W. Karl, J. Keller (Eds.): PARS 2017
Proc. 27th PARS-Workshop

Tracing of Multi-Threaded Java Applications in Score-P Using JVMTI and User Instrumentation

Jan Frenzel[1], Kim Feldhoff[2], René Jäkel[3], Ralph Müller-Pfefferkorn[4]

Abstract:

Over the past years, parallel Java applications received a substantial boost in the research field of High Performance Computing, especially in the field of Big Data Analytics by the development of Java-based frameworks, i. e., Apache Hadoop, Flink or Spark, amongst others, for processing large-scale datasets. Analyzing the performance of said Big Data frameworks in particular, and multi-threaded Java applications in general, is indispensable for efficient execution. Due to the high number of threads, this requires a scalable runtime performance measurement infrastructure. The established, open-source tracing framework Score-P provides such an infrastructure, but did not support (parallel) Java applications, previously. We added support for tracing multi-threaded Java applications to Score-P by implementing two instrumentation approaches. The first instrumentation approach is based on the Java Virtual Machine tool interface (JVMTI) and allows to easily trace an application without source code modifications. The second instrumentation approach allows to manually modify sources via API functions such that only those parts of an application are recorded which the user is interested in. Both instrumentation approaches were successfully applied to the LU kernel of the established Java benchmark suite SPECjvm2008 at a modern HPC machine. We show the quality of the implementations by determining the tracing overheads of the instrumented versions for different test scenarios using varying numbers of Java threads, and thus, varying numbers of recorded events.

Keywords: Tracing, Java, Multi-Threaded, Score-P, Performance Analysis, Profiling, Instrumentation, JVMTI

1 Introduction

The programming language Java received a substantial boost in the research field of High Performance Computing mainly due to features like networking and multi-threading support [TRE+13], [SB01], [BGS06].

[1] Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, Germany, 01062 Dresden, jan.frenzel@tu-dresden.de
[2] Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, Germany, 01062 Dresden, kim.feldhoff@tu-dresden.de
[3] Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany, rene.jaekel@tu-dresden.de
[4] Center for Information Services and High Performance Computing (ZIH), Technische Universität Dresden, 01062 Dresden, Germany, ralph.mueller-pfefferkorn@tu-dresden.de

The development of Big Data frameworks like Flink[5], Hadoop[6], or Spark[7] increased the usage of Java to process large-scale datasets in parallel in the past years. Due to their generic nature, these frameworks leave room for performance improvements for particular use cases or environments, e. g., when Remote Direct Memory Access (RDMA) is available [IRJ+12]. However, we perceive that their performance has not been systematically analysed up to now. Performance analysis of programs written with these frameworks is often based on measuring their runtimes, seldom on profiling [HB11]. Thus, the current research within this field focuses on new methods for debugging and recording [Lei14]. In general, the performance analysis of parallel Java applications is an active research field.

Profiling can provide statistics wrt. function runtimes accumulated over the total runtime of an application and can already help to find bottlenecks. However, it introduces some recording overhead. Tracing retains each event of each thread with its timestamp and provides more details, and can therefore be considered to be better suited to find dependencies between threads, changed behavior caused by transient noise or skew during processing. Unfortunately, tracing influences the runtime behavior even more than profiling since more information is collected.

To overcome the challenge of recording Java applications utilizing many threads, a scalable performance measurement runtime infrastructure is required. Furthermore, to serve the most common pattern of use, this infrastructure should require no or only minimal code modifications and has to be applicable fast. For more advanced measurements, a more fine-grained level of recording performance data should be offered to users which can be used to manually select code regions of interest.

Our paper is structured as follows. Sect. 2 gives an overview about related work and summarizes the main goals. Sect. 3 describes the methods needed to generate events during the program's execution, Sect. 4 describes the evaluation of the implementations and the obtained results. Sect. 5 concludes this paper and sketches our future work.

