Analysis and Optimization of Task Granularity on the Java Virtual Machine

Doctoral Dissertation submitted to the
Faculty of Informatics of the Università della Svizzera Italiana
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

presented by
Andrea Rosà

under the supervision of
Prof. Walter Binder

August 2018

Dissertation Committee

Prof. Fernando Pedone	Università della Svizzera italiana, Switzerland
Prof. Robert Soulé	Università della Svizzera italiana, Switzerland
Prof. Petr Tůma	Charles University, Czech Republic
Prof. Giuseppe Serazzi	Politecnico di Milano, Italy

Dissertation accepted on 2 August 2018

Research Advisor	PhD Program Director
Prof. Walter Binder	**Prof. Olaf Schenk**

I certify that except where due acknowledgement has been given, the work presented in this thesis is that of the author alone; the work has not been submitted previously, in whole or in part, to qualify for any other academic award; and the content of the thesis is the result of work which has been carried out since the official commencement date of the approved research program.

Andrea Rosà
Lugano, 2 August 2018

To Eleonora

Abstract

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.

We focus on task-parallel applications running in a single Java Virtual Machine on a shared-memory multicore. Despite their performance may considerably depend on the granularity of their tasks, this topic has received little attention in the literature. Our work fills this gap, analyzing and optimizing the task granularity of such applications.

In this dissertation, we present a new methodology to accurately and efficiently collect the granularity of each executed task, implemented in a novel profiler. Our profiler collects carefully selected metrics from the whole system stack with low overhead. Our tool helps developers locate performance and scalability problems, and identifies classes and methods where optimizations related to task granularity are needed, guiding developers towards useful optimizations.

Moreover, we introduce a novel technique to drastically reduce the overhead of task-granularity profiling, by reifying the class hierarchy of the target application within a separate instrumentation process. Our approach allows the instrumentation process to instrument only the classes representing tasks, inserting more efficient instrumentation code which decreases the overhead of task detection. Our technique significantly speeds up task-granularity profiling and so enables the collection of accurate metrics with low overhead.

We use our novel techniques to analyze task granularity in the DaCapo, ScalaBench, and Spark Perf benchmark suites. We reveal inefficiencies related to fine-grained and coarse-grained tasks in several workloads. We demonstrate that the collected task-granularity profiles are actionable by optimizing task granularity in numerous benchmarks, performing optimizations in classes and methods indicated by our tool. Our optimizations result in significant speedups (up to a factor of 5.90×) in numerous workloads suffering from fine- and coarse-grained tasks in different environments. Our results highlight the importance of analyzing and optimizing task granularity on the Java Virtual Machine.

Acknowledgements

First of all, I would like to thank my research advisor Prof. Walter Binder for providing me with the opportunity to pursue my PhD in his group. I thank Walter for his guidance through the doctoral studies and his support towards the achievements described in this dissertation. I am also grateful to Dr. Lydia Y. Chen, who supported by work during the first years of my doctoral studies.

I would like to thank the members of the dissertation committee, including Prof. Petr Tůma, Prof. Giuseppe Serazzi, Prof. Fernando Pedone and Prof. Robert Soulé for their valuable comments that improved this dissertation.

I am grateful to the students who contributed to this research and related topics, in particular Eduardo Rosales, Haiyang Sun, and Samuele Decarli.

I am thankful to all the past and present members of the Dynamic Analysis Group, with whom I had a pleasure to work at Università della Svizzera italiana. In particular, I would like to express my gratitude to Prof. Lubomír Bulej, Prof. Alex Villazon and Dr. Yudi Zheng for their support and the inspiring and critical discussions. I also want to thank Sebastiano Spicuglia, whose support has been fundamental many times at the very beginning of my doctoral studies.

Special thanks go to Elisa and Janine of the faculty's Dean Office for their constant administrative support and help.

Finally, I would like to thank Eleonora and my family for their support during my doctoral studies.

This work has been funded by the Swiss National Science Foundation (project 200021_141002), by the EU commission (contract ACP2-GA-2013-605442) and by Oracle (ERO project 1332).

Contents

Contents **ix**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Goals and Challenges . . . 3
1.3 Contributions . . . 3
1.3.1 Task-Granularity Profiling . . . 4
1.3.2 Reification of Complete Supertype Information . . . 4
1.3.3 Task-Granularity Analysis and Optimization . . . 5
1.4 Dissertation Outline . . . 6
1.5 Publications . . . 6

2 State-of-the-Art **9**
2.1 Task Granularity . . . 9
2.1.1 Estimating Task Granularity . . . 9
2.1.2 Adapting Task Granularity . . . 11
2.1.3 Profiling Task Granularity . . . 14
2.2 Work-Span Model . . . 14
2.3 Analyses of Parallel Applications . . . 15
2.4 Profilers for Parallel Applications . . . 18
2.5 Reification of Supertype Information . . . 21

3 Task-Granularity Profiling **25**
3.1 Background . . . 25
3.1.1 JVMTI and JNI . . . 25
3.1.2 DiSL and Shadow VM . . . 26
3.2 Task Model . . . 27
3.2.1 Tasks . . . 27
3.2.2 Task Granularity . . . 28
3.2.3 Task Submission . . . 28

3.2.4 Task Aggregation . . . 28
3.2.5 Multiple Task Executions . . . 29
3.3 Metrics . . . 29
3.4 Profiling Methodology . . . 31
3.4.1 Metric Collection . . . 31
3.4.2 Task Aggregation . . . 33
3.4.3 Trace Alignment . . . 33
3.4.4 Calling-Context Profiling . . . 33
3.5 Instrumentation . . . 34
3.5.1 Data Structures . . . 34
3.5.2 Challenges in Task-Granularity Profiling . . . 37
3.5.3 Instrumentation for Task-Granularity Profiling . . . 38
3.6 Implementation . . . 41
3.6.1 Efficient Shadow Stacks . . . 42
3.6.2 Task-Granularity Profiling . . . 44
3.6.3 Task-Submission Profiling . . . 46
3.6.4 Calling-Context Profiling . . . 46
3.7 Discussion . . . 46
3.7.1 Excluded Metrics . . . 46
3.7.2 Limitations . . . 47
3.8 Summary . . . 48

4 Reification of Complete Supertype Information 51
4.1 Background . . . 52
4.1.1 Classloaders . . . 52
4.1.2 Reflective Information . . . 53
4.2 Motivation . . . 54
4.2.1 Compile-time Instrumentation . . . 55
4.2.2 Load-time In-process Instrumentation . . . 56
4.2.3 Load-time Out-of-process Instrumentation . . . 57
4.2.4 Classloader Namespaces . . . 58
4.2.5 Our Solution . . . 59
4.3 The DiSL Reflection API . . . 60
4.4 Implementation . . . 62
4.4.1 Forced Loading of Supertypes . . . 64
4.4.2 Classloader Namespaces . . . 67
4.4.3 Preprocessing Java Core Classes . . . 69
4.5 Efficient Task-Granularity Profiling . . . 71
4.6 Evaluation . . . 73

4.6.1 Methodology and Setup . . . 73
4.6.2 Profiling Overhead and Speedup . . . 77
4.6.3 Perturbation . . . 79
4.7 Discussion . . . 84
4.7.1 Reclamation of Classloader Namespaces . . . 84
4.7.2 Preprocessing Classes Outside `java.*` . . . 84
4.7.3 Instrumentation State . . . 85
4.7.4 Checking Classloader IDs . . . 85
4.7.5 Limitations . . . 85
4.8 Summary . . . 86

5 Task-Granularity Analysis and Optimization 89
5.1 Analysis . . . 89
5.1.1 Methodology . . . 89
5.1.2 Fine-Grained Tasks . . . 90
5.1.3 Coarse-Grained Tasks . . . 97
5.2 Optimization . . . 102
5.2.1 pmd . . . 102
5.2.2 lusearch . . . 104
5.2.3 Spark Perf Benchmarks . . . 105
5.2.4 Evaluation . . . 107
5.3 Discussion . . . 113
5.3.1 Platform-dependent Results . . . 113
5.3.2 Optimization of DaCapo and ScalaBench . . . 114
5.3.3 CPU Utilization . . . 114
5.4 Summary . . . 114

6 Conclusion 117
6.1 Summary of Contributions . . . 117
6.2 Future Work . . . 119

Bibliography 123

Index 137

Chapter 1

Introduction

In this chapter, we introduce the work presented in this dissertation. Section 1.1 motivates the need for analyzing and optimizing task granularity on the Java Virtual Machine (JVM). Section 1.2 discusses our goals and the related challenges. Section 1.3 presents an overview on the contributions made by our work. Section 1.4 outlines the structure of the dissertation. Finally, Section 1.5 lists the scientific publications supporting the work here presented.

1.1 Motivation

Due to technological limitations complicating further advances in single computing cores (such as the clock rate and the amount of exploitable instruction-level parallelism), nowadays processors offer an increasing number of cores. While modern multicore machines provide extensive opportunities to speed up workloads, developing or tuning a parallel application to make good use of all 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 computing cores. Tasks can execute either the same or different code on the same or different data, and are run by different threads in parallel. This work focuses on *task-parallel*[1] applications running on a single JVM in a shared-memory multicore. Task parallelism is implemented in many applications running on the JVM, and is eased by the presence of dedicated frameworks, such as e.g. thread pools [105] and fork-join pools [104], which significantly lower the programming effort for

[1]We denote as *task-parallel* any application resorting to task parallelism.

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 [72]. 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 synchronization and communication that may be involved between them. Such overheads may 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, such techniques present multiple limitations (such as lack of accuracy [90; 31] or limited applicability [1; 141; 89]), may rely on asymptotic complexity functions that can be difficult to compute or provide [1; 80], or may need custom systems [22; 144; 76; 83]. Moreover, they fall short in highlighting and optimizing performance drawbacks caused by coarse-grained tasks. Finally, only few of them support the JVM [90; 150].

On the other hand, despite the presence of studies [48; 122; 146] based on the *work-span model*[2] [62; 23] 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 workloads, overlooking the overhead caused by fine-grained tasks. Finally, while numerous authors provide detailed studies on parallel applications running on the JVM [29; 19; 67] or propose profiling tools for different performance

[2]The *work-span model* computes the maximum theoretical speedup 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 length of the longest chain of tasks that must be executed sequentially). More information on the model is given in Section 2.2.

attributes [26; 127; 18; 57], none of them focuses on task granularity. As a result, task granularity and its performance impact on task-parallel applications running on the JVM remain largely unexplored yet crucial topics.

1.2 Goals and Challenges

The goal of our work is analyzing and optimizing the task granularity of parallel applications running on a single JVM in a shared-memory multicore. In particular, we aim at 1) characterizing the task granularity of task-parallel applications, 2) analyzing the impact of task granularity on the application performance, 3) locating workloads where suboptimal task granularity causes negative effects on application performance, and 4) optimizing task granularity in such workloads, ultimately enabling significant speedups.

Our work faces notable challenges. Task-parallel applications may use tasks in complex ways. For example, applications may employ *nested tasks* (i.e., tasks fully executing in the dynamic extent of another task's execution), may use recursion within tasks, or may execute a single task multiple times. While such practices may be motivated by design principles or code reuse, they significantly complicate task-granularity profiling, and may lead to incorrect measurements of task granularity if not handled correctly. Our work identifies patterns where special care is needed, and employs efficient instrumentation that guarantees correct and accurate task-granularity profiling.

Moreover, the metrics considered by our study may be susceptible to perturbations caused by the inserted instrumentation code. In particular, such perturbations may alter the collected values of task granularity, thus biasing our results. While being of paramount importance, minimizing measurement perturbation is challenging. Our work takes several measures to keep perturbations low, including efficient and accurate profiling techniques, instrumentation and data structures to increase the accuracy of the collected task-granularity profiles.

1.3 Contributions

To enable our goal of analyzing and optimizing the task granularity of task-parallel applications running on a single JVM, this dissertation makes the following contributions.

1.3.1 Task-Granularity Profiling

We develop a new methodology for profiling the task granularity of applications running on the JVM. The goals of our methodology are to collect metrics characterizing task granularity, to pinpoint the impact of task granularity on application performance, to help developers locate performance and scalability problems, and to guide them towards effective optimizations.

We implement our profiling technique in tgp, a new task-granularity profiler for the JVM. Our profiler is built upon the DiSL [82] Java bytecode instrumentation framework, which ensures the detection of all spawned tasks, including those in the Java class library (which is notoriously hard to instrument [13; 69]).[3] Our tool enables an accurate collection of task-granularity profiles even for tasks showing complex patterns, such as nested tasks, tasks executed multiple times, and tasks with recursive operations.

To enable a detailed and accurate analysis of task granularity, tgp resorts to *vertical profiling* [47],[4] collecting a carefully selected set of metrics from the whole system stack, aligning them via offline analysis. Moreover, thanks to *calling-context profiling* [5],[5] tgp identifies classes and methods where optimizations related to task granularity are needed, guiding developers towards useful optimizations through *actionable profiles* [88].[6] 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.

1.3.2 Reification of Complete Supertype Information

We introduce a novel approach to decrease the overhead of task detection as well as the perturbation of the collected task-granularity profiles. Our technique accurately reifies the class hierarchy of an instrumented application within a separate instrumentation process, such that complete *reflective supertype information*

[3]The *Java class library* is the set of core classes offered by the Java platform, which can be used by every application running on a JVM.

[4]According to Hauswirth et al. [47], *vertical profiling* is an approach that collects and correlates information about system behavior from different system *layers*. Our work correlates metrics from the following layers: application, framework, virtual machine, operating system and hardware.

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

[6]According to Mytkowicz et al. [88], profiles are *actionable* if acting on the classes and methods indicated by the profiles yields performance improvements.

(RSI), i.e., information about all direct and indirect supertypes of the class under instrumentation, is available for each class to be instrumented. This information is usually not available in frameworks performing bytecode instrumentation in a separate process, causing the framework to instrument many more classes than those falling in the scope of the analysis and to insert expensive runtime checks into the instrumentation code, which introduce additional runtime overhead and increase the perturbation of the measurements performed.

Our technique enables the instrumentation process to instrument only the classes that are relevant for the analysis, inserting more efficient instrumentation code which in turns decreases the overhead of task detection as well as the perturbations of the collected metrics. Moreover, our technique exposes *classloader namespaces*[7] (usually unavailable) to the instrumentation process, allowing the instrumentation framework to deal correctly with homonym classes defined by different classloaders.

Our approach results in a new API—the *DiSL Reflection API*—included in an extension of the DiSL framework. Evaluation results show that the API leads to significant speedups (up to a factor of 6.24×) when profiling task-granularity with tgp, enabling the collection of task-granularity profiles with low overhead. While we use the API primarily for optimizing tgp, the API is beneficial also for other *type-specific analyses* (i.e., those targeting objects of specific types) on the JVM.

1.3.3 Task-Granularity Analysis and Optimization

We analyze task granularity in two well-known Java benchmark suites, DaCapo [15] and ScalaBench [123], as well as in several applications from Spark Perf [25], a benchmark suite for the popular Apache Spark [149] big-data analytics framework. 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.

Our analysis shows that several applications either employ a small number of coarse-grained tasks that underutilize CPU and result in idle cores, or a large number of fine-grained tasks suffering from noticeable contention, which leads to significant parallelization overheads. We identify coarse-grained tasks that can be split into several smaller ones to better leverage idle CPU cores, as well as fine-grained tasks that can be merged to reduce parallelization overheads.

[7]The *namespace* of a classloader is the set of all classes loaded by the classloader. More information is given in Section 4.1.1.

We use the actionable profiles collected by tgp to guide the optimization of task granularity in numerous workloads. 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 coarse- and fine-grained tasks.

1.4 Dissertation Outline

This dissertation is structured as follows:

- Chapter 2 discusses the state-of-the-art in the domain of this dissertation, i.e., task granularity, work-span model, analysis of parallel applications, parallel profilers, as well as reification of reflective information.
- Chapter 3 describes our approach to profile task granularity on the JVM. The chapter presents the metrics collected, our methodology to obtain accurate task-granularity profiles, as well as the instrumentation logic employed and its implementation in tgp.
- Chapter 4 details our technique for reifying complete supertype information in a separate instrumentation process, its implementation in DiSL, and the DiSL Reflection API. In addition, the chapter shows how our technique can lower profiling overhead and measurement perturbation for task-granularity profiling with tgp.
- Chapter 5 describes our task-granularity analysis on the DaCapo, ScalaBench, and Spark Perf benchmarks, details our optimizations related to task granularity, and discusses the achieved speedups.
- Chapter 6 concludes the dissertation and outlines future research directions inspired by this work.

