Analysis and Optimization of Task Granularity on the Java Virtual Machine

ANDREA ROSÀ, EDUARDO ROSALES, and WALTER BINDER,
Università della Svizzera italiana

Task granularity, i.e., the amount of work performed by parallel tasks, is a key performance attribute of parallel applications. On the one hand, fine-grained tasks (i.e., small tasks carrying out few computations) may introduce considerable parallelization overheads. On the other hand, coarse-grained tasks (i.e., large tasks performing substantial computations) may not fully utilize the available CPU cores, leading to missed parallelization opportunities. In this article, we provide a better understanding of task granularity for task-parallel applications running on a single Java Virtual Machine in a shared-memory multicore. We present a new methodology to accurately and efficiently collect the granularity of each executed task, implemented in a novel profiler (available open-source) that collects carefully selected metrics from the whole system stack with low overhead, and helps developers locate performance and scalability problems. We analyze task granularity in the DaCapo, ScalaBench, and Spark Perf benchmark suites, revealing inefficiencies related to fine-grained and coarse-grained tasks in several applications. We demonstrate that the collected task-granularity profiles are actionable by optimizing task granularity in several applications, achieving speedups up to a factor of 5.90×. Our results highlight the importance of analyzing and optimizing task granularity on the Java Virtual Machine.

CCS Concepts: • **Software and its engineering** → **Software performance**; *Dynamic analysis*; • **General and reference** → *Measurement*; *Metrics*; *Evaluation*;

Additional Key Words and Phrases: Task granularity, task parallelism, performance analysis and optimization, vertical profiling, actionable profiles, Java virtual machine

ACM Reference format:
Andrea Rosà, Eduardo Rosales, and Walter Binder. 2019. Analysis and Optimization of Task Granularity on the Java Virtual Machine. *ACM Trans. Program. Lang. Syst.* 41, 3, Article 19 (July 2019), 47 pages.
https://doi.org/10.1145/3338497

1 INTRODUCTION

Due to technological limitations complicating further advances in single CPU cores (such as the clock rate and the amount of exploitable instruction-level parallelism), nowadays processors offer an increasing number of CPU cores. While modern multicore machines provide extensive opportunities to speed up applications, developing or tuning a parallel application to make good use of

The research presented in this article was supported by Oracle (ERO project 1332).
Authors' address: A. Rosà, E. Rosales, and W. Binder, Università della Svizzera italiana, via Giuseppe Buffi 13, Lugano 6900, Switzerland; emails: {andrea.rosa, rosale, walter.binder}@usi.ch.

0164-0925/2019/07-ART19 $15.00
https://doi.org/10.1145/3338497

all CPU cores remains challenging. A popular way to speed up application execution in multicore machines is *task parallelism*, i.e., dividing the computation to be performed into units of work called *tasks*, executing each task in parallel across different CPU cores. Tasks can execute either the same or different code on the same or different data, and can be run by different threads in parallel.

This work focuses on *task-parallel*[1] applications running on a single Java Virtual Machine (JVM) in a shared-memory multicore. Task parallelism is implemented in many applications running on the JVM, also thanks to the convenient programming abstractions offered by frameworks such as thread pools [48] and fork-join pools [47], which significantly lower the programming effort for exploiting task parallelism. As a result, task-parallel applications are widespread nowadays.

A key performance attribute of task-parallel applications is their *task granularity*, i.e., the amount of work performed by each spawned task [27]. Task granularity relates to the tradeoff between the overhead of a parallel task execution and the potential performance gain. If the overall computation is divided into many *fine-grained tasks* (i.e., small tasks carrying out few computations), the application can better utilize the available CPU cores, as there are more tasks that can be distributed among the computing resources. Unfortunately, this solution may lead to considerable parallelization overheads, due to the cost of creating and scheduling a large number of tasks and the substantial contention that may occur between them. Such overheads can be mitigated by dividing the work into few *coarse-grained tasks* (i.e., large tasks performing substantial computations). However, this solution may miss some parallelization opportunities, as CPU cores may be underutilized due to the lack of tasks to be executed or due to an unbalanced division of work to tasks.

The performance of task-parallel applications may considerably depend on the granularity of their tasks. Hence, understanding task granularity is crucial to assess and improve the performance of task-parallel applications. Despite this fact, the analysis and optimization of the task granularity for applications running on the JVM has received little attention in the literature. While several researchers have proposed techniques to estimate or control task granularity in task-parallel applications, these techniques suffer from multiple limitations, such as a lack of accuracy [13, 40] or limited applicability [9, 70]. Moreover, they fall short in highlighting and optimizing performance drawbacks caused by coarse-grained tasks, and only few approaches support the JVM [40, 75]. On the other hand, although there are studies [17, 58, 73] based on the *work-span model* [24] to find the maximum speedup theoretically obtainable for an application by optimizing the longest sequential tasks, such studies focus mainly on the analysis and optimization of large sequential portions of applications, paying little attention to the overhead caused by fine-grained tasks. Finally, while numerous authors provide detailed studies on parallel applications running on the JVM [8, 12] or propose profiling tools for different performance attributes [7, 11, 22], task granularity is not their main focus. As a result, task granularity and its performance impact on task-parallel applications running on the JVM remain largely unexplored yet crucial topics.

This article aims at filling this gap, analyzing and optimizing the task granularity of parallel applications running on a single JVM in a shared-memory multicore. Our work faces notable challenges. Task-parallel applications may use tasks in complex ways. For example, applications may employ nested tasks,[2] may use tasks presenting nested calls to their own execution methods,[3] or may execute a single task multiple times.[4] While such practices may be motivated by design

[1] We denote as *task-parallel* any application resorting to task parallelism.
[2] *Nested tasks* are tasks fully executing in the dynamic extent of another task's execution, as further discussed in Section 3.4.
[3] Such tasks are detailed in Section 3.5.
[4] More information on this pattern is given in Section 3.6.

principles or code reuse, they significantly complicate task-granularity profiling. Moreover, the considered metrics may be susceptible to measurement perturbations caused by the inserted instrumentation code. In particular, such perturbations may alter the collected values of task granularity, thus biasing the results. While being of paramount importance, minimizing measurement perturbations is challenging.

Our work makes the following contributions. We develop a new methodology for profiling the task granularity of applications running on the JVM. We implement our profiling technique in the novel task-granularity profiler tgp. Our tool is built upon the DiSL [35] Java bytecode instrumentation framework, enabling the detection of all spawned tasks, including those in the Java class library (which is notoriously hard to instrument [5, 25]). To enable a detailed and accurate analysis of task granularity, tgp resorts to *vertical profiling* [16],[5] collecting a carefully selected set of metrics from the whole system stack, aligning them via offline analysis. Moreover, thanks to *calling-context profiling* [3],[6] tgp identifies classes and methods where optimizations related to task granularity are needed, guiding developers toward useful optimizations through *actionable profiles* [39].[7] Our technique resorts to a novel and efficient profiling methodology, instrumentation, and data structures to collect accurate task-granularity profiles with low profiling overhead. Overall, our tool helps developers locate performance and scalability problems related to task granularity. To the best of our knowledge, tgp is the first task-granularity profiler for the JVM.

With tgp, we analyze task granularity in two well-known benchmark suites for the JVM, i.e., DaCapo [6] and ScalaBench [59], as well as in several applications from Spark Perf [10], a benchmark suite for the popular Apache Spark [74] big-data analytics framework. Our analysis shows that several applications either employ a large number of fine-grained tasks which suffer from noticeable contention leading to significant parallelization overheads, or a small number of coarse-grained tasks that underutilize CPU and result in idle cores. We identify fine-grained tasks that can be merged to reduce parallelization overheads, as well as coarse-grained tasks that can be split into several smaller ones to better leverage idle CPU cores. To the best of our knowledge, we provide the first analysis of task granularity for task-parallel applications on the JVM. Moreover, we reveal performance issues of the target applications that were previously unknown.

We use the actionable profiles collected by tgp to guide the optimization of task granularity in numerous applications. We collect and analyze the calling contexts upon the creation and submission of tasks causing performance drawbacks, locating classes and methods to modify to perform optimizations related to task granularity. Our optimizations result in significant speedups (up to a factor of 5.90×) in several applications suffering from fine- and coarse-grained tasks.

This article significantly extends our previous work on task-granularity profiling and optimization on the JVM [53, 54]. In this article, we describe our approach to instrument tasks and collect their granularities in-depth (Section 6), as well as its implementation in DiSL (Section 7), which are not presented in our previous work. Moreover, we present an in-depth evaluation of tgp (Section 8) in terms of its profiling overhead (which is only partially outlined in our previous work) and the measurement perturbations introduced by our tool (which is missing in the previous publications). Finally, we significantly extend the analysis (Section 9) and the optimization (Section 10) of task granularity to new parallel applications, which further demonstrate the impact and the benefits of our work on different multicore machines (Section 11).

[5] According to Hauswirth et al. [16], *vertical profiling* is an approach that collects and correlates information about system behavior from different system layers.

[6] A *calling context* is the set of all methods open on the call stack at a specified point during the execution of a thread.

[7] According to Mytkowicz et al. [39], profiles are *actionable* if acting on the classes and methods indicated by the profiles yields performance improvements.

This article is structured as follows. Section 2 introduces background information on the used frameworks. Section 3 defines the scope of our analysis and the terminology used in the article. Section 4 details the metrics of interest. Section 5 discusses our profiling methodology. Section 6 describes the instrumentation for profiling task granularity. Section 7 details the implementation of our approach in DiSL. Section 8 discusses the profiling overhead and the measurement perturbations introduced by tgp. Section 9 details our empirical task-granularity analysis. Section 10 presents our approach to optimize task granularity. Section 11 evaluates the speedup achieved by our task-granularity optimizations. Section 12 discusses other aspects of our work. Section 13 compares our approach with related work. Finally, we conclude in Section 14.

2 BACKGROUND

Here, we introduce background information on the DiSL and Shadow VM frameworks used by tgp.

2.1 DiSL

The implementation of our profiling methodology resorts to DiSL to insert profiling code into the observed application. DiSL [35] is a dynamic program-analysis framework based on Java bytecode instrumentation. In DiSL, developers write instrumentation code in the form of *code snippets*, based on *aspect-oriented programming* (AOP) principles [26] that allow a concise implementation of runtime monitoring tools. DiSL allows developers to specify where a code snippet shall be woven through *markers* (specifying which parts of a method to instrument, such as method bodies, basic blocks, etc.), *annotations* (specifying where a code snippet must be inserted w.r.t. a marker, e.g., before or after method bodies), *scopes* (specifying which classes or methods shall be instrumented based on a pattern-matching scheme), and *guards* (predicate methods enabling the evaluation of conditionals at instrumentation-time to determine whether a code snippet should be woven into the method being instrumented or not).

Code snippets and guards have access to *context information* provided via method arguments. Context information can be either static (i.e., static information limited to constants) or dynamic (i.e., including local variables and the operand stack). Dynamic context information can be accessed only by code snippets. DiSL supports also *synthetic local variables* (enabling data passing between different code snippets woven into the same method body) and *thread-local variables* (implemented by additional instance fields in `java.lang.Thread`). Both variables can be expressed as annotated static fields (i.e., `@SyntheticLocal` and `@ThreadLocal`, respectively).

DiSL performs the instrumentation in a separate JVM process, the *DiSL server*. A native JVMTI[8] agent attached to the observed JVM intercepts classloading, sending each loaded class to the DiSL server. There, the instrumentation logic determines which methods to instrument to collect the desired metrics. Instrumented classes are then sent back to the observed JVM. The DiSL weaver guarantees full bytecode coverage of an analysis.[9] In particular, DiSL enables the instrumentation of classes in the Java class library, which are notoriously hard to instrument [5, 25].

[8] JVMTI (JVM Tool Interface) [44] is an interface that enables a native *agent* (attached to a JVM) to inspect the state of a JVM and to control the execution of applications running on top of it. JVMTI agents can intercept certain events occurring in the observed JVM, such as the loading of a class (allowing one to modify the final representation of a class before it is linked in the JVM), the termination of a thread, the shutdown of the observed JVM, or the activation of the garbage collector.

[9] *Full bytecode coverage* is the ability of a framework to guarantee the instrumentation of every Java method with a bytecode representation.

2.2 Shadow VM

DiSL offers a deployment setting to isolate the execution of analysis code from application code, executing analysis code asynchronously with respect to the application code in a separate JVM process, the *Shadow VM* [34]. The observed application is instrumented using DiSL to emit the events of interests, which are then forwarded to the analysis executing in the separate Shadow VM via a native JVMTI agent attached to the observed JVM. This setting avoids sharing states between the analysis and the observed application, preventing certain problems that may be introduced by less isolated approaches [25]. Moreover, Shadow VM eases proper handling of all thread life cycle events, and guarantees that all thread termination events are received even during the shutdown phase of the JVM. We rely on Shadow VM to increase the isolation of analysis code and to guarantee that the granularity of all threads can be detected and registered (even during the shutdown phase), ensuring the complete detection of all spawned tasks.

3 TASKS

In this section, we outline the entities of interest to our work and tasks that require special handling.

3.1 Tasks in Java

Our work targets *tasks* created by parallel applications running on a JVM. We consider only those entities as tasks that are expected to be executed in parallel with other tasks. In Java, parallel applications usually specify objects to be potentially executed in parallel (by different threads) as subtypes of at least one of the following interfaces or classes:

- interface `java.lang.Runnable`;
- interface `java.util.concurrent.Callable`;
- abstract class `java.util.concurrent.ForkJoinTask`.

According to the Java API [43], `Runnable` should be implemented by objects intended to be executed by a thread, `Callable` provides a semantics analogous to `Runnable`, with the difference that objects can declare a non-void return type, while `ForkJoinTask` should be extended by entities run by a fork-join pool. Hence, we consider all instances of subtypes of the above Java interfaces/classes as tasks. We use the term *task interfaces* to collectively refer to `Runnable`, `Callable`, and `ForkJoinTask`.[10] Java threads themselves are also considered tasks, as `java.lang.Thread` implements `Runnable`.

3.2 Task Granularity

Task granularity represents the amount of work carried out by each task. Following the indications of the Java API [43], the starting points for the computation of a task are the methods:

- `Runnable.run`;
- `Callable.call`;
- `ForkJoinTask.exec`.

We refer to the above methods, as well as all implementations of `Runnable.run` and `Callable.call` and all overriding `exec` methods in subtypes of `ForkJoinTask` as *execution methods*. When a thread executes a task, it will always call such a method. Consequently, all code executed in the dynamic extent of an execution method contributes to the granularity of a task.

[10] Entities that are not a subtype of any task interface, but are used like tasks, are not in the scope of our work, as further discussed in Section 12.2.

We use the term *task execution* to denote the execution of an execution method (by a thread). We denote a task as *executed* if one of its execution method has been executed until (normal or abnormal) completion at least once.

3.3 Task Submission

We also detect when tasks are submitted to a *task execution framework*, i.e., any subtype of the interface `java.util.concurrent.Executor`, such as `ThreadPoolExecutor` or `ForkJoinPool`. A task is *submitted* if it is passed as argument to a *submission method*. We consider as submission methods all (overriding) implementations of the following methods:

- `Executor.execute`;
- `ExecutorService.submit`;
- `ForkJoinPool.execute`;
- `ForkJoinPool.invoke`;
- `ForkJoinPool.submit`.

Note that the Java API [46, 47] defines multiple `ExecutorService.submit`, `ForkJoinPool.execute` and `ForkJoinPool.submit` methods. We consider all of them as submission methods, along with all their (overriding) implementations. We use the term *task submission* to refer to the submission of a task to a task execution framework.

3.4 Nested Tasks

Some tasks may be *nested*, i.e., they fully execute within the dynamic extent of the execution method of another task, which we call *outer task*. The outer and nested tasks cannot execute in parallel, as the execution of the outer task cannot proceed before the execution of the nested task has completed.

A nested task is effectively used as a normal object by the outer task, rather than as a task that could execute in parallel with the outer task. For this reason, we employ a heuristic for *task aggregation*, which aggregates the execution of a nested task to its outer task. While profiling an application, a nested task is treated as any other task, and its granularity is collected separately from the one of its outer task. After the profiling, we apply our heuristic (offline), to offer a condensed view of task executions, where a nested task is aggregated to its outer task, resulting in a single, larger task.

Task aggregation sums up the granularity of the nested task to the one of the outer task. If the outer task is itself nested, we recursively aggregate it until a non-nested task is found. Task aggregation occurs for nested tasks satisfying one of the following conditions:

(1) the outer task is not a thread;
(2) the outer task is a thread, and both of the following conditions are true:
    - the nested task has not been submitted;
    - the nested task is created and executed by the same thread.

The first condition is motivated by the fact that each task necessarily executes in the dynamic extent of a thread; hence, aggregating tasks to threads would result in a final profile composed only of threads. The second condition indicates code patterns similar to `new MyRunnable().run()`, where the created entity is effectively used by the thread as a normal object rather than as a task. For this reason, we aggregate the entity to the executing thread.

Our heuristic for task aggregation aims at reducing the complexity of the resulting task-granularity profiles, removing tasks that cannot run in parallel. Examples of nested tasks that

are aggregated are *helper tasks* and *adapter tasks*. The former are tasks that carry out small auxiliary operations, such as setting or cleaning up data or handling exceptions, before executing other tasks.[11] The latter are tasks that act as adapters to another task interface.[12] By aggregating such tasks and removing them from the profiles, our heuristic allows the user to focus on the important tasks running in parallel, easing task-granularity analysis in applications that use common abstractions from the Java class library such as task execution frameworks.

3.5 Nested Calls to Execution Methods

Tasks may present *nested calls* to their own execution methods, i.e., before the execution of a task `t` completes, one of `t`'s execution methods calls one of its own execution methods. This situation may occur due to:

- direct or indirect recursive calls to execution methods, e.g., `t.run()` calls `t.run()`;
- calls to an (overridden) task execution method defined in the superclass, e.g., `t.run()` calls `super.run()`;
- nested calls to different execution methods, if `t` is an instance of multiple task interfaces, e.g., `t.run()` calls `t.call()`, with `t` being subtype of both `Runnable` and `Callable`.

If a task presents nested calls, we consider the task as executed only when the first (outermost) call to its execution method completes, collecting its task granularity only at that moment. Our approach ensures that all work executed in the dynamic extent of an execution method is measured and considered as task granularity, complying with the specifications given in Section 3.2.

3.6 Multiple Task Executions

Finally, a task can be executed multiple times, i.e., after task execution has completed, one of its execution method is called again, and the task executes to completion. In such tasks, each execution may occur in a different thread and operate on different data. For these reasons, we treat each execution as if it was a separate task.