2 Related work and main goal

Many proprietary and open-source software tools are available for analyzing the performance of Java applications based on profiling and tracing. VisualVM[8] provides CPU and memory profiling, HPROF can profile the heap. BTrace[9] can apply user-defined trace scripts to Java programs. InTrace[10] only adds output instructions to programs. A tool for monitoring is jMonitor which allows to add user-defined tracing logic to applications at bytecode level [KF05]. DynaTrace[11] is an established, but proprietary profiling and tracing tool. VampirTrace [JBK+07] is open-source and capable to trace multi-threaded Java

[5] http://flink.apache.org
[6] http://hadoop.apache.org
[7] http://spark.apache.org
[8] https://visualvm.github.io/
[9] https://kenai.com/projects/btrace
[10] http://mchr3k.github.io/org.intrace
[11] https://www.dynatrace.com/technologies/java-monitoring

applications using an JVMTI[12]-based instrumentation approach. More popular profiling tools are AppPerfect Java Profiler or Eclipse Memory Analyzer. SLF4J[13] provides an extension for basic profiling. However, many of these tools lack support for collecting and storing large traces for later analysis with other visualization tools, are not suited for multi-threaded applications, or require to change the code, when switching from profiling to tracing. hTrace[14] and Zipkin[15] follow the idea of Dapper [SBB+10] and provide an API for user-defined trace generation and collection. However, both frameworks require to run an additional service for collecting events. Additionally, Dapper's intended use is to only trace requests and responses. IBM JTrace[16] allows the manual instrumentation of Java applications via a given set of API functions such that JVM internal methods, applications, and Java methods can be traced.

We decided to implement the proposed instrumentation approaches within the Score-P framework [KRM+12] for the following reasons: Score-P is an established, open-source profiling and tracing framework[17]. It is open-source and runs on modern HPC machines with the possibility to implement extensions to it. In particular, an intent during design was to unify existing measurement environments such that various analysis and visualization tools, like Vampir[KBD+08][18], Cube[19], Tau[20] or Periscope Tuning Framework[21] are supported. Furthermore, the design of Score-P allows that added instrumentation approaches can reuse all basic features of Score-P like scalability, trace collection or trace file compressions.

Summarizing, the main goal of this paper is to present and discuss the support for profiling and tracing of multi-threaded Java applications to the performance measurement infrastructure Score-P by implementing two instrumentation approaches.

3 Implementation of instrumentation types in Score-P

Estimating the behavior of a program by tracing or profiling techniques requires to generate, collect, and store runtime events. Event collection and storage is common for all instrumentation approaches and therefore the core of Score-P. Only the event generation method is specific to the measurement environment and needs to adapt events for the rest of the Score-P framework. Thus, we focused on the generation of events, which can be done

[12] http://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html
[13] https://www.slf4j.org/extensions.html
[14] http://htrace.apache.org
[15] http://zipkin.io
[16] https://www.ibm.com/support/knowledgecenter/SSYKE2_7.0.0/
[17] http://www.vi-hps.org/projects/score-p/
[18] http://www.vampir.eu
[19] http://www.scalasca.org
[20] http://www.cs.uoregon.edu/Research/tau/home.php
[21] http://periscope.in.tum.de/

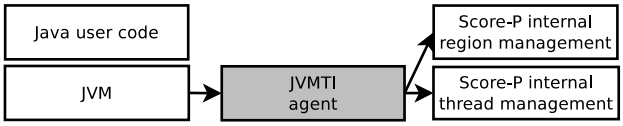

Fig. 1: Design of the JVMTI-based instrumentation. As the user code executes on the JVM, callbacks of the JVMTI agent are invoked triggering appropriate actions in the Score-P framework. The new software component is shown in gray.

1. externally, by using the unmodified source code and external tools which generate events at runtime (e. g., via interpreter programs, library substitution, or other means available in the execution context) and
2. internally, where the source code regions are annotated to generate tracing events.