1.5 Publications

This dissertation is based on the following conference papers. The work on task-granularity profiling, analysis, and optimization (Chapters 3 and 5) has been published at GPCE'16 and CGO'18:

- Andrea Rosà, Lydia Y. Chen, and Walter Binder. Actor Profiling in Virtual Execution Environments. In *Proceedings of 15th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pp. 36–46. 2016. Amsterdam, The Netherlands. ACM. DOI: 10.1145/2993236.2993241.

- Andrea Rosà, Eduardo Rosales, and Walter Binder. Analyzing and Optimizing Task Granularity on the JVM. In *Proceedings of the 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 27–37. Vienna, Austria. ACM. DOI: 10.1145/3168828.

The work on reification of complete supertype information (Chapter 4) has been published at GPCE'17:

- Andrea Rosà, Eduardo Rosales, and Walter Binder. Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE)*, pp. 104–116. 2017. Vancouver, Canada. ACM. DOI: 10.1145/3136040.3136061.

Chapter 2

State-of-the-Art

This chapter describes the state-of-the-art in the domain of the dissertation, i.e., analysis and optimization of task granularity for task-parallel applications running in shared-memory multicores. The chapter is organized as follows. Section 2.1 presents work related to task granularity. Section 2.2 describes research efforts based on the work-span model. Section 2.3 reviews analyses of parallel workloads. Section 2.4 describes existing profilers for parallel applications. Finally, Section 2.5 details related frameworks that reify reflective supertype information during instrumentation.

2.1 Task Granularity

Task granularity has been studied in different domains (such as many-cores [107; 42; 4], grids [63; 86; 125; 87], cloud-based systems [20; 110] and reconfigurable architectures [9]) and for different kinds of workloads (especially distributed applications [37; 43; 109; 145]). In this section, we present related approaches for estimating (Section 2.1.1), adapting (Section 2.1.2), and profiling (Section 2.1.3) task granularity in applications in the scope of the dissertation, i.e., task-parallel applications running in shared-memory multicores.

2.1.1 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*. This technique is based on an *oracle* that estimates the granularity of each task declared by developers. Oracle scheduling requires users to provide the asymptotic complexity of each function. The technique resorts to profiling information to increase the accuracy of the estimations. Unfortunately, programs without an easily derivable asymptotic complexity are less likely to benefit from oracle scheduling, as accurate complexity functions are fundamental to apply this technique. On the other hand, our analysis does not rely on asymptotic complexity functions, and thus can better benefit complex or large applications.

Lopez et al. [80] describe a static method to estimate task granularity. Their approach computes cost functions and performs static program transformations accordingly, such that the transformed program automatically controls granularity. Their work resorts to static analysis to derive asymptotic cost functions, which may be hard to compute with high accuracy in large or complex programs. Similar to oracle scheduling, this technique may yield poor results for real-world workloads, differently from our work (that does not require any cost function).

Huelsbergen et al. [51] 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 and a run-time component. The compile-time component identifies the parallel functions whose execution time depends on the size of the associated data structures. The dynamic component approximates the size of the data structures, thus estimating function execution times, and eventually task granularity. Another approach by Zoppetti et al. [152] 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 implicitly parallel languages, i.e., languages that allow developers to specify parallelism through tasks, but defer the decision whether a new task is spawned to the runtime framework. Examples of such languages are Cilk [40], Manticore [39], Multilisp [45], X10 [55], Chapel [24] and Jade [116]. Other languages (including JVM languages) may not benefit from the above techniques. On the other hand, our work focuses on task-parallel workloads running on the JVM, where parallelism can be declared explicitly (i.e., every task specified by developers is executed at runtime).

2.1.2 Adapting Task Granularity

Several authors aim at finding an optimal level of task granularity, and propose techniques to adapt the granularity of a task to the optimal value.

Thoman et al. [141] present an approach that enables automatic granularity control for recursive OpenMP [92] applications. A compiler generates multiple versions of a task, each with increasing granularity obtained through task unrolling. Superfluous synchronization primitives are removed in each version. At runtime, a framework selects the version to execute according to the size of the task queues. Cong et al. [22] propose the X10 Work Stealing framework (XWS), an open-source runtime for the X10 parallel programming language. XWS extends the Cilk work-stealing framework with several features to improve the execution of graph-based algorithms, including a strategy to adaptively control the granularity of parallel tasks in the work-stealing scheme, depending on the instantaneous size of the work queues. Lifflander et al. [76] present an approach to dynamically merge tasks in fork-join work-stealing-based Cilk programs. The authors show that their approach improves spatial locality and sequential overheads by combining many fine-grained tasks into coarser tasks while ensuring sufficient concurrency for a locality-optimized load balance.

While the above techniques enable automatic task-granularity adaptation, they can mainly benefit recursive *divide-and-conquer* [23] 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. Our work highlights inefficiencies and provides optimizations related to task granularity in a large variety of applications, including those where the above techniques would yield poor results or cannot be applied.

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, the runtime system decides whether to spawn and execute the task in parallel or to *inline* it in the caller (i.e., execute it in the context of the caller). The latter operation has the effect to increase the granularity of the caller task. The original implementation by Mohr et al. [83] spawns a new task only if computing resources become idle.

Several other techniques are based on lazy task creation. Noll et al. [90] present a methodology to estimate the optimal parallel task granularity at runtime on the JVM. Their approach is based on the notion of *concurrent calls*, special language constructs that defer the decision of whether executing concurrent calls sequentially or concurrently as parallel tasks to the runtime framework, which can merge two (or more) concurrent calls, hence coarsening task granularity. Zhao et al. [150] present a similar framework for Habanero-Java [17], which is based on static analysis. An alternative implementation for OmpSs [10] (an extension of OpenMP) is provided by Navarro et al. [89], which propose heuristics that, combined with the profiling of execution metrics, provide information to the runtime system to decide when it is worth to instantiate a task.

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 [90], thus decreasing the benefits of task-granularity adaptation. On the contrary, the profiling overhead caused by tgp is very low in most of the analyzed applications. In addition, optimized applications resulting from our approach do not need to be profiled again to monitor task granularity, in contrast to the aforementioned techniques that require continuous profiling to enable task-granularity adaptation.

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. [31] propose an adaptive cut-off technique for OpenMP. Their approach is based on profiling information collected at runtime to discover the granularity of the tasks created by the application. Tasks whose granularity is lower than the cut-off are pruned from the target application to reduce creation and scheduling overhead. A similar approach is proposed by Iwasaki et al. [61] and implemented as an optimization pass in LLVM [74], while Bi et al. [11] present a similar technique for Function Flow [36]. These techniques base their decisions on collected metrics that can be significantly perturbed, decreasing the accuracy of the proposed approaches. While also our analysis is based on metrics that can be biased, we resort to accurate and efficient profiling techniques, instrumentation and data structures that help reducing measurement perturbation, differently from the above work.

Wang et al. [144] propose AdaptiveTC, an adaptive task-creation strategy for work-stealing schedulers. AdaptiveTC can create three kinds of tasks. A *task* is a regular task that is added to a task queue, and is responsible for keeping idle threads busy. Tasks can be stolen by idle threads. A *fake task* is a plain recursive function that is never added to any task queue. Fake tasks are responsible for improving performance. Finally, a *special task* is added to a task queue to indicate a transition from a fake task to a task. AdaptiveTC adaptively switches between tasks and fake tasks to achieve better load balancing and performance. The framework manages task creation so as to keep all threads busy most of the time, to reduce the number of tasks spawned, and to optimize task granularity.

A limitation of the above technique is that it requires the introduction of custom constructs in the source code and executes an application on a modified runtime system, which may be impractical for some programs. Other approaches previously presented [22; 90; 150; 89; 144; 76; 83; 31; 141] suffer from the same limitation. In contrast, our analysis targets unmodified workloads (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.

Opsommer [93] proposes a static technique to adjust task granularity according to a metric called *attraction*. This metric is defined between two tasks and is proportional to the benefit of aggregating them into a coarser task. This methodology combines only tasks with an attraction value higher than a certain threshold. In turn, the threshold is based on task size, amount of communication involved, and number of forks and joins. This work is based on static analysis, which may lead to a poor accuracy in estimating the threshold. On the other hand, our work resorts to dynamic analysis in conjunction to an efficient profiling technique to measure the granularity of the spawned tasks with higher accuracy.

Ansaloni et al. [6] propose *deferred methods*, a methodology to speed up dynamic analyses on multicores. Deferred methods postpone the invocation of analysis methods (often representing fine-grained tasks), aggregating them in thread-local buffers. When a buffer is full, all the tasks in the buffer are merged into a single coarser-grained task executed on an idle core, leading to a lower communication overhead. A similar methodology, called *buffered advice* [7],[1] target applications based on the aspect-oriented programming model [71]. Unfortunately, such techniques are ineffective for analysis methods or advice that must execute synchronously.

Finally, similarly to approaches for estimating task granularity (Section 2.1.1), the aforementioned techniques shed no light on the performance drawbacks of coarse-grained tasks, differently from our work.

[1]According to the terminology used in *aspect-oriented programming (AOP)* [71], an *advice* is a piece of code to be executed whenever a specified point in the execution of a program (called *join point*) is reached.

2.1.3 Profiling Task Granularity

Profiling task granularity is fundamental to investigate related performance drawbacks. Some of the aforementioned techniques [90; 89; 31] 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. [46] 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. [85] develop *grain graphs*, a performance analysis tool that visualizes the granularity of OpenMP tasks, highlighting drawbacks such as low parallelism, work inflation and poor parallelization benefits.

Unfortunately, the above tools collect only a limited set of metrics that does 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 layers (i.e., application, framework, JVM, operating system, and hardware), allowing more detailed task-granularity analyses.

Moreover, our profiler 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.

2.2 Work-Span Model

The *work-span model* [62; 23] is a way to characterize task execution in a parallel application. The model can be used to determine the maximum theoretical speedup of a parallel application wrt. a sequential execution. The model compares two quantities: the *work*, i.e., the time a sequential execution would take to complete all tasks, and the *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, which the model considers the main factor limiting speedup. The maximum theoretical speedup is given by the ratio of the work to the span.

Several authors rely on the work-span model to locate the critical path of an application, focusing their optimization effort on it. He et al. propose the Cilkview scalability analyzer [48]. Cilkview analyzes the logical dependencies within an application to determine its work and span, allowing one to estimate the maximum speedup and predict how the application will scale with an increasing number of computing cores. Schardl et al. present Cilkprof [122], 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, enabling developers to quickly diagnose scalability bottlenecks. A main limitation of Cilkprof is that it runs the profiled application sequentially, resulting in significant slowdown wrt. a parallel execution of the original application. Both Cilkview and Cilkprof can only benefit Cilk applications. Yoga et al. propose TaskProf [146], 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 (TBB) [115] 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 perturbation. Finally, none of these tools support 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.

2.3 Analyses of Parallel Applications

Several researchers have conducted analyses to shed light on the parallel behavior of an application. While they focus on different aspects of parallel workloads, we are not aware of any detailed analysis on the granularity of tasks in task-parallel applications running in shared-memory multicores. In the following text, we present some major analyses on parallel workloads.

Dufour et al. [29] propose dynamic and platform-independent metrics to describe the runtime behavior on an application running on the JVM in five areas: size and control structure, data structures, polymorphism, memory use, and concurrency and synchronization. Apart for understanding program behavior, they use the profiled metrics to guide and evaluate compiler optimizations. Unfortunately, task granularity is not among the considered metrics. They also present *J [30], a profiling tool enabling the collection of the proposed metrics. Unfortunately, *J introduces excessive runtime overhead and is hardly applicable to complex workloads, unlike tgp.

Kalibera et al. [67] present several platform-independent concurrency-related metrics to enable a black-box understanding on the parallel behavior of Java applications. With the proposed metrics, they conduct an observational study of parallel Java workloads, providing more insights on their degree of concurrency, their scalability, and how they synchronize and communicate via shared memory. Despite task granularity is a key attribute of parallel workloads, their work does not shed light on the task granularity of the analyzed applications.

Chen et al. [19] analyze scalability issues of multithreaded Java applications on multicore systems. Their study pinpoints that lock contention is a strong limiting factor of scalability, determines that memory stalls are mostly produced by L2 cache misses and cache-to-cache transfers, and identifies an important factor causing slowdowns in minor garbage collections. They also highlight the importance of thread-local allocation buffers to increase cache utilization. While they target applications running on the JVM, as we do, their analysis overlooks task granularity as a possible factor causing performance drawbacks. On the other hand, our work demonstrates that task granularity can significantly impair application performance if overlooked.

Roth et al. [118] observe performance factors that are common to most parallel programs, and propose a hierarchical framework to organize these factors with the goal of helping users locate scalability issues in parallel applications. The top of their hierarchy is composed of three key factors: *work* (i.e., the time spent on executing computations), *distribution* (i.e., the overhead caused by distributing work to processors, idleness or load imbalance), and *delay* (i.e., the overhead caused by resource contention or failed transactions). They use their framework to discover inefficient barrier implementations and improve work distribution in some PARSEC [12] applications. Unfortunately, the authors do no consider task granularity as a performance factor common to many parallel workloads. On the other hand, our results indicate that task granularity is a crucial performance attribute of many task-parallel applications, and that optimizing it is of paramount importance.

Eyerman et al. [35] and Heirman et al. [49] propose *speedup stacks* to identify the impact of scalability bottlenecks on the speedup of parallel applications. A speedup stack is a stacked bar composed of various scaling delimiters. The larger a delimiter is on the stack, the more it contributes to the application slowdown; hence, it is likely to yield the largest speedup if optimized. The authors identify several important scaling delimiters that are represented on the stack: spinning (time spent spinning on locks and barriers), yielding (time spent due to yielding on locks and barriers), last-level cache, memory interference, cache coherency, work imbalance, and parallelization overhead. The work by Eyerman et al. requires custom hardware support to obtain speedup stacks, which may severely limit their applicability. On the other hand, the methodology used by Heirman et al. relies on multiple expensive simulations of the analyzed applications which may be impracticable for large multithreaded programs. Eklov et al. [34] provide an alternate method to obtain speedup stacks based on standard hardware performance counters (HPCs) commonly available in contemporary processors. Their methodology relies on cache pirating [33] to measure application performance as a function of the amount of shared cache capacity that the application receives. Unfortunately, the resulting speedup stacks are less comprehensive than those proposed by Eyerman et al., as they cannot represent scaling delimiters such as yielding, memory interference, and cache coherency.

While useful for locating *which* scaling delimiters to optimize, speedup stacks do not provide insights on *how* to perform the optimization. On the other hand, our work enables the collection of actionable profiles, indicating the classes and methods where optimizations are needed to optimize task granularity. Differently from the above work, our approach significantly reduces the effort for implementing optimizations, and does not require in-depth knowledge of the target application. Overall, task granularity is a complementary performance attribute to those considered by the above techniques, and could be integrated in speedup stacks to enable a comprehensive performance analysis of parallel workloads.

Du Bois et al. [28] introduce *bottle graphs* to show the performance of multithreaded applications. Each thread is represented as a box, with height equal to the share of the thread in the total application execution time, and width equal to its parallelism (here defined as the average number of threads that run concurrently with that thread, including itself). Boxes for all threads are stacked upon each other. Bottle graphs expose threads representing scalability bottlenecks as narrow and tall boxes, intuitively pointing developers to the threads with the greatest optimization potential. The authors use bottle graphs to pinpoint scalability bottlenecks in Java applications. The same authors propose *criticality stacks* [27] which relate the running time of a thread to the number of threads waiting for its termination. Criticality stacks support the identification of *critical threads*, i.e., those making other threads wait on locks or barriers for significant time. Optimizing critical threads can speed up the whole application. The authors use criticality stacks to remove parallel bottlenecks, identify the most critical thread of an application and accelerate it through frequency scaling, and lower energy consumption.

Both bottle graphs and criticality stacks suffer from limitations similar to speedup stacks, i.e., they fall short in suggesting to developers how to perform optimizations. Moreover, the aforementioned approaches target only threads, while our work focuses on parallel tasks,[2] allowing finer-grained analyses than the ones enabled by bottle graphs and criticality stacks.

