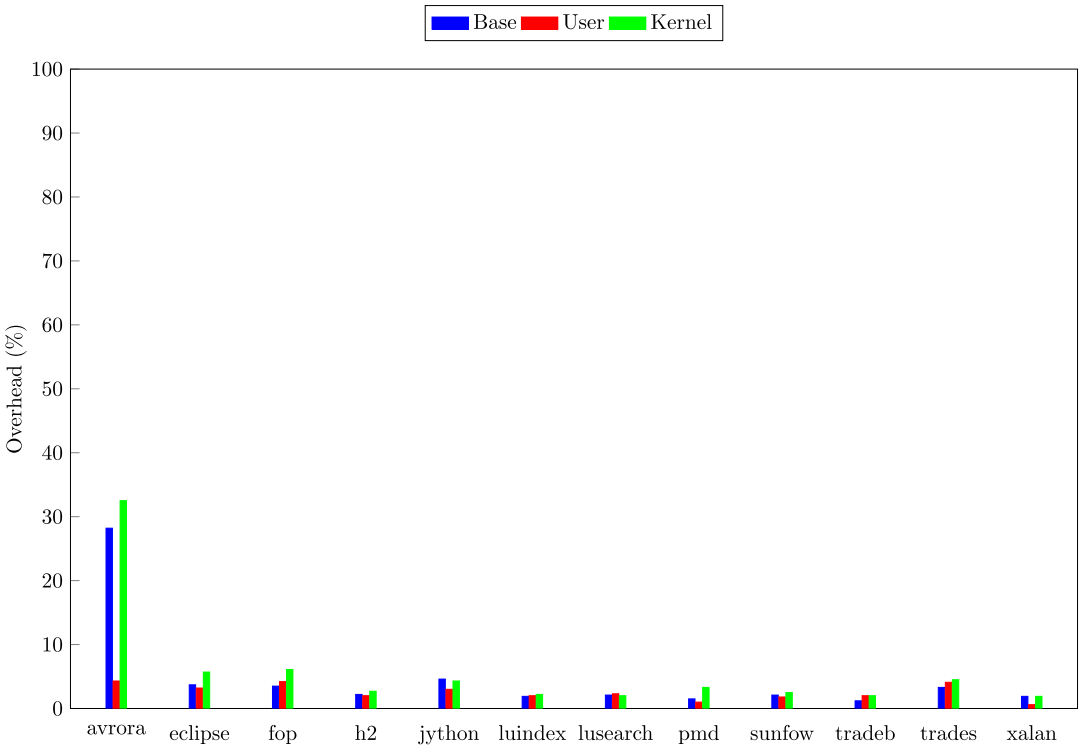

Fig. 32. Tracing overhead.

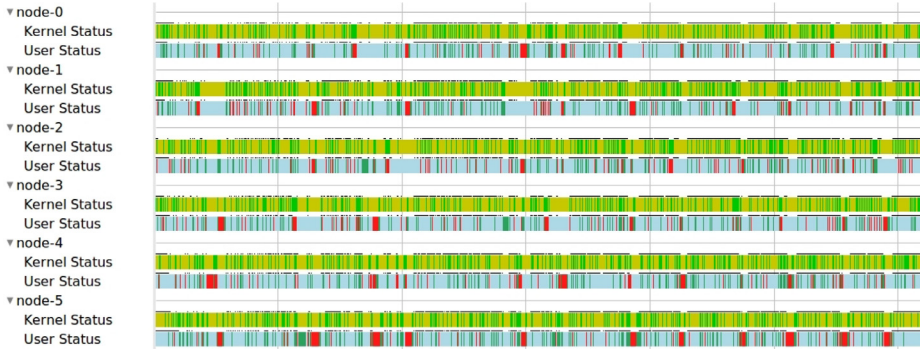

Fig. 33. *avrora* workload threads view.