We propose methods for external and internal event generation using an instrumentation approach based on the Java Virtual Machine Tools Interface (JVMTI) (Subsection 3.1) and an approach based on Java API methods (Subsection 3.2), i. e., users have to add method calls to their source code to generate entry and exit events for each region they are interested in.

Recording method entries and exits requires the creation of events for entering and leaving the methods. Entry and exit events have to be linked with correct thread information, i. e., Score-P locations. Therefore, events for each thread start have to be created. Score-P uses these thread start events to initialize the location. Later, this location can be passed to methods which create events for region entry and exit events. For visualization purposes, we annotate the location with the thread's name like it is specified in the Java source code. Regions can be annotated with e. g. method name, source file name, and line numbers.

3.1 Instrumentation using JVMTI

The first proposed instrumentation approach is based on the Java Virtual Machine Tools Interface and can be used to automatically record events for code regions of a program at runtime. The JVMTI specification describes how a shared library, a so-called JVMTI agent, is loaded into the Java Virtual Machine (JVM) and how functions of that library can be registered as callbacks for various JVM events. The JVMTI agent receives calls at runtime which are used to generate events for the measurement system. Callback parameters contain additional information about the JVM event. The required thread context for the callback is stored in thread local storage. Fig. 1 shows the details of our design.

Callbacks are registered for the entry and exit of methods, thread start and end, garbage collection start and finish, object allocation and freeing. A detailed list of all registered callbacks and their corresponding responsibilities is shown in Tab. 1.

Filtering rules can be used to steer the collection of events. This can be necessary when a user is only interested in some regions, e. g. wants to ignore all methods in the `java.lang` package, or wants to reduce the overhead of collecting and storing too many events. A filter rule applies to either region names, source file names, or thread names and decides

JVM event	Callback responsibilities
Method Entry	Records an entry event for the method. Additionally, registers the method when called for the first time.
Method Exit	Records an exit event for the method.
Thread Start	Registers the thread as a new location stored in thread local storage.
Thread End	Marks the end of the thread.
Garbage Collection Start	Sets the garbage collection metrics to true.
Garbage Collection Finish	Sets the garbage collection metrics to false.
Object Allocation	Adds the size of the allocated object to the object allocation metrics, increments the number of allocations.
Object Free	Subtracts the size of the object from the object allocation metrics, decrements the number of allocations.

Tab. 1: JVM events and description of corresponding callback code.

whether events for that region or thread are collected. To simplify the process of specifying filters, users are allowed to use the wild card characters `?` and `*` matching any character and strings of arbitrary length including the empty string, respectively. Filter rules can be established by setting the environment variable `SCOREP_FILTERING_FILE` to the path of a file containing these filter rules. If this variable is not set, a set of default filter rules is automatically applied.

When a new region is registered, its name and the name of the source file containing that region are checked against the filter rules. If the region should be filtered out, any events for that region are discarded, so that no events for that region appear in the trace. Filtering by location is implemented in a similar fashion. When a new thread has been started, the thread start callback receives a handle of that thread so it can check whether the filter accepts the thread's name or not. If the filter allows recording of events for this thread, a new location is created and stored in the thread local storage. When a method entry or exit has been registered, the thread local storage of the current thread has to be accessed to get the corresponding location. If a location is present in thread local storage, the event can be recorded for that location. Otherwise, method entry or exit events are not recorded.

In summary, the JVMTI-based instrumentation approach offers the following benefits: For the user, instrumentation is easy, as no source file modifications are required. The memory size of classes and objects remains the same. Additional metrics, e. g., garbage collection times, can be easily obtained. Tracing a different set of methods requires to change the filter rules and rerun the program, but no recompilation is necessary. However, the approach is limited to the method level, and filtering at runtime is required to exclude many events (of core classes).

Further event generation, e. g., for synthetic or even native methods or object allocations, can be controlled by setting the environment variable `SCOREP_JAVA_ENABLE` to `synthetic`, `native`, or `memtrace`, respectively. Combinations of these values are also possible.