Kambadur et al. [68] propose *parallel block vectors*, which establish a mapping between static basic blocks in a multithreaded application and the number of active threads. The mapping is performed each time a basic block executes. The authors use parallel block vectors to separate sequential and parallel portions of a program for individual analysis, and to track changes in the number of threads in execution over time in small code regions. The vectors are generated through Harmony, an instrumentation pass for the LLVM compiler. While their work allows developers to locate code regions executed by many threads, our approach enables the identification of the code portions where fine-grained tasks are created and submitted, which typically need modifications during task-granularity optimization.

Overall, unlike the above work, we select metrics to both characterize the granularity of all spawned tasks accurately and analyze their impact on application performance. Moreover, we implement a vertical profiler to collect such metrics from multiple system layers with low overhead. Finally, some of the aforementioned authors [67; 19; 28] describe the parallel characteristics of the DaCapo benchmarks [15], as we do. Our work is complementary to them, as it reveals features of such benchmarks that were previously unknown, including the presence of fine-grained tasks causing significant parallelization overheads, as well as coarse-grained tasks with suboptimal CPU utilization. Our findings pinpoint new parallelization opportunities leading to noticeable speedups.

2.4 Profilers for Parallel Applications

In Section 2.1.3, we presented the profiling tools most related to task granularity. Researchers from both industry and academia have developed several comple-

[2]Our definition of tasks include threads, as detailed in Section 3.2.1.

mentary tools to analyze diverse characteristics of a parallel application. This section discusses the major ones. Note that the aforementioned authors focusing on the work-span model also propose profilers for parallel applications, which we already discussed in Section 2.2.

Free Lunch [26] is a lock profiler for production Java server applications. The tool identifies phases where the progress of threads is significantly impeded by a lock, indicating a loss in performance. Free Lunch profiles locks by modifying the internal lock structures of the JVM. To maintain a low overhead (as required by analyses performed at production-time), Free Lunch relies on statistical sampling, which may significantly reduce the accuracy of the tool. On the other hand, tgp collects metrics without resorting to sampling (apart from CPU utilization), resulting in more accurate profiles while still guaranteeing low profiling overhead thanks to efficient instrumentation code and data structures.

THOR [127] helps developers understand the state of a Java thread, i.e., whether the thread is running on a core or is idling. The tool relies on vertical profiling to trace events across different layers of the execution stack (such as context switches and lock contention), reconstructing the traces obtained from different layers through offline analysis. Unfortunately, the overhead and the measurement perturbation caused by THOR as well as the required memory can be significant. For this reason, the authors recommend using the tool only for short periods of time (e.g., 20 seconds), which may be inconvenient for large or complex workloads. In contrast, our tool does not suffer from this limitation, and can be used to collect accurate task-granularity metrics even on long-running workloads.

jPredictor [18] detects concurrency errors in Java applications. The tool instruments a program to generate relevant events at runtime. The resulting trace is analyzed by jPredictor through static analysis. The tool can “predict” concurrency errors, i.e., detect errors that did not occur in an observed execution, but which could have happened under a different thread scheduling. Unfortunately, the tool does not detect suboptimal task granularities, thus missing related performance drawbacks.

Inoue et al. [57] propose a sampling-based profiler for parallel applications running on the IBM J9 virtual machine [54]. The profiler detects Java-level events and correlates them with metrics collected by HPCs. Aiming at obtaining more valuable information from HPCs to understand and optimize the running application, the tool tracks the calling context of each event received by an HPC. Our tool uses a similar approach, collecting calling contexts on a JVM and querying HPCs. However, tgp focuses on the collection of task granularities, while this work targets other events such as object creation or lock activities.

HPCToolkit [2] is a suite of tools for the analysis of application performance. HPCToolkit aims at locating and quantifying scalability bottlenecks in parallel programs. The suite instruments the binaries of the target application to achieve language-independency, and relies massively on HPCs. A subsequent work by Liu et al. [79] enhances the suite to support performance analysis and optimization on Non-Uniform Memory Access (NUMA) architectures. Similarly to HPCToolkit, our approach uses HPCs to collect accurate task-granularity values, and leverages the presence of different NUMA nodes to bind the execution of the observed application to an exclusive node, executing other components of tgp on a separate node. Our approach 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), ultimately increasing the accuracy of the collected metrics. Unfortunately, HPC-Toolkit is not suitable to characterize task granularity in task-parallel applications on the JVM.

Kremlin [41] provides recommendations on which regions of a sequential program can benefit from parallelization. The tool extends critical-path analysis [73] to quantify the benefit of parallelizing a given region of code, providing as output a ranked order of regions that are likely to have the largest performance impact when being parallelized. Kismet [64] is an extension of Kremlin that enhances the capability to predict the expected speedup of a code region after parallelization. To this end, Kismet employs a parallel execution model to compute an approximated upper bound for speedup, representing constraints arising from hardware characteristics or the internal program structure. Unfortunately, similarly to profilers based on the work-span model (Section 2.2), such tools do not allow to discover performance shortcomings caused by fine-grained tasks, unlike tgp.

SyncProf [148] locates portions of code where bottlenecks are caused by threads suffering from contention and synchronization issues. SyncProf repeatedly executes a program with various inputs and summarizes the observed performance behavior with a graph-based representation that relates different critical sections. SyncProf aids the process of computing the performance impact of critical sections, identifying the root cause of a bottleneck, and suggesting possible optimization strategies. Similarly to SyncProf, our approach allows the identification of tasks incurring significant contention and synchronization, which decrease application performance. While this tool only targets threads, tgp focuses on every spawned task, allowing a finer-grained performance analysis.

Other prevailing Java profilers are Health Center [53], JProfiler [32], YourKit [147], vTune Amplifier [60], and Mission Control [106]. They are optimized for common analyses, such as CPU utilization monitoring, object and memory profiling, memory-leak detection, and heap walking. However, they fall short in performing more specific analyses, including task-granularity profiling.

Overall, most of the aforementioned work considers processes or threads as the main computing entities, providing little information about individual tasks and their impact on application performance. On the contrary, our work focuses on tasks, identifying fine- and coarse-grained tasks and enabling the diagnosis of related performance shortcomings. Moreover, our tool provides actionable profiles, easing the identification of classes and methods that can benefit from task-granularity optimizations. Finally, tgp employs efficient instrumentation and profiling data structures that help reduce profiling overhead and measurement perturbation, unlike most of the above tools.

2.5 Reification of Supertype Information

The availability of complete and accurate RSI while instrumenting Java classes allows more efficient type-specific analyses. Unfortunately, most of the existing instrumentation frameworks for the JVM cannot access RSI or can inspect it only partially, and may result in analyses with increased runtime overhead and measurement perturbation. Moreover, frameworks capable of accessing complete RSI usually cannot offer *full bytecode coverage*,[3] and may lead to incomplete analyses that miss relevant events. In this section, we discuss the limitations of the major Java bytecode instrumentation frameworks capable of accessing RSI at instrumentation time.

AspectJ [70] is a mainstream AOP language and weaver.[4] In addition to AOP, AspectJ has been used for various instrumentation tasks. Since version 5, AspectJ provides a reflection API which is fully aware of the AspectJ type system [138]. AspectJ can perform the instrumentation either at compile-time or at load-time. The AspectJ compile-time weaver resorts to static analysis to precompute the type hierarchy of an application. Unfortunately, static analysis cannot guarantee accurate and complete RSI, since information on classloaders and on dynamically loaded classes is missing. On the other hand, the load-time weaver of AspectJ can access complete RSI. However, the weaver is unable to instrument classes in the

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

[4]We use the term *weaver* to denote the component of a framework performing the instrumentation. We use the term *weaving* to denote the insertion of instrumentation code into Java classes.

Java class library, resulting in limited bytecode coverage.[5] This issue is reported as a major limitation of AspectJ by the authors of the DJProf profiler [112].

The AspectBench Compiler (abc) [8] is an extensible AspectJ compiler that eases the implementation of extensions to the AspectJ language and of optimizations. It uses the Polyglot [91] framework as its front-end and the Soot framework [143] as its back-end for improving code generation. As abc is based on the compile-time weaver of AspectJ, only limited RSI is available.

Several runtime monitoring and verification tools for the JVM rely on AOP to weave the monitoring logic into the observed program, such as JavaMOP [66], Tracematches [16], or MarQ [114]. These frameworks are based on AspectJ and suffer from the same limitations. In particular, Tracematches is based on AspectJ's compile-time weaver (resulting in limited RSI), while JavaMOP and MarQ rely on the load-time weaver of AspectJ (resulting in limited bytecode coverage).

RoadRunner [38] is a framework for composing dynamic analysis tools aimed at checking safety and liveness properties of concurrent programs. Each analysis is represented as a filter over a set of event streams which can be chained together. RoadRunner performs the instrumentation at load-time in the same JVM running application code. While the framework may access RSI, it suffers from limited bytecode coverage [81], similarly to the AspectJ's load-time weaver.[6] DPAC [65] is a dynamic analysis framework for the JVM. Similarly to DiSL, DPAC performs the instrumentation at load-time in a JVM running in a separate process. To the best of our knowledge, DPAC does not provide access to complete RSI at instrumentation-time. Our approach can benefit frameworks like DPAC, providing complete RSI in the weaver.

Several bytecode engineering libraries facilitate bytecode instrumentation. Javassist [21] is a load-time bytecode manipulation library that enables structural reflection, i.e., altering the definition of classes or methods. Javassist provides convenient source-level abstractions and also supports a bytecode-level API allowing one to directly edit a classfile. ASM [108] and BCEL [128] provide low-level APIs to analyze, create, and transform Java class files. Java classes are represented as objects that contain all the information of the given class: constant pool, methods, fields, and bytecode instructions. Additionally, ASM supports load-time transformation of Java classes. Soot [143] is a bytecode optimization framework supporting multiple bytecode representations in order to simplify the analysis and

[5]The load-time weaver of AspectJ prevents the instrumentation of classes in the packages `java.*`, `javax.*`, and `sun.reflect.*`. More details are available at `https://eclipse.org/aspectj/doc/released/devguide/ltw-specialcases.html`.

[6]Classes inside the following packages cannot be instrumented by RoadRunner: `java.*`, `javax.*`, `com.sun.*`, `org.objectweb.asm.*`, `sun.*`.

the transformation of Java bytecode. Spoon [111] is a framework for program transformation and static analysis in Java, which reifies the program with respect to a meta-model. This allows direct access and modification of its structure at compile-time and allows inserting code using an AOP-based notation.

In contrast to the above frameworks, our approach enables the provision of accurate and complete RSI at instrumentation time. Thus, type-specific analyses running on DiSL can benefit from the new DiSL Reflection API to decrease the profiling overhead while offering full bytecode coverage, resulting in analyses that are both efficient and complete.

Moreover, the aforementioned frameworks based on compile-time instrumentation [70; 8; 16; 111] cannot access complete classloader namespaces while performing the instrumentation, and may fail to detect and instrument classes loaded by custom classloaders. On the other hand, our approach allows the weaver to inspect accurate and complete classloader namespaces, enabling the selective instrumentation of classes loaded by custom classloaders. In addition, our technique allows DiSL to correctly handle homonym classes defined by different classloaders, which may cause inaccurate or wrong profiling in other prevailing instrumentation frameworks due to the lack of completely reified classloader namespaces in the weaver.

Chapter 3

Task-Granularity Profiling

This chapter describes our approach to profile task granularity. Section 3.1 introduces background information on the used frameworks and technologies. Section 3.2 presents the model used for identifying tasks and accounting their granularities. Section 3.3 details the metrics of interest. Section 3.4 discusses our profiling methodology. Section 3.5 describes the instrumentation for profiling task granularity. Section 3.6 details the implementation of our approach in DiSL. Section 3.7 discusses other metrics initially considered and later disregarded, as well as the limitations of our work. Finally, Section 3.8 summarizes the achievements presented in this chapter. The profiling technique presented here is fully implemented in tgp, our novel task-granularity profiler for the JVM.

3.1 Background

Here, we introduce background information on the JVMTI and JNI interfaces (Section 3.1.1) and on the DiSL and Shadow VM frameworks (Section 3.1.2).

3.1.1 JVMTI and JNI

JVMTI (JVM Tool Interface) [98] is an interface that enables inspecting the state of a JVM and controlling the execution of applications running on top of it. JVMTI exposes an API to a native *agent* (written in C or C++) to be attached to a JVM. The agent runs in the same process and directly communicates with the JVM it is attached to. Among the features offered by the interface, a JVMTI agent 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. Moreover, JVMTI supports *heap tagging*: agents can assign a unique long value (i.e., a *tag*) to any object allocated on the heap, as well as retrieve or unset the tag associated to an object. Untagged objects have a tag of 0. Agents can also be notified when a tagged object is reclaimed by the garbage collector, executing custom code when this occurs.

JNI (Java Native Interface) [96] is a standard interface for writing native Java methods. Similarly to JVMTI, JNI exposes an API to a native agent, to be attached to a JVM and executing in the same process. The main purpose of JNI is enabling Java code to call native code and vice versa. JNI is not available during the early initialization phase of the JVM (i.e., the *primordial phase*). Our approach uses features of JVMTI and JNI to instrument classes and signaling events such application shutdown to a separate analysis process (the Shadow VM, see below). In Chapter 4, we will resort to such interfaces to expose reflective supertype information and classloader namespaces to the instrumentation process.

3.1.2 DiSL and Shadow VM

Our profiling methodology resorts to DiSL to insert profiling code into the observed application. DiSL [82] 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 AOP principles 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 wrt. a marker, e.g., before or after method bodies), *scope* (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 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. In particular, DiSL enables the instrumentation of classes in the Java class library, which are notoriously hard to instrument [13; 69].

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* [81]. 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, which helps avoiding various known classes of bugs that may be introduced by less isolated approaches [69]. Moreover, Shadow VM eases proper handling of all thread lifecycle events, and guarantees that all thread termination events are received even during the shutdown phase of the JVM. We rely on Shadow VM to both increase the isolation of analysis code and guarantee that the granularity of all threads can be detected and registered (even during the shutdown phase), ensuring complete detection of all spawned tasks.

3.2 Task Model

In this section, we present the *task model* used by our approach. The model specifies the entities of interest, defines rules for measuring task granularity, and outlines tasks requiring special handling.

3.2.1 Tasks

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. Accordingly, we consider as task every instance of the Java interfaces `java.lang.Runnable` (which should be implemented by objects intended to be executed by a thread), `java.util.concurrent.Callable` (analogous to `Runnable`, with the difference that tasks can declare a non-void return type), and the abstract class `java.util.concurrent.ForkJoinTask` (which defines tasks running within a fork-join pool). We use the term *task interfaces* to collectively refer to `Runnable`, `Callable`, and `ForkJoinTask`. Java threads themselves are also considered tasks, as `java.lang.Thread` implements `Runnable`.

3.2.2 Task Granularity

Task granularity represents the amount of work carried out by each task. Following the indication of the Java API [97], the starting point for the computation of a task is the method `Runnable.run`, `Callable.call`, or `ForkJoinTask.exec`. We refer to these 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. We use the term *task execution* to denote the execution of an execution method (by a thread). We denote a task as *executed* if its execution method has been executed until (normal or abnormal) completion (at least once).

3.2.3 Task Submission

We also detect tasks 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*, i.e., all implementations of methods `Executor.execute`, `ExecutorService.submit`, and methods `execute`, `invoke`, and `submit` in class `ForkJoinPool`, along with all overriding implementations of such methods in subtypes of `ForkJoinPool`. Note that the Java API [103; 104] defines multiples `ExecutorService.submit`, `ForkJoinPool.execute` and `ForkJoinPool.submit` methods. We consider all of them as submission methods, along with all their implementations or overriding methods. We use the term *task submission* to refer to the submission of a task to a task execution framework.

3.2.4 Task Aggregation

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. Therefore, we aggregate a nested task to its outer task, resulting in a single, larger task in the profile. If the outer task is itself nested, we recursively aggregate it until a not-nested task is found.[1]

With the goal of better recognizing the key tasks of an application and studying their granularity accurately, we do not aggregate any task whose outer task is a thread (with an exception, see below). This choice 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 trace composed only of threads. The only exception are tasks that are created and sequentially executed by the creating thread, but not submitted to any task execution framework. Such tasks indicate 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.

3.2.5 Multiple Task Executions

Finally, a task can be executed multiple times (i.e., its execution method is executed to completion more than once). 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.

3.3 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 synchronization between tasks. We employ a form of *vertical profiling* [47], 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.