4 METRICS

Analyzing task granularity involves understanding its impact on application performance. To this end, we efficiently profile a comprehensive set of metrics from the whole system stack to provide a deeper understanding of task granularity, the utilization of CPU cores, and the contention between tasks. We employ a form of *vertical profiling* [16], logically dividing the system into several *layers* and collecting metrics from each of them. In the following text, we present the layers involved in the profiling process and the metrics collected, while subsequent sections detail our methodology to collect such metrics. Figure 1 summarizes the different components of tgp and the techniques they are based on.

4.1 Application Layer

For each task, we collect its starting and ending execution timestamps (in nanoseconds), as well as the threads which create and execute the task (denoted as *creating thread* and *executing thread*, respectively) and the outer task (if any). The timestamps allow us to identify the time intervals

[11] For example, `FutureTask` is a helper task used by `ThreadPoolExecutor`.
[12] Examples of adapter tasks are `RunnableAdapter` (used by `ThreadPoolExecutor`), `AdaptedRunnable` and `AdaptedCallable` (used by `ForkJoinPool`).

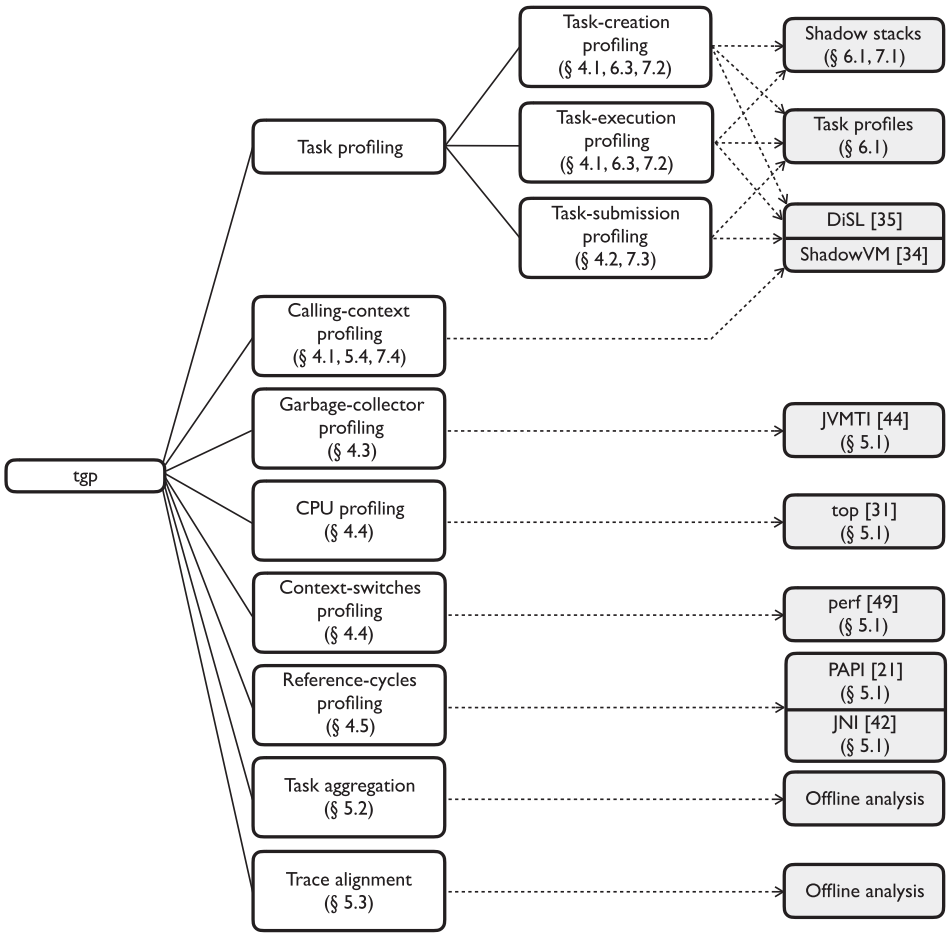


Fig. 1. Components of tgp (white background) and techniques used by them (gray background). Text in parenthesis refers to the sections providing more information about each component or technique.

where tasks are executed, enabling us to correlate task execution to JVM- and OS-layer metrics, while the other information is used by our heuristic for task aggregation to detect nested tasks that are created and executed by the same thread (which may be aggregated according to the conditions outlined in Section 3.4). We also collect the type of each task.

In an optional second profiling run, we collect the *calling contexts* [3] upon task creation, submission, and execution, as well as upon thread start. This information helps the user identify the classes and methods to target when optimizing task granularity.

4.2 Framework Layer

We profile all task submissions. For each of them, we track both the submitted task and the task execution framework the task is submitted to. Detecting task submissions provides useful information on the usage of task execution frameworks in the observed application. Moreover, this information is used by our heuristic for task aggregation when deciding which tasks should be aggregated (see Section 3.4).

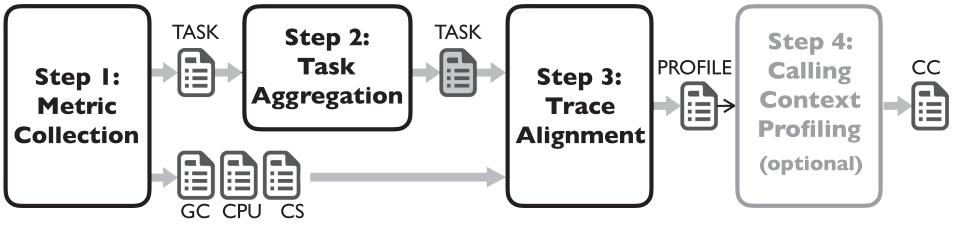


Fig. 2. Overview of the profiling methodology.

4.3 JVM Layer

We detect all time intervals when the garbage collector (GC) makes a *stop-the-world* collection, during which all threads cease to modify the state of the JVM. Stop-the-world collections can significantly alter the collected metrics, particularly those related to the OS and hardware layers. Tracking such GC activities allows us to attribute unexpected metrics fluctuations to garbage collection. During task-granularity analysis, OS- and hardware-layer metrics profiled during a stop-the-world GC are ignored, as they are collected when application threads are not in execution.

4.4 OS Layer

We detect the CPU utilization (including both user and kernel components) and the number of context switches (CSs) experienced by the observed application. The former metric allows us to determine whether the CPU is well utilized by the application, particularly when it is executing coarse-grained tasks. We use the latter metric as a measure of contention among tasks, as an excessive number of context switches indicates that tasks executing in parallel significantly interfere with each other due to the presence of numerous blocking primitives (including synchronization and message passing).

4.5 Hardware Layer

We profile the number of *reference cycles* elapsed during the execution of each task. A reference cycle elapses at the nominal frequency of the CPU, even if the actual CPU frequency is scaled up or down. We use reference cycles to measure task granularity. Using reference cycles ensures that the granularity of different tasks can be compared, even if the tasks were executed under different CPU frequencies. Moreover, this metric well represents the work carried out by a task, as it also accounts for instruction complexity and for latencies introduced by cache misses or misalignments. Finally, reference cycles can be converted into a temporal value if the nominal frequency of the CPU is known.

5 PROFILING METHODOLOGY

Figure 2 depicts our methodology to profile task granularity. First, our profiler collects metrics during application execution, producing four different *traces* containing information on tasks, stop-the-world GC activations, CPU utilization, and the amount of context switches observed. After application termination, tgp performs offline processing in two steps: task aggregation and trace alignment. The former step implements our heuristic for task aggregation, producing a refined task trace as described in Section 3.4, while the latter step aligns and integrates all traces into a single *profile*, which enables one to accurately analyze task granularity and identify the tasks to optimize. As an optional step, tgp profiles the calling contexts upon the creation, submission and execution of such tasks, aiming at locating application code to be modified to optimize task

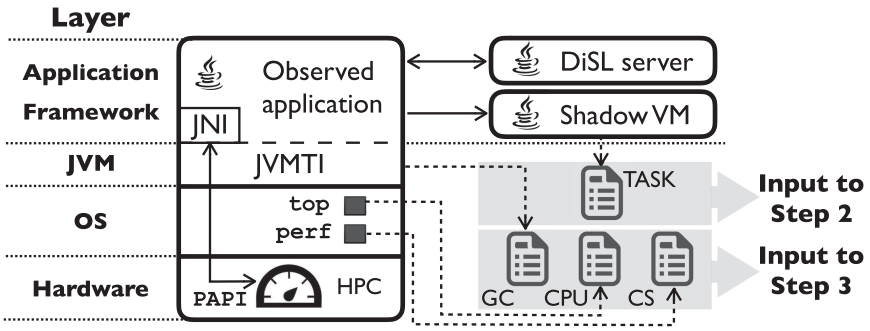


Fig. 3. Components of Step 1 (metric collection).

granularity. Each of these steps is performed separately by different components of tgp. The rest of this section details the four steps of our methodology.

5.1 Metric Collection

Figure 3 shows the different components involved in metric collection. Metrics collected at the application and framework layers are obtained by instrumenting tasks and task execution frameworks (as detailed in Sections 6 and 7). At these layers, the metrics are sent to Shadow VM upon collection, which contains most of the profiling logic and data structures. Our profiler ensures that events signaling execution start and end are observed in Shadow VM for all executed tasks.

At the JVM layer, we rely on JVMTI to profile stop-the-world GC activations. A dedicated agent attached to the observed application subscribes to the events `GarbageCollectionStart` and `GarbageCollectionFinish` exposed by JVMTI, storing the timestamps where collection starts and finishes, respectively.

At the OS layer, tgp employs two existing tools to query Linux performance counters. We rely on top [31] to acquire CPU utilization, while we resort to perf [49] to collect context switches. Both metrics are collected periodically at the highest rate allowed by the tools (i.e., every ~150ms for the former and every 100ms for the latter). Each sample taken by top represents the instantaneous CPU utilization of the system, whereas each measurement performed by perf collects the number of context switches experienced by the observed application since the last measurement.

At the hardware layer, the profiler makes use of hardware performance counters (HPCs) integrated in most modern processors, which store accurate low-level metrics that can be read efficiently. We rely on PAPI [21] to manage HPCs. In particular, PAPI enables one to access per-thread virtualized counters storing low-level metrics. We use PAPI to query the reference cycles elapsed during the execution of a task. The activation and the reading of such counters is governed by the inserted instrumentation logic (see Sections 6 and 7). A dedicated JNI[13] agent enables the management of the HPCs from the instrumentation code through PAPI.

The metrics are stored in four independent traces. Shadow VM generates a *task trace*, including metrics on tasks collected from the application, framework, and hardware layers. The JVMTI agent responsible to profile stop-the-world GC activations collects them in a *GC trace*. Finally, top and perf each store the measurements in a separate trace, the *CPU trace* and the *CS trace*, respectively. The traces undergo further offline elaboration (i.e., task aggregation and trace alignment) by other components of tgp.

[13]Java Native Interface (JNI) [42] is a standard interface for writing native Java methods. The main purpose of JNI is enabling Java code to call native code and vice versa.

5.2 Task Aggregation

As described in Section 3.4, we use a heuristic approach to aggregate nested tasks to their outer tasks, such that tasks that cannot run in parallel do not appear in the profile. In this step, we apply our heuristic to nested tasks. To avoid the overhead of aggregating nested tasks during application execution, nested tasks appear as normal tasks in the task trace and are aggregated via an offline analysis in this step.

For each nested task that has to be aggregated, the profiler adds its granularity to the outer task and afterwards removes the nested task from the task trace. This step also filters out tasks created but not executed. The output of this step is a new refined task trace where nested tasks are aggregated.

5.3 Trace Alignment

The four traces obtained during metric collection are produced by independent components, each using a different trace format and temporal reference. Trace alignment is an offline process that produces a single unified profile of the application behavior suitable to analyze task granularity.

In all traces, each observation is associated with at least one timestamp, indicating either the starting/ending execution timestamps of a task (task trace), the start/finish of a garbage collection (GC trace), or the time instant when a metric was measured (CPU and CS traces). While such timestamps are obtained by querying a unique high-accuracy clock, the initial temporal reference (i.e., the zero) in each trace is different, as it may refer to the JVM default *origin time* [41] (task and GC traces), to the program starting time (CS trace), or to the default OS-clock starting time [32] (CPU trace).

We align traces such that the initial temporal reference in all of them is the starting time of the observed application. We also filter out observations occurred when the application was not in execution (e.g., values of CPU utilization obtained before starting the observed JVM or after its termination). The profile resulting from this process contains comprehensive information on task granularity, which can be analyzed by the user to locate tasks to optimize.

5.4 Calling-Context Profiling

Finally, calling-context profiling is an optional profiling pass that runs the observed application to obtain complete calling contexts. Our approach first identifies the types of tasks of suboptimal granularity by analyzing the profile resulting from trace alignment, and then it obtains the calling contexts upon creation, submission, and execution of all tasks of such types, to help locate application code where they are created, submitted, or executed; such code often needs to be modified to optimize task granularity. In the case of a thread, we also collect the calling context when it is started (i.e., when its `start` method is called by another thread). This information allows tgp to provide actionable profiles, which enable users to identify the classes and methods to target when optimizing task granularity.

6 INSTRUMENTATION

Here, we detail our approach to instrument tasks and collect their granularities. To ease the comprehension of our profiling technique, this section presents our approach by means of abstract data types and pseudocode. We discuss how our technique can be implemented in DiSL in Section 7.

6.1 Data Structures

Task-granularity profiling relies mainly on two data structures. We describe them by means of abstract data types. In the rest of the article, we assume the existence of types `INT`, `LONG`, `THREAD`,

Table 1. Main Fields and Functions Defined on a Task Profile (TP)

Field	Description
init	Stores metrics collected at task creation.
submit	Stores metrics collected at task submission.
exec	Stores metrics collected at task execution.
Function	**Description**
`registerCreation(TASK` ta, `THREAD` th`)`	Creates a new TP tp associated with ta, and registers th as its creating thread in $tp.init$. The operation has no effect if a TP associated with ta already exists.
`registerSubmission(TASK` ta, `TEF` tef`)`	Retrieves the TP tp associated with ta and registers the submission of ta to tef in a new entry of $tp.submit$. The function is undefined if tp is not found.
`registerExecution(TASK` ta_{current}, `TASK` ta_{outer}, `LONG` c, `LONG` ti_{start}, `LONG` ti_{end}, `THREAD` th`)`	Retrieves the TP tp associated with ta_{current}, and registers the execution of ta_{current} in a new tuple of $tp.exec$. The execution of ta_{current} has occurred within the execution of task ta_{outer}, has taken c cycles, started at time ti_{start} and ended at time ti_{end}. Task ta_{current} has been executed by thread th. ta_{outer} can be $\perp_{\text{TASK}}$; in this case, ta_{current} is not nested. Otherwise, ta_{outer} is the outer task of ta_{current}. The function is undefined if tp is not found.

TASK, and TEF (the latter representing a task execution framework). We also assume that each type T has a *null value* $\perp_T$, denoting that the value of a variable or parameter of type T is undefined. We use the notation $\perp$ when referring to the null value without considering any specific type.[14]

Task Profile. We store information on each spawned task in a *Task Profile (TP)*. Each TP is associated with exactly one task, storing data related to its creation, submission(s), and execution(s). A new TP instance must be created along with the creation of a new task. Since each task has a unique identifier (i.e., a reference to the task instance), each task can be mapped to the corresponding TP. We store all task profiles in the Shadow VM.

Table 1 describes the main fields and functions defined on a TP. Conceptually, a TP instance can be represented as a data structure composed of three fields: *init*, *submit*, and *exec*, which store metrics collected at task creation, submission, and execution, respectively. The first field stores a reference to the task and to the thread that created the task. This information is registered in the TP upon its creation by calling `registerCreation`. The second field stores an ordered list of TEF instances, representing the task execution frameworks the task was submitted to. Each call to `registerSubmission` appends a new task execution framework to the list. Finally, *exec* stores an ordered list of tuples. Each tuple contains all other metrics collected at the application and hardware layers (see Section 4) related to a single task execution, apart from calling contexts. Tasks executed multiple times have multiple such tuples in their TP, one for each task execution. Function `registerExecution` stores metrics related to a single task execution, creating a new tuple at the end of the list.

Apart from being fundamental to analyze task granularity, the information stored inside task profiles is used in the second and third steps of our profiling methodology to aggregate nested tasks to their outer task (if needed) and to align metrics collected from different traces (via the collected timestamps).

[14]Unless otherwise noted, the functions listed in Tables 1, 2, and 3 are undefined if any of their input parameters is $\perp$.

Table 2. Functions Defined on a Shadow Stack (SS)

Function	Description
`createSS(`T`, INT` n`):` $\mathscr{S}_T^n$	Creates an SS that allows the inspection of the top n elements. The stack can contain only elements of type T, and is associated to the thread executing the function. Pushes ϵ_T n times on the stack. The function is undefined if $n < 1$.
`push(`$\mathscr{S}_T^n\ \bar{s}, T\ e$`)`	Pushes element e on the top of the stack $\bar{s}$. The function is undefined if $e = \epsilon_T$.
`top(`$\mathscr{S}_T^n\ \bar{s}$`, INT` i`):` T	Returns the element stored i positions from the top of the stack $\bar{s}$. $i = 0$ denotes the top of $\bar{s}$. The stack is not modified by this function. The function can return ϵ_T. The function is undefined if $i < 0$ or $i \geq n$.
`pop(`$\mathscr{S}_T^n\ \bar{s}$`)`	Removes the element at the top of the stack $\bar{s}$. The function is undefined if such element is ϵ_T.

Shadow Stack. Our instrumentation technique needs to store information related to executed methods and to make them available to subsequent callees. As this operation is done frequently, we define an auxiliary data structure, the *shadow stack (SS)*, to support storing and accessing this kind of information. As we will show in Section 7.1, SSs can be efficiently implemented by embedding them into the frames of the call stack and into thread-local variables [51].

SSs are similar to a usual parametrized stack, but they support access to several top elements rather than just the top of the stack. We use the notation $\mathscr{S}_T^n$ to denote an SS storing elements of type T and providing access to the $n \geq 1$ top elements. To better highlight SSs, variables used as shadow stacks in this article are overlined (e.g., $\bar{s}$). The functions on SSs are summarized in Table 2. Upon creation of the data structure (via method `createSS`), one must define the element type T of the stack, and how many top elements $n \geq 1$ it provides access to. Upon creation, n special elements are pushed onto the stack; we call them *empty elements* ϵ_T of type T, whose value is always $\perp_T$. The empty elements cannot be popped from the stack (hence, the stack is never empty) and cannot be pushed after stack creation. Elements on the stack can be inspected with the `top` operation, which provides access to the top of the stack as well as to subsequent elements (up to the $n - 1^{th}$ element from the top). Note that `top` may return empty elements in case the stack has not been filled up enough with `push` operations.