3.2 Java user instrumentation

Java user instrumentation, the second proposed instrumentation approach, allows users to record only selected code regions of their applications using a set of API functions. These functions invoke procedures of the measurement system directly, i. e., call internal region and thread handling procedures of Score-P via the Java Native Interface (JNI)[22]. The design is shown in detail in Fig. 2. The user code calls functions of the proposed Score-P Java API (in gray), which uses JNI to send information to the backend of Score-P. If the user code calls C, C++ or Fortran functions, these native functions can use Score-P's native API to collect further events. The API for the programming languages C, C++, and Fortran (native) is also depicted.

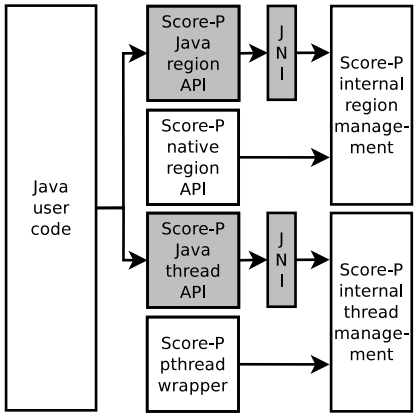

Fig. 2: Design of the Java user instrumentation approach: API functions call internal Score-P functions via the Java Native Interface. Newly developed software components are shown in gray boxes.

Essentially, users only have to use two Java functions for marking code regions. The start and end of a selected region can be marked by adding a call to the API function `enterRegion` and `leaveRegion`, respectively. To distinguish events of different threads, users have to wrap `java.lang.Thread` objects with `scorep.Thread` objects. The complexity of handling and passing around thread context objects is then hidden from the user. The thread context is stored in thread local storage and automatically used by `enterRegion` and `leaveRegion`.

The Java user instrumentation method is illustrated for the simple Java class `Foo`. List. 1 shows the source code of the original, non-instrumented class `Foo`. List. 2 shows the source code of the Java user instrumented version of the method `Foo.bar`. The Score-P instrumentation class must be imported (step 1), the instrumentation environment has to be initialized (step 2) before any other method is called. Additionally, the source file (step 3) and all regions (step 4) have to be registered before their use. Steps 1 to 4 have to be done only once per source file. Then, every code region that should be included in the measurement has to be surrounded by a `try-finally` clause (steps 5 and 6) to ensure that there is a corresponding call to `leaveRegion` for each call to `enterRegion`, since exceptions can

[22] https://docs.oracle.com/javase/8/docs/technotes/guides/jni/

```
public class Foo extends Thread {
    public void bar() {
        // [...] Original code of method 'bar' has to be added here.
    }
    @Override public void run() {
        bar();
    }
    public static void main(String[] args) {
        // Normal method body.
        new Foo().start();
    }
}
```

List. 1: Original, non-instrumented version of the example class `Foo`.

```
import scorep.Instrumentation; // Step 1. Import Score-P instrumentation class
    .
public class Foo extends Thread {
    static {
        Instrumentation.initialize(); // Step 2. Initialize the environment.
    }
    private static final long srcFileHandle = Instrumentation.
        getFileHandleOfThisSourceFile(); // Step 3. Register the source file.
    private static final long barRegion = Instrumentation.prepareRegion("Foo.
        bar()", srcFileHandle, 8, 15); // Step 4. Register the region.
    public void bar() {
        Instrumentation.enterRegion(barRegion); // Step 5. Enter region.
        try {
            // [...] Original code of method 'bar' has to be added here.
        } finally {
            Instrumentation.leaveRegion(barRegion); // Step 6. Leave region.
        }
    }
    @Override public run() {
        bar(); // not instrumented
    }
    public static void main(String[] args) {
        new scorep.Thread(new Foo()).start(); // Step 7. Wrap the thread.
    }
}
```

List. 2: Application of the Java user instrumentation to the method `Foo.bar`.

appear during the execution of normal code. Threads can be instrumented by using wrapper objects (step 7). Cleanup is automatically done via a shutdown hook thread, which is registered in method `Instrumentation.initialize` (not shown).