Application Layer For each task, we collect its starting and ending execution timestamps, as well as the threads which create and execute the task (denoted as *creating thread* and *executing thread*, respectively). The former information allows us to identify the time intervals where tasks are executed, which enables us to

[1]Figure 3.3 in Section 3.5.2 shows an example of nested and outer tasks.

correlate task execution to OS-layer metrics, while the latter is fundamental to detect tasks created and executed by the same thread (which may be aggregated according to our task model). Moreover, we track the outer task (if any) of each task, to enable aggregation of nested tasks following the rules outlined in the task model. We also collect the type of each task.

In an optional second profiling run, we collect the *calling contexts* [5] 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.

Framework Layer We profile all task submissions, detecting the usage of task execution frameworks. For each task submission, we track both the submitted task and the task executor framework the task was submitted to. Since tasks created and executed by the same thread are aggregated unless submitted (according to our task model), detecting task submissions is fundamental to ensure correct task aggregation.

JVM Layer We detect all time intervals when the garbage collector (GC) is active. GC can significantly alter the collected metrics, particularly those related to the OS layer. Tracking all GC activities allows us to attribute unexpected metrics fluctuations to garbage collection. Note that we collect GC activations only for *stop-the-world* collections, i.e., collections during which all threads cease to modify the state of the JVM.

OS Layer We detect the CPU utilization (including both user and kernel components) and the number of context switches (CS) experienced by the observed application. The former allows one to determine whether the CPU is well utilized by the application, particularly when it is executing coarse-grained tasks. We use the latter as a measure of contention and synchronization 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 I/O, synchonization, and message passing).

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. This ensures that profiling remains consistent for the whole application execution. We use reference cycles to measure task granularity. This metric well represents the work carried

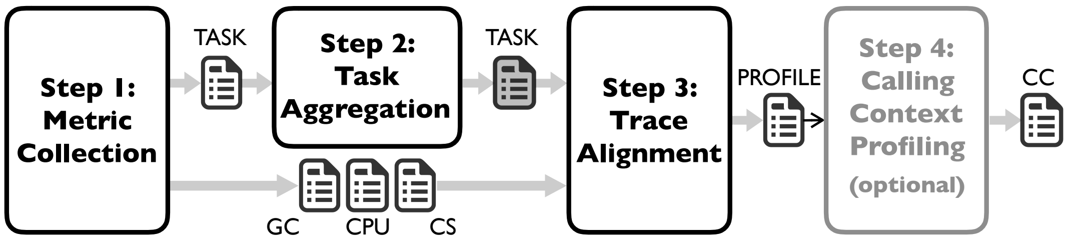


Figure 3.1. Overview of the profiling methodology.

out by a task, as it also accounts for instruction complexity as well as latencies introduced by cache misses or misalignments. Moreover, reference cycles can be converted into a temporal value if the nominal frequency of the CPU is known.

3.4 Profiling Methodology

Figure 3.1 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, i.e., task aggregation and trace alignment. The former step aggregates nested tasks, producing a modified task trace that complies with our task model, 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 granularity. Each of the above steps is performed separately by different components of tgp. The rest of this section details the four steps of our methodology.

3.4.1 Metric Collection

Figure 3.2 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 3.5 and 3.6). 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.

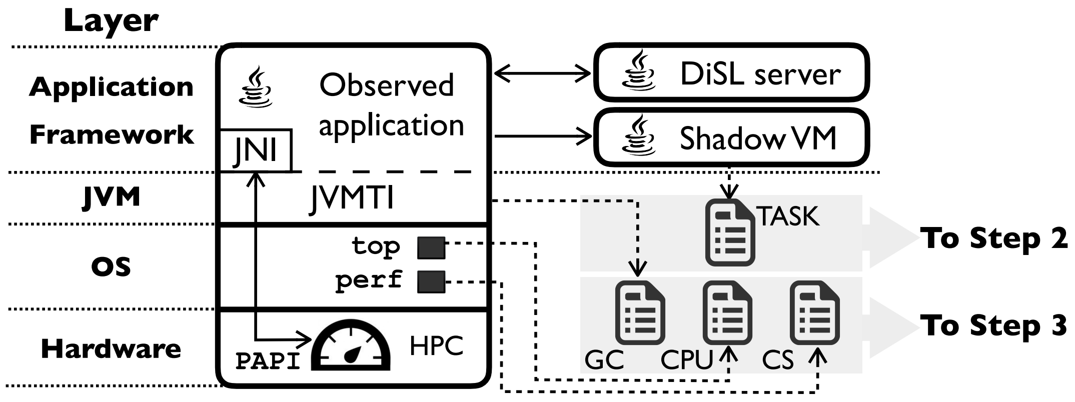


Figure 3.2. Components of the metric collection step.

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 integrates two tools that query Linux performance counters. We rely on top [77] to acquire CPU utilization, while we resort to perf [113] to collect context switches. Both metrics are collected periodically at the minimum period 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 amount of context switches experienced by the observed application since the last measurement.

At the hardware layer, the profiler makes use of HPCs integrated in most modern processors, which store accurate low-level metrics that can be read efficiently. We rely on PAPI [56] 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 3.5 and 3.6). A dedicated JNI 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.

3.4.2 Task Aggregation

According to our task model, nested tasks may need aggregation. 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, following the rules described in Section 3.2.4. If a nested task has to be aggregated, the profiler adds its granularity to the outer task and afterwards removes the nested task from the trace. The result of this step is a refined task trace compliant with our task model. This step also filters out from the trace tasks spawned but not executed.

3.4.3 Trace Alignment

The four traces obtained during metrics 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* [95] (task and GC traces), to the program starting time (CS trace), or to the default OS-clock starting time [78] (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.

3.4.4 Calling-Context Profiling

Finally, calling-context profiling is an optional profiling pass that runs again 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; 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. We present our approach to profile calling contexts in Section 3.6.4.

3.5 Instrumentation

Here, we detail our approach to instrument tasks and collect their granularities. First, we discuss the data structures used by our technique (Section 3.5.1). Then, we present the main challenges in profiling task granularity (Section 3.5.2). Finally, we detail our instrumentation approach to accurately account the granularity of each executed task (Section 3.5.3). 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 3.6.[2]

3.5.1 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 chapter, we assume the existence of types `INT`, `LONG`, `THREAD`, `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.[3]

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),

[2]We outline our approach to profile calling contexts in Section 3.6.4. We do not provide an in-depth discussion on such topic because similar instrumentation has been presented in several related studies and can be easily implemented [121; 84].

[3]Unless otherwise noted, the functions listed in Tables 3.1, 3.2 and 3.3 are undefined if any of their input parameters is $\perp$.

Table 3.1. 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 $\bot_{\texttt{TASK}}$; in this case, the task is not nested. Otherwise, ta_{outer} is the outer task of ta_{current}. The function is undefined if tp is not found.

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. As we will detail in Section 3.6.2, we store all task profiles in the Shadow VM.

Table 3.1 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 3.3) related to a single task execution, apart from calling contexts. Tasks

Table 3.2. Functions defined on a shadow stack (SS).

Function	Description
`createSS(`T`, INT`n`):` $\mathscr{S}_T^n$	Creates a shadow stack 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.

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).

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 3.6.1, shadow stacks can be efficiently implemented by embedding them into the frames of the call stack and into thread-local variables, without requiring any heap-allocated array.

Shadow stacks are similar to an usual parametrized stack, but support access to several top elements rather than just the top of the stack. We use the notation $\mathscr{S}_T^n$ to denote a shadow stack storing elements of type T and providing access to the $n \geq 1$ top elements. To better highlight shadow stacks, variables used as shadow stacks in this chapter are overlined (e.g., $\bar{s}$). The functions on shadow stacks are summarized in Table 3.2. 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 shadow stack is thread-local, accessible only by the owning thread. In each method, there can be at most one `push` operation for each shadow stack; 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 shadow stack is the same as upon method entry.

3.5.2 Challenges in Task-Granularity Profiling

Correctly accounting task granularity is complicated by the presence of special tasks that may lead to wrong profiling if not handled carefully. First, following our task model, nested tasks may be aggregated to their outer-most task, depending on their characteristics. To avoid expensive checks in the instrumentation code, we postpone aggregation of nested tasks to the second step of the profiling methodology (Section 3.4.2). This implies that all nested tasks must be correctly 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, an execution method of a task `t` may present *nested calls* to one of `t`'s execution methods. This situation may occur due to 1) (indirectly) recursive calls to execution methods (e.g., `t.run()` calls `t.run()`), 2) calls to an (overridden) task execution method defined in the superclass (e.g., `t.run()` calls `super.run()`), and 3) 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`). In such tasks, the profiler must determine when task execution is completed, collecting its granularity only at that moment.

The pseudocode in Figure 3.3 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

```
 1
 2 class A implements Runnable {
 3   public void run() {...}          // execution method of A
 4 }
 5
 6 class B implements Runnable {
 7   public void run() {...}          // execution method of B
 8 }
 9
10 class C extends B {
11   public void run() {              // execution method of C
12     super.run();                   // account to current task
13     ...
14     a.run();                       // account to a
15     ...
16     b.run();                       // account to b
17   }
18 }
```

Figure 3.3. Nested tasks and nested calls to execution methods.

method of c (i.e., c.run) is executed, the execution method defined in B is called as first operation (line 12). 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 14 and 16).[4] 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 workloads (including those analyzed in Chapter 5); hence, it is very important to detect and handle the above situations correctly, as failure in doing so may lead to incorrect task-granularity analysis.

3.5.3 Instrumentation for Task-Granularity Profiling

In this section, we present the instrumentation code ensuring correct task-granularity profiling. Note that our approach to profile task submissions and calling contexts is described in Sections 3.6.3 and 3.6.4, respectively.

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

Table 3.3. Auxiliary functions used in Figure 3.4.

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.

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 3.4. In addition to the functions defined on task profiles and shadow stacks, we use the auxiliary functions shown in Table 3.3. To ease the explanation 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 $\perp_{\mathtt{TASK}}$ if ta is not nested).

The instrumentation logic relies on five thread-local variables, four of them being shadow stacks. 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 variable $c_{\text{nested-thread}}$ to store the total granularity of nested tasks executed by th, while the shadow stack $\overline{\mathsf{c}_{\text{nested-outer}}}$ 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 $\overline{\mathsf{c}_{\text{nested-outer}}}$ at execution method entry (line 18). Note that $c_{\text{nested-thread}}$ (and hence `top(`$\overline{\mathsf{c}_{\text{nested-outer}}}$`,0)`) is 0 if ta is not nested.

The purpose of the shadow stack $\overline{\mathsf{c}_{\text{entry}}}$ is to memorize the value of the cycle counter at execution method entry for later use (line 20). The other shadow stacks are used to store the tasks being executed ($\overline{\mathsf{tasks}}$; line 16) and their execution starting timestamps ($\overline{\mathsf{times}}$; line 19). Following the rules for manipulating shadow stacks, 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 shadow stacks provide access only to the topmost element, with the exception of $\overline{\mathsf{tasks}}$ which also provides access to one element below the top.

TL :	LONG $c_{\text{nested-thread}}$	stores the granularity of nested tasks executed by the thread.
	$\mathscr{S}^2_{\texttt{TASK}}$ $\overline{\text{tasks}}$	stores the tasks in execution.
	$\mathscr{S}^1_{\texttt{LONG}}$ $\overline{c_{\text{entry}}}$	stores the value of the thread cycle counter at method entry.
	$\mathscr{S}^1_{\texttt{LONG}}$ $\overline{c_{\text{nested-outer}}}$	stores the granularity of nested tasks executed by the outer task.
	$\mathscr{S}^1_{\texttt{LONG}}$ $\overline{\text{times}}$	stores the task execution starting timestamp.

```
at threadInitialization() begin
    $c_{\text{nested-thread}} \leftarrow 0$
    $\overline{\text{tasks}}$ ← createSS(TASK, 2)

    $\overline{c_{\text{entry}}}$ ← createSS(LONG, 1)
    $\overline{c_{\text{nested-outer}}}$ ← createSS(LONG, 1)
    $\overline{\text{times}}$ ← createSS(LONG, 1)
end

after taskCreation() begin
    registerCreation(thisTask(), thisThread())
end


before executionMethod() begin
    push($\overline{\text{tasks}}$, thisTask())

    push($\overline{c_{\text{nested-outer}}}$, $c_{\text{nested-thread}}$)
    push($\overline{\text{times}}$, getTime())
    push($\overline{c_{\text{entry}}}$, readCycleCounter(thisThread()))
end


after executionMethod() begin
    LONG $c_{\text{nested-task}} \leftarrow c_{\text{nested-thread}}$ - top($\overline{c_{\text{nested-outer}}}$, 0)
    LONG $c_{\text{current}}$ ← readCycleCounter(thisThread()) - top($\overline{c_{\text{entry}}}$, 0) - $c_{\text{nested-task}}$
    if top($\overline{\text{tasks}}$, 1) $= \epsilon_{\texttt{TASK}}$ then
        registerExecution(top($\overline{\text{tasks}}$, 0), $\perp_{\texttt{TASK}}$, $c_{\text{current}}$, top($\overline{\text{times}}$, 0), getTime(),
          thisThread())
        $c_{\text{nested-thread}} \leftarrow 0$
    else if top($\overline{\text{tasks}}$, 1) $\neq$ top($\overline{\text{tasks}}$, 0) then
        registerExecution(top($\overline{\text{tasks}}$, 0), top($\overline{\text{tasks}}$, 1), $c_{\text{current}}$, top($\overline{\text{times}}$, 0),
          getTime(), thisThread())
        $c_{\text{nested-thread}} \leftarrow c_{\text{nested-thread}} + c_{\text{current}}$
    end
    pop($\overline{c_{\text{entry}}}$)
    pop($\overline{\text{times}}$)
    pop($\overline{c_{\text{nested-outer}}}$)
    pop($\overline{\text{tasks}}$)
end
```

Figure 3.4. 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 3.5.

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 $\overline{\mathtt{c}_{\text{nested-outer}}}$ 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 $\mathtt{top}(\overline{\mathtt{c}_{\text{nested-outer}}}, 0)$ 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 $\overline{\mathtt{c}_{\text{entry}}}$). This difference is reduced by the granularity of the nested tasks executed by ta, stored in $c_{\text{nested-task}}$ (line 26).

The shadow stack $\overline{\mathsf{tasks}}$ 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 $\mathtt{top}(\overline{\mathsf{tasks}}, 1) = \mathtt{top}(\overline{\mathsf{tasks}}, 0)$ (implying $\mathtt{top}(\overline{\mathsf{tasks}}, 1) \neq \epsilon_{\mathtt{TASK}}$), 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 $\mathtt{top}(\overline{\mathsf{tasks}}, 1) \neq \epsilon_{\mathtt{TASK}} \wedge \mathtt{top}(\overline{\mathsf{tasks}}, 1) \neq \mathtt{top}(\overline{\mathsf{tasks}}, 0)$ (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 $\mathtt{top}(\overline{\mathsf{tasks}}, 0)$ and $\mathtt{top}(\overline{\mathsf{tasks}}, 1)$, respectively), the granularity of ta (stored in c_{current}), the starting and ending execution timestamps (the former stored at the top of $\overline{\mathsf{times}}$, 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 $\mathtt{top}(\overline{\mathsf{tasks}}, 1) = \epsilon_{\mathtt{TASK}}$ (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.

3.6 Implementation

Here, we detail how the instrumentation scheme presented in the previous section is implemented in tgp using DiSL code. First, we discuss how shadow stacks can be efficiently implemented in DiSL (Section 3.6.1); then, we detail how our instrumentation technique can be translated into DiSL code (Section 3.6.2). We also present our approach to profile task submissions (Section 3.6.3) and calling contexts (Section 3.6.4).

3.6.1 Efficient Shadow Stacks

In Java, thread-local shadow stacks can be efficiently embedded within the frames of the call stack and in thread-local variables. In this section, we present a translation of the functions defined on shadow stacks (see Table 3.2) that avoids heap allocations and leverages synthetic-local and thread-local variables offered by DiSL.

Table 3.4 reports the general scheme to translate functions on shadow stacks to DiSL code, assuming that the preconditions of the functions are met. A shadow stack $\mathscr{S}_T^n\ \bar{s}$ can be implemented with $n-1$ thread-local variables and one synthetic-local variable. For $n \geq 2$, the thread-local 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 shadow stack, only $n-1$ elements are stored in thread-local variables. All other elements are embedded within the frames of the Java call stack, thanks to the synthetic local variable. All thread-local variables are initialized to $\perp_T$.