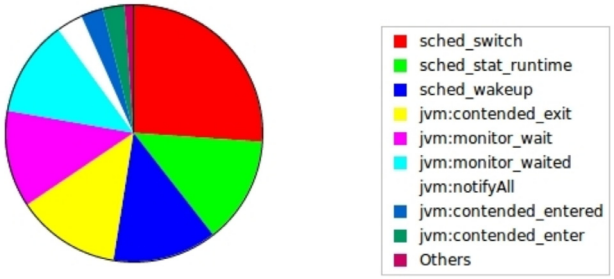

Fig. 34. *avrora* workload event statistics.

analysis time for the different benchmarks presented previously. Results show that the trace size depends on the frequency of events. The trace size does not exceed 10 MB for most benchmarks, except *avrora* which generates a trace file of 178MB.

The analysis time is the time required to read the different events in order to generate the data model. Table 8 shows that the analysis time is a linear function of the trace size. Bigger traces require more time to read and analyze.

7. Conclusion

The Java virtual machine is a very powerful, yet complex application. The internal behavior of the JVM is not fully controlled by the developer and, as a result, many performance issues may happen without being noticed.

Existing analysis tools offer a limited visibility and do not efficiently collect data from the multiple layers of the system.

In this paper, we proposed a multilevel performance analysis tool for Java applications that covers the whole application stack. Tracing is used to collect data from the operating system and from the different JVM components, including the garbage collector and the JIT compiler. The analysis module synchronizes and correlates the traces collected from different sources, based on timestamps and event matching rules. The generated model is saved in a disk-based data structure that can be efficiently accessed afterward to

generate graphical views. The source code of the tool is publicly available on Github[1].

Different use cases are provided to demonstrate that our tool is able solve real problems, which are challenging to diagnose using traditional tools. In addition, we used the *dacapo* benchmarking suite to evaluate the tracing overhead and the analysis cost. The findings showed that the tracing overhead is acceptable.

The proposed framework is subject to two main limitations. First, the tracer used to collect the data is only available on Linux. Porting the framework to another operating system is possible, but it would be major task. Second, the userspace tracepoints required for the analysis are statically linked to OpenJDK. Using our tool with another JVM implementation, such as JRockit and OpenJ9, requires the recompilation of the JVM with the required tracepoints. As a future work, it would be very interesting to investigate the possibility of using dynamic instrumentation techniques to insert tracepoints at runtime. This improvement will make our tool usable with any JVM without the need of a prior instrumentation.

Declaration of Competing Interest

The authors declare that there is no conflict of interests regarding the publication of this paper.

CRediT authorship contribution statement

Houssem Daoud: Conceptualization, Methodology, Software, Writing - original draft. **Michel Dagenais:** Supervision, Writing - review & editing.

Acknowledgments

This research is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), Prompt, Ericsson, Ciena and EfficiOS.