Once, all code regions have been instrumented, the corresponding source files can be compiled and executed as usual. Users only have to ensure that the Score-P instrumentation library in terms of the JAR file `scorepInstrumentation.jar` can be found at the class path, compare List. 3. Compared to the JVMTI-based approach, benefits of the Java user instrumentation approach can be summarized as follows: Users can add event generation to an arbitrary collection of statements and are not limited to the method level. Any sequence of statements or expressions can be chosen. Only those code regions marked by the user will be recorded. There is no need to specify filter rules. Additionally, there is no overhead of runtime filter checking. However, the approach shows the following limita-

```
javac -cp .:scorepInstrumentation.jar <sourcefile> # Compilation
java  -cp .:scorepInstrumentation.jar <classname>  # Execution
```

List. 3: Compilation of a Java user source file `<sourcefile>` and the execution of the corresponding class `<classname>`.

tions: Source code line numbers cannot automatically be obtained at runtime. Especially, a region has to be defined before it is visited, because Score-P uses region handles to reuse region definitions with their annotations. A workaround is to allow users to specify the line numbers manually. However, line numbers can change many times because performance analysis is mainly done during the software development process. In addition, events from JVM internal threads like `Destroy-Java-VM` or `Signal-Dispatcher` cannot easily be collected during the measurement. Several metrics, such as garbage collection activity or object allocations are not included, because they would require adaptation of Java internal classes. Constructors can also be only instrumented partially as the call to the constructor of the parent class must be the first statement in a constructor.

4 Overhead estimation of performance measurements

To demonstrate the applicability of the two proposed instrumentation approaches, we investigated the tracing overhead of a reference application. We have chosen the thread-parallel, scientific kernel *scimark.lu.small* of the established open-source benchmark suite SPECjvm2008 v1.01[23] developed by the Standard Performance Evaluation Corporation. Each thread of the kernel performs a prescribed number of LU decompositions, i. e., a matrix will be decomposed into the product of a lower triangular matrix and an upper triangular matrix.

We measured the tracing overhead in terms of wall clock times needed for the execution of each instrumented program version. According to [MK07], the tracing overhead is mainly dependent on the trace buffer size and on the number of recorded events, whereby the overhead for writing the trace buffer increases with increasing number of processors. Thus, we have chosen a fixed trace buffer size of 4 GB, such that the buffer was not flushed during the recording process to avoid additional hard disk writes. For scalability tests, we varied the number of used threads and with that the number of recorded events.

We compared the tracing overhead wrt. the uninstrumented version (“none”) for the following instrumentation types:

1. JVMTI-based instrumentation implemented in VampirTrace v.5.14.4 (“vt-jvmti”),
2. JVMTI-based instrumentation implemented in Score-P (“scorep-jvmti”),
3. Java user instrumentation implemented in Score-P (“scorep-user”).