Manipulation of the shadow stack can occur only on method entry and exit, as first (resp. last) 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 thread-local variable suffices (see Table 3.5), while for $n = 1$, no thread-local variable is needed (see Table 3.6).

For $n > 2$, an alternative translation of shadow stacks would store variables `s_tl_<i>` ($i = [0, n-2]$) in a single thread-local 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 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 ($n \leq 2$).

Table 3.4. 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 = `$\bot_T$`;`
	`...`
	`@ThreadLocal T s_tl_<`$n-2$`> = `$\bot_T$`;`
	`@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 3.5. 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 = `$\bot_T$`;`
	`@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 3.6. 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

3.6.2 Task-Granularity Profiling

Figure 3.5 shows the translation of our approach for profiling task granularity (Figure 3.4) to DiSL code snippets. The numeric abstract data types used in Section 3.5 (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 `ExecutionMethod` (lines 14 and 23) allows the code snippet to be inserted only in executions methods, according to the definition given in Section 3.2.2.[5]

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 3.1. 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.

[5]Guard implementation is discussed in Section 4.5.

```java

@ThreadLocal static long c_nested_thread = 0;
@ThreadLocal static Object tasks_tl = null;
@SyntheticLocal static Object tasks_sl;
@SyntheticLocal static long c_entry_sl;
@SyntheticLocal static long c_nested_outer_sl;
@SyntheticLocal static long times_sl;

@After(marker = AfterInitBodyMarker.class, scope = "<init>",
    guard = TaskGuard.class)
public static void afterTaskCreation(DynamicContext dc) {
 Profiler.registerCreation(dc.getThis(), Thread.currentThread());
}

@Before(marker = BodyMarker.class, guard = ExecutionMethod.class)
public static void beforeExecutionMethod(DynamicContext dc) {
 tasks_sl = tasks_tl;
 tasks_tl = dc.getThis();
 c_nested_outer_sl = c_nested_thread;
 times_sl = System.nanoTime();
 c_entry_sl = PAPI.readCycleCounter(Thread.currentThread());
}

@After(marker = BodyMarker.class, guard = ExecutionMethod.class)
public static void afterExecutionMethod() {
 final long c_nested_task = c_nested_thread - c_nested_outer_sl;
 final long c_current = PAPI.readCycleCounter(
     Thread.currentThread()) - c_entry_sl - c_nested_task;
 if (tasks_sl == null) {
  Profiler.registerExecution(tasks_tl, null, c_current, times_sl,
      System.nanoTime(), Thread.currentThread());
  c_nested_thread = 0;
 } else if (tasks_sl != tasks_tl) {
  Profiler.registerExecution(tasks_tl, tasks_sl, c_current, times_sl,
       System.nanoTime(), Thread.currentThread());
  c_nested_thread += c_current;
 }



 tasks_tl = tasks_sl;
}
```

Figure 3.5. DiSL code for task-granularity profiling. Blank lines are added to ease line-by-line comparison with Figure 3.4.

3.6.3 Task-Submission Profiling

We detect task submission by instrumenting all submission methods (as defined in Section 3.2.3). The inserted instrumentation code calls function `registerSubmission` to store the task as well as the executor 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 correspondent task profile.

3.6.4 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.

3.7 Discussion

In this section, we present the metrics initially considered in our approach but later disregarded (Section 3.7.1), and discuss the limitations of our profiling methodology (Section 3.7.2).

3.7.1 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 little affected by perturbations caused by the instrumentation (as the bytecodes introduced by the instrumentation can be excluded from the count), ensuring accurate results in the presence of full bytecode coverage; 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*).

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 synchronization among tasks. 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.

3.7.2 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 avoids any heap allocation; use of low-overhead HPCs; profiling data structures built in a separate process. 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 4.6.

Most of the metrics used to analyze task granularity are platform-dependent; hence, the reproducibility of any task granularity analysis may be limited. However, the drawbacks of using platform-dependent metrics are well justified by their efficacy in describing task granularity, CPU utilization, and synchronization among 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.

Finally, other processes executing concurrently with the observed application may interfere with our profiling methodology. 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 wrt. an execution without concurrent applications). Ensuring that no other computational-intensive process executes concurrently with the observed application reduces the chances of experiencing this limitation; we apply this approach in 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 this setting when evaluating our approach (Section 4.6) and conducting task-granularity analysis (Chapter 5).

3.8 Summary

In this chapter, we have presented a novel profiling methodology to measure the granularity of every executed task spawned in a task-parallel application running on a JVM in a shared-memory multicore. Our profiling technique resorts to vertical profiling to collect carefully selected metrics from the whole system stack, aligning them via offline analysis. The collected metrics enable one to analyze task granularity and its impact on application performance. Moreover, the calling contexts profiled enable actionable profiles, which indicate classes and methods where optimizations related to task granularity are needed, guiding developers towards useful optimizations.

We have implemented our profiling methodology in tgp, a novel task-granularity profiler for the JVM. Our tool is built on top of the DiSL and Shadow VM frameworks, which support accurate profiling thanks to full bytecode coverage and strong isolation of analysis code. Our methodology resorts to efficient data structures to reduce the profiling overhead, storing most of the collected metrics in a separate process that executes analysis code asynchronously wrt. application code. Our instrumentation ensures accurate profiling even in tasks showing complex patterns, such as nested tasks and tasks with nested calls to execution methods.

The work presented in this chapter demonstrates that our methodology can collect task-granularity profiles that are both complete (i.e., all tasks spawned are detected) and accurate (i.e., granularity is correctly accounted to each task).

However, we have not discussed the efficiency of our methodology, i.e., how much the application is slowed down by tgp, and to which extent task granularity is biased by the inserted instrumentation logic. While the data structures used by our profiler can be efficiently implemented, the instrumentation code needed to detect tasks and task executor frameworks may significantly increase the overhead of our profiling approach, which in turn may increase the perturbation of the collected metrics.

We deal with this issue in the next chapter, which presents a novel methodology that allows us to efficiently detect tasks and task execution frameworks. The next chapter also discusses the overhead of tgp and the perturbation of the collected task granularities.

Chapter 4

Reification of Complete Supertype Information

Reflective supertype information (RSI) is useful for many instrumentation-based, type-specific analyses on the JVM, including task-granularity profiling. If complete RSI is available at instrumentation time, the weaver can access the type hierarchy of the observed application, instrumenting a class depending on its supertypes. For example, in task-granularity profiling (when calling contexts are not collected), only subtypes of `Runnable`, `Callable`, and `ForkJoinTask` should be instrumented, leaving other classes untouched. If the weaver can access complete RSI of the class under instrumentation, it can check whether one of the task interfaces is among the supertypes of the class, instrumenting it only in such a case. On the other hand, if complete RSI is not available during instrumentation, the weaver may be unable to determine whether the class to be instrumented falls in the scope of the analysis (e.g., whether the class represents a task). To guarantee a complete analysis, the weaver may still instrument classes for which it cannot determine all supertypes, introducing runtime checks (e.g., ensuring that the receiver of a method call is a subtype of a task interface) to be executed before the inserted instrumentation code. Unfortunately, such checks can result in significant overhead and thus can cause serious perturbations of the collected metrics.

On the one hand, while RSI can be obtained when performing the instrumentation within the same JVM executing the observed application, in-process instrumentation can interfere with the instrumentation of the Java class library, class loading, and JVM initialization [13; 69]. To mitigate these problems, frameworks performing in-process instrumentation (such as the AspectJ [70] load-time weaver; see Section 2.5) often prevent any instrumentation of the Java class library, leading to limited bytecode coverage. On the other hand, performing the instrumentation in a separate process (like in DiSL [82]; see Section 3.1.2) can achieve full bytecode coverage, but complete RSI is generally not available, often requiring expensive runtime checks in the instrumented program. Providing accurate and complete RSI in the instrumentation process is challenging because the observed application may make use of custom classloaders and the set of all loaded classes is generally only known upon termination of an application.

In this chapter, we present a novel technique to accurately reify complete RSI in a separate instrumentation process. We implement our technique as an extension of DiSL. Our approach enables type-specific analyses that are both complete (thanks to full bytecode coverage provided by DiSL) and efficient (thanks to the provision of RSI in the weaver ensured by our work). We use our technique to speed up task-granularity profiling with tgp and to reduce measurement perturbations. Thanks to our work, task-granularity profiling introduces only small overhead in the observed application, also reducing perturbations of the collected metrics.

The chapter is organized as follows. Section 4.1 introduces background information. Section 4.2 motivates the need for our approach. Section 4.3 presents a new API to access complete RSI within the DiSL server. Section 4.4 describes our technique in depth. Section 4.5 discusses how our approach can be used in task-granularity profiling, while Section 4.6 shows the benefits of applying our methodology to tgp (i.e., low overhead and reduced measurement perturbations). Finally, Section 4.7 discusses advanced features and limitations of our approach, while Section 4.8 summarizes the achievements presented in this chapter.

4.1 Background

Before presenting our work, we introduce some preliminary information on classloaders (Section 4.1.1) and reflective information (Section 4.1.2).

4.1.1 Classloaders

A Java class[1] is created by loading a binary representation of the class (i.e., *classfile*) using a *classloader*, which is responsible for locating the corresponding classfile, parsing it, and constructing a class representation within the JVM. A

[1]In this chapter, we use the term *class* to indiscriminately refer to both Java classes and interfaces.

classloader can either be the *bootstrap classloader* or a *user-defined* classloader.[2] The bootstrap classloader, supplied by the JVM, is responsible for loading the Java core classes (e.g., the `java.*` package), while a user-defined classloader is responsible for loading application classes. All user-defined classloaders are subtypes of `java.lang.ClassLoader`. The standard user-defined classloader used to load the main class of an application (i.e., the *application classloader*) can only locate classfiles in the application *classpath* (specified by the user at JVM startup). Developers can create their own user-defined classloaders to extend this behavior. This is needed if an application fetches classfiles from other locations (such as a remote server) or loads special classes (such as encrypted or dynamically generated ones).

When a classloader CL is requested to load a class C, it can either load the class itself or delegate the loading to another classloader. In the former case, CL is known as the *defining classloader* of C. Each class has a single defining classloader. The bootstrap classloader cannot delegate classloading. Normally, a class C is loaded when another previously loaded class D references it. The first classloader that attempts loading C is the defining classloader of D.[3] A class may be loaded before some of its supertypes. Loading C triggers the subsequent loading of its direct supertypes (i.e., its superclass and all directly implemented interfaces) if they are not yet loaded. Due to delegation, it is possible that the defining classloader of a class is different from the one of its supertypes.

At runtime, a class is determined by its *fully qualified name* (i.e., the class name, including the package where the class is defined) and its defining classloader. Several classes with the same fully qualified name may exist at runtime, provided that their defining classloaders are different. To this end, classloaders maintain different *namespaces*: a classloader is only aware of the classes it has defined or has delegated loading. Our approach exposes classloader namespaces to an instrumentation process, ensuring that homonym classes defined by different classloaders can be handled correctly, and that RSI is always accessible even if a supertype and a subtype are defined by different classloaders.

4.1.2 Reflective Information

Reflective information refers to any information related to a Java class or method available at runtime. Developers can obtain reflective information through the

[2]We use the terminology of the JVM specification [101]. Note that some user-defined classloaders are part of the Java class library.

[3]The use of the Java Reflection API may cause exceptions to this behavior. See the documentation of `java.lang.Class` for more information.

Java Reflection API [94]. Any loaded class is represented by an instance of `java.lang.Class`, which allows one to inspect reflective information of the corresponding class. Our work focuses on *reflective supertype information (RSI)*, i.e., complete information about all direct and indirect supertypes of a class. Such information could be retrieved by calling `Class.getSuperclass` and `Class.getInterfaces`, which return `Class` instances corresponding to the direct superclass and the directly implemented interfaces of a class, respectively. Complete RSI of a class can be obtained by repeatedly calling these methods on all the supertypes of the class. This approach only works in the process running the application.

If complete RSI is available at instrumentation-time, the weaver can selectively instrument a class based on its position in the type hierarchy, enabling more efficient dynamic analyses. Unfortunately, frameworks performing instrumentation in a separate process cannot access the `Class` instances of the classes loaded in the observed application (because such classes are not loaded in the instrumentation process). Thus, they cannot normally inspect complete RSI. Our approach fills this gap, making complete RSI available to the instrumentation process.

4.2 Motivation

While RSI can significantly reduce the overhead of type-specific analyses on the JVM, existing approaches to expose RSI at instrumentation-time suffer from serious limitations. In the following text, we discuss the limitations of existing instrumentation techniques in offering full bytecode coverage and accessing RSI, motivating the need for a new approach. As running example, we consider a subset of the profiling logic implemented in tgp that instruments the execution of method `run` (with no input arguments and return type `void`) in all subtypes of `java.lang.Runnable`.[4] To correctly determine the classes to instrument, the weaver has to retrieve complete RSI for each class. We first discuss the benefits and limitations of implementing such an analysis with prevailing techniques, and highlight the importance of taking classloader namespaces into account; then, we outline how our approach overcomes these limitations.

[4]While we exemplify our approach by considering only `Runnable.run` for simplicity, the same observations hold for all other execution and submission methods defined in Section 3.2 as well as for task constructors.

```java
1 public aspect Instrumentation {
2   pointcut exec(): execution(void Runnable+.run());
3   before(): exec() {
4     // Inserted instrumentation code - omitted here
5   }
6 }
```

(a) AspectJ [70].

```java
1 @Before(marker = BodyMarker.class, scope = "void run()")
2 public static void beforeRunMethod(DynamicContext dc) {
3   if (dc.getThis() instanceof Runnable) { // Inserted
4     // Inserted instrumentation code - omitted here
5   }
6 }
```

(b) DiSL [82].

Figure 4.1. AspectJ and DiSL code for instrumenting method `void run()` in every subtype of `Runnable`.

4.2.1 Compile-time Instrumentation

A possible approach is to implement the analysis with compile-time instrumentation (i.e., replacing existing classfiles with instrumented ones), for example resorting to the compile-time weaver of the AspectJ AOP framework [70]. Figure 4.1(a) shows the code for implementing the analysis in AspectJ. Compile-time instrumentation typically resorts to static analysis to derive the type hierarchy of an application, so as to insert instrumentation code (line 4) only in subtypes of `Runnable`. Unfortunately, static analysis cannot guarantee accurate and complete RSI, as it lacks information about which classes will be loaded at runtime (due to dynamic class loading). Moreover, it may not handle classloader namespaces correctly (e.g., it may not process classes loaded by custom classloaders), although classloaders play an important role in many frameworks such as in the dynamic module system OSGi [124]. Such limited RSI may lead to failures in instrumenting all relevant classes (e.g., every subtype of `Runnable`), resulting in an analysis with limited bytecode coverage.

4.2.2 Load-time In-process Instrumentation

Another approach is load-time instrumentation, performing the instrumentation within the observed JVM process (an approach known as *in-process instrumentation*). In this setting, instrumentation occurs right after a class has been loaded (before it is linked by the JVM). Interfaces such as JVMTI [98] or those offered by the *Java Instrumentation API* [99] enable intercepting and instrumenting each loaded class, including dynamically loaded classes or those defined by custom classloaders. For example, the AspectJ load-time weaver operates in such a setting.

While obtaining complete RSI using load-time instrumentation is possible, such an approach has serious drawbacks. In-process weavers implemented in Java have access to the classloader used to load a class (provided as argument to a `ClassFileTransformer`), and can call `Class.forName` (passing the classloader as an argument) to access the `Class` instances of the direct supertypes; then, they can use this approach recursively to get the `Class` instances of all indirect supertypes.

Unfortunately, this approach may significantly interfere with the instrumentation of Java classes. Instrumentation code may use classes that are themselves (to be) instrumented, which may cause infinite recursions (as reported by several researchers [126; 14] and developers [136]). This is a particular concern when instrumenting classes that are commonly used by weavers implemented in Java, such as those in the Java class library [13; 69]. Existing approaches to avoid infinite recursions when instrumenting the Java class library have serious drawbacks. One approach is to prevent the instrumentation of the Java class library altogether, as in the AspectJ load-time weaver. Unfortunately, this solution results in limited bytecode coverage; for example, subtypes of `Runnable` inside the Java class library (such as `Thread`) cannot be instrumented using the AspectJ load-time weaver. Another solution is to instrument the Java class library via class redefinition [13]; however, such an approach exacerbates the complexity of the instrumentation and constrains the instrumentation due to the limitations of class redefinition (e.g., class redefinition may only replace method bodies and must not introduce any new methods or fields).