Each SS is thread-local, accessible only by the owning thread. In each method, there can be at most one `push` operation for each SS; such `push` is only allowed on method entry. For each `push`, there must be a corresponding `pop` on method completion. No other `push` or `pop` is allowed. Consequently, at the end of each method the state of each SS is the same as upon method entry.

6.2 Challenges in Task-Granularity Profiling

Task-granularity profiling is complicated by the presence of special tasks that must be detected and handled carefully. First, our heuristic for task aggregation may aggregate nested tasks to their outer-most task, depending on their characteristics (see Section 3.4). To avoid expensive checks in the inserted instrumentation code, we postpone aggregation of nested tasks to the second, offline step of the profiling methodology (Section 5.2). This implies that all nested tasks must be detected at runtime, and their granularity must be accounted accurately. In particular, the profiler must separate the granularity of the nested task from the one of the outer task. Second, a task may execute nested calls to its own execution methods (see Section 3.5). In such tasks, the profiler must determine when task execution completes, collecting its granularity only at that moment.

```
1  class A implements Runnable {
2    public void run() {...}                // execution method of A
3  }
4
5  class B implements Runnable {
6    public void run() {...}                // execution method of B
7  }
8
9  class C extends B {
10   public void run() {                    // execution method of C
11     super.run();                         // account to current task
12     ...
13     a.run();                             // account to a
14     ...
15     b.run();                             // account to b
16   }
17 }
```

Fig. 4. Nested tasks and nested calls to execution methods.

Table 3. Auxiliary Functions Used in Figure 5

Function	Description
`readCycleCounter(THREAD` *th*`): LONG`	Returns the reference cycles elapsed so far during the execution of thread *th*.
`thisThread(): THREAD`	Returns the thread executing this function.
`thisTask(): TASK`	Returns the task currently executed by `thisThread()`.
`getTime(): LONG`	Returns the current system time (in nanoseconds) as a long.

The pseudocode in Figure 4 exemplifies the above situations. Suppose that `a`, `b`, and `c` are three tasks of class `A`, `B`, and `C`, respectively. In turn, the three classes are subtypes of `Runnable`, and `C` is also subtype of `B`. When the execution method of `c` (i.e., `c.run`) is executed, the execution method defined in `B` is called as the first operation (line 11). In this nested call, the profiling logic must ensure that the work (i.e., the reference cycles elapsed) executed in the dynamic extent of `B.run` is still accounted to the task being executed. Moreover, `c` calls the execution methods of two other tasks (`a` and `b`) within its execution method (lines 13 and 15).[15] The work executed in the context of `a` and `b` shall be accounted only to `a` and `b`, respectively, and not also to `c`, so as to avoid counting elapsed cycles multiple times. We rely on shadow stacks to detect the aforementioned situations, ensuring accurate accounting of task granularity. Both cases occur often in task-parallel applications (including those analyzed in Section 9); hence, it is very important to detect and handle the above situations properly.

6.3 Instrumentation for Task-Granularity Profiling

In this section, we present the instrumentation code ensuring correct task-granularity profiling. Note that this section details task-creation and task-execution profiling, while our approach to profile task submissions and calling contexts is described in Sections 7.3 and 7.4, respectively.

We describe our approach by means of snippets of pseudocode using AOP notations to express where instrumentation code is inserted. We report the code in Figure 5. In addition to the functions defined on TPs and SSs, we use the auxiliary functions shown in Table 3. To ease the explanation

[15] In this example, `a` and `b` are nested tasks, while `c` is their outer task.

```
TL: LONG c_nested-thread          stores the granularity of nested tasks executed by the thread.
    S^2_TASK tasks                stores the tasks in execution.
    S^1_LONG c_entry              stores the value of the thread cycle counter at method entry.
    S^1_LONG c_nested-outer       stores the granularity of nested tasks executed by the outer task.
    S^1_LONG times                stores the task execution starting timestamp.

1  at threadInitialization() begin
2      c_nested-thread ← 0
3      tasks ← createSS(TASK, 2)
4
5      c_entry ← createSS(LONG, 1)
6      c_nested-outer ← createSS(LONG, 1)
7      times ← createSS(LONG, 1)
8  end
9
10 after taskCreation() begin
11     registerCreation(thisTask(), thisThread())
12 end
13
14
15 before executionMethod() begin
16     push(tasks, thisTask())
17
18     push(c_nested-outer, c_nested-thread)
19     push(times, getTime())
20     push(c_entry, readCycleCounter(thisThread()))
21 end
22
23
24 after executionMethod() begin
25     LONG c_nested-task ← c_nested-thread - top(c_nested-outer, 0)
26     LONG c_current ← readCycleCounter(thisThread()) - top(c_entry, 0) - c_nested-task
27     if top(tasks, 1) = ε_TASK then
28         registerExecution(top(tasks, 0), ⊥_TASK, c_current, top(times, 0), getTime(), thisThread())
29         c_nested-thread ← 0
30     else if top(tasks, 1) ≠ top(tasks, 0) then
31         registerExecution(top(tasks, 0), top(tasks, 1), c_current, top(times, 0), getTime(), thisThread())
32         c_nested-thread ← c_nested-thread + c_current
33     end
34     pop(c_entry)
35     pop(times)
36     pop(c_nested-outer)
37     pop(tasks)
38 end
```

Fig. 5. Instrumentation code to profile task granularity. Thread-local (TL) variables are reported at the top of the figure. Blank lines are added to ease line-by-line comparison with Figure 6.

of our technique, we denote with *th* and *ta* the thread and the task in execution, respectively, and with *ot* the outer task of *ta* (which is $\perp_{\mathsf{TASK}}$ if *ta* is not nested).

The instrumentation logic relies on five thread-local variables, four of them being SSs. All thread-local variables are initialized upon thread creation (lines 1–8). Code at lines 10–12 is inserted after the creation of a new task. Here, the code creates a new TP for the task being created (via `registerCreation`) and tracks the thread which created the task. The other code snippets are inserted before (lines 15–21) or after (lines 24–38) each execution method of every task.

To measure task granularity, we query a *thread cycle counter* at selected points during thread execution, via `readCycleCounter`. The counter stores the total amount of cycles elapsed from the

start of th. We use the thread-local (TL) variable $c_{\text{nested-thread}}$ to store the total granularity of nested tasks executed by th, while the SS $\overline{c_{\text{nested-outer}}}$ tracks the granularity of all nested tasks executed by ot (excluding ta). To obtain such value, the value of $c_{\text{nested-thread}}$ is pushed on $\overline{c_{\text{nested-outer}}}$ at execution method entry (line 18). Note that $c_{\text{nested-thread}}$ (and hence $\texttt{top}(\overline{c_{\text{nested-outer}}}, \texttt{0})$) is 0 if ta is not nested.

The purpose of the SS $\overline{c_{\text{entry}}}$ is to memorize the value of the cycle counter at execution method entry for later use (line 20). The other SSs are used to store the tasks being executed ($\overline{\text{tasks}}$; line 16) and their execution starting timestamps ($\overline{\text{times}}$; line 19). Following the rules for manipulating SSs, all stacks are pushed at the beginning of an execution method (lines 16–20) and are popped at method end (lines 34–37). Note that all SSs provide access only to the topmost element, with the exception of $\overline{\text{tasks}}$ which also provides access to one element below the top.

The local variable $c_{\text{nested-task}}$ stores the total granularity of the nested tasks executed by ta (line 25). If ta has not executed nested tasks, $c_{\text{nested-thread}}$ has not been modified since the beginning of the execution method; hence, $c_{\text{nested-task}} = 0$ (as the top of $\overline{c_{\text{nested-outer}}}$ stores the value that $c_{\text{nested-thread}}$ had at the beginning of the method, see line 18). If ta has executed nested tasks, the difference between $c_{\text{nested-thread}}$ and $\texttt{top}(\overline{c_{\text{nested-outer}}}, \texttt{0})$ represents the granularity of all nested tasks executed by ta, which is stored in $c_{\text{nested-task}}$. The granularity c_{current} of ta is computed as the difference between the value of the cycle counter at the end and at the beginning of the execution method (the latter has been stored in $\overline{c_{\text{entry}}}$). This difference is reduced by the granularity of the nested tasks executed by ta, stored in $c_{\text{nested-task}}$ (line 26).

The SS $\overline{\text{tasks}}$ is necessary to identify nested calls to different execution methods of ta, as well as to determine whether ta is nested. The former case occurs if $\texttt{top}(\overline{\text{tasks}}, 1) = \texttt{top}(\overline{\text{tasks}}, 0)$ (implying $\texttt{top}(\overline{\text{tasks}}, 1) \neq \epsilon_{\text{TASK}}$), meaning that the method being executed has been called by another execution method of ta. In this case, no specific action is taken, as the execution of ta will be registered when its outmost execution method completes. On the other hand, ta is nested if $\texttt{top}(\overline{\text{tasks}}, 1) \neq \epsilon_{\text{TASK}} \wedge \texttt{top}(\overline{\text{tasks}}, 1) \neq \texttt{top}(\overline{\text{tasks}}, 0)$ (lines 30–33), which indicates that the method being executed has been called within the execution method of another task, as the two tasks on the stack are different. In this case, the profiler registers the execution of ta (line 31), storing in the TP the objects representing the task itself ta and the outer task ot (stored in $\texttt{top}(\overline{\text{tasks}}, 0)$ and $\texttt{top}(\overline{\text{tasks}}, 1)$, respectively), the granularity of ta (stored in c_{current}), the starting and ending execution timestamps (the former stored at the top of $\overline{\text{times}}$, the latter retrievable with `getTime`) and the current thread th. Moreover, the profiler adds the granularity of ta to $c_{\text{nested-thread}}$ (line 32). Following this mechanism, the granularity of ot will exclude the cycles elapsed during the execution of the nested task ta. Finally, the condition $\texttt{top}(\overline{\text{tasks}}, 1) = \epsilon_{\text{TASK}}$ (line 27) indicates that ta is not nested within any other task. Here, the profiler registers its execution (line 28) and resets $c_{\text{nested-thread}}$ (line 29), such that the granularity of subsequently executed tasks can be accounted correctly.

7 IMPLEMENTATION

Here, we detail how the instrumentation scheme presented in the previous section is implemented in tgp using DiSL code.

7.1 Efficient Shadow Stacks

In Java, TL SSs can be efficiently embedded within the frames of the call stack and in TL variables. In this section, we present a translation of the functions defined on SSs (see Table 2) that avoids heap allocations and leverages synthetic-local and TL variables offered by DiSL.

Table 4. Translations of Shadow-Stack Functions (General Case)

Function	Translation to DiSL code
$\bar{s}$ = createSS(INT n, T):$\mathscr{S}_T^n$	@ThreadLocal T s_tl_0 = $\perp_T$; . . . @ThreadLocal T s_tl_<$n-2$> = $\perp_T$; @SyntheticLocal T s_sl;
push($\mathscr{S}_T^n$ $\bar{s}$, T e)	s_sl = s_tl_<$n-2$>; s_tl_<$n-2$> = s_tl_<$n-3$>; . . . s_tl_0 = e;
top($\mathscr{S}_T^n$ $\bar{s}$, INT i):T	if (i == $n-1$) return s_sl else return s_tl_<i>;

pop($\mathscr{S}_T^n$ $\bar{s}$)	s_tl_0 = s_tl_1;
	. . .
	s_tl_<$n-2$> = s_sl;

Table 5. Translations of Shadow-Stack Functions ($n = 2$)

Function	Translation to DiSL code
$\bar{s}$ = createSS(2, T):$\mathscr{S}_T^2$	@ThreadLocal T s_tl = $\perp_T$; @SyntheticLocal T s_sl;
push($\mathscr{S}_T^2$ $\bar{s}$, T e)	s_sl = s_tl; s_tl = e;
top($\mathscr{S}_T^2$ $\bar{s}$, INT i):T	if (i == 1) return s_sl; else return s_tl;
pop($\mathscr{S}_T^2$ $\bar{s}$)	s_tl = s_sl;

Table 4 reports the general scheme to translate functions on SSs to DiSL code, assuming that the preconditions of the functions are met. An SS $\mathscr{S}_T^n$ $\bar{s}$ can be implemented with $n-1$ TL variables and one synthetic-local variable. For $n \geq 2$, the TL variable s_tl_<i> provides access to the element top($\bar{s}$, i) ($i = [0, n-2]$). The element top($\bar{s}$, $n-1$) is stored in the synthetic local variable s_sl. Regardless of the size of the SS, only $n-1$ elements are stored in TL variables. All other elements are embedded within the frames of the Java call stack, thanks to the synthetic local variable. All TL variables are initialized to $\perp_T$.

Manipulation of the SS can occur only on method entry and exit, as first (last, respectively) instructions executed in the method. At the beginning of each method that manipulates the stack, a push occurs by copying s_tl_<$n-2$> into s_sl, then copying the value of s_tl_<i> into s_tl_<$i+1$> ($i = [0, n-3]$), and finally assigning the pushed element to s_tl_0. When the method ends, it must undo the push operation, by first copying s_tl_<$i+1$> into s_tl_<i> ($i = [0, n-3]$), then copying s_sl into s_tl_<$n-2$>. For $n = 2$, a single TL variable suffices (see Table 5), while for $n = 1$, no TL variable is needed (see Table 6).

For $n > 2$, an alternative translation of SSs would store variables s_tl_<i> ($i = [0, n-2]$) in a single TL array s_tl. The array serves as a ring buffer where the top $n-1$ elements are accessible, while elements falling out from it due to push operations are saved into synthetic local variables. On method exit, fallen-out elements are restored into the array. This solution would avoid n data copying operations at each push and pop, at the cost of accessing an array on the heap, and is

Table 6. Translations of Shadow-Stack Functions ($n = 1$)

Function	Translation to DiSL code
$\bar{s}$ = createSS(1,T):$\mathscr{S}_T^1$	@SyntheticLocal T s_sl;
push($\mathscr{S}_T^1$ $\bar{s}$, T e)	s_sl = e;
top($\mathscr{S}_T^1$ $\bar{s}$,INT i):T	return s_sl;
pop($\mathscr{S}_T^1$ $\bar{s}$)	no action

likely to be more efficient for large values of n. We do not show this translation here, because it is not needed in our approach ($n \leq 2$).

7.2 Task-Granularity Profiling

Figure 6 shows the translation of our approach for profiling task granularity (Figure 5), along with task creation and execution, to DiSL code snippets. The numeric abstract data types used in Section 6 (i.e., INT and LONG) are translated into the correspondent primitive types in Java (i.e., int and long, respectively), THREAD and TEF are translated into Thread and Executor, respectively, while we use Object for representing TASK, to avoid casts to multiple task interfaces. We consider −1 as null value for numeric primitive types, and null for all other types.

The annotations @Before and @After specify where the code snippets shall be woven (i.e., at method beginning or end, respectively), while guards and scopes specify into which methods the code snippets shall be woven. In particular, the annotation at line 9 specifies that the code snippet must be applied only to the constructors of classes being subtype of Runnable, Callable, or ForkJoinTask, while guard ExecutionMethodGuard (lines 14 and 23) allows the code snippet to be inserted only in executions methods, according to the definition given in Section 3.2. Guards TaskGuard and ExecutionMethodGuard (executed in the DiSL server to determine whether a class shall be instrumented) resort to the DiSL Reflection API [50, 52] to access complete reflective supertype information about the class under instrumentation, performing accurate checks on the type hierarchy to detect subtypes of task interfaces at instrumentation time. Such checks are fundamental to avoid the insertion of expensive runtime type checks (i.e., the use of the instanceof operator) in the inserted instrumentation code, thus reducing the profiling overhead of tgp [50, 52].

In the code snippets, we use the class PAPI to query the virtualized cycle counter of a thread through readCycleCounter. The class executes native code through a JNI agent to read and manage HPCs via the API provided by PAPI. Class DynamicContext (provided by DiSL) allows one to retrieve the object currently being executed (i.e., this) through getThis. Since our instrumentation guarantees that getThis is only called in a task constructor or execution method, function thisTask can be translated to a call to getThis. Functions getTime and thisThread can be translated to calls to System.nanoTime and Thread.currentThread, respectively, which can be called by any application running on the JVM.

Operations defined on task profiles are managed by class Profiler. Methods of class Profiler (i.e., registerCreation and registerExecution) send the collected metrics to the Shadow VM via the attached JVMTI agent. The Shadow VM stores and manages all task profiles, implementing the functions as detailed in Table 1. In particular, the Shadow VM maintains a mapping from tasks to task profiles and inserts metrics received upon task creation and execution into the corresponding task profile, creating it if it does not exist. Moreover, Shadow VM detects and manages tasks executed multiple times, registering each execution separately in the task profile. Finally, when the observed application terminates, the Shadow VM creates a task trace, containing all metrics collected for each task profile.

```
1
2  @ThreadLocal static long c_nested_thread = 0;
3  @ThreadLocal static Object tasks_tl = null;
4  @SyntheticLocal static Object tasks_sl;
5  @SyntheticLocal static long c_entry_sl;
6  @SyntheticLocal static long c_nested_outer_sl;
7  @SyntheticLocal static long times_sl;
8
9  @After(marker = AfterInitBodyMarker.class, scope = "<init>",
       guard = TaskGuard.class)
10 public static void afterTaskCreation(DynamicContext dc) {
11   Profiler.registerCreation(dc.getThis(), Thread.currentThread());
12 }
13
14 @Before(marker = BodyMarker.class, guard = ExecutionMethodGuard.class)
15 public static void beforeExecutionMethod(DynamicContext dc) {
16   tasks_sl = tasks_tl;
17   tasks_tl = dc.getThis();
18   c_nested_outer_sl = c_nested_thread;
19   times_sl = System.nanoTime();
20   c_entry_sl = PAPI.readCycleCounter(Thread.currentThread());
21 }
22
23 @After(marker = BodyMarker.class, guard = ExecutionMethodGuard.class)
24 public static void afterExecutionMethod() {
25   final long c_nested_task = c_nested_thread - c_nested_outer_sl;
26   final long c_current = PAPI.readCycleCounter(Thread.currentThread()) -
         c_entry_sl - c_nested_task;
27   if (tasks_sl == null) {
28     Profiler.registerExecution(tasks_tl, null, c_current, times_sl,
           System.nanoTime(), Thread.currentThread());
29     c_nested_thread = 0;
30   } else if (tasks_sl != tasks_tl) {
31     Profiler.registerExecution(tasks_tl, tasks_sl, c_current, times_sl,
           System.nanoTime(), Thread.currentThread());
32     c_nested_thread += c_current;
33   }
34
35
36
37   tasks_tl = tasks_sl;
38 }
```

Fig. 6. DiSL code for task-granularity profiling. Blank lines are added to ease line-by-line comparison with Figure 5.