[23] https://www.spec.org/jvm2008/

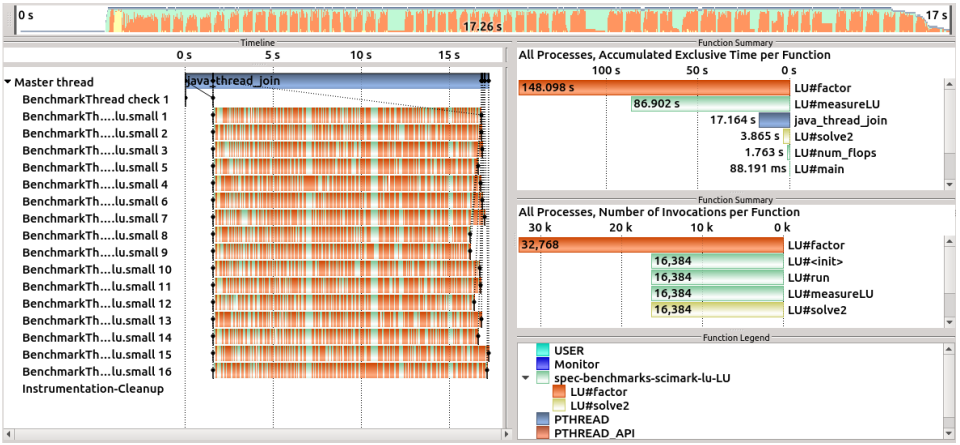

Fig. 3: Screenshots of Vampir displays showing the process summaries of all recorded Java threads (left), the accumulated exclusive times of all recorded methods (top right), the number of invocations of all recorded methods (middle right), and the legend of all methods and methods groups (bottom right) for the kernel *scimark.lu.small* executed at HPC machine Venus with 16 Java threads and instrumented using "scorep-user" with filter rules set "compareJvmtiToUser".

Used set of filter rules: "compareJvmtiToUser"									
n_T	n_E	n_L	"none"	"vt-jvmti"		"scorep-jvmti"		"scorep-user"	
			–	Prof.	Trac.	Prof.	Trac.	Prof.	Trac.
1	8,197	4	10.65	382.61	381.30	406.11	405.20	10.72	10.92
8	65,555	11	11.20	1,478.57	1,482.81	1,579.06	1,436.46	11.71	11.75
16	131,107	19	11.37	—	—	—	—	11.42	11.73
32	252,211	35	12.11	—	—	—	—	12.28	12.51
64	524,419	67	13.75	—	—	—	—	14.12	15.30
128	1,048,835	131	19.24	—	—	—	—	19.47	20.95
256	2,097,667	259	24.61	—	—	—	—	25.21	26.92

Tab. 2: Wall clock times in seconds for profiling ("Prof.") and tracing ("Trac.") the kernel *scimark.lu.small* executed on Venus using the instrumentations "none", "vt-jvmti", "scorep-jvmti", "scorep-user", using the filter rules set "compareJvmtiToUser", and specifying different numbers of used threads n_T for the kernel. n_E indicates the number of recorded events, n_L the number of actually recorded threads.

All tests were executed on the HPC machine Venus[24], a shared-memory system of the ZIH. The machine contains of 512 Sandy Bridge cores and has a total of 8 TiB shared memory. The OpenJDK 8 Runtime Environment[25] was used for compiling and executing the kernel. For the comparison of all three instrumentation types, the set of filter rules for the JVMTI instrumentations had to be adapted such that these instrumentations record the same events as the Score-P Java user instrumentation. The adapted set of filter rules is named "compareJvmtiToUser". Fig. 3 shows a screenshot of the performance analysis and visualization tool Vampir displaying recorded performance data of the instrumenta-

[24] https://doc.zih.tu-dresden.de/hpc-wiki/bin/view/Compendium/SystemVenus
[25] http://openjdk.java.net/

tion type “scorep-user”. Tab. 2 compares the instrumentation types for different thread teams. As expected, the Java user instrumentation performs better than the JVMTI-based instrumentations “vt-jvmti” and “scorep-jvmti”, and these instrumentations show similar profiling and tracing overheads. This can be explained by the checks of the specified set of filter rules at runtime which are not needed for the Java user instrumentation. These checks have to be done for each method, including the method calls of the Java core classes. Since the overhead is already large for a small number of threads, we did not measure the runtimes of the instrumented program versions for more than 8 threads. In contrast, the Java user instrumentation does not require filter checking and introduces a comparably low runtime overhead, even for large numbers of threads. The runtime overhead for profiling is less than 5 percent, for tracing less than 15 percent. In particular, the profiling overhead is smaller than the tracing overhead. Fig. 4 visualizes of the overall runtimes of the uninstrumented version (“none”) and the Java user instrumented code (“scorep-user”) with profiling or tracing enabled.