References

Akritidis, P., 2010. Cling: A memory allocator to mitigate dangling pointers.. In: USENIX Security Symposium, pp. 177–192.
Arnold, M., Fink, S.J., Grove, D., Hind, M., Sweeney, P.F., 2005. A survey of adaptive optimization in virtual machines. Proc. IEEE 93 (2), 449–466.
Attanasio, C.R., Bacon, D.F., Cocchi, A., Smith, S., 2001. A comparative evaluation of parallel garbage collector implementations. In: International Workshop on Languages and Compilers for Parallel Computing. Springer, pp. 177–192.
Blackburn, S.M., Cheng, P., McKinley, K.S., 2004. Myths and realities: the performance impact of garbage collection. ACM SIGMETRICS Performance Evaluation Review 32 (1), 25–36.
Blackburn, S.M., Garner, R., Hoffmann, C., Khang, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., et al., 2006. The dacapo benchmarks: Java benchmarking development and analysis. In: ACM Sigplan Notices, 41. ACM, pp. 169–190.
Boehm, H.-J., Demers, A.J., Shenker, S., 1991. Mostly parallel garbage collection. ACM SIGPLAN Notices 26 (6), 157–164. http://www.dorsal.polymtl.ca/fr/blog/yannick-brosseau/userspace-tracing-comparison-dtrace-vs-lttng-ust
Brosseau, Y.,, 2020. A userspace tracing comparison: Dtrace vs LTTng ust. [10 February 2020].
Caballero, J., Grieco, G., Marron, M., Nappa, A., 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In: Proceedings of the 2012 International Symposium on Software Testing and Analysis. ACM, pp. 133–143.
Cass, S., 2017. The 2017 top programming languages. IEEE Spectr.
Desnoyers, M., 2009. Low-impact operating system tracing. École Polytechnique de Montréal.
Desnoyers, M., Dagenais, M.R., 2006. The LTTng tracer: A low impact performance and behavior monitor for gnu/linux. In: OLS (Ottawa Linux Symposium), 2006. Citeseer, pp. 209–224.
Desnoyers, M., McKenney, P.E., Stern, A.S., Dagenais, M.R., Walpole, J., 2011. User-level implementations of read-copy update. IEEE Trans. Parallel Distrib. Syst. 23 (2), 375–382.
Dtrace4linux, 2016. [23 February 2016]. https://github.com/dtrace4linux/linux
Eclipse openj9, 2018. [28 July 2018]. https://www.eclipse.org/openj9/
Eigler, F.C., Hat, R., 2006. Problem solving with systemtap. In: Proc. of the Ottawa Linux Symposium. Citeseer, pp. 261–268.
Ezzati-Jivan, N., Dagenais, M.R., 2012. A stateful approach to generate synthetic events from kernel traces. Advances in Software Engineering 2012, 6.
Gebai, M., Dagenais, M.R., 2014. Virtual machines cpu monitoring with kernel tracing. In: Electrical and Computer Engineering (CCECE), 2014 IEEE 27th Canadian Conference on. IEEE, pp. 1–6.
Gebai, M., Dagenais, M.R., 2018. Survey and analysis of kernel and userspace tracers on linux: design, implementation, and overhead. ACM Computing Surveys (CSUR) 51 (2), 1–33.
Giraldeau, F., Dagenais, M., 2016. Wait analysis of distributed systems using kernel tracing. IEEE Trans. Parallel Distrib. Syst. 27 (8), 2450–2461.
Goulet, D., 2012. Unified Kernel/User-Space Efficient Linux Tracing Architecture. École Polytechnique de Montréal.
Gregg, B., Mauro, J., 2011. Dtrace: Dynamic tracing in oracle solaris, mac OS x, and freebsd. Prentice Hall Professional.
Gregg, B., Spier, M., 2020. Java in flames. [10 February 2020] https://netflixtechblog.com/java-in-flames-e763b3d32166.
Griswold, D., 1998. The java hotspot virtual machine architecture. Sun Microsystems Whitepaper.
Hansen, L.T., 2000. Older-first garbage collection in practice. In: In The 2002 International Symposium on Memory Management (ISMM 2002. Citeseer.
Hastings, R., Joyce, B., 1991. Purify: Fast detection of memory leaks and access errors. In: In Proc. of the Winter 1992 USENIX Conference. Citeseer.
Hertz, M., Blackburn, S.M., Moss, J.E.B., McKinley, K.S., Stefanović, D., 2002. Error-free Garbage Collection Traces: How to Cheat and Not Get Caught. In: Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems. ACM, New York, NY, USA, pp. 140–151.
Hertz, M., Blackburn, S.M., Moss, J.E.B., McKinley, K.S., Stefanović, D., 2006. Generating object lifetime traces with merlin. ACM Trans. Program. Lang. Syst. 28 (3), 476–516. doi:10.1145/1133651.1133654.
Honest profiler, 2018. [28 July 2018]. https://github.com/jvm-profiling-tools/honest-profiler/wiki
Hprof: A heap/cpu profiling tool, 2018. [28 July 2018] https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html.
Jantz, M.R., Kulkarni, P.A., 2013. Exploring single and multilevel jit compilation policy for modern machines 1. ACM Transactions on Architecture and Code Optimization (TACO) 10 (4), 22.