7.3 Task-Submission Profiling

We detect task submission by instrumenting all submission methods (as defined in Section 3.3). The inserted instrumentation code calls function `registerSubmission` to store the task as well as the task execution framework the task was submitted to in a task profile. In all submission methods, the task execution framework is the object being executed (i.e., `this`), while the submitted task is the object passed as first argument. Both objects can be retrieved by querying context information through methods `getThis` and `getMethodArgumentValue`, respectively, both defined in the class `DynamicContext` provided by DiSL. Similarly to task creation and execution, the collected metrics are sent to Shadow VM, which registers the submission in the task profile.

7.4 Calling-Context Profiling

Calling contexts are profiled separately from all other metrics in a separate run of the observed application. We profile calling contexts by instrumenting all methods of each loaded class. The full

bytecode coverage ensured by DiSL allows the collection of complete calling contexts, including methods executed inside Java library classes. Upon method entry, an identifier of the method is pushed onto a stack-like data structure (on the observed JVM's heap); upon method completion, it is popped. The data structure is sent to Shadow VM upon task creation, submission, execution, or thread start. Our approach allows profiling complete calling contexts at the price of numerous memory accesses, which may significantly slow down application execution, biasing the collection of other metrics. For this reason, we profile calling contexts in a separate application run. We do not provide further details on calling-context profiling because similar instrumentation has been presented in several related studies and can be easily implemented [37, 57].

8 EVALUATION OF TGP

In this section, we evaluate the overhead of tgp and discuss the measurement perturbations introduced by our approach.

8.1 Methodology and Setup

Our evaluation targets multiple real-world applications running on a single JVM in a shared-memory multicore machine. In particular, we apply our tool to benchmarks from the DaCapo [6], ScalaBench [59], and Spark Perf [10] suites. DaCapo and ScalaBench are composed of well-known JVM applications written in Java and Scala, respectively, while Spark Perf is a collection of benchmarks for the popular Apache Spark [74] big-data analytics framework, released by the company responsible for Spark development.

We use the latest versions of the suites at the time of writing, i.e., DaCapo version 9.12-MR1 (released in January 2018), ScalaBench version 0.1.0 (released in February 2012), and the latest build of Spark Perf (dated December 2015). Following the recommendation of the DaCapo developers,[16] we execute the benchmark lusearch-fix in place of lusearch.[17] Both DaCapo and ScalaBench can be executed with different input sizes. We use the largest input defined for each application. Benchmarks can execute multiple iterations. Each iteration can either be considered as *warm-up* or *steady-state*. We run warm-up iterations until dynamic compilation and GC ergonomics are stabilized, following the approach described by Lengauer et al. [29]. All other iterations after warm-up are classified as steady-state. The evaluation presented in this section, as well as task-granularity analysis presented in Section 9 target only steady-state iterations. Table 7 lists the benchmarks considered, along with the used input size (in DaCapo and ScalaBench), and the number of warm-up iterations (in the last column).

In all applications, we use the stop-the-world parallel collector [45] as GC (i.e., the default GC on multicores). Our choice allows filtering out cycles elapsed when GC is active, since no application- or framework-layer tasks are in execution during a stop-the-world garbage collection. We conduct our evaluation on a server-class machine equipped with two NUMA nodes, each with an Intel Xeon E5-2680 (2.7GHz) processor with eight physical cores and 64GB of RAM, running under Ubuntu 16.04.03 LTS (kernel GNU/Linux 4.4.0-112-generic x86_64). When profiling a benchmark, no other CPU-, memory-, or IO-intensive applications are in execution on the system, to reduce measurement perturbations. Moreover, we pin the observed application to an exclusive NUMA node (i.e., other processes, including the DiSL server, Shadow VM, perf, and top run on a different NUMA node). This deployment setting increases the isolation of the observed application, reducing performance interference caused by other processes in execution (as the observed application exclusively utilizes the cores and the memory of its NUMA node).

[16] See http://dacapobench.org for more information.
[17] Results reported on lusearch have been obtained on lusearch-fix.

Table 7. Benchmarks Considered in the Article

Benchmark DaCapo	Input size	# w.u.	Benchmark ScalaBench	Input size	# w.u.	Benchmark Spark Perf	# w.u.
avrora	large	20	actors	huge	10	AlternatingLeastSquares	10
batik	large	20	apparat	gargantuan	5	ChiSquare	60
eclipse	large	20	factorie	gargantuan	5	ClassificationDecisionTree	20
fop	default	40	kiama	default	40	GaussianMixtureEM	40
h2	huge	10	scalac	large	20	KMeansClustering	35
jython	large	20	scaladoc	large	20	LogRegression	20
luindex	default	40	scalap	large	20	MultinomialNaiveBayes	20
lusearch	large	20	scalariform	huge	10	PrincipalComponentAnalysis	20
pmd	large	20	scalatest	default	40	StreamingWordCount	5
sunflow	large	20	scalaxb	huge	10		
tomcat	huge	10	specs	large	20		
tradebeans	huge	10	tmt	huge	10		
tradesoap	huge	10					
xalan	large	20					

We set up top to collect CPU utilization only for the NUMA node where the observed application is executing, such that computational resources used by the DiSL server, Shadow VM, perf, and top are not accounted in the measurements, increasing the accuracy of the resulting CPU trace. We disable Turbo Boost and set the CPU governor to "performance" to disable frequency scaling, such that CPU cores run at the nominal speed for the whole application execution. In addition, we disable Hyper-Threading, ensuring that each logical CPU core seen by the OS maps to a physical CPU core. We use Java OpenJDK 1.8.0_161-b12.

The Spark Perf benchmarks are run on Spark 2.3.0 (compiled with Scala 2.11), set up in local mode on a single machine with as many executors as available cores. Note that in local mode, all Spark components, including the driver and all executors, run in a single JVM.

8.2 Profiling Overhead

In this section, we show the profiling overhead caused by tgp. We present overheads in the form of *overhead factors*, defined as the execution time of the observed application with profiling enabled divided by the execution time of the observed application with profiling disabled. We show our results in Figure 7. For each application, we report the average overhead obtained on 20 steady-state runs (within the same JVM process). Error bars indicate 95% confidence intervals.

In most benchmarks, the overhead factor does not exceed 1.04×, with the following exceptions. In DaCapo, avrora incurs an average overhead of 1.13×. In ScalaBench, there are four benchmarks suffering from higher overhead, i.e., actors (1.34×), scalatest (1.41×), specs (1.34×), and tmt (1.12×). In Spark Perf, the overhead of GaussianMixtureEM is 1.16×. These overheads are caused by the presence of many spawned tasks (especially in the ScalaBench applications), whose creation, submission, and execution are instrumented by tgp. On average, profiling task granularity incurs a slowdown of 1.02× (DaCapo), 1.11× (ScalaBench), 1.04× (Spark Perf), and 1.05× (overall).

In summary, the overhead introduced by tgp is low for most evaluated applications. In our profiling methodology, a high overhead can jeopardize the accuracy of the collected task granularities, which are sensitive to measurement perturbations caused by the inserted instrumentation code. The typically low overhead introduced by tgp is less likely to significantly perturb the collected metrics, enabling more accurate task-granularity analyses, as discussed in the next section.

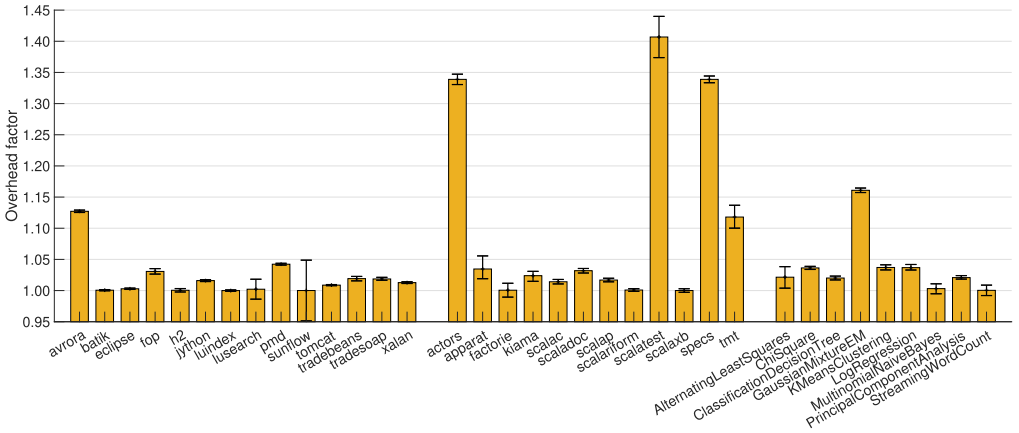


Fig. 7. Profiling overhead on DaCapo (first group), ScalaBench (second group), and Spark Perf (third group).

8.3 Perturbations

The instrumentation code inserted for profiling the metrics of interest causes the execution of additional computations that are not part of the original observed application. While this approach is necessary to collect the desired metrics, the computations inserted may alter the metrics collected, leading to results that may not be representative of the original application behavior. Among all metrics profiled by tgp, reference cycles (used as a measure of task granularity) are particularly susceptible to perturbations caused by the profiling logic, because the additional computations inserted by the instrumentation may result in extra cycles elapsed during the execution of a task, which may be accounted in the granularity of the task. Reducing the amount of extra cycles elapsed due to instrumentation code is therefore very important to collect accurate task-granularity profiles.

The goal of this section is to estimate the amount of extra cycles introduced in the observed application when profiling task granularity with tgp. To this end, we measure the total amount of reference cycles elapsed when executing the observed application with and without tgp enabled. The measured reference cycles represent both the sequential and parallel work performed on all cores during the execution of an application. We collect reference cycles by attaching perf to the observed JVM, using a configuration that enables profiling the amount of cycles elapsed during program execution, regardless of whether tgp is used. We do not consider cycles elapsed during garbage collection, when no application- and framework-layer tasks are in execution.

To relate the amount of extra cycles introduced by tgp to the cycles originally elapsed in the target application, we introduce a new metric, the *perturbation factor*, defined as the number of reference cycles elapsed with tgp enabled, divided by the number of cycles elapsed with tgp disabled. We use perturbation factors as an indication of the quality of the collected profiles. High perturbation factors indicate that the collected metrics are likely to be biased, while low perturbation factors increase our confidence in the results. Note that the purpose of this metric is not to exactly quantify the (relative) amount of extra cycles inserted by tgp in the collected task-granularity profiles. A high perturbation factor indicates that a significant amount of extra cycles has been introduced by the profiling logic; however, such cycles are not necessarily all counted in the collected task profiles. On the other hand, a low perturbation factor does not guarantee that the metrics are perturbed only little, as tgp might overestimate the granularity of some tasks while underestimating the granularity of some other tasks, without this being reflected in a high

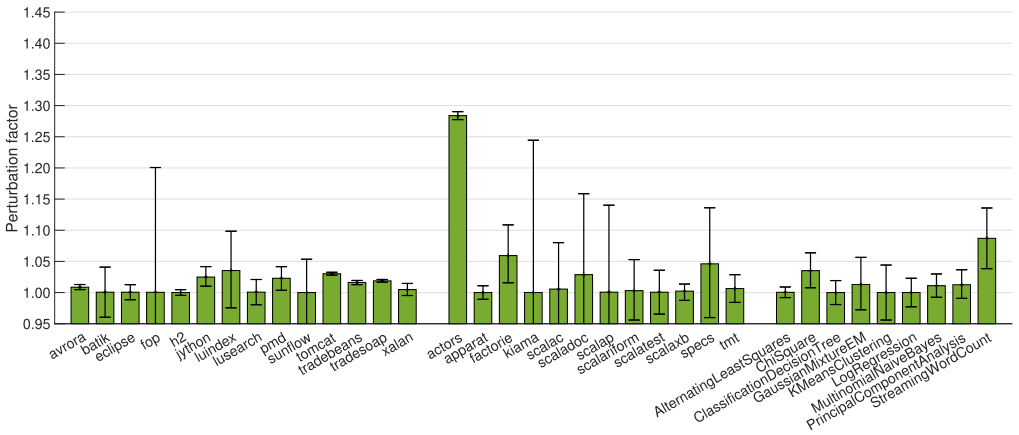


Fig. 8. Perturbation factors in DaCapo (first group), ScalaBench (second group), and Spark Perf (third group).

perturbation factor. Moreover, even with a low perturbation factor, thread scheduling may change in different application runs, and some computations in the instrumented application may be executed in different tasks (in comparison to the original application). Finally, attaching perf to collect reference cycles may itself introduce extra reference cycles in both the original and instrumented application, perturbing the collected values. Therefore, it is not possible to collect a fully accurate baseline.

Figure 8 summarizes our results. For each application, we report the average perturbation factor obtained on 20 steady-state runs (within the same JVM process). Error bars indicate 95% confidence intervals. Overall, the perturbation factor is moderate for many evaluated applications. In all DaCapo benchmarks, the perturbation factor is close to 1×, with the highest factor observed in luindex (1.03×). In ScalaBench, the only application where the perturbation factor is significant is actors (1.28×). This value can be explained by the fact that the workload of actors is dominated by the creation and execution of millions of very fine-grained tasks (see Section 9.2), each of them introducing extra cycles elapsed due to profiling code. Apart from actors, the perturbation factor observed in all other ScalaBench applications is below 1.06×. Finally, the only Spark Perf benchmark with a significant perturbation factor is StreamingWordCount (1.09×), while in all other Spark Perf benchmarks, the factor is no higher than 1.03×. The large confidence interval in some of the DaCapo and ScalaBench benchmarks can be motivated by the short duration of the workloads (below ~500ms), which amplifies the variance of the measurements even in presence of moderate differences in the collected values (fop, luindex, kiama, and scalap) or by the presence of outliers in the performed measurements (scalac, scaladoc, spec, and sunflow).

On average, profiling task granularity with tgp causes only a low perturbation factor, equal to 1.01× (DaCapo), 1.03× (ScalaBench), 1.02× (Spark Perf), and 1.02× (overall). These low perturbation factors are likely to result in collected task-granularity profiles that closely represent the original application behavior.

9 TASK-GRANULARITY ANALYSIS

In this section, we use tgp to characterize task granularity in multiple task-parallel applications.

9.1 Methodology

Our analysis targets task-parallel benchmarks of the latest DaCapo, ScalaBench, and Spark Perf suites. We analyze the same applications used in the previous section for the evaluation of tgp. The

Table 8. Benchmarks Spawning Fine-Grained (FG) or Coarse-Grained (CG) Tasks

Benchmark DaCapo [6]	Problematic Task Class	# tasks spawned	CG/FC
avrora	avrora.sim.SimulatorThread	26	CG
eclipse	java.lang.Thread	468	FG
h2	org.dacapo.h2.TPCC$3	8	CG
lusearch	org.dacapo.lusearch.Search$QueryThread	8	CG
pmd	net.sourceforge.pmd.PMD$PmdRunnable	570	FG
sunflow	org.sunflow.core.renderer.BucketRenderer$BucketThread	8	CG
tomcat	org.apache.tomcat.util.net.NioBlockingSelector$BlockPoller$3	200,816	FG
tradebeans	org.apache.geronimo.samples.daytrader.dacapo.DaCapoTrader	8	CG
tradesoap	org.mortbay.jetty.nio.SelectChannelConnector$ConnectorEndPoint	128,308	FG
"	org.apache.geronimo.samples.daytrader.dacapo.DaCapoTrader	8	CG
Benchmark ScalaBench [59]	**Problematic Task Class**	**# tasks spawned**	**CG/FC**
actors	scala.actors.ActorTask	5,197,993	FG
apparat	scala.actors.ActorTask	122,626	FG
tmt	scala.concurrent.ThreadRunner$$anon$2	16,184	FG
Benchmark Spark Perf [10]	**Problematic Task Class**	**# tasks spawned**	**CG/FC**
ChiSquare	org.apache.spark.executor.Executor$TaskRunner	5	CG
GaussianMixtureEM	org.apache.spark.executor.Executor$TaskRunner	35	CG
KMeansClustering	org.apache.spark.executor.Executor$TaskRunner	32	CG
LogRegression	org.apache.spark.executor.Executor$TaskRunner	45	CG
MultinomialNaiveBayes	org.apache.spark.executor.Executor$TaskRunner	14	CG
PrincipalComponentAnalysis	org.apache.spark.executor.Executor$TaskRunner	9	CG
StreamingWordCount	org.apache.spark.executor.Executor$TaskRunner	1,852	FG

experimental setup and our evaluation methodology are the same ones presented in Section 8.1. Table 8 reports all benchmarks where we found fine- or coarse-grained tasks, along with the number of problematic tasks spawned by the application and their class. All applications not appearing in the table are either explicitly labeled as single-threaded in the benchmark documentation (factorie, kiama, scalap, scalariform, and scalaxb), or they spawn neither coarse- nor fine-grained tasks (AlternatingLeastSquares, batik, ClassificationDecisionTree, fop, jython, luindex, scalac, scaladoc, scalatest, specs, and xalan).

9.2 Fine-Grained Tasks

We identify fine-grained tasks as large groups of tasks of the same class showing similarly low granularities. Our analysis reveals that several benchmarks spawn many fine-grained tasks in all the three benchmark suites: eclipse, pmd, tomcat, and tradesoap (from DaCapo); actors, apparat, and tmt (from ScalaBench); and StreamingWordCount (from Spark Perf). In the following text, we detail the class of such tasks and their purpose.

Task Description. Regarding DaCapo, eclipse runs a series of performance tests for the well-known Eclipse integrated development environment (IDE) [68]. As part of the workload, a `ReadManager`[18] creates many `Thread` instances to read the content (i.e., source code) of different

[18]We omit package declaration and outer classes to improve readability. Table 8 reports the fully qualified class names.