Finally, the above limitations may be prevented by implementing instrumentation logic using only native code. For example, this can be achieved by attaching a native-code agent to the JVM using JVMTI, and avoiding any calls into bytecode. However, this solution would require the implementation of a complex weaver purely in C/C++, which is extremely tedious and error-prone.

4.2.3 Load-time Out-of-process Instrumentation

Another alternative is performing the instrumentation at load-time in a JVM instance running in a different process (an approach known as *out-of-process instrumentation*). This setting prevents the limitations of in-process instrumentation outlined above and enables full bytecode coverage. For example, DiSL employs load-time out-of-process instrumentation and can instrument every loaded class, including Java core classes.

Unfortunately, the instrumentation process has no access to complete RSI, because it cannot access the `Class` instances of the observed application. For example, the DiSL server receives a byte array upon class loading, containing the binary representation of the class to be instrumented (as encoded in the classfile). As a result, the weaver can only inspect the RSI that are contained in the classfile, i.e., the *names* of the *direct* supertypes of a class, while any other information on the direct and indirect supertypes is missing. While in principle more advanced weavers may request missing RSI from the observed JVM (which can retrieve the requested information from the `Class` instances of the supertypes), some supertypes of the class under instrumentation may not have been loaded yet, leading to failures when retrieving reflective information of such classes. To the best of our knowledge, there is no instrumentation framework resorting to load-time out-of-process instrumentation which can guarantee complete and accurate RSI in the instrumentation process.

To implement the target analysis in the absence of complete RSI (while ensuring full bytecode coverage), the weaver must insert expensive runtime checks (i.e., using the `instanceof` operator) in the bytecode of the class under instrumentation, to ensure that instrumentation code is executed only in subtypes of `Runnable`. Figure 4.1(b) shows a DiSL code snippet implementing the analysis. The code snippet instructs the weaver to instrument all methods named `run` (with no arguments and `void` return type) of *every* loaded class (line 1), executing instrumentation code (line 4) only if the current object (i.e., `this`) is an instance of `Runnable` (line 3).[5] The performance penalty introduced by such checks may be significant, especially if method `run` is defined in many classes, or if the analysis targets a constructor, which is present in any loaded class. While guaranteeing full bytecode coverage, the resulting analysis can be inefficient.

[5]The snippet shown in Figure 4.1(b) could be optimized if the class under instrumentation 1) is a direct subtype of `Runnable`, or 2) is a direct subtype of `Object` and does not implement any interface.

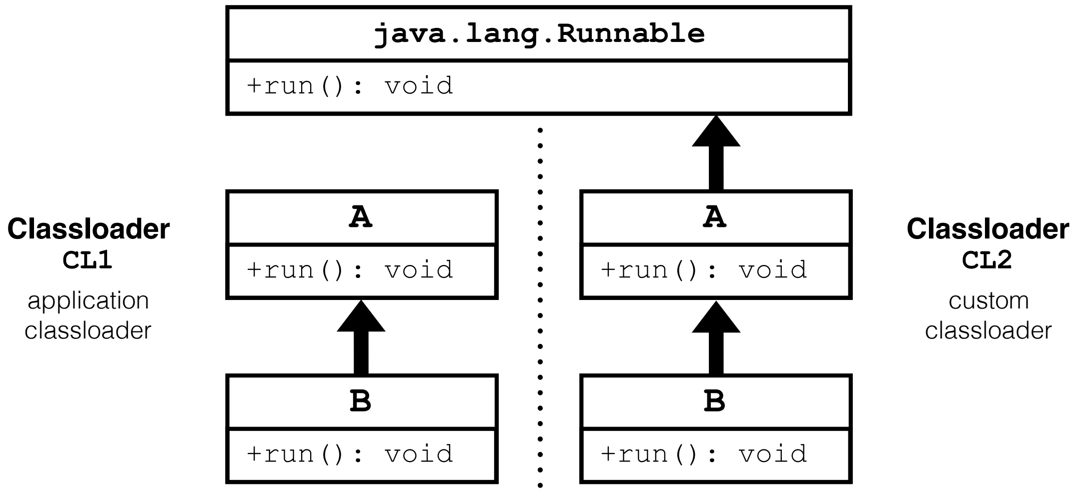


Figure 4.2. Simplified UML class diagram showing an example of homonym classes defined by different classloaders.

4.2.4 Classloader Namespaces

Classloaders maintain their own namespaces and are unaware of the classes defined by other classloaders. As a result, two classloaders can define different classes with the same fully qualified name. This situation can lead to incomplete instrumentation (i.e., classes in the scope of the analysis are not instrumented) or wrong instrumentation (i.e., classes not in the scope of the analysis are instrumented) if not carefully handled by the weaver.

To better understand this issue, consider the classes shown in Figure 4.2, and suppose that all of them are loaded by the application. Classes `A` and `B` in the left part of the figure are located in the classpath and are defined by classloader CL1 (i.e., the application classloader), while those in the right part are located outside the classpath and are defined by classloader CL2 (i.e., a custom classloader). The defining classloader of `Runnable` is the bootstrap classloader, as classes inside the `java.*` package can be loaded only by such classloader [101]. To guarantee a correct analysis, the weaver must ensure that only classes `A` and `B` defined by CL2 are instrumented (as they are subtypes of `Runnable`), leaving untouched those defined by CL1 (which are not subtypes of `Runnable`).

Unfortunately, compile-time instrumentation may not process classes loaded by custom classloaders such as CL2. As a result, it may miss the instrumentation of `A` and `B` loaded by CL2. Moreover, load-time weavers not handling classloader namespaces properly may fail to distinguish classes with the same fully qualified name but different defining classloader, treating them as if they were the same class. In such a case, both `A` and `B` classes could be instrumented (even though those defined by `CL1` do not fall within the scope of the analysis) or ignored by the weaver (even though those defined by CL2 should be instrumented).

Moreover, suppose now that our analysis targets subtypes of `A` rather than `Runnable`. A flexible instrumentation framework should provide users a way to define at instrumentation-time which of the two `A` classes falls within the scope of the analysis (e.g., instrument only subtypes of `A` defined by CL2), such that the weaver may instrument classes based on their defining classloaders. Both AspectJ and DiSL do not offer such functionality, requiring the insertion of runtime checks to retrieve the defining classloader of a class and, if the classloader matches the one targeted by the analysis, execute instrumentation code.

4.2.5 Our Solution

Our approach reconciles the benefit offered by out-of-process instrumentation (i.e., full bytecode coverage) with the provisioning of accurate and complete RSI and classloader namespaces. We propose a novel technique to accurately reify the class hierarchy (including classloader namespaces) of the instrumented application within a separate instrumentation process, such that accurate and complete RSI is available for each class to be instrumented. Our solution avoids the use of expensive runtime checks required otherwise, enabling efficient analyses that guarantee full bytecode coverage. Moreover, our approach correctly deals with homonym classes defined by different classloaders and allows the developer to identify classloader namespaces (if desired) when writing dynamic analyses.

We implement our technique in an extended version of DiSL, where the weaver reifies classloader namespaces. Figure 4.3 shows the architecture of DiSL (as in version 2.1, the latest at the time of writing) and our contribution. The original DiSL does not have access to complete RSI and is unaware of classloader namespaces, because it cannot access the `Class` (resp. `ClassLoader`) instances of the classes (resp. classloaders) of the observed JVM. As a consequence, it cannot instrument classes based on their position in the type hierarchy, and treats as equal classes with the same fully qualified name but different defining classloaders. Our work tackles this issue, providing complete RSI and classloader namespaces to the instrumentation process (i.e., the DiSL server). This information can be inspected by the DiSL server thanks to our new DiSL Reflection API (described in the next section), which mirrors `Class` and `ClassLoader` instances in the observed JVM. Our technique allows DiSL analyses to reconcile full bytecode coverage with efficiency.

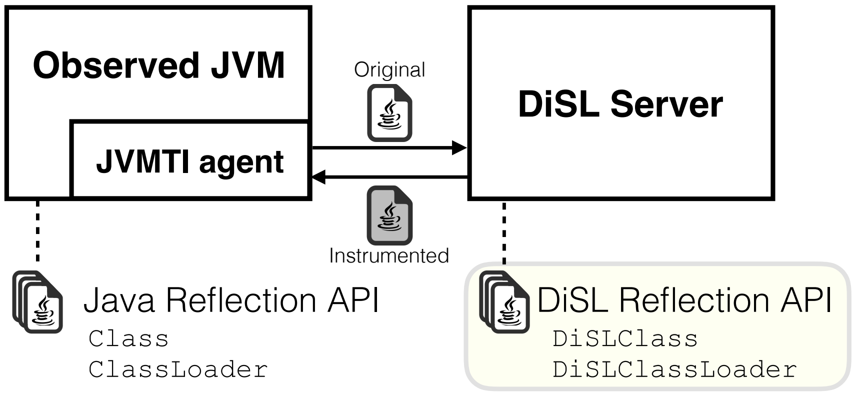


Figure 4.3. DiSL architecture. DiSL (version 2.1) cannot access complete RSI and is unaware of classloader namespaces. Our work introduces the new DiSL Reflection API, which enables the DiSL server to inspect complete RSI and classloader namespaces by mirroring `Class` and `ClassLoader` instances that can be obtained with the Java Reflection API in the observed JVM.

4.3 The DiSL Reflection API

A key feature of our approach is the new *DiSL Reflection API*, which provides an interface to access RSI within the DiSL server, as if the server could access the Java Reflection API in the observed JVM, retrieving the (future) `Class` instance corresponding to the class being instrumented (see Figure 4.3). Moreover, our API allows inspecting classloader namespaces.

Figure 4.4 summarizes the key classes and methods of the DiSL Reflection API. The purpose of the classes `DiSLClassLoader` and `DiSLClass` is to mirror their counterparts in the observed application (i.e., `ClassLoader` and `Class`, respectively), such that RSI and classloader namespaces can be inspected from the DiSL server. Each time a new classloader is used to load a class, the corresponding instance of `DiSLClassLoader` is created in the DiSL server. Similarly, when a class is loaded by the observed application, the server creates the corresponding `DiSLClass`.

`DiSLClassLoader` represents the namespace of the corresponding classloader within the DiSL server; in particular, it maintains the mapping from fully qualified class names to the corresponding `DiSLClass` instances. Thanks to our API, an instrumentation is able to correctly handle homonym classes defined by different classloaders. Internally, `DiSLClassLoader` instances are identified by a unique ID (line 2). ID 0 corresponds to the bootstrap classloader, ID 1 corresponds

```
 1 public interface DiSLClassLoader {
 2   long getID();
 3   DiSLClass forName(String fullyQualifiedClassName);
 4   boolean isBootstrapLoader();
 5   boolean isApplicationLoader();
 6   ...
 7 }
 8
 9 public interface DiSLClass {
10   String getName();
11   DiSLClassLoader getClassLoader();
12   DiSLClass getSuperclass();
13   Stream<DiSLClass> getInterfaces();
14   Stream<DiSLClass> getSupertypes();
15   ...
16 }
17
18 public class ReflectionContext extends AbstractStaticContext {
19   public DiSLClass thisClass() { ... };
20   ...
21 }
```

Figure 4.4. Classes and methods of the new DiSL Reflection API. Methods not relevant for the dissertation are omitted.

to the application classloader, while subsequent IDs identify other user-defined classloaders.

A `DiSLClass` allows inspecting RSI of a class from the DiSL server. In particular, it exposes the superclass (line 12) and the directly implemented interfaces (line 13) of a class. Our approach ensures that when a class is being instrumented, the corresponding `DiSLClass` and those of all its supertypes are available. Supertype information returned by a `DiSLClass` is fully compliant with the JVM specification [101].[6] `DiSLClass` also offers the convenience method `getSupertypes` to retrieve all (direct and indirect) supertypes of a class, including the class itself (line 14).

The DiSL Reflection API is typically used in guards at instrumentation-time. To this end, our API introduces new context information than can be accessed

[6]The contracts of the methods `getSuperclass` and `getInterfaces` are the same of the homonym methods in `java.lang.Class`, with the exception that interfaces are returned using a `Stream` rather than an array.

Table 4.1. Abstract data types used in Figure 4.5 and their translation into Java, C or JNI types.

Type	**Description**	**Java equivalent**	**C/JNI equivalent**
LONG	A long integer.	`long`	`jlong`
STRING	A string.	`java.lang.String`	`char *`
C	A class.	`java.lang.Class`	`jclass`
B_c	A byte array representing class c.	`byte[]`	`char *`
L	A classloader ($\perp_L$ represents the bootstrap classloader).	`java.lang.ClassLoader`	`jobject`
$\mathscr{S}$	A list of STRINGs.	`List<String>`	`char **`
P	An ordered pair <STRING,LONG>. The first element represents a fully qualified class name; the second element the ID of a classloader.	`class P { String name; long id; }`	`typedef struct { char *name; jlong id; } P`
$\mathscr{P}$	A list of P pairs.	`List<P>`	`P *`

by guards through the `ReflectionContext` (line 18). In particular, this class provides the `DiSLClass` (line 19) of the class currently under instrumentation, allowing guards to perform their checks based on RSI. We show how to use the DiSL Reflection API in tgp in Section 4.5.

4.4 Implementation

The DiSL Reflection API enables efficient analyses thanks to the provisioning of RSI and classloader namespaces. In this section, we discuss the key implementation details of our approach, which ensures that such information is always available in the DiSL server. We first outline the algorithms involved by means of abstract data types; then, we expand the discussion showing C and Java code. The abstract data types involved in this section are shown in Table 4.1, and the functions defined on them are reported in Table 4.2. Note that lists are ordered and support concatenation through the || operator. The notation $\perp_T$ refers to the null value of type T (see Section 3.5.1). We consider -1 as null value for LONG (in both

Table 4.2. Functions on abstract data types used in Figure 4.5. () denotes the empty list.

Function	**Description**	**C code**
`superclass(`B_c b`)`: STRING	Returns the fully qualified name of c's direct superclass, or $\perp_{\text{STRING}}$ if c is `java.lang.Object`.	-
`interfaces(`B_c b`)`: $\mathscr{S}$	Returns the list of fully qualified names of c's directly implemented interfaces, in the order they are implemented, or () if c does not implement any interface.	-
`loadClass(`STRING s`,`L l`)`: C	Loads a class with fully qualified name s using classloader l, returning the corresponding `java.lang.Class`. l can be $\perp_L$, denoting the bootstrap classloader.	Figure 4.6.
`classloader(`C c`)`: L	If c is defined by an user-defined classloader, returns the corresponding `java.lang.ClassLoader`. If c is defined by the bootstrap classloader, returns $\perp_L$.	Figure 4.7
`id(`L l`)`: LONG	Returns 0 if $l = \perp_L$. Otherwise, returns the unique ID (> 0) associated with l. If no ID has been associated to l, this function assigns a unique ID to l before returning it.	Figure 4.8
`instrumentRemotely(`B_c b`,` LONG i`,` P p_s`,` $\mathscr{P}$ p_i`)`: B_c	Sends b to the server for instrumentation, along with the ID of c's defining classloader (i, which must be ≥ 0) and information to univocally identify c's superclass (p_s, which can be $\perp_P$ if c is `java.lang.Object`) and c's directly implemented interfaces (p_i). Returns the instrumented byte array of c.	-

Java and C), and `null` (resp. `NULL`) for all other types in Java (resp. C).[7] In the following text, we often assume that the server is instrumenting a class c.[8]

4.4.1 Forced Loading of Supertypes

The DiSL server creates a new `DiSLClass` when it receives the corresponding class for instrumentation. To access RSI of c at instrumentation-time (e.g., in guards), the `DiSLClass` instances of all the supertypes of c must be available in the server when c is being instrumented. This implies that every supertype of c must be instrumented before c itself. Unfortunately, DiSL sends classes to the server in the same order as they are loaded by the JVM; hence, some of the supertypes of c may not have been received by the server when instrumenting c. In such a case, attempts of retrieving RSI at instrumentation-time will fail, as some required `DiSLClass` instances are not yet available. We tackle this issue by modifying the order in which classes are sent to the server for instrumentation, ensuring that all the supertypes of c are sent before c. To this end, our framework forces the loading of each supertype of c that has not yet been loaded.