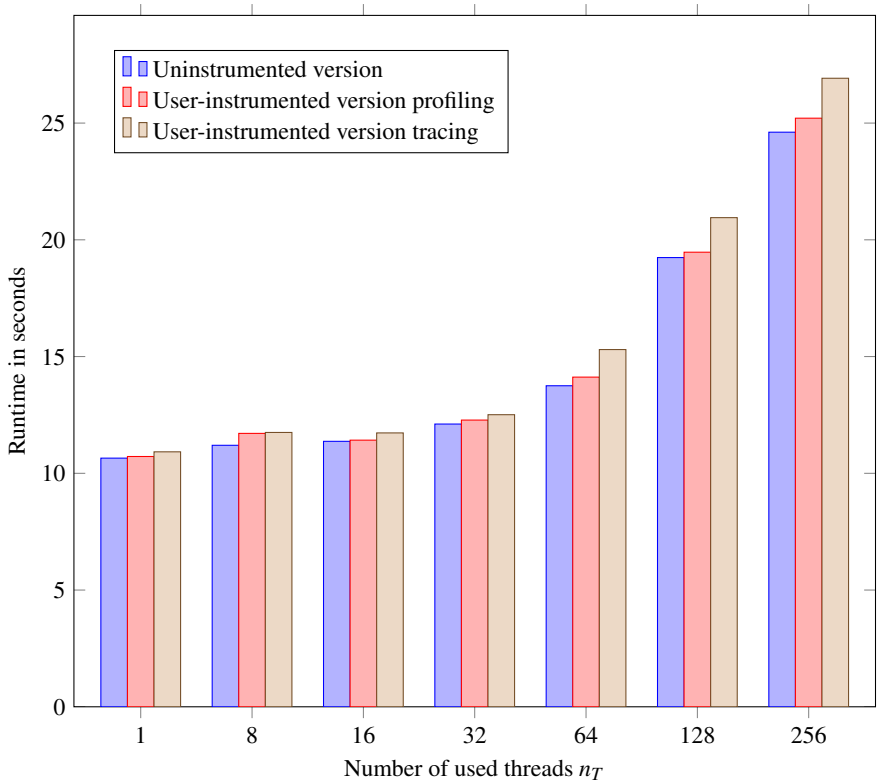

Fig. 4: Visualization of the overall runtimes of the uninstrumented version (blue boxes) and the Java user instrumented code with profiling (red boxes) or tracing (yellow boxes) enabled for different numbers of used threads n_T.

5 Conclusions and future work

We presented two instrumentation approaches for performance evaluation of Java applications and successfully applied our implementation to a thread-parallel benchmark. Generated trace files include thread, method, object allocation, and garbage collection events forming an integrated view of the runtime behavior of a thread-parallel Java application. Besides, the filter mechanism allows users to specify which methods and threads should be recorded in a trace file. This simplifies choosing the event granularity level: Users can decide on their own whether they want to see only long but seldom called methods or small but frequently called methods. The test results related to computed tracing overheads show that the JVMTI-based instrumentation type can be used to get an first overview over all methods within a Java application for a small number of threads. Once, hot spots have been identified, these can be further analyzed at a larger scale using the Java user instrumentation.

In the future, we plan to extend our approach with methods to automatically inject calls from user code to our API, making manual instrumentation unnecessary. This requires either source-to-source transformations or bytecode manipulations.

The advances made by our work presented here are related to the support of tracing multi-threaded Java applications and serve as a first step into the direction of analyzing the performance of Big Data frameworks. Thus, further investigations are required to extend the Java tracing support towards measuring distributed frameworks such as Flink, Hadoop, and Spark. Challenges are multi-process event generation and distributed fast event collection. For example, if an application is distributed over various data centers the time measurements might differ. Therefore, the timestamps for the events might differ also which complicates trace generations or interpretations.