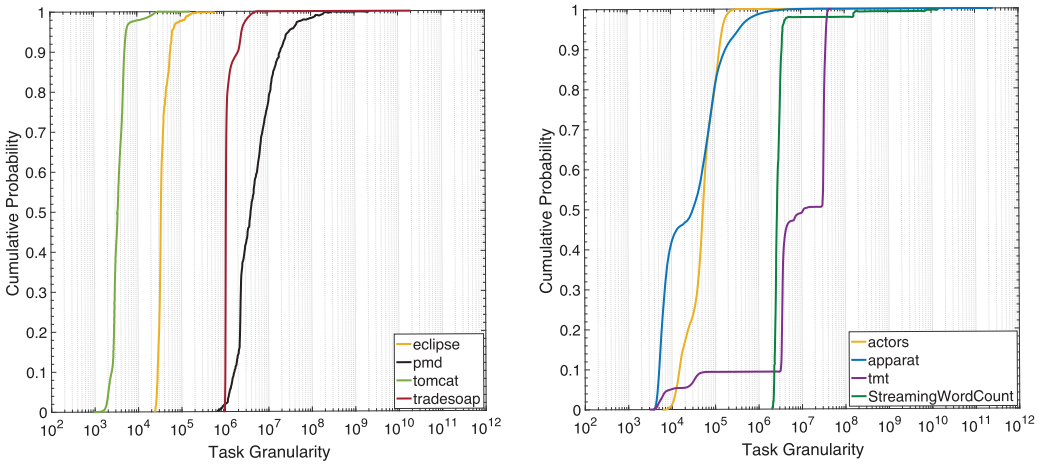
compilation units. The goal of pmd is to analyze a set of source-code files, detecting errors or bad coding practices. Each file is processed in parallel by a `PmdRunnable`, which produces a report containing the problems found. tomcat requests several pages to an Apache Tomcat [62] web-server, verifying the correctness of the received pages afterwards. Operations over sockets are managed by the `BlockPoller` class, which creates many tasks to add or remove packets from the socket. Tasks responsible for handling packet removal (of class `BlockPoller$3`) are particularly fine grained. Finally, tradesoap executes the DayTrader [20] online stock trading benchmark on the H2 in-memory database [14]. Users execute transactions on the database in different sessions, coordinated by SOAP messages encapsulated in HTTP packets, which are in turn managed by the Jetty [67] web server. Numerous fine-grained tasks of class `ConnectorEndPoint` are used by Jetty to handle HTTP packets. In total, we found 468 fine-grained tasks in eclipse, 570 tasks in pmd, 200,816 tasks in tomcat, and 128,308 tasks in tradesoap.

Regarding ScalaBench, actors and apparat rely on the *actor model* [18] to execute computations in parallel. Actors are a form of lightweight parallel tasks [2]; indeed, they are considered as very fine-grained tasks by tgp. Both benchmarks use an actor implementation provided by the Scala library, where an actor is an instance of `ActorTask` (which, in turn, is a subtype of both `Runnable` and `Callable`). Actors are used to test the performance of inter-actor communication (actors) or to optimize multimedia files (apparat). On the other hand, tmt uses the Stanford Topic Modeling Toolbox [69] to learn a topic model from a document composed of different records. Each record is processed in parallel by a `ThreadRunner` provided by the Scala library. actors and apparat make use of a large number of tasks, i.e., 5,197,993 and 122,626, respectively, while tmt resorts to 16,184 instances of `ThreadRunner`.

Finally, StreamingWordCount (from Spark Perf) executes performance tests on the streaming library of Spark [65] by reading a stream of words and producing a word frequency histogram. A computation in Spark is divided in several individual units of execution (called *task* in the Spark terminology, as explained in Section 10.3), each encapsulated in a `Runnable` of class `TaskRunner`. In this benchmark, the computation is divided in 1,852 small tasks, resulting in 1,852 fine-grained `TaskRunner` instances detected by tgp.

Task Granularity. Figure 9 shows the Cumulative Distribution Function (CDF) of the granularity of tasks outlined above. Among the DaCapo benchmarks (Figure 9(a)), the lowest granularities can be observed in tomcat and eclipse, where ~98% of the tasks spawned do not execute more than 10^4 cycles and 10^5 cycles, respectively. The low granularities of such tasks are caused by the small size of the compilation units read (eclipse) and by the few operations needed to remove packets from the socket (tomcat). Task granularity in tradesoap is around 10^6 for 90% of the tasks, a sign that most of the tasks handle small HTTP messages. In pmd, ~75% of the `PmdRunnable` objects feature a granularity lower than 10^7 cycles, and most of the tasks (98%) execute less than 10^8 cycles. The granularity of a `PmdRunnable` depends on the length and complexity of the file that the task is in charge of processing. The presence of diverse files in the input set of this benchmark leads to tasks with different granularities.

In ScalaBench (Figure 9(b)), actors and apparat spawn actors of small granularity. In both benchmarks, the amount of cycles elapsed does not exceed 10^5 in ~80% of the actors spawned, which is a sign that most actors execute little computation. The CDF of tmt shows the presence of two large groups of tasks with similar granularity, executing $\sim 3 \cdot 10^6$ cycles (~35% of the tasks) and $\sim 3 \cdot 10^7$ cycles (~50% of the tasks), respectively. Similarly to pmd, the granularity of a task in tmt is proportional to the length and complexity of the record that is assigned to the task. Finally, the granularity of most of the tasks spawned in StreamingWordCount (98%) is around $3 \cdot 10^6$. Overall, these CDFs pinpoint the presence of large groups of tasks executing few computations.



(a) DaCapo [6]. (b) ScalaBench [59] and Spark Perf [10].

Fig. 9. Cumulative Distribution Function (CDF) of the task granularity for benchmarks spawning fine-grained tasks. The figure shows only the granularities of fine-grained tasks.

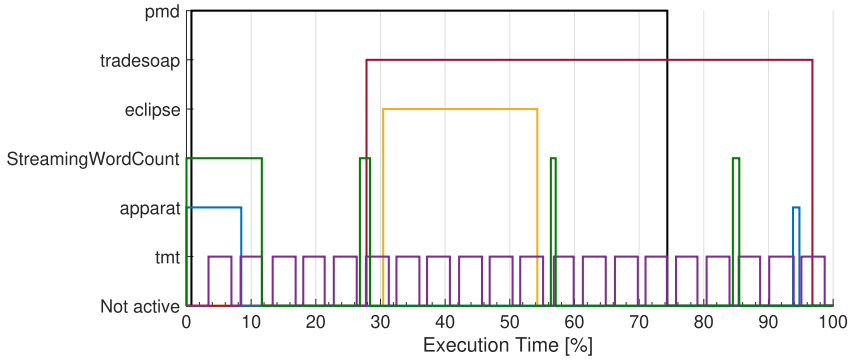
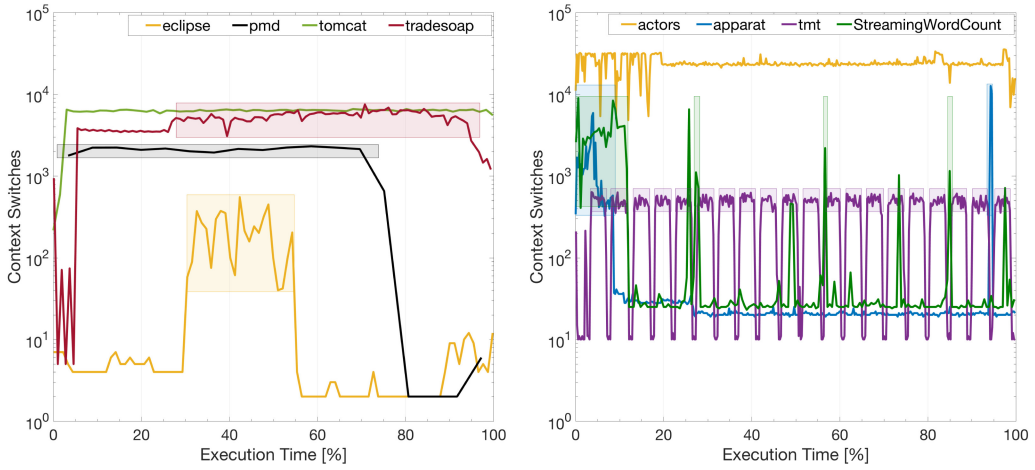


Fig. 10. Portion of workloads where fine-grained tasks are executed. Tasks are always in execution in benchmarks not reported here (actor and tomcat).

Task Execution. In tomcat and actors, fine-grained tasks are spawned and executed for the whole duration of the benchmark, while in all other applications, tasks are used only for a portion of the total workload. Figure 10 shows the portions of workloads that make use of fine-grained tasks. In pmd, tasks are used for the first ~74% of the benchmark execution (measured in wall time) where source code is loaded and parsed. In the last part, the benchmark executes sequentially, collecting the reports produced by each `PmdRunnable` and producing a single human-readable log file. In tradesoap, tasks are executed once the benchmark starts processing user transactions. The initial and final parts of the workload (performing setup or cleaning operations) do not make use of such tasks. Tasks in eclipse are used only in the central portion of the workload (from ~30% to ~54% of the execution time), when compilation units are read.

Fine-grained tasks in StreamingWordCount are spawned in four distinct time intervals. Most of the tasks (92%) are executed at the beginning of the workload (i.e., in the first 12% of the benchmark execution), while other tasks are spawned in short spikes occurring at the ~27%, ~56%, and ~85% of the execution time, where 3%, 3%, and 2% of the remaining tasks are spawned, respectively.



(a) DaCapo [6]. (b) ScalaBench [59] and Spark Perf [10].

Fig. 11. Context switches over execution time in benchmarks spawning fine-grained tasks. Measurements sampled during stop-the-world garbage collections have been removed. The colored regions superimposed to the lines represent the time intervals where fine-grained tasks are executed (shown in Figure 10). Tasks are executed for the whole duration of the workload in benchmarks with no colored regions. Data for some benchmarks has been downsampled to 10% (actors), 20% (eclipse, tomcat, and tradesoap), or 40% (apparat).

Similarly, apparat spawns most of the tasks (85%) in the first 8% of the workload, while others are created in a short spike toward the end of the execution. Finally, the tasks in tmt are executed in different batches (20 in total), each batch consisting of ~800 tasks. This behavior is due to the benchmark performing several cascading operations on the same dataset (such as mapping and reductions), each representing a batch. Each operation spawns new tasks.

Task Interference. Executing many fine-grained tasks in parallel may cause significant interference if there is need for synchronization. We measure the interference between fine-grained tasks by studying the number of context switches (cs) occurred during benchmark execution, reporting our results in Figure 11. To better relate context switches experienced with the execution of fine-grained tasks, we highlight the time intervals where fine-grained tasks are in execution (reported in Figure 10) with colored regions superimposed to the trends shown in the figure. A region matches only the trend of the same color. The absence of a colored region for a benchmark denotes that tasks are always in execution in such application.

As shown in the figure, in pmd, eclipse, apparat, and tmt we can observe noticeable increments in the amount of context switches experienced in the time intervals where fine-grained tasks are in execution, resulting in an average amount of context switches observed equal to 2,120cs/100ms (pmd), 225cs/100ms (eclipse), 1,359cs/100ms (apparat), and 491cs/100ms (tmt),[19] in contrast to the very few context switches experienced when fine-grained tasks are not executed. Similarly, the execution of tomcat and actors (where tasks are used for the whole duration of the workload) results in many context switches occurring throughout benchmark execution, with an average of 6,326cs/100ms (tomcat) and 24,118cs/100ms (actors). While contention in StreamingWordCount remains low for most of the workload execution, we can observe several short time intervals where

[19]Unless otherwise noted, the average amounts of context switches reported in this section consider only time intervals where fine-grained tasks are in execution.

a sudden increase of context switches can be observed. Most of these spikes occur in each time interval where tasks are used, resulting in an average of 3,343cs/100ms.

These observations suggest that in these seven benchmarks fine-grained tasks significantly interfere with each other, as the execution of fine-grained tasks is accompanied by a significant increase in the amount of context switches experienced by the application, while contention remains low when fine-grained tasks are not executed. On the other hand, while also the execution of fine-grained tasks in tradesoap causes a significant increment in the amount of context switches observed (i.e., from an average of 3,568cs/100ms to 5,500cs/100ms), high contention is present before fine-grained tasks are executed, and continues after they are not used anymore. This behavior suggests that, although fine-grained tasks increase the amount of context switches experienced by the benchmark, they are not the only cause of the high contention in tradesoap.

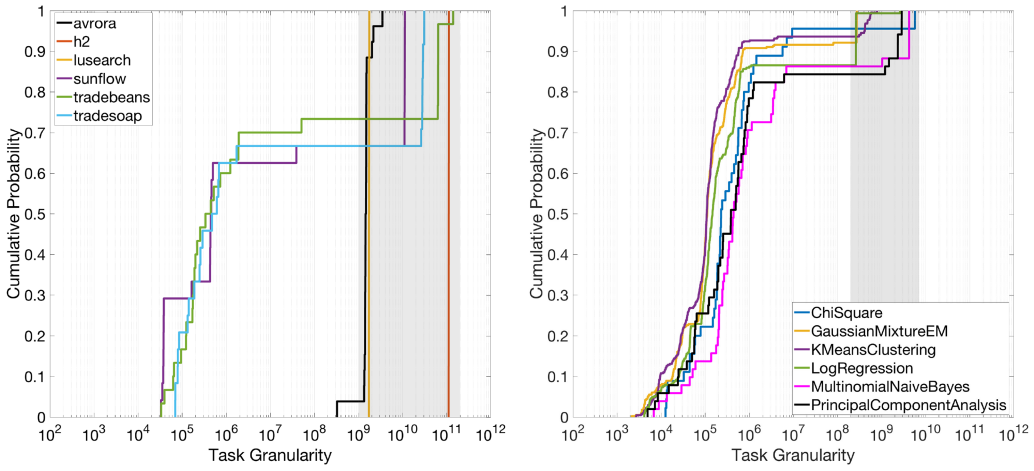
Summary. Overall, our analysis reveals that fine-grained tasks in eclipse, pmd, tomcat, actors, apparat, tmt, and StreamingWordCount cause significant contention during their execution. Such contention is highest in actors (experiencing many context switches throughout benchmark execution) and more modest in eclipse (due to the lower amount of context switches observed). To reduce the interference between fine-grained tasks, they could be merged together into a smaller number of larger tasks. This optimization is eased by the fact that fine-grained tasks in several benchmarks process either a single piece of data (e.g., pmd and tmt) or a small dataset (e.g., eclipse and StreamingWordCount). To improve task granularity in such applications, it would be sufficient to assemble data in groups (e.g., in case of pmd and tmt), modifying each task to process a group rather than a single element (in tmt, this optimization could be applied to tasks within each batch), or to increase the size of the dataset processed by a single task (e.g., in case of eclipse and StreamingWordCount). Section 11 confirms the benefit of such an optimization on pmd and StreamingWordCount. On the other hand, while in tradesoap fine-grained tasks interfere with each other as well, merging fine-grained tasks may not significantly decrease the contention experienced by the benchmark, which is caused only partially by the fine-grained tasks.

9.3 Coarse-Grained Tasks

We identify coarse-grained tasks as relatively large groups of tasks of the same class, with a similar and high granularity. We observe the presence of coarse-grained tasks in six benchmarks of the DaCapo suite (avrora, h2, lusearch, sunflow, tradebeans, and tradesoap) and six applications from Spark Perf (ChiSquare, GaussianMixtureEM, KMeansClustering, LogRegression, MultinomialNaiveBayes, and PrincipalComponentAnalysis).

Task Granularity. Figure 12 shows the CDF of the task granularity for the applications outlined above. To better see the presence of coarse-grained tasks in tradesoap, we remove fine-grained tasks from its CDF. In all DaCapo benchmarks (Figure 12(a)), the vertical trends in the rightmost portion of their CDFs (in the shaded region) pinpoint the presence of a small group of tasks with similar granularity. Moreover, the granularity of such tasks is several orders of magnitude higher that the one of other tasks spawned by the benchmarks. Such trends identify coarse-grained tasks, whose average granularity ranges from $\sim 10^9$ cycles (avrora) to $\sim 10^{11}$ cycles (h2). Similar observations can be made on Spark Perf (Figure 12(b)). All benchmarks in the suite feature similar CDFs, with the presence of few outlier tasks (in the shaded region) with a granularity significantly higher than the one of other tasks spawned, ranging from an average of $\sim 3 \cdot 10^8$ (GaussianMixtureEM and LogRegression) to $\sim 6 \cdot 10^9$ (ChiSquare).

Task Description. All coarse-grained tasks within a benchmark have the same class, whose purpose is discussed in the following text. The goal of avrora is to simulate the execution of



(a) DaCapo [6]. (b) Spark Perf [10].

Fig. 12. Cumulative Distribution Function (CDF) of the task granularity for benchmarks spawning coarse-grained tasks. The shaded region marks the presence of coarse-grained tasks. The CDF of tradesoap does not include fine-grained tasks.

26 programs on a grid of AVR microcontrollers. The benchmark runs each simulation in parallel in a separate `SimulatorThread`. h2 executes the TPC-C [71] online transaction benchmark on H2. Database transactions are divided into different groups, each of them executed in parallel by a task of class `TPCC$3`. lusearch uses the Apache Lucene [63] text search engine to query the presence of keywords over a corpus of data. Queries are divided into groups, each of them executed in parallel by a `QueryThread`. sunflow renders a set of images by dividing subparts of images into independent groups, each of them processed in parallel by a `BucketThread`.

The purpose of tradebeans is the same of tradesoap (explained in Section 9.2), with the difference that transactions are executed directly on the server without using SOAP messages. In both tradebeans and tradesoap, transactions are divided into groups, each of them carried out in parallel by a `DaCapoTrader`. Apart from executing transactions, such tasks are also responsible to populate and tear down the database. Except avrora, all benchmarks in DaCapo spawn as many coarse-grained tasks as available CPU cores (i.e., 8 in the machine used for the evaluation). Such tasks are used throughout benchmark execution, with the exception of h2, where they are not utilized in the final part of the workload (i.e., the last ~15% of the execution).

Finally, the purpose of each Spark Perf application is to run performance tests on different algorithms offered by the machine learning library of Spark (MLlib [61]). All coarse-grained tasks in these benchmarks are of class `TaskRunner`. As discussed in Section 9.2, each `TaskRunner` encapsulates the execution of a single *task* (under Spark terminology). The computation carried out by each Spark Perf application is divided into a variable number of Spark tasks, resulting in diverse instances of `TaskRunner` spawned, ranging from a minimum of 5 (ChiSquare) to a maximum of 45 (LogRegression), as summarized in Table 8. Coarse-grained tasks are used during the whole execution of each Spark Perf benchmark.

CPU Utilization. Table 9 depicts the average CPU utilization of each benchmark. In most of them, the CPU is far from being fully utilized, especially in avrora, GaussianMixtureEM, and h2 where the average CPU utilization is 9%, 18%, and 19%, respectively. The highest average CPU utilization occurs in sunflow (96%), followed by MultinomialNaiveBayes (76%). All other benchmarks

Table 9. Average CPU Utilization for Benchmarks Spawning Coarse-Grained Tasks, with 95% Confidence Intervals

Benchmark DaCapo [6]	CPU Util. [%] (± 95% conf.)	Benchmark Spark Perf [10]	CPU Util. [%] (± 95% conf.)
avrora	9.01 ± 2.81	ChiSquare	45.52 ± 12.90
h2	18.94 ± 0.98	GaussianMixtureEM	18.16 ± 12.47
lusearch	58.34 ± 11.31	KMeansClustering	28.75 ± 12.24
sunflow	95.83 ± 8.17	LogRegression	38.33 ± 10.55
tradebeans	29.81 ± 2.19	MultinomialNaiveBayes	76.11 ± 16.11
tradesoap	44.75 ± 3.59	PrincipalComponentAnalysis	26.82 ± 14.68

In h2, the values shown are obtained by considering only measurements sampled when coarse-grained tasks are in execution. In all benchmarks, measurements sampled during stop-the-world garbage collections are not considered.

feature an average utilization ranging from 27% (PrincipalComponentAnalysis) to 58% (lusearch), pinpointing that a significant portion of the available processing capacity is not utilized on average. These observations remark that the use of coarse-grained tasks results in missed parallelization opportunities in most benchmarks (excluded sunflow and MultinomialNaiveBayes, where the CPU is overall better utilized), as the applications fail to fully utilize the available computing resources.