Figure 4.5 reports the algorithm used to force the loading and instrumentation of c's supertypes. The algorithm is executed in a callback, invoked when c has been loaded in the observed JVM (but before the JVM has created a `Class` representing c). The callback has access to the byte array representing c as loaded from the classfile (which can be modified by the DiSL server and cannot be $\perp_B$), and the defining classloader of c. First, the algorithm retrieves the fully qualified name of c's superclass (line 2) and triggers its loading through the helper function `loadAndGetClassloaderID` (line 5), unless c is `java.lang.Object`, which has no superclass [101]. In particular, the function asks the JVM to load the supertype using the defining classloader of c via `loadClass` (line 17). This process is then repeated for each of the interfaces directly implemented by c (lines 8–10). Note that `loadClass` triggers the nested execution of the callback in the context of a supertype of c, resulting in the recursive loading of *every* supertype of c, including indirect ones. When the foreach loop terminates (line 13), every supertype of c has been loaded and instrumented. At this time, complete RSI of c is available in the instrumentation process, and the algorithm can send the class to the server for instrumentation (line 14).

The above algorithm is implemented in the JVMTI agent attached to the observed JVM (see Figure 4.3). In particular, the callback and its arguments

[7]Unless otherwise noted, the functions listed in Table 4.2 are undefined if any of their input parameters is $\perp$.

[8]For simplicity, error-handling code is not shown here.

```
input  : b         original byte array of the loaded class
         l         defining classloader of the loaded class
output : B_c       instrumented byte array of the loaded class

callback onClassLoad($B_c$ $b$, $L$ $l$) begin
    STRING $s_s \leftarrow$ superclass($b$)
    $P$ $p_s \leftarrow \perp_P$
    if $s_s \neq \perp_{\text{STRING}}$ then
        $p_s \leftarrow <s_s,$ loadAndGetClassloaderID($s_s, l$) $>$
    end
    $\mathscr{P}$ $p_i \leftarrow ()$
    $\mathscr{S}$ $s_i \leftarrow$ interfaces($b$)
    foreach $s_j \in s_i$ do
        $P$ $p_j \leftarrow <s_j,$ loadAndGetClassloaderID($s_j, l$) $>$
        $p_i \leftarrow p_i \parallel (p_j)$
    end
    /* Here, all supertypes have been (recursively)
    instrumented */
    return instrumentRemotely($b$, id($l$), $p_s$, $p_i$)
end
```

```
input  : s         fully qualified class name of the class to be loaded
         l         classloader used for initiating the loading of s
output : LONG      ID of the defining classloader of the loaded class

function loadAndGetClassloaderID(STRING $s$, $L$ $l$) begin
    return id(classloader(loadClass($s$, $l$)))
end
```

Figure 4.5. Algorithm to force supertype loading and retrieve classloader IDs.

```
 1 jclass loadClass(char *s, jobject l) {
 2   if (l == NULL) {
 3     return (*jni)->FindClass(jni,s);
 4   }
 5   jclass loaderClass = (*jni)->GetObjectClass(jni,l);
 6   jmethodID methodID = (*jni)->GetMethodID(jni,loaderClass,
 7     "loadClass", "(Ljava/lang/String;)Ljava/lang/Class;");
 8   s = formatName(s); // Replace '/' with '.'
 9   jstring className = (*jni)->NewStringUTF(jni,s);
10   return (jclass) (*jni)->CallObjectMethod(jni,l,methodID,className);
11 }
```

Figure 4.6. Native code of `loadClass`(STRING s,L l): C.

are provided by JVMTI, which allows an agent to be notified and to execute custom code upon class loading.[9] Functions `superclass` and `interfaces` are implemented by looking up the byte array representing c at specific indices, following the class file format [101]. Finally, the C code for `loadClass` is shown in Figure 4.6.[10] The purpose of this function is to call `l.loadClass(s)` from native code through JNI, triggering the loading of a class named `s` by classloader `l` (lines 5–10). Note that no class is loaded if `l` was already requested to load a class named `s`, and that `l` can delegate loading to another classloader. Special handling is needed if `l` is the bootstrap classloader. No object representing the bootstrap classloader is accessible in Java, thus no JNI calls can be made on `l` (`l` is `NULL`). In this case, we resort to the JNI function `FindClass` (line 3), which can look up and load classfiles located in the classpath using the bootstrap classloader.

According to the JVM specification, all supertypes of c must have been loaded before c is initialized and used by the observed JVM. Our algorithm does not load any class that would not be loaded otherwise, resulting in no observable difference from the application perspective (apart from calls to the classloader possibly arriving in a slightly different order). Moreover, the JVM would initiate the loading of the supertypes of c using the same defining classloader of c. Our algorithm follows the same approach, resulting in no alteration of classloader namespaces in the observed application.

[9]See the JVMTI documentation of the `ClassFileLoadHook` event for more information.

[10]In all C functions shown in this chapter, the variables `jni` and `jvmti` refer to function pointers offered by the JNI or JVMTI API, respectively.

4.4.2 Classloader Namespaces

The design behind DiSL mandates that the DiSL server be completely independent from the observed application; thus, any information related to the classes used by the application must be explicitly sent to the server by the JVMTI agent. The communication protocol employed by DiSL just sends (to the server) the byte array of the class to be instrumented, without any classloader information. As a result, DiSL is unaware of classloader namespaces.

Our approach maintains different classloader namespaces in the DiSL server, ensuring that homonym classes defined by different classloaders are handled correctly. While maintaining separate namespaces, the server must ensure that the `DiSLClass` of a supertype can be retrieved by a subtype even if the two are defined by different classloaders. This situation occurs frequently; for example any application class (loaded by a user-defined classloader) has at least one supertype (`Object`) in the `java.*` package, and classes within `java.*` can be loaded only by the bootstrap classloader [101]. To handle such cases correctly, we modify the agent-to-server communication protocol, including additional data to identify classes univocally. The rest of this section describes the rationale of our approach and some implementation details.

Retrieving and Sending Classloader Data

Apart from the byte array representing c (needed for instrumenting the class), the agent in our framework sends (to the server) the defining-classloader ID of c and complete information (fully qualified name and defining-classloader ID) of all direct supertypes of c. This information is retrieved during the forced loading of supertypes, as shown in Figure 4.5. As soon as a supertype of c has been loaded, the agent retrieves the ID of its defining classloader (line 17). Note that such classloader can be different from l in case of classloader delegation. The agent stores the ID along with the fully qualified name of the supertype just loaded in a dedicated structure (p_s) for the superclass (line 5) or in a dedicated list (p_i) for the interfaces implemented by c (line 11). When all the supertypes have been loaded, p_s and p_i contain all the necessary information to identify each supertype of c univocally. Such information is then sent to the server (line 14) along with the defining-classloader ID of c.

Figure 4.7 reports the C code for retrieving the defining classloader of c. The purpose of function `classloader` is to call `c.getClassLoader()` from native code (lines 5–8) through JNI. The call is avoided if c is defined inside the `java.*` package (lines 2–4) for two reasons: 1) classes inside that package can only

```
jobject classloader(jclass c) {
  if (isInJavaPackage(c)) {
    return NULL;
  }
  jclass kl = (*jni)->FindClass(jni,"java/lang/Class");
  jmethodID methodID = (*jni)->GetMethodID(jni,kl,
    "getClassLoader","()Ljava/lang/ClassLoader;");
  return (*jni)->CallObjectMethod(jni,c,methodID);
}
```

Figure 4.7. Native code of `classloader(`$C\ c$`):` L.

be loaded by the bootstrap classloader, and 2) the call can occur during JVM bootstrap (when Java classes are being initialized), interfering with their loading. In this case, the function returns `NULL` (to represent the bootstrap classloader).

Finally, Figure 4.8 details the process of retrieving the ID of classloader l. The agent makes use of the JVMTI heap tagging feature to assign or retrieve a unique long tag to/from each classloader, using the tags as IDs. According to the specification of the DiSL Reflection API, the bootstrap classloader has ID 0 (lines 3–5). For all other classloaders, the agent first checks whether the classloader has already been tagged, returning the corresponding tag in such a case (lines 6–10). Otherwise, it tags the classloader with a successive unique ID (lines 11–14), returning the assigned tag. This implementation ensures that classloaders are tagged in the same order they are used by the JVM, such that the application classloader has ID 1, while subsequent user-defined classloaders have successive IDs. The code is atomically executed in a critical section, to avoid any race condition.

Maintaining Classloader Namespaces

The DiSL server uses the additional data sent by our agent to maintain classloader namespaces, enabling the inspection of RSI. For each loaded class c, this process involves: 1) creating a `DiSLClassLoader` instance cl corresponding to the defining loader of c, 2) creating a `DiSLClass` instance kl corresponding to c, and 3) inserting kl into cl.

Figure 4.9 details the above operations. Note that this process occurs before c is actually instrumented (line 38), such that guards can access complete RSI during the instrumentation of c. First, the server obtains cl, (atomically) creating it if c is the first class loaded by the classloader (line 34). Then, the server creates

```
static jlong next_tag = 1;
jlong id(jobject l) {
  if (l == NULL) {
    return 0;
  }
  jlong id;
  (*jvmti)->GetTag(jvmti,l,&id);
  if (id != 0) {
    return id;
  }
  // l still untagged
  jlong tag = next_tag++;
  (*jvmti)->SetTag(jvmti,l,tag);
  return tag;
}
```

Figure 4.8. Native code of `id(`*L l*`)`: LONG. The code is atomically executed in a critical section (not shown here for simplicity).

a new `DiSLClass` (line 36) using the information received from the agent. A `DiSLClass` instance is immutable, since all the information required to completely represent its state are known at creation time. Upon creation, an implementation of a `DiSLClass` (shown as `DiSLClassImpl` in the figure) needs to obtain the `DiSLClass` instances of its supertypes (lines 13–16). To this end, the server queries the defining classloader of each direct supertype for the corresponding `DiSLClass` (via method `getDiSLClass`; lines 20–26). Due to the forced loading of supertypes, `DiSLClass` and `DiSLClassLoader` lookup is guaranteed to succeed (apart from the corner cases described in Section 4.4.3). As last step, the server (atomically) inserts kl into cl (line 37).

4.4.3 Preprocessing Java Core Classes

The forced loading of supertypes makes RSI available in the DiSL server. This process relies on JNI to call Java methods from native code (see Figure 4.6). Unfortunately, JNI is not available during the primordial phase of the JVM, when the JVM and the core Java classes are being initialized. As a result, the forced loading of supertypes cannot be ensured for all the Java classes loaded during the primordial phase of the JVM. This fact can lead to failures when retrieving RSI, if c is loaded during the primordial phase and one or more supertypes of c have not been loaded yet. For example, in the JVM used in the evaluation (more details

```java
class DiSLClassImpl implements DiSLClass {
  final private String className;
  final private DiSLClassLoader classloader;
  final private DiSLClass superclass;
  final private List<DiSLClass> interfaces =
                 new LinkedList<>();


  DiSLClassImpl(String name, DiSLClassLoader cl, P ps,
                 List<P> pi) {
    className = name;
    classloader = cl;
    superclass = getDiSLClass(ps);
    for (P p: pi) {
       interfaces.add(getDiSLClass(p));
    }
  }


  DiSLClass getDiSLClass(P p) {
    if (p == null) {
       return null;
    }
    DiSLClassLoader cl = getClassLoaderFromID(p.id);
    return cl.forName(p.name);
  }

  ...

}


byte[] instrument(byte[] b, long i, P ps, List<P> pi) {
  DiSLClassLoader cl = getOrCreateClassLoader(i);
  String name = parseClassName(b);
  DiSLClass kl = new DiSLClassImpl(name, cl, ps, pi);
  insert(cl, kl); // Inserts kl in the namespace of cl
  ... // Instrument class
}
```

Figure 4.9. Code for maintaining classloader namespaces within the DiSL server.

in Section 4.6), `java.lang.Thread` (which implements `java.lang.Runnable`) is loaded before `Runnable` during the primordial phase, leading to failures when a guard attempts to retrieve RSI of `Thread`.

Our framework tackles this issue by precomputing the `DiSLClass` of each class inside package `java.*`. The correctness of our approach is supported by the fact that all classes inside `java.*` can be loaded only by the bootstrap classloader (such that their defining classloader is always known). Precomputing reflective information of the Java core classes ensures the correct provisioning of complete RSI, assuming that the Java class library used for the precomputation and the one of the observed application is the same.

4.5 Efficient Task-Granularity Profiling

In this section, we describe how tgp can leverage reification of RSI to decrease profiling overhead and the perturbations of the collected task-granularity metrics caused by the inserted instrumentation code. We focus on task constructors (which must be instrumented to create a new task profile and register the creating thread), comparing two implementations of the `afterTaskCreation` DiSL code snippet (see Figure 3.5, lines 9–12) resorting to runtime checks and to the new DiSL Reflection API, respectively. Similar observations apply to execution and submission methods. While this section focuses only on task profiling, the same approach can be used to recast existing type-specific analyses written in DiSL (such as actor profiling [117]) into more efficient versions using the DiSL Reflection API.

Figure 4.10(a) reports the implementation of the code snippet in DiSL 2.1, where the DiSL Reflection API is not available. The call to `Profiler.registerCreation` (defined in Table 3.1) allows the creation of a task profile associated to a new task. The call should occur when the constructor of a newly created task terminates. While DiSL allows users to insert instrumentation code at the end of a constructor (line 1), the DiSL server has no access to complete RSI, and may not be able to determine whether the class under instrumentation is a subtype of a task interface. To guarantee a complete analysis (i.e., to detect all spawned tasks), DiSL instruments *every* constructor *all* loaded classes, adding runtime checks (lines 3–5) to ensure that `Profiler.registerCreation` is called only in task constructors. While this approach guarantees a complete analysis, executing runtime checks each time a new object is created can introduce significant profiling overhead.

```
@After(marker = AfterInitBodyMarker.class, scope = "<init>")
public static void afterTaskCreation(DynamicContext dc) {
 if (dc.getThis() instanceof Runnable ||
     dc.getThis() instanceof Callable<?> ||
     dc.getThis() instanceof ForkJoinTask<?>) {
       Profiler.registerCreation(dc.getThis(), Thread.currentThread());
 }
}
```

(a) Using runtime checks.

```
@After(marker = AfterInitBodyMarker.class, scope = "<init>",
       guard = TaskGuard.class)
public static void afterTaskCreation(DynamicContext dc) {
       Profiler.registerCreation(dc.getThis(), Thread.currentThread());
}

// Executed in the DiSL server at instrumentation-time
public final class TaskGuard {
 @GuardMethod
 public static boolean guard(ReflectionContext rc) {
  return rc.thisClass().getSupertypes().anyMatch(
   s -> (s.getName().equals("java.lang.Runnable") ||
         s.getName().equals("java.util.concurrent.Callable") ||
         s.getName().equals("java.util.concurrent.ForkJoinTask"))
  );
 }
}
```

(b) Resorting to the DiSL Reflection API.

Figure 4.10. DiSL code in tgp for instrumenting task constructors.

Figure 4.10(b) shows how to recast the above code snippet to leverage the DiSL Reflection API. Instead of inserting runtime checks in every class, the code snippet inserts a call to `Profiler.registerCreation` (line 4) only if the associated guard (`TaskGuard`, line 2) evaluates to true. The guard (executed in the DiSL server to determine whether a class shall be instrumented) resorts to the DiSL Reflection API to instrument only task interfaces and their subtypes by retrieving the `DiSLClass` of the class being instrumented through the `ReflectionContext` (line 11), and checking whether a task interface is among its supertypes (lines 12–14). Note that there is no need for the guard to specify the namespace where task interfaces should be defined, as task interfaces are part of the `java.*` package, which can be defined only by the bootstrap classloader. Our approach guarantees that when a `DiSLClass` is retrieved (e.g., via `ReflectionContext.thisClass`), complete RSI for the class are available in the DiSL server, such that guards can always determine all supertypes of the class under instrumentation.

Thanks to the provision of RSI and classloader namespaces, DiSL-based analyses running on our framework can avoid slowdowns caused by the execution of expensive runtime type checks inserted in many loaded classes, as the weaver can insert instrumentation code only into classes in the scope of the analysis. Avoiding unnecessary slowdowns is fundamental in tgp, as the metrics profiled (reference cycles in particular) are very sensitive to perturbations introduced by the inserted instrumentation code; hence, low profiling overhead is fundamental to obtain accurate measurements, as discussed in the next section.