Java, 2018. language and virtual machine specifications. [28 July 2018] https://docs.oracle.com/javase/specs/.
Johnstone, M.S., Wilson, P.R., 1998. The memory fragmentation problem: Solved? In: Acm Sigplan Notices, 34. ACM, pp. 26–36.
Jones, R., Hosking, A., Moss, E., 2016. The garbage collection handbook: The art of automatic memory management. Chapman and Hall/CRC.
Jones, R., Lins, R., 1996. Garbage collection: Algorithms for automatic dynamic memory management, 208. Wiley Chichester.
JVM tool interface, 2018. [28 July 2018] https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.
Knuth, D.E., 1971. An empirical study of fortran programs. Software: Practice and experience 1 (2), 105–133.
Kotzmann, T., Wimmer, C., Mössenböck, H., Rodriguez, T., Russell, K., Cox, D., 2008. Design of the java hotspot client compiler for java 6. ACM Transactions on Architecture and Code Optimization (TACO) 5 (1), 7.
Krishnakumar, R., 2005. Kernel korner: kprobes-a kernel debugger. Linux Journal 2005 (133), 11.
Lee, B., Song, C., Jang, Y., Wang, T., Kim, T., Lu, L., Lee, W., 2015. Preventing use-after-free with dangling pointers nullification.. In: NDSS.
Lengauer, P., Bitto, V., Mössenböck, H., 2015. Accurate and efficient object tracing for Java applications. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering. ACM, pp. 51–62.
Lengauer, P., Bitto, V., Mössenböck, H., Weninger, M., 2017. A comprehensive Java benchmark study on memory and garbage collection behavior of dacapo, dacapo scala, and specjvm2008. In: Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering. ACM, pp. 3–14.
Lieberman, H., Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. Commun ACM 26 (6), 419–429.
Montplaisir, A., Ezzati-Jivan, N., Wininger, F., Dagenais, M., 2013. Efficient model to query and visualize the system states extracted from trace data 8174, 219–234.
Montplaisir, A., Ezzati-Jivan, N., Wininger, F., Dagenais, M., 2013b. State history tree : an incremental disk-based data structure for very large interval data.
Morris, F.L., 1978. A time-and space-efficient garbage compaction algorithm. Commun. ACM 21 (8), 662–665.
Nemati, H., Dagenais, M.R., 2018. Virtflow: guest independent execution flow analysis across virtualized environments. IEEE Trans. Cloud Comput..
Oracle Hotspot, 2018. [28 July 2018] https://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html.
Oracle jrockit, 2018. [28 July 2018] https://www.oracle.com/technetwork/middleware/jrockit/overview/index.html.
Ricci, N.P., Guyer, S.Z., Moss, J.E.B., 2013. Elephant tracks: portable production of complete and precise gc traces. ACM Sigplan Notices 48 (11), 109–118.
Sullins, B.G., Whipple, M., 2002. JMX In action. Manning Publications Co..
The common trace format (CTF), 2018. [28 July 2018]. http://diamon.org/ctf/
The Java monitoring and management console, 2018. [28 July 2018] http://openjdk.java.net/tools/svc/jconsole/.
The libunwind project, 2018. [28 July 2018] http://www.nongnu.org/libunwind/.

[1] https://github.com/houssemmh/tracecompass-inclubator-thesis

Toupin, D., 2011. Using tracing to diagnose or monitor systems. IEEE Software 28 (1), 87.
Visualvm, 2018. [28 July 2018] https://visualvm.github.io/.
Wang, Y., Kent, K.B., Johnson, G., 2015. Improving j9 virtual machine with LTTng for efficient and effective tracing. Software: Practice and Experience 45 (7), 973–987.

Houssem Daoud recieved his Master and PhD degree in computer engineering from Ecole Polytechnique de Montreal. His main interests are operating systems, parallel computing and low latency programming.

Michel Dagenais is a professor at Ecole Polytechnique de Montreal in Computer and Software Engineering. He authored or co-authored over one hundred scientific publications, as well as numerous free documents and free software packages in the fields of operating systems, distributed systems, and multicore systems, particularly in the area of tracing and monitoring Linux systems for performance analysis. In 1995–1996, during a leave of absence, he was the director of software development at Positron Industries and chief architect for the Power911, object oriented, distributed, fault-tolerant, call management system with integrated telephony, and databases. The Linux Trace Toolkit next generation, developed under his supervision, is now used throughout the world and is part of several specialised and general purpose Linux distributions.