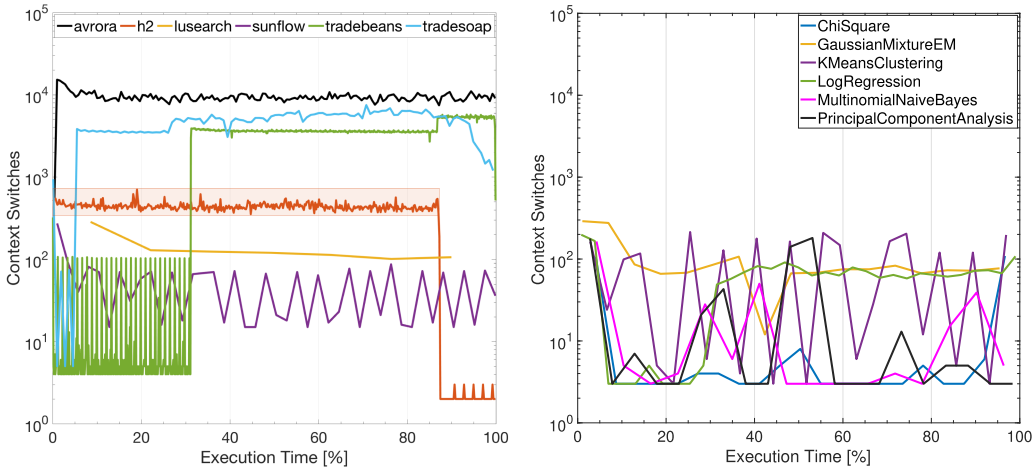
Task Interference. With the goal of better utilizing the available CPU cores, the coarse-grained tasks outlined above could be split into multiple tasks with smaller granularity. However, the need for synchronization among tasks may severely limit the benefits of a lower task granularity, as more active tasks may result in more blocking primitives called, increasing the contention among tasks. To this end, we measure the interference between the executed tasks by studying the amount of context switches experienced during benchmark execution.

Figure 13 reports our results. Regarding DaCapo (Figure 13(a)), the applications experiencing less context switches over their execution are sunflow and lusearch, with an average of 51cs/100ms and 140cs/100ms, respectively. In contrast, the application mostly affected by context switches is avrora, with an average of 9,421cs/100ms, suggesting that avrora makes extensive use of blocking primitives that cause severe contention during task execution. On the other hand, the much lower amount of context switches experienced by sunflow and lusearch pinpoints that the interference between their tasks is small.

Regarding tradebeans and tradesoap, these benchmarks are little subjected to context switches in the first part of the workload (i.e., the first ∼31% of execution in tradebeans, and the first ∼5% in tradesoap), but experience a noticeable increase in the trend in the second part, resulting in an average of 3,948cs/100ms and 5,036cs/100ms for tradebeans and tradesoap, respectively. This behavior is caused by the fact that the first part of the two benchmarks is dominated by database population (which incurs in low contention between tasks), while the rest is dedicated to the execution of transactions (where contention is higher). h2 incurs in an average of 441cs/100ms in most of the workload, while this number drops suddenly in the final part, where tasks are not used, resulting in a sequential execution that incurs context switches only occasionally.

Finally, all Spark Perf benchmarks (Figure 13(b)) exhibit a modest amount of context switches throughout their execution, ranging from an average of 14cs/100ms (ChiSquare) to a maximum of 97cs/100ms (GaussianMixtureEM), which indicates small contention occurring between different parallel tasks in the six Spark applications.

Summary. Overall, our analysis suggests that in several benchmarks coarse-grained tasks could be split into multiple smaller tasks to better leverage the available CPU and improve application



(a) DaCapo [6]. (b) Spark Perf [10].

Fig. 13. Context switches over execution time in benchmarks spawning coarse-grained tasks. Measurements sampled during stop-the-world garbage collections have been removed. Data on tradesoap has been down-sampled to 20%. The colored region superimposed to the trend of h2 represents the time interval where coarse-grained tasks are executed.

performance. The optimization is eased by the fact that coarse-grained tasks in all benchmarks (except avrora) process independent groups of data. To improve task granularity, it would be sufficient to divide data in a higher number of groups of lower size, keeping each group processed by a single task; we evaluate the benefits of such an optimization in Section 11.

This optimization is likely to yield major benefits in lusearch, ChiSquare, GaussianMixtureEM, KMeansClustering, LogRegression, and PrincipalComponentAnalysis, where the interference between tasks is lower and computing resources can be better utilized. A similar approach could also speed up database population in tradebeans and tradesoap, where interference between tasks is low. On the other hand, while applying the proposed approach to other benchmarks is possible, it is less likely that modifying task granularity there results in noticeable optimizations, as the CPU is already well utilized in sunflow and MultinomialNaiveBayes, and contention in avrora, h2, and the second part of tradebeans and tradesoap is significant, which may overcome the benefits of processing the application with a higher number of tasks.

10 TASK-GRANULARITY OPTIMIZATION

In this section, we show how tgp eases the optimization of task granularity, focusing on one benchmark where many fine-grained tasks interfere with each other (pmd) and on another benchmark where the presence of coarse-grained tasks limits the utilization of the available computing resources (lusearch). We also describe our approach to optimize task granularity in Spark Perf.

10.1 pmd

Each `PmdRunnable` (despite its name, the class implements `Callable`) receives the source-code file to process as an argument to its constructor. Our profiler detects that such tasks are submitted to a single `ThreadPoolExecutor`. Thanks to calling-context profiling enabled by tgp, we determine the class and method where the tasks are created and submitted. In particular, such tasks are created in the method `processFiles` of the class `PMD`. The creation calling context of all `PmdRunnable` objects is the same, indicating that all tasks are likely to be created altogether in a loop-like

construct in `processFiles`. Moreover, the submission calling context of all `PmdRunnable` objects is equal to their creation calling context, a sign that a task is submitted right after its creation in `PMD.processFiles`.

To optimize task granularity in pmd, we perform the following actions. First, we modify the implementation of class `PmdRunnable`, enabling the processing of multiple source-code files. In particular, we modify the constructor (taking a list of files as input) as well as its `call` method (returning a list of reports, one for each processed file). Second, since the input data is provided to each task upon its creation, we modify the code where such tasks are created (in `PMD.processFiles`). Our modification allows the user to set the number of tasks spawned (n) through a command-line parameter. Input files are then clustered in n groups (a group contains approximately $570/n$ files), each of them processed by a single task. Third, since each task will now produce multiple reports, we modify task submission in `PMD.processFiles`, enabling the collection of multiple reports upon the completion of each task. Our modifications concern only task granularity; they do not alter the behavior of the application or the results produced, require the modification of only 32 lines of code, and do not require any specific knowledge of the benchmark.

10.2 lusearch

The benchmark carries out a total of 128 queries, dividing them into equally sized groups, each of them processed by a `QueryThread`. The benchmark spawns as many `QueryThread` instances as available CPU cores. As the name suggests, such tasks subclass `Thread`. Similarly to `PmdRunnable`, all `QueryThread` instances feature the same creation calling context. Moreover, the starting calling contexts of all `QueryThread` instances match their creation calling context. This fact indicates that all threads are created together and started right after creation. Creation and starting occur in class `Search` in method `main`.

Optimizing task granularity in lusearch involves spawning more `QueryThread` objects, each of them processing a smaller number of queries. To this end, we modify lusearch as follows. First, we make `QueryThread` implement `Runnable` rather than subclassing `Thread`, transforming that class into a more lightweight entity that can be scheduled more efficiently. Second, we allow the user to specify the number of `QueryThread` instances created by the benchmark. To this end, we modify the code portions inside `Search.main` responsible to create the tasks. Finally, we introduce a thread-pool making use of as many threads as available cores, submitting a `QueryThread` to such thread-pool right after its creation. This modification still occurs in `Search.main`. If more `QueryThread` instances than available cores are spawned (as specified by the user), our modifications lead to a lower number of queries carried out by each `QueryThread` wrt. the original application, as queries are equally distributed among all spawned `QueryThread` objects. Similarly to pmd, the modifications applied to lusearch are only aimed at optimizing task granularity (i.e., they do not alter the logic of the benchmark or the results produced) and alter only 13 lines of code.

10.3 Spark Perf Benchmarks

In Spark, each application performs computations on *resilient distributed datasets* (RDDs) [74], i.e., on immutable collections of elements divided into *partitions*, which represent the portions of data that can be processed in parallel. Data must be inserted into an RDD before being processed. A Spark application can be seen as a sequence of operations over RDDs, organized in *jobs* composed of a variable number of *stages*. The number of stages inside a job depends on which operations are used by the job (e.g., in general, `reduceByKey` or `groupByKey` introduce additional stages, while `map` or `filter` do not). Between each stage, data inside an RDD is repartitioned in a new RDD. Each stage is composed of a collection of *tasks* executing the same code on different partitions of

the RDDs. Overall, the final number of tasks into which a Spark application is divided depends on several variables, such as the number of jobs created, the number of stages a job requires, and the number of partitions in each RDD.[20]

Coarse-Grained Tasks. To optimize task granularity in the Spark Perf applications spawning coarse-grained tasks, we modify the number of partitions of the RDD created at the beginning of the application to store the initial dataset. In such applications, the initial RDD is created from a textual file stored in the local filesystem, via the `textFile` method (defined in `SparkContext` [66]), which allows setting the desired number of partitions of the RDD as an optional argument. In the original applications, such an argument is not passed to `textFile`, resulting in a default number of partitions being generated by Spark in the RDD (this number depends on the size and format of the input data). We control the number of partitions generated by setting this optional argument, modifying a single line of code in each benchmark. Since a different number of partitions results in a different number of tasks spawned, our approach enables us to decrease task granularity by increasing the number of partitions of the RDD.

Fine-Grained Tasks. Differently from all other Spark Perf benchmarks considered in this article, in StreamingWordCount (spawning fine-grained tasks) the input data is received in the form of a continuous stream of textual data from a socket, leveraging the Spark Streaming API [65]. Data received is assembled in RDDs with multiple partitions periodically (every 100ms), resulting in a high number of fine-grained tasks to process data in the RDDs.

To increase task granularity in StreamingWordCount, we modify the number of partitions of the RDDs created between different stages. Each Spark operation introducing additional stages accepts an optional parameter specifying the desired number of partitions of the new RDD. If no number is specified, Spark uses the value of the `spark.default.parallelism` configuration variable. StreamingWordCount neither specifies any explicit partition number, nor sets the value of `spark.default.parallelism`, resulting in a default number of partitions being generated by Spark in the new RDDs (in local mode, this is equal to the number of available cores [64]). We modify the number of partitions created after each stage by explicitly setting `spark.default.parallelism`. Since the number of partitions determines the amount and the granularity of the spawned tasks, our approach allows us to control the number of tasks spawned by StreamingWordCount by adding a single line of code in the application (to set the value of `spark.default.parallelism`).

Similarly to pmd and lusearch, our modifications to Spark Perf benchmarks concern only task granularity. They neither modify the behavior of the application nor the results produced. Moreover, they do not require any specific knowledge of the benchmark. Finally, they require only minimum modifications to the target application (i.e., a single line of code) to be implemented.

11 EVALUATION OF OPTIMIZED APPLICATIONS

Here, we discuss the speedup enabled by our task-granularity optimizations presented in the previous section.

11.1 Methodology and Setup

We execute the modified benchmarks on different environments. Apart from the machine used for task-granularity analysis, where applications are set up to use 8 cores on a single NUMA node as described in Section 8.1, we use two additional environments: the same machine, where applications are not bound to run on a single NUMA node (i.e., they can use 16 physical cores), and an

[20] More details on the scheme used by Spark to divide computations into tasks can be found in [55, 56, 60].

additional machine, equipped with an Intel i7-4710MQ (2.5GHz) processor with 4 physical cores, 8GB of RAM, running under Ubuntu 16.04.4 LTS (kernel GNU/Linux 4.4.0-116-generic x86_64), with Turbo Boost and Hyper-Threading disabled, where the governor is set to "performance" to disable frequency scaling. The version of all other software used is the one described in Section 8.1.

11.2 Approach

In each environment, we run the modified benchmarks in different settings, each with a different number of tasks spawned.

DaCapo. In pmd and lusearch, we start from the same number of tasks used in the original applications (i.e., 570 `PmdRunnable` objects and as many `QueryThread` instances as available cores). In each setting, we iteratively halve the number of `PmdRunnable` objects used by pmd and double the number of `QueryThread` instances employed by lusearch. Our approach materializes in a progressively higher task granularity in pmd, as each task processes more files in subsequent settings. Similarly, task granularity in lusearch becomes smaller, as the number of queries processed by each task is halved in each setting. In pmd, we continue this process until the benchmark would spawn less tasks than available CPU cores. In lusearch, we stop adding tasks when 128 tasks are used (more tasks would not be utilized, since the total number of queries to be processed is 128).

Spark Perf (Coarse-Grained Tasks). In all Spark Perf benchmarks using coarse-grained tasks, we decrease task granularity as follows. We start by setting the number of partitions of the initial RDD to 8 (lower values do not result in a higher number of tasks spawned, as they are lower that the default number of partitions used by Spark in the original application). In subsequent settings, we iteratively double the number of partitions, resulting in a progressively higher number of tasks spawned, each with decreasing granularity. Note that the final number of `TaskRunner` objects created by each application is proportional but is not equal to the number of partitions set, as the use of multiple stages or jobs in the application can cause a much larger number of tasks spawned, as outlined in Section 10.3. We continue our approach until the execution time of the modified benchmark is higher than the original one (denoting the presence of fine-grained tasks causing interference between each other).[21]

Spark Perf (Fine-Grained Tasks). The original StreamingWordCount repartitions RDDs between different stages in as many partitions as available cores. We increase task granularity as follows. We start by setting `spark.default.parallelism` to the number of available cores. In subsequent settings, we iteratively halve the value of `spark.default.parallelism`, which results in a lower amount of tasks spawned, each with increasing granularity. We continue our approach until `spark.default.parallelism` is set to 1, or no speedup can be observed.

11.3 Results

Tables 10, 11, and 12 report the results of task-granularity optimization on 8 cores, 16 cores, and 4 cores, respectively. For each setting, the table reports the number of tasks spawned and the speedup obtained w.r.t. the original (unmodified) application. Speedup is presented in terms of *speedup factors*, defined as the execution time of the original application divided by the execution time of the modified application. The values shown are the average over 20 steady-state runs (within the same JVM process). We also report 95% confidence intervals. The number of tasks spawned has been measured with tgp in a separate run (tgp is not active when measuring speedups).

[21] We do not apply the proposed optimization to MultinomialNaiveBayes because the benchmark well utilizes CPU (as shown in Section 9.3).

Table 10. Speedup (Including 95% Confidence Intervals) Resulting from Task-granularity Optimization on an 8-Core Machine

	pmd		StreamingWordCount			
Fine-grained tasks	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)		
	570	Baseline	1,852	Baseline		
	285	1.2577 ± 0.0085	976	1.2176 ± 0.0113		
	143	1.3548 ± 0.0136	543	1.0527 ± 0.0098		
	72	1.3597 ± 0.0142				
	36	1.3644 ± 0.0089				
	18	1.3736 ± 0.0117				
	9	1.3799 ± 0.0094				
	lusearch		ChiSquare		GaussianMixtureEM	
Coarse-grained tasks	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)
	8	Baseline	5	Baseline	35	Baseline
	16	1.0500 ± 0.0097	9	1.9644 ± 0.0047	69	1.7934 ± 0.0048
	32	1.0552 ± 0.0172	17	3.7961 ± 0.0694	137	2.2173 ± 0.1083
	64	1.0613 ± 0.0174	33	3.2505 ± 0.0938	273	2.1599 ± 0.0229
	128	1.0438 ± 0.0117	65	2.9895 ± 0.0954	545	2.1397 ± 0.0340
			129	2.8045 ± 0.0350	1,089	1.9778 ± 0.0273
			257	2.2582 ± 0.0185	2,177	1.6650 ± 0.0199
			513	1.5004 ± 0.0066		
	KMeansClustering		LogRegression		PrincipalComponentAnalysis	
	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)
	32	Baseline	45	Baseline	9	Baseline
	64	1.8196 ± 0.0040	89	1.8236 ± 0.0046	22	2.0475 ± 0.0305
	128	2.7234 ± 0.0844	220	2.4127 ± 0.0928	41	2.0101 ± 0.0476
	256	2.4808 ± 0.0536	440	2.1737 ± 0.0279	75	1.9527 ± 0.0384
	512	2.0943 ± 0.0226	814	2.0375 ± 0.0168	145	1.9472 ± 0.0465
	1,024	1.7579 ± 0.0171	1584	1.7709 ± 0.0147	277	1.8387 ± 0.0502
	2,048	1.3598 ± 0.0069	3037	1.4438 ± 0.0108	545	1.7668 ± 0.0136
					1,069	1.6269 ± 0.0073
					2,113	1.2963 ± 0.0135

The best result for each benchmark is highlighted.