Acknowledgements

This work was supported in parts by the Sächsische Aufbaubank (SAB) via the research project “PTPT Workbench” (order number 100220482) and by the German Federal Ministry of Education and Research (BMBF, 01IS14014A-D) by funding the competence center for Big Data “ScaDS Dresden/Leipzig”. Additionally, we want to thank Ronny Tschüter (TU Dresden, ZIH) and Bert Wesarg (TU Dresden, ZIH) for their introduction to the Score-P framework and their pointers to already available functionality in Score-P.

References

[BGS06] Mark A. Baker, Matthew Grove, and Aamir Shafi. Parallel and Distributed Computing with Java. In *Proceedings of The Fifth International Symposium on Parallel and Distributed Computing*, ISPDC ’06, pages 3–10, Washington, DC, USA, 2006. IEEE Computer Society.

[HB11] Herodotos Herodotou and Shivnath Babu. Profiling, What-if Analysis, and Cost-based Optimization of Mapreduce Programs. *Proceedings of the VLDB Endowment*, 4(11):1111–1122, 2011.

[IRJ+12] N. S. Islam, M. W. Rahman, J. Jose, R. Rajachandrasekar, H. Wang, H. Subramoni, C. Murthy, and D. K. Panda. High Performance RDMA-based Design of HDFS over InfiniBand. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 35:1–35:35, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[JBK+07] Matthias Jurenz, Ronny Brendel, Andreas Knüpfer, Matthias Müller, and Wolfgang E. Nagel. *Memory Allocation Tracing with VampirTrace*, pages 839–846. Springer, Berlin, Heidelberg, 2007.

[KBD+08] Andreas Knüpfer, Holger Brunst, Jens Doleschal, Matthias Jurenz, Matthias Lieber, Holger Mickler, Matthias S. Müller, and Wolfgang E. Nagel. The Vampir Performance Analysis Tool-Set. In Michael Resch, Rainer Keller, Valentin Himmler, Bettina Krammer, and Alexander Schulz, editors, *Tools for High Performance Computing*, pages 139–155. Springer Berlin Heidelberg, 2008.

[KF05] Murat Karaorman and Jay Freeman. jMonitor: Java Runtime Event Specification and Monitoring Library. *Electronic Notes in Theoretical Computer Science*, 113:181 – 200, 2005.

[KRM+12] Andreas Knüpfer, Christian Rössel, Dieter an Mey, Scott Biersdorff, Kai Diethelm, Dominic Eschweiler, Markus Geimer, Michael Gerndt, Daniel Lorenz, Allen Malony, Wolfgang E. Nagel, Yury Oleynik, Peter Philippen, Pavel Saviankou, Dirk Schmidl, Sameer Shende, Ronny Tschüter, Michael Wagner, Bert Wesarg, and Felix Wolf. *Score-P: A Joint Performance Measurement Run-Time Infrastructure for Periscope,Scalasca, TAU, and Vampir*, pages 79–91. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Lei14] Marcus Leich. Runtime Analysis of Distributed Data Processing Programs. *Proceedings of the VLDB Phd Workshop*, 2014.

[MK07] Kathryn Mohror and Karen L. Karavanic. A Study of Tracing Overhead on a High-performance Linux Cluster. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2007, San Jose, California, USA, March 14-17, 2007*, pages 158–159, 2007.

[SB01] L. A. Smith and J. M. Bull. Java for High Performance Computing, 2001. UKHEC Technology Watch Report.

[SBB+10] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a Large-scale Distributed Systems Tracing Infrastructure. 2010.

[TRE+13] Guillermo L. Taboada, Sabela Ramos, Roberto R. Expósito, Juan Touriño, and Ramon Doallo. Java in the High Performance Computing arena: Research, practice and experience. *Sci. Comput. Program.*, 78(5):425–444, 2013.