For all benchmarks, we observe significant speedups in all the environments considered. Regarding DaCapo, employing less tasks with larger granularity in pmd leads to noticeable speedups, reaching a peak when 9 (in the 8-core and 4-core machine) or 18 tasks (in the 16-core machines) are used. These results are a sign of a reduced need for synchronization among tasks, resulting in less contention as well as reduced scheduling overhead. Oppositely, reducing task granularity in lusearch allows significant speedups, which are maximum when 64 (in the 8-core machine), 32 (in the 16-core machine) or 8 `QueryThread` objects (in the 4-core machine) are used. These results suggest that the higher number of tasks used enables better CPU utilization. Further creating tasks results in lower speedups, a sign that the contention between tasks increases, jeopardizing

Table 11. Speedup (Including 95% Confidence Intervals) Resulting from Task-Granularity Optimization on a 16-core Machine

	pmd		StreamingWordCount			
Fine-grained tasks	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)		
	570	Baseline	3,661	Baseline		
	285	1.1623 ± 0.0286	1,833	1.0102 ± 0.0050		
	143	1.2340 ± 0.0372	986	1.0274 ± 0.0060		
	72	1.2591 ± 0.0309	576	1.0703 ± 0.0057		
	36	1.2777 ± 0.0325	380	1.1345 ± 0.0063		
	18	1.2955 ± 0.0386				
	lusearch		ChiSquare		GaussianMixtureEM	
Coarse-grained tasks	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)
	16	Baseline	5	Baseline	35	Baseline
	32	1.3326 ± 0.0484	9	1.8014 ± 0.0150	69	1.7791 ± 0.0158
	64	1.2805 ± 0.0438	17	3.8854 ± 0.0462	137	2.8545 ± 0.0336
	128	1.2682 ± 0.0374	33	5.8952 ± 0.1138	273	3.2997 ± 0.0938
			65	5.8903 ± 0.0922	545	3.0110 ± 0.0343
			129	4.6151 ± 0.0622	1,089	2.8445 ± 0.0410
			257	3.1957 ± 0.0589	2,177	2.5317 ± 0.0253
			513	2.3752 ± 0.0380	4,353	1.7381 ± 0.0147
			1,025	1.5354 ± 0.0216		
	KMeansClustering		LogRegression		PrincipalComponentAnalysis	
	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)
	32	Baseline	45	Baseline	9	Baseline
	64	1.7906 ± 0.0115	89	1.8107 ± 0.0106	22	2.1849 ± 0.0109
	128	2.8504 ± 0.0374	220	2.7036 ± 0.0233	41	3.5418 ± 0.1045
	256	3.3416 ± 0.0439	440	3.4413 ± 0.0833	75	3.3867 ± 0.0995
	512	3.1176 ± 0.0422	814	2.9703 ± 0.0364	145	3.2746 ± 0.0857
	1,024	2.9112 ± 0.0276	1,584	2.7148 ± 0.0223	277	3.2475 ± 0.0874
	2,048	2.3676 ± 0.0161	3,037	2.3428 ± 0.0174	545	3.2371 ± 0.0246
	4,096	1.7848 ± 0.0089	5,984	1.7437 ± 0.0188	1,069	2.8603 ± 0.0198
					2,113	2.2715 ± 0.0168
					4,185	1.7741 ± 0.0122

The best result for each benchmark is highlighted.

the benefits of a better CPU utilization. The maximum speedup achievable in pmd and lusearch is 1.38× (8-core machine) and 1.33× (16-core machine), respectively.

Similar observations hold for Spark Perf benchmarks. Decreasing the amount of tasks spawned in StreamingWordCount yields a maximum speedup of 1.22× (8-core machine) when 976 tasks are used. All other Spark Perf applications benefit from task-granularity optimization to a greater extent than the DaCapo benchmarks and StreamingWordCount, as indicated by the higher speedups observed. Here, increasing task granularity on the 8-core machine leads to a progressively higher speedup, reaching peaks above 2.00× in all benchmarks, and above 3.00× (3.80×) in ChiSquare.

Table 12. Speedup (Including 95% Confidence Intervals) Resulting from Task-Granularity Optimization on a 4-core Machine

	pmd		StreamingWordCount			
Fine-grained tasks	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)		
	570	Baseline	876	Baseline		
	285	1.1568 ± 0.0112	423	1.0393 ± 0.0693		
	143	1.2518 ± 0.0277				
	72	1.2701 ± 0.0157				
	36	1.2894 ± 0.0110				
	18	1.3154 ± 0.0126				
	9	1.3285 ± 0.0122				
	4	1.2237 ± 0.0082				
	lusearch		ChiSquare		GaussianMixtureEM	
Coarse-grained tasks	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)
	4	Baseline	5	Baseline	35	Baseline
	8	1.1427 ± 0.0284	9	1.4685 ± 0.0067	69	1.3716 ± 0.0455
	16	1.1048 ± 0.0263	17	1.5185 ± 0.0134	137	1.2497 ± 0.0164
	32	1.0915 ± 0.0345	33	1.3400 ± 0.0241	273	1.2001 ± 0.0150
	64	1.0709 ± 0.0349	65	1.2736 ± 0.0102	545	1.0896 ± 0.0121
	128	1.0620 ± 0.0293	129	1.1778 ± 0.0075		
	KMeansClustering		LogRegression		PrincipalComponentAnalysis	
	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)	# tasks	Speedup (± 95% conf.)
	32	Baseline	45	Baseline	9	Baseline
	64	1.3797 ± 0.0503	89	1.3377 ± 0.0226	22	1.0199 ± 0.0611
	128	1.2780 ± 0.0500	220	1.2204 ± 0.0232	41	1.0401 ± 0.0455
	256	1.1826 ± 0.0261	440	1.1323 ± 0.0130	75	1.0366 ± 0.0468
	512	1.0381 ± 0.0390	814	1.0368 ± 0.0117		

The best result for each benchmark is highlighted.

Increasing the number of available cores makes the benefit of optimizing task granularity more evident (see Table 11), with speedups above 3.00× in all five Spark Perf benchmarks suffering from coarse-grained tasks, and equal to 5.90× in ChiSquare, sign that tasks in Spark Perf can well exploit the available computing resources if an optimal number of tasks is used. Task-granularity optimization leads to improved application performance also when a lower number of cores is available, as indicated by Table 12. On a 4-core machine, the maximum speedup observable is 1.52× (ChiSquare). The lower speedups obtained on this environment can be determined by the lower performance of the used machine (i.e., less computing cores and reduced memory), which may limit the resources exploitable by an optimized task granularity, especially in StreamingWordCount.

In summary, our evaluation results show that optimizing task granularity can lead to significant performance improvements, resulting in speedups up to a factor of 5.90×. Our profiler plays a fundamental role to this end, as it enables one to locate tasks suffering from too fine- or coarse-grained granularities, and assists the user in their optimization. In pmd and lusearch, all

the classes and methods modified during task-granularity optimization are directly indicated by tgp, which avoids the need of manually locating the application code to be modified.

12 DISCUSSION

In this section, we discuss design decisions behind our approach as well as the limitations of our profiling methodology.

12.1 Excluded Metrics

The metrics collected by tgp result from a careful selection process considering a larger set of metrics. Before focusing on reference cycles, we considered other metrics to represent task granularity: bytecode count, machine-instruction count, and wall time.

Bytecode count (i.e., the number of bytecodes executed by a task) is less affected by perturbations caused by the instrumentation than reference-cycle count (as the number of bytecodes introduced by the instrumentation is known and can be excluded from the count); however, it cannot track code without a bytecode representation (such as native methods), may account bytecodes that are not executed due to optimizations performed by the JVM's dynamic compiler, and represents bytecodes of different complexity with the same unit (e.g., a complex floating-point division has the same weight as a simple *pop*). Moreover, collecting bytecode count requires the insertion of instrumentation code in the basic blocks of code in the methods of the loaded classes. Such an extensive instrumentation may significantly perturb dynamic compilation and impair optimizations performed by the dynamic compiler, such as escape analysis, scalar replacement of objects, and method inlining, unless the dynamic compiler is specifically designed to avoid such perturbations (e.g., by recognizing explicitly marked instrumentation code [76]).

Similarly to bytecode count, also machine-instruction count (i.e., the number of machine instructions executed by a task) represents computations of different complexity with the same unit. Moreover, it cannot track latencies caused by cache misses or misalignments. Finally, wall time (i.e., the difference between the ending and starting task-execution timestamps) may include time intervals where a task is not scheduled, resulting in an overestimation of task granularity. In contrast, reference-cycle count does not suffer from such limitations.

At the OS layer, we originally considered the number of core migrations incurred by the observed application. This metric is subsumed by context switches (a core migration may happen only during a context switch), which can better show contention among tasks, as it accounts also for the execution of blocking primitives not causing a core migration.

Finally, we considered also the number of cache misses caused by the observed application. We excluded that metric because we found it to be little related to task granularity or task contention.

12.2 Task Interfaces, Execution Frameworks, Aggregation and Multiple Task Executions

Our definition of task interfaces includes `Runnable`, `Callable`, and `ForkJoinTask`, because they are the interfaces/classes most commonly used to define parallel entities in multithreaded applications on the JVM. However, the Java API [43] does not oblige developers to represent tasks as subtypes of such task interfaces. As a result, entities that are subtypes of neither `Runnable`, nor `Callable`, nor `ForkJoinTask` may still be used as parallel tasks, without being profiled by tgp. A similar observation holds for entities used as task execution frameworks without being subtype of `Executor`. This particularly concerns old applications written before Java 5 was released, when the interfaces `Callable`, `ForkJoinTask`, and `Executor` were not provided by the Java class library, forcing developers to define custom interfaces to represent tasks and task execution frameworks.

To enable the profiling of such entities, we plan to allow users to provide custom specifications of task interfaces, execution methods, task execution frameworks, and submission methods.

Similarly, our heuristic for task aggregation reduces the complexity of the resulting profiles for applications using common coding patterns enabled by the Java class library. For example, our heuristic assumes that the application uses the task execution frameworks defined by the Java API (such as `ThreadPoolExecutor` and `ForkJoinPool`, which make use of helper and adapter tasks). For applications not making use of such coding patterns, our heuristic may not be appropriate, e.g., it may aggregate important tasks whose granularities should be studied separately. In such applications, the heuristic can be disabled, effectively skipping the second step of our profiling methodology (see Figure 2). We have verified that the results presented in the article would not have changed with our heuristic disabled.

Among the benchmarks analyzed in Section 9, the heuristic proved particularly useful when analyzing task granularity in tradesoap, where 209,975 helper tasks resulting from the use of thread pools were aggregated to their outer task. Without our heuristic, such tasks would have resulted as very fine-grained tasks in the profile. However, such tasks do not execute any user-defined workload, and their purpose is limited to set up the execution of `ConnectorEndPoint` tasks and handle exceptions triggered by them. Since their execution is tied to `ConnectorEndPoint` tasks, users should focus on such tasks during task-granularity analysis. This is eased by our heuristic, which removes helper tasks from the profile.

Finally, an alternative approach could treat multiple task executions as a single task, summing up the different collected granularities in a single task profile, in contrast to considering them as separate tasks, as we do (see Section 3.6). We have verified that such an approach does not impact any of the results presented in this article.

12.3 JVM Services

Our tool does not aim at detecting all GC activities; rather, tgp monitors only stop-the-world collections. This is motivated by the fact that application threads are stopped during such collections; hence, OS-layer metrics sampled during these activities are not related to task execution, and are typically not of interest during task-granularity analysis. Detecting stop-the-world collections allows one to ignore such measurements, enabling a better study of task interference and CPU utilization. While concurrent collections (i.e., those occurring along with application execution) are not monitored by tgp, the analysis presented in Section 9 is conducted using the parallel GC [45], which does not perform concurrent collections.

In contrast, tgp does not remove OS-layer metrics collected during dynamic compilation. This is motivated by the fact that dynamic compilation is typically performed concurrently with application execution; hence, filtering out OS-layer metrics collected during dynamic compilation would also remove metrics related to the execution of application code. To minimize the chances of considering OS-layer metrics related to dynamic compilation, the task-granularity analysis presented in this article focuses only on the steady-state, where compiler threads are rarely active and are not likely to alter significantly the collected metrics. Moreover, in the JVM used for our analysis, reference cycles related to dynamic compilation are not accounted, because compilation occurs in dedicated native threads, which are not monitored by tgp.

12.4 Limitations

Several metrics collected by tgp can be biased by perturbations caused by the inserted instrumentation code. Besides, instrumentation may influence thread scheduling. Our profiling methodology takes several measures to keep perturbations low, including minimal and efficient instrumentation

that minimizes heap allocations in the observed JVM, use of low-overhead HPCs, and profiling data structures built in a separate process (Shadow VM). Moreover, expensive operations such as task aggregation, trace alignment, and calling-context profiling are performed after application execution. These measures result in low profiling overhead and reduce the chances for significant perturbations, as discussed in Section 8. In applications with a high *task spawn rate* (defined as the number of created tasks over unit of time), such as actors, profiling overhead and measurement perturbations can still be significant (see Figures 7 and 8), reducing the accuracy of the collected profiles. Nonetheless, a high task spawn rate can be sign of suboptimal performance. Our tool enables one to detect applications with a high task spawn rate, and aids their optimization thanks to calling-context profiling.

Other processes executing concurrently with the observed application may interfere with tgp. In particular, such processes may increase the CPU utilization of the system (biasing the measurements performed by top, as the values observed do not refer only to the execution of the observed application) and the contention on blocking primitives (which may result in more context switches experienced by the target application w.r.t. an execution without concurrent applications). Ensuring that no other CPU-, memory-, or IO-intensive process executes concurrently with the observed application reduces the chances of experiencing this limitation, as we did when conducting our task-granularity analysis. Moreover, in environments with multiple NUMA nodes, binding the execution of the observed JVM to an exclusive node can further reduce the effect of this limitation. We use such a setting when evaluating our approach (Section 8) and conducting task-granularity analysis (Section 9).

The applications analyzed in this article may exhibit different behaviors in different environments. The number of coarse-grained tasks spawned in h2, lusearch, sunflow, tradebeans, and tradesoap is determined by the number of available CPU cores. On machines with more CPU cores, more tasks will be spawned, which may result in a reduced task granularity. On the contrary, task granularity may increase on machines with fewer CPU cores. In addition, a different number of available CPU cores may change the resulting CPU utilization. More computing resources may decrease the overall CPU utilization of the target application, thus increasing the benefits of our task-granularity optimizations on coarse-grained tasks, as indicated by the higher speedups obtained on the 16-core machine (Table 11). On the other hand, fewer CPU cores may limit the benefits of our approach on coarse-grained tasks. While we observe such an effect in our evaluation (Table 12), the achieved speedups in all settings show that the considered applications suffered from suboptimal task granularity in all our measurement environments before optimization.

Our approach uses context-switches as a measure for contention among tasks. While our results show a strong correlation between periods of many context switches and task execution in many benchmarks, tgp cannot detect whether such context switches originate from task interference or from other causes (such as frequent I/O activities unrelated to tasks). Nonetheless, for all the applications analyzed in Section 9, we manually verified that periods of many context switches are caused by contention among tasks, unless otherwise noted.

Finally, most of the metrics used to analyze task granularity are platform-dependent; hence, the reproducibility of any task-granularity analysis may be limited. However, our choice of using platform-dependent metrics is well justified by their efficacy in describing task granularity, CPU utilization, and contention between tasks. While profiling some metrics requires the presence of performance counters either in the OS (e.g., for measuring context switches) or in the hardware (e.g., for measuring reference cycles), such counters are available in most modern operating systems and processors.

13 RELATED WORK

Here, we discuss work related to our approach. First, we consider related techniques for estimating, adapting, and profiling task granularity in task-parallel applications running in shared-memory multicores. Then, we outline research efforts based on the work-span model.

13.1 Estimating Task Granularity

Several authors focus on estimating the granularity of parallel tasks before they are spawned at runtime. The main motivation for estimating task granularity is to predict whether a task would execute significant work before actually creating it. The estimations can be used by an underlying framework to avoid spawning a task predicted as very fine-grained (i.e., if the overhead of creating and scheduling the task is higher than the expected benefits of executing work in parallel).

Acar et al. [1] propose *oracle scheduling*, a technique based on an oracle that estimates the granularity of each task declared by developers. While oracle scheduling can automatically determine an optimal task granularity and avoid creation of unnecessary tasks, it requires users to provide asymptotic cost functions of the target applications, and resorts to profiling to increase the accuracy of the estimations. Lopez et al. [33] introduce a static method to estimate task granularity, computing cost functions via static analysis and performing static transformations such that the transformed program automatically controls granularity. Unfortunately, programs without easily derivable asymptotic cost functions are less likely to benefit from the above techniques, as both approaches heavily rely on accurate cost functions. On the other hand, our analysis does not require the provision of any cost function and thus can better benefit complex or large applications.

Huelsbergen et al. [19] present Dynamic Granularity Estimation, a technique to estimate task granularity by examining the runtime size of data structures. Their approach relies on a framework composed of a compile-time component (which identifies the functions whose execution time depends on the size of the associated data structures) and a runtime component (which approximates the size of the data structures and thus task granularity). Another approach by Zoppetti et al. [78] estimates task granularity to generate threads executing large enough tasks, such that the context switching cost is relatively small compared to the cost of performing the actual computation.

A common limitation of the above techniques is that they fall short in pinpointing missed parallelization opportunities related to coarse-grained tasks, as they mainly focus on avoiding the overhead of creating and scheduling fine-grained tasks. In contrast, our work enables one to identify and optimize performance drawbacks of both fine- and coarse-grained tasks. Moreover, they mainly target languages that allow developers to specify parallelism through tasks but defer the decision regarding whether a new task is spawned to the runtime framework. Other languages (including JVM languages) may not benefit from the above techniques. On the other hand, our work focuses on task-parallel applications running on the JVM, where every task specified by developers is executed at runtime.

13.2 Adapting Task Granularity

Several authors propose techniques to adapt task granularity at runtime to an optimal estimated value. Thoman et al. [70] present an approach that enables automatic granularity control for recursive OpenMP applications. A compiler generates multiple versions of a task, each with increasing granularity obtained through task unrolling, removing superfluous synchronization primitives in each version. At runtime, a framework selects the version to execute according to the size of the task queues. Cong et al. [9] propose the X10 Work Stealing framework (XWS), an open-source runtime for the X10 parallel programming language, which extends the Cilk work-stealing framework with a strategy to adaptively control task granularity according to the instantaneous size of

the work queues. Lifflander et al. [30] present an approach to dynamically merge tasks in fork-join work-stealing–based Cilk programs, which are known to suffer significant overheads caused by the synchronized deques from which tasks are taken out [28].

While the above techniques enable automatic task-granularity adaptation, they can mainly benefit recursive divide-and-conquer applications. Other kinds of applications are less suited to dynamic task-granularity control. On the other hand, our focus is not limited to recursive divide-and-conquer applications.

An alternative approach to adapt task granularities at runtime is *lazy task creation*. This technique allows developers to express fine-grained parallelism. Lazy task creation works best with fork-join tasks. At runtime, when a new task should be forked, a runtime system decides whether to spawn and execute the task in parallel or to execute it in the context of the caller (thus increasing the granularity of the caller task). The original implementation by Mohr et al. [36] spawns a new task only if computing resources become idle. Several other techniques are based on lazy task creation. Noll et al. [40] present *concurrent calls*, language constructs that can be executed either sequentially (as a coarser-grained task) or concurrently (as finer-grained tasks), based on task-granularity estimations performed by a runtime framework. Zhao et al. [75] present a similar approach for Habanero-Java, which is based on static analysis.

Lazy task creation is less effective in applications that do not make use of fork-join tasks. On the other hand, our work benefits fork-join tasks as well as other kinds of construct (such as thread pools or custom task execution frameworks) that may not benefit from lazy task creation. Moreover, most of the above techniques rely on profiling to refine their runtime decisions, which may introduce significant runtime overhead [40], thus decreasing the benefits of task-granularity adaptation. On the contrary, the profiling overhead caused by tgp is low in most of the analyzed applications.

Another approach to avoid the overhead of creating fine-grained tasks is to determine the minimum task granularity that makes parallel task execution worthwhile, i.e., the *cut-off*. Several authors have proposed different solutions to determine the cut-off and manage finer task granularities. Duran et al. [13] propose an adaptive cut-off technique for OpenMP, based on profiling information collected at runtime to discover task granularity. Tasks whose granularity is lower than the cut-off are pruned from the target application to reduce creation and scheduling overheads. A similar approach is proposed by Iwasaki et al. [23] and implemented as an optimization pass in LLVM, while Bi et al. [4] present a similar technique for Function Flow. These approaches base their decisions on collected metrics that can be significantly perturbed, decreasing the accuracy of the proposed approaches. While our analysis is also based on metrics that can be biased, we resort to accurate and efficient profiling techniques, instrumentation and data structures that help reducing measurement perturbations, differently from the above work.

Another limitation of many techniques presented above [9, 13, 30, 36, 40, 70, 75] is that they require the introduction of custom constructs in the source code of the target application, or execute it on a modified runtime system, which may be impractical for some programs. In contrast, our analysis targets unmodified applications (as distributed by the developers) in execution on standard JVMs. Our work requires neither the addition of custom constructs nor the use of a modified runtime. Finally, similarly to approaches for estimating task granularity (Section 13.1), the aforementioned techniques focus only little on the performance drawbacks of coarse-grained tasks, differently from our work.

13.3 Profiling Task Granularity

Profiling task granularity is fundamental to investigate related performance drawbacks. Some of the aforementioned techniques [13, 40] profile task granularity as part of their adaptive strategies.

Unfortunately, the resulting profiles are not made available to the user and cannot be used to conduct further analysis of task granularity.

To the best of our knowledge, there are only two profilers for task granularity, apart from ours. Hammond et al. [15] describe a set of graphical tools to help analyze task granularity in terms of a temporal profile correlating thread execution with time. More recently, Muddukrishna et al. [38] develop *grain graphs*, a performance analysis tool that visualizes the granularity of OpenMP tasks.

Unfortunately, the above tools collect only a limited set of metrics that do not allow one to fully understand the impact of task granularity on application performance. On the other hand, our profiler collects comprehensive metrics from the whole system stack simultaneously, which aid performance analysts to correlate task granularity with its impact on application performance at multiple system layers. Moreover, tgp incurs only little profiling overhead and provides actionable profiles, which are used to optimize task granularity in several real-world applications, unlike the work of Hammond et al. Finally, grain graphs are better suited for recursive fork-join applications, while tgp targets any task-parallel application running on the JVM in shared-memory multicores.

13.4 Work-Span Model

The *work-span model* [24] aims at finding the maximum theoretical parallelism of a parallel application by dividing its *work* (i.e., the time a sequential execution would take to complete all tasks) by its span (i.e., the time a parallel execution would take on an infinite number of processors). The span is also equal to the time to execute the *critical path*, i.e., the longest chain of tasks that must be executed sequentially.

Several authors rely on the work-span model to locate and optimize the critical path of an application. He et al. propose Cilkview [17], a tool for analyzing the logical dependencies within an application to determine its work and span, estimating the maximum speedup and predicting how the application will scale with an increasing number of computing cores. Schardl et al. present Cilkprof [58], an extension of Cilkview that collects the work and span for each call site of the application, to assess how much each call site contributes to the overall work and span. A main limitation of Cilkprof is that it runs the profiled application sequentially, resulting in significant slowdown w.r.t. a parallel execution of the original application. Both Cilkview and Cilkprof can only benefit Cilk applications. Yoga et al. propose TaskProf [73], a profiler that identifies parallelism bottlenecks and estimates possible parallelism improvements by computing work and span in task-parallel applications. TaskProf only supports C++ applications using the Intel Threading Building Blocks task-parallel library.

Differently from Cilkview, our profiler does not aim at computing the expected speedup of the whole program; instead, it aims at locating suboptimal task granularities. In contrast to both Cilkprof and TaskProf, our profiler requires neither compiler support nor library modification. Moreover, the overhead of tgp is significantly lower than the one reported by the authors of all three tools, allowing the collection of more accurate metrics with less measurement perturbations. Finally, none of these tools supports the JVM.

Overall, the above work detects bottlenecks and predicts speedups by mainly focusing on the longest tasks of an application (i.e., coarse-grained tasks), paying little attention to the possible performance drawbacks caused by short tasks (i.e., fine-grained tasks). In contrast, our work focuses on both coarse-grained and fine-grained tasks, enabling the detection of performance problems caused by a too fine-grained task parallelism.

14 CONCLUSION

Our work bridges the gap between the need for a better understanding of the task granularity for parallel applications running on the JVM and the lack of dedicated techniques and tools focusing on the analysis and optimization of task granularity.

Summary of Contributions. We present a novel methodology to accurately profile the granularity of all tasks spawned in task-parallel applications on a JVM, helping developers locate performance problems. Our approach produces actionable profiles indicating the classes and methods where optimizations related to task granularity are needed, guiding developers towards useful optimizations.

We implement our profiling technique in tgp, a novel task-granularity profiler for the JVM, built on top of the DiSL and Shadow VM frameworks, which enable accurate and complete task-granularity profiling thanks to full bytecode coverage and strong isolation of analysis code. We use efficient data structures to decrease the profiling overhead of tgp (i.e., 1.05×, on average) and so to reduce perturbations of the collected task-granularity profiles (i.e., average perturbation factor equal to 1.02×).

With tgp, we analyze task granularity in numerous task-parallel applications from the DaCapo, ScalaBench, and Spark Perf suites, revealing inefficiencies related to fine-grained and coarse-grained tasks in many applications. We identify the presence of coarse-grained tasks that underutilize CPU and result in idle core, and of fine-grained tasks suffering from noticeable contention and leading to significant parallelization overheads. We identify coarse-grained tasks that can be split into several smaller ones to better exploit idle CPU, and fine-grained tasks that can be merged to reduce contention.

We use the actionable profiles collected by tgp to optimize task granularity in numerous applications. We modify the classes and methods indicated by our tool, implementing optimizations by modifying only a few lines of code and with limited knowledge of the target application. Our approach enables significant speedups in numerous applications suffering from fine- and coarse-grained tasks in different environments, up to a factor of 5.90× (in ChiSquare on a 16-core machine).

Overall, our evaluation results demonstrate the importance of analyzing and optimizing task granularity on the JVM using an accurate and efficient profiling methodology that enables the collection of actionable profiles. Moreover, our task-granularity analysis and profiler can help the maintainers of the analyzed benchmark suites to improve future releases by including additional applications with varying task granularities.

An open-source release of tgp is available at https://github.com/fithos/tgp. Apart from task-granularity analysis and optimization, tgp has proved useful for supporting different research purposes, such as domain-specific benchmark synthesis [77] and automated large-scale dynamic program analysis [72].

Future Work. The work presented in this article opens up new research opportunities. Our optimization results indicate that our work can benefit various task-parallel applications in different fields. As future work, one could extend the analysis and optimization presented here by conducting a large-scale characterization of task granularity, performing related optimizations on a broad range of applications. From the collected profiles, one could extract common anti-patterns leading to suboptimal task granularity in different applications, deriving guidelines for developers to prevent such anti-patterns in future releases and in new software.

Moreover, future research could automate the characterization and optimization of task granularity, reducing the effort for analyzing numerous task-granularity profiles. Thanks to automated task-granularity analysis, one could derive patterns and anti-patterns to classify task granularity as optimal or suboptimal from numerous task-parallel applications. These patterns could be used to define accurate performance models through machine-learning techniques to enable the automatic classification of task granularity in new open-source applications.

REFERENCES

[1] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. 2011. Oracle scheduling: Controlling granularity in implicitly parallel languages. In *OOPSLA*. 499–518.

[2] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press.

[3] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI*. 85–96.

[4] Jianmin Bi, Xiaofei Liao, Yu Zhang, Chencheng Ye, Hai Jin, and Laurence T. Yang. 2014. An adaptive task granularity based scheduling for task-centric parallelism. In *HPCC*. 165–172.

[5] Walter Binder, Jarle Hulaas, and Philippe Moret. 2007. Advanced Java bytecode instrumentation. In *PPPJ*. 135–144.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA*. 169–190.

[7] Feng Chen, Traian Florin Serbanuta, and Grigore Rosu. 2008. jPredictor: A predictive runtime analysis tool for Java. In *ICSE*. 221–230.

[8] Kuo-Yi Chen, J. Morris Chang, and Ting-Wei Hou. 2011. Multithreading in Java: Performance and scalability on multicore systems. *IEEE Trans. Comput.* 60, 11 (2011), 1521–1534.

[9] Guojing Cong, Sreedhar Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay Saraswat, and Tong Wen. 2008. Solving large, irregular graph problems using adaptive work-stealing. In *ICPP*. 536–545.

[10] Databricks. 2015. Spark Performance Tests. Retrieved from https://github.com/databricks/spark-perf.

[11] Florian David, Gael Thomas, Julia Lawall, and Gilles Muller. 2014. Continuously measuring critical section pressure with the free-lunch profiler. In *OOPSLA*. 291–307.

[12] Bruno Dufour, Karel Driesen, Laurie Hendren, and Clark Verbrugge. 2003. Dynamic metrics for Java. In *OOPSLA*. 149–168.

[13] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. An adaptive cut-off for task parallelism. In *SC*. 1–11.

[14] H2. 2018. H2 Database Engine. Retrieved from http://www.h2database.com.

[15] Kevin Hammond, Hans-Wolfgang Loidl, and Andrew S Partridge. 1995. Visualising granularity in parallel programs: A graphical winnowing system for Haskell. In *HPFC*. 208–221.

[16] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical profiling: Understanding the behavior of object-oriented applications. In *OOPSLA*. 251–269.

[17] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview scalability analyzer. In *SPAA*. 145–156.

[18] Carl Hewitt, Peter Bishop, and Richard Steiger. 1973. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*. 235–245.

[19] Lorenz Huelsbergen, James R. Larus, and Alexander Aiken. 1994. Using the run-time sizes of data structures to guide parallel-thread creation. In *LSP*. 79–90.

[20] IBM. 2007. DayTrader. Retrieved from https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaag/wascrypt/l0wscry00_daytrader.htm.

[21] ICL. 2017. PAPI. Retrieved from http://icl.utk.edu/papi/.

[22] Hiroshi Inoue and Toshio Nakatani. 2009. How a Java VM can get more from a hardware performance monitor. In *OOPSLA*. 137–154.

[23] Shintaro Iwasaki and Kenjiro Taura. 2016. Autotuning of a cut-off for task parallel programs. In *MCSoC*. 353–360.

[24] Joseph JaJa. 1992. *Introduction to Parallel Algorithms*. Addison-Wesley Professional.

[25] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. 2012. The JVM is not observable enough (and what to do about it). In *VMIL*. 33–38.

[26] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. 1997. Aspect-oriented programming. In *ECOOP*. 220–242.

[27] Clyde P. Kruskal and Carl H. Smith. 1988. On the notion of granularity. *J. Supercomput.* 1, 4 (1988), 395–408.

[28] Vivek Kumar, Daniel Frampton, Stephen M. Blackburn, David Grove, and Olivier Tardieu. 2012. Work-stealing without the baggage. In *OOPSLA*. 297–314.

[29] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A comprehensive Java benchmark study on memory and garbage collection behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *ICPE*. 3–14.

[30] Jonathan Lifflander, Sriram Krishnamoorthy, and Laxmikant V. Kale. 2014. Optimizing data locality for fork/join programs using constrained work stealing. In *SC*. 857–868.

[31] Linux man. 2013. top(1). Retrieved from https://linux.die.net/man/1/top.

[32] Linux man. 2018. Documentation of `CLOCK_MONOTONIC` in `clock_gettime()`. Retrieved from https://linux.die.net/man/3/clock_gettime.

[33] Pedro Lopez, Manuel Hermenegildo, and Saumya K. Debray. 1996. A methodology for granularity-based control of parallelism in logic programs. *J. Symbolic Comput.* 21, 4 (1996), 715–734.
[34] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tuma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. 2013. ShadowVM: Robust and comprehensive dynamic program analysis for the Java platform. In *GPCE*. 105–114.
[35] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A domain-specific language for bytecode instrumentation. In *AOSD*. 239–250.
[36] Eric Mohr, David A. Kranz, and Robert H. Halstead Jr. 1991. Lazy task creation: A technique for increasing the granularity of parallel programs. *IEEE Trans. Parallel Distrib. Syst.* 2, 3 (1991), 264–280.
[37] Philippe Moret, Walter Binder, and Alex Villazon. 2009. CCCP: Complete calling context profiling in virtual execution environments. In *PEPM*. 151–160.
[38] Ananya Muddukrishna, Peter A. Jonsson, Artur Podobas, and Mats Brorsson. 2016. Grain graphs: OpenMP performance analysis made easy. In *PPoPP*. 28:1–28:13.
[39] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *PLDI*. 187–197.
[40] Albert Noll and Thomas Gross. 2013. Online feedback-directed optimizations for parallel Java code. In *OOPSLA*. 713–728.
[41] Oracle. 2017. Documentation of `System.nanotime()`. Retrieved from https://docs.oracle.com/javase/9/docs/api/java/lang/System.html.
[42] Oracle. 2017. Java Native Interface. Retrieved from https://docs.oracle.com/javase/9/docs/specs/jni/index.html.
[43] Oracle. 2017. Java Platform, Standard Edition & Java Development Kit Version 9 API Specification. Retrieved from https://docs.oracle.com/javase/9/docs/api/.
[44] Oracle. 2017. Java Virtual Machine Tool Interface (JVM TI). Retrieved from https://docs.oracle.com/javase/9/docs/specs/jvmti.html.
[45] Oracle. 2017. The Parallel Collector. Retrieved from https://docs.oracle.com/javase/9/gctuning/parallel-collector1.htm.
[46] Oracle. 2017. `ExecutorService`. Retrieved from https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ExecutorService.html.
[47] Oracle. 2017. `ForkJoinPool`. Retrieved from https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ForkJoinPool.html.
[48] Oracle. 2017. `ThreadPoolExecutor`. Retrieved from https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ThreadPoolExecutor.html.
[49] perf. 2015. Linux profiling with performance counters. Retrieved from https://perf.wiki.kernel.org.
[50] Andrea Rosà and Walter Binder. 2018. Optimizing type-specific instrumentation on the JVM with reflective supertype information. *J. Visual Lang. Comput.* 49 (2018), 29–45.
[51] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. Actor profiling in virtual execution environments. *SIGPLAN Not.* 52, 3 (Oct. 2016), 36–46.
[52] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2017. Accurate reification of complete supertype information for dynamic analysis on the JVM. *SIGPLAN Not.* 52, 12 (Oct. 2017), 104–116.
[53] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2018. Analyzing and optimizing task granularity on the JVM. In *CGO*. 27–37.
[54] Eduardo Rosales, Andrea Rosà, and Walter Binder. 2017. tgp: A task-granularity profiler for the Java virtual machine. In *APSEC*. 570–575.
[55] Sandy Ryza. 2015. How-to: Tune Your Apache Spark Jobs (Part 1). Retrieved from http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/.
[56] Sandy Ryza. 2015. How-to: Tune Your Apache Spark Jobs (Part 2). Retrieved from http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/.
[57] Aibek Sarimbekov, Andreas Sewe, Walter Binder, Philippe Moret, and Mira Mezini. 2014. JP2: Call-site aware calling context profiling for the Java virtual machine. *Sci. Comput. Program.* 79 (Jan. 2014), 146–157.
[58] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The cilkprof scalability profiler. In *SPAA*. 89–100.
[59] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and analysis of a scala benchmark suite for the Java virtual machine. In *OOPSLA*. 657–676.
[60] The Apache Software Foundation. 2018. Apache Spark—RDD Programming Guide. Retrieved from https://spark.apache.org/docs/latest/rdd-programming-guide.html.
[61] The Apache Software Foundation. 2018. Apache Spark MLlib. Retrieved from https://spark.apache.org/mllib/.
[62] The Apache Software Foundation. 2018. Apache Tomcat. Retrieved from http://tomcat.apache.org.

[63] The Apache Software Foundation. 2018. Lucene. Retrieved from https://lucene.apache.org.
[64] The Apache Software Foundation. 2018. Spark Configuration. Retrieved from https://spark.apache.org/docs/latest/configuration.html.
[65] The Apache Software Foundation. 2018. Spark Streaming. Retrieved from https://spark.apache.org/streaming/.
[66] The Apache Software Foundation. 2018. `SparkContext` API. Retrieved from https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/SparkContext.html.
[67] The Eclipse Foundation. 2016. Jetty. Retrieved from http://www.eclipse.org/jetty/.
[68] The Eclipse Foundation. 2018. Eclipse. Retrieved from https://www.eclipse.org.
[69] The Stanford Natural Language Processing Group. 2010. Stanford Topic Modeling Toolbox. Retrieved from https://nlp.stanford.edu/software/tmt/tmt-0.4/.
[70] Peter Thoman, Herbert Jordan, and Thomas Fahringer. 2013. Adaptive granularity control in task parallel programs using multiversioning. In *Euro-Par*. 164–177.
[71] TPC. 2010. TPC-C. Retrieved from http://www.tpc.org/tpcc/.
[72] Alex Villazón, Haiyang Sun, Andrea Rosà, Eduardo Rosales, Daniele Bonetta, Isabella Defilippis, Sergio Oporto, and Walter Binder. 2019. Automated large-scale multi-language dynamic program analysis in the wild. In *ECOOP*. 1–26.
[73] Adarsh Yoga and Santosh Nagarakatte. 2017. A fast causal profiler for task parallel programs. In *ESEC/FSE*. 15–26.
[74] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*. 1–14.
[75] Jisheng Zhao, Jun Shirako, V. Krishna Nandivada, and Vivek Sarkar. 2010. Reducing task creation and termination overhead in explicitly parallel programs. In *PACT*. 169–180.
[76] Yudi Zheng, Lubomír Bulej, and Walter Binder. 2015. Accurate profiling in the presence of dynamic compilation. In *OOPSLA*. 433–450.
[77] Yudi Zheng, Andrea Rosà, Luca Salucci, Yao Li, Haiyang Sun, Omar Javed, Lumobír Bulej, Lydia Y. Chen, Zhengwei Qi, and Walter Binder. 2016. AutoBench: Finding workloads that you need using pluggable hybrid analyses. In *SANER*. 639–643.
[78] Gary M. Zoppetti, Gagan Agrawal, Lori Pollock, Jose Nelson Amaral, Xinan Tang, and Guang Gao. 2000. Automatic compiler techniques for thread coarsening for multithreaded architectures. In *ICS*. 306–315.

Received August 2018; revised March 2019; accepted May 2019