4.6 Evaluation

In this section, we evaluate the benefits offered by our approach. We present the workloads considered in the evaluation and the experimental setup in Section 4.6.1. We discuss the overhead of tgp and the speedup enabled by the DiSL Reflection API in Section 4.6.2. Finally, Section 4.6.3 shows that our approach helps reducing measurement perturbations when profiling task granularity.

4.6.1 Methodology and Setup

We demonstrate the benefits offered by our technique by comparing two versions of the instrumentation implemented in tgp, i.e., resorting to runtime checks and using the DiSL Reflection API, respectively. 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 [15], ScalaBench [123], and Spark Perf [25] suites. DaCapo and ScalaBench are composed of well-known JVM workloads written in Java and Scala, respectively, while Spark Perf is a collection of benchmarks for the popular Apache Spark [149] 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 Decemeber 2015). Following the recommendation of the DaCapo developers,[11] we execute benchmark lusearch-fix in place of lusearch.[12] Both DaCapo and ScalaBench can be executed with different input sizes. We use the largest input defined for each workload. 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. [75]. All other iterations after warm-up are classified as steady-state. The evaluation presented in this section, as well as task-granularity analysis and optimization presented in Chapter 5 target only steady-state iterations. Tables 4.3, 4.4 and 4.5 list the workloads considered, along with a brief description, the input size used (in DaCapo and ScalaBench), and the number of warm-up iterations (in the last column).

In all workloads, we use the stop-the-world parallel collector [102] as GC (i.e., the default GC on multicores). Our choice allows filtering out cycles elapsed when GC is active (which are not related to the execution of application code). We conduct our evaluation on a server-class machine equipped with two NUMA nodes, each with an Intel Xeon E5-2680 (2.7 GHz) processor with 8 physical cores and 64 GB 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 computational-intensive application is in execution on the system, to reduce the perturbation on OS-layer metrics as discussed in Section 3.7.2. 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). 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,

[11]See `http://dacapobench.org` for more information.

[12]Results reported on lusearch have been obtained on lusearch-fix.

Table 4.3. DaCapo [15] workloads considered in the dissertation. Benchmark description is taken from `http://dacapobench.org/benchmarks.html`.

Benchmark	**Description**	**Input**	**# w.u.**
avrora	Simulates a number of programs run on a grid of AVR microcontrollers.	large	20
batik	Produces a number of Scalable Vector Graphics (SVG) images based on the unit tests in Apache Batik.	large	20
eclipse	Executes some of the (non-gui) jdt performance tests for the Eclipse IDE.	large	20
fop	Takes an XSL-FO file, parses it and formats it, generating a PDF file.	default	40
h2	Executes a JDBCbench-like in-memory benchmark, executing a number of transactions against a model of a banking application, replacing the hsqldb benchmark.	huge	10
jython	Inteprets the pybench Python benchmark.	large	20
luindex	Uses lucene to index a set of documents; the works of Shakespeare and the King James Bible.	default	40
lusearch	Uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible.	large	20
pmd	Analyzes a set of Java classes for a range of source code problems.	large	20
sunflow	Renders a set of images using ray tracing.	large	20
tomcat	Runs a set of queries against a Tomcat server retrieving and verifying the resulting webpages.	huge	10
tradebeans	Runs the daytrader benchmark via a Java Beans to a GERONIMO backend with an in memory h2 as the underlying database.	huge	10
tradesoap	Runs the daytrader benchmark via a SOAP to a GERONIMO backend with in memory h2 as the underlying database.	huge	10
xalan	Transforms XML documents into HTML.	large	20

Table 4.4. ScalaBench [123] workloads considered in the dissertation. Benchmark description is taken from `http://www.benchmarks.scalabench.org`.

Benchmark	**Description**	**Input**	**# w.u.**
actors	Trading sample with Scala and Akka actors.	huge	10
apparat	Framework to optimize ABC, SWC, and SWF files.	gargantuan	5
factorie	Toolkit for deployable probabilistic modeling.	gargantuan	5
kiama	Library for language processing.	default	40
scalac	Compiler for the Scala 2 language.	large	20
scaladoc	Scala documentation tool.	large	20
scalap	Scala classfile decoder.	large	20
scalariform	Code formatter for Scala.	huge	10
scalatest	Testing toolkit for Scala and Java programmers.	default	40
scalaxb	XML data-binding tool.	huge	10
specs	Behaviour-driven design framework.	large	20
tmt	Stanford Topic Modeling Toolbox.	huge	10

Table 4.5. Spark Perf [25] workloads considered in the dissertation.

Benchmark	**Description**	**# w.u.**
AlternatingLeastSquares	Runs the ALS algorithm from mllib.	10
ChiSquare	Runs the chi-square test from mllib.	60
ClassificationDecisionTree	Runs the Random Forest algorithm from mllib.	20
GaussianMixtureEM	Computes a Gaussian mixture model using expectation-maximization.	40
KMeansClustering	Runs K-Means++ algorithm from mllib.	35
LogRegression	Runs the logistic regression workload from mllib.	20
MultinomialNaiveBayes	Runs the multinomial naive Bayes algorithm from mllib.	20
PrincipalComponentAnalysis	Runs the principal component analysis algorithm.	20
StreamingWordCount	Produces a word frequency histogram from a stream of words.	5

perf, and top are not accounted in the measurements, increasing the accuracy of the resulting CPU trace. We disable Turbo Boost [59] 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 [58], 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.

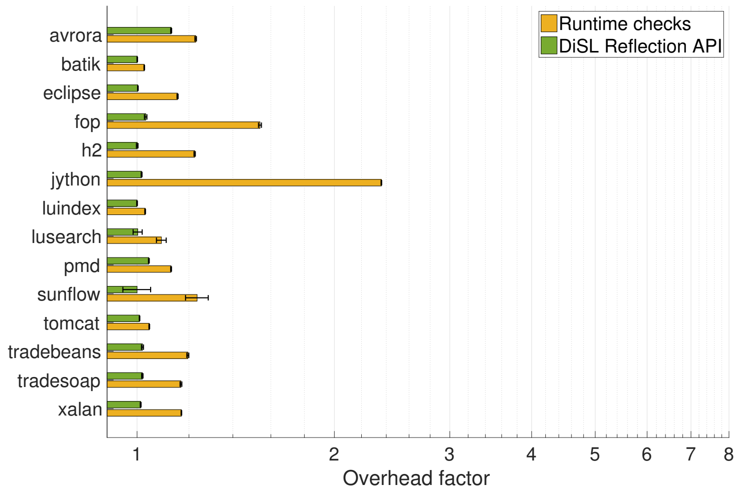
Spark Perf benchmarks are run on Spark version 2.3.0 (compiled with Scala 2.11), set up in *local mode* on a single machine with as many executors as available cores. Note that in local mode, all Spark components, including the driver and all executors, run in a single JVM.

4.6.2 Profiling Overhead and Speedup

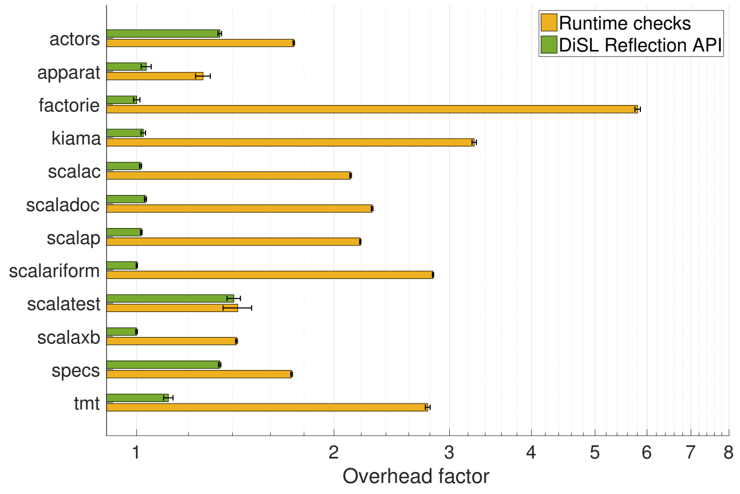
In this section, we compare the profiling overhead caused by tgp when using runtime checks and when resorting to the DiSL Reflection API, evaluating the speedup enabled by our approach wrt. runtime checks. We present overheads in the form of *overhead factors*, defined as the execution time of the observed application with profiling enabled divided by the execution time of the observed application with profiling disabled. We show our results in Figure 4.11. For each workload, we report the average overhead obtained on 20 steady-state runs. Error bars indicate 95% confidence intervals.

Overall, the overhead introduced by the DiSL Reflection API is low for most of the applications. In general, the overhead factor does not exceed 1.04×, with the following exceptions. In DaCapo (Figure 4.11(a)), avrora incurs an average overhead of 1.13×. In ScalaBench (Figure 4.11(b)), 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 (Figure 4.11(c)), 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.

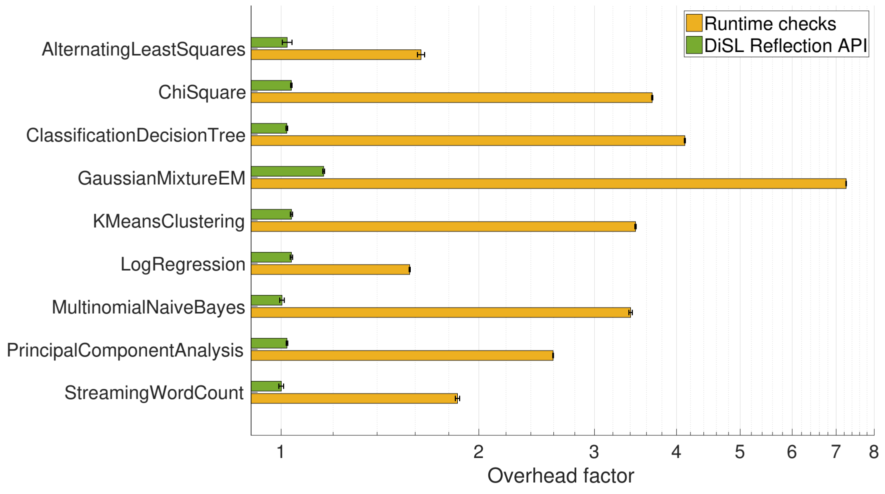
On the other hand, the overhead caused by runtime checks is always higher than the one introduced by the DiSL Reflection API, causing significant slowdowns in many benchmarks. In DaCapo, the workload which is mostly slowed down by runtime checks is jython (2.36×). In ScalaBench, the slowdown is significant in all benchmarks, with overheads above 2.00× in scalac, scaladoc, scalap, scalariform, and tmt, and above 3.00× in kiama and factorie, which suffers from the highest slowdown (5.81×). Similarly, runtime checks have a strong negative effect in Spark Perf, causing overhead factors above 2.00× in PrincipalComponentAnalysis, above 3.00× in ChiSquare, KMeansClustering, and MultinomialNaiveBayes, and



(a) DaCapo [15].



(b) ScalaBench [123].



(c) Spark Perf [25].

Figure 4.11. Profiling overhead using the DiSL Reflection API vs. resorting to runtime checks.

above 4.00× in ClassificationDecisionTree and GaussianMixtureEM, where the slowdown is maximum (7.25×).

On average, using the DiSL Reflection API incurs a minor slowdown of 1.02× (DaCapo), 1.11× (ScalaBench), and 1.04× (Spark Perf), whereas runtime checks cause a non-negligible overhead of 1.26×, 2.40×, and 3.28×, in DaCapo, ScalaBench, and Spark Perf, respectively.

Table 4.6 reports the speedup of tgp resorting to the DiSL Reflection API wrt. using runtime checks. The values shown are *speedup factors*, obtained by dividing, for each workload, the execution time of the analysis using runtime checks by the correspondent execution time using the DiSL Reflection API. We also report 95% confidence intervals. As can be read from the table, the DiSL Reflection API enables significant speedups in most of the workloads. In DaCapo, the highest speedup can be observed in jython (2.32×). In ScalaBench, our technique leads to noticeable speedups in most of the workloads, i.e., scalac (2.09×), scaladoc (2.22×), scalap (2.16×), scalariform (2.83×), tmt (2.49×), kiama (3.20×), with the maximum speedup observed in factorie (5.80×). Finally, all benchmarks in Spark Perf benefit significantly from the DiSL Reflection API, with a minimum speedup of 1.51× (LogRegression), a maximum of 6.24× (GaussianMixtureEM), and an average speedup factor above 3.00× in several workloads (ChiSquare, ClassificationDecisionTree, KMeansClustering, and MultinomialNaiveBayes). On average, the speedup enabled by the DiSL Reflection API is 1.23× in DaCapo, 2.25× in ScalaBench, and 3.12× in Spark, while the maximum is 6.24× (GaussianMixtureEM). Such high speedups stem from the fact that expensive runtime checks need to be executed in the constructors of each new object if the DiSL Reflection API is not used.

Overall, our experimental results show that the DiSL Reflection API can significantly speed up certain analyses that use runtime checks. In the case of tgp, the high overhead introduced by runtime checks can jeopardize the accuracy of the collected task granularities, which are very sensitive to measurement perturbations caused by the inserted instrumentation code. On the other hand, the much lower overhead introduced by the DiSL Reflection API is less likely to significantly perturb the collected metrics, enabling more accurate task-granularity analyses, as discussed in the next section.

4.6.3 Perturbation

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

Table 4.6. Speedup enabled by the DiSL Reflection API wrt. using runtime checks.

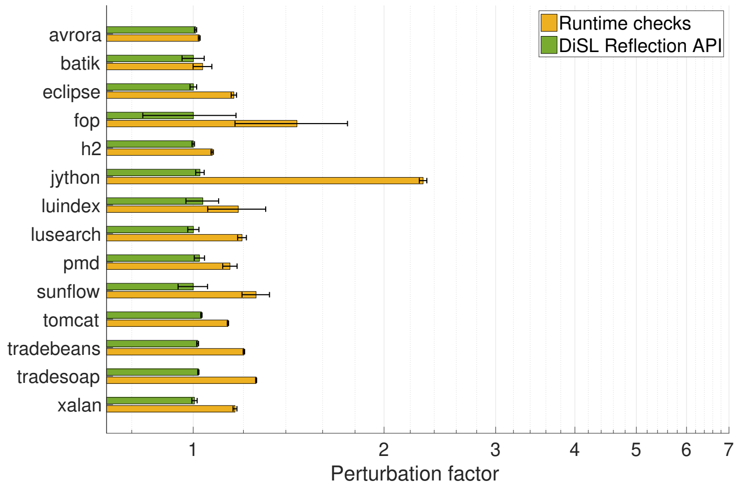
Benchmark DaCapo [15]	**Speedup Factor (± 95% conf.)**	**Benchmark ScalaBench [123]**	**Speedup Factor (± 95% conf.)**
avrora	1.0899 ± 0.0029	actors	1.2968 ± 0.0060
batik	1.0250 ± 0.0011	apparat	1.2205 ± 0.0244
eclipse	1.1495 ± 0.0022	factorie	5.8030 ± 0.0357
fop	1.4941 ± 0.0063	kiama	3.1951 ± 0.0067
h2	1.2236 ± 0.0036	scalac	2.0901 ± 0.0058
jython	2.3218 ± 0.0032	scaladoc	2.2156 ± 0.0061
luindex	1.0288 ± 0.0014	scalap	2.1569 ± 0.0056
lusearch	1.0868 ± 0.0179	scalariform	2.8282 ± 0.0048
pmd	1.0811 ± 0.0018	scalatest	1.0137 ± 0.0170
sunflow	1.2348 ± 0.0529	scalaxb	1.4201 ± 0.0037
tomcat	1.0348 ± 0.0011	specs	1.2872 ± 0.0043
tradebeans	1.1728 ± 0.0032	tmt	2.4857 ± 0.0375
tradesoap	1.1453 ± 0.0042		
xalan	1.1538 ± 0.0015		

Benchmark Spark Perf [25]	**Speedup Factor (± 95% conf.)**
AlternatingLeastSquares	1.5989 ± 0.0191
ChiSquare	3.5443 ± 0.0064
ClassificationDecisionTree	4.0370 ± 0.0102
GaussianMixtureEM	6.2445 ± 0.0185
KMeansClustering	3.3403 ± 0.0134
LogRegression	1.5137 ± 0.0068
MultinomialNaiveBayes	3.3945 ± 0.0193
PrincipalComponentAnalysis	2.5422 ± 0.0060
StreamingWordCount	1.8551 ± 0.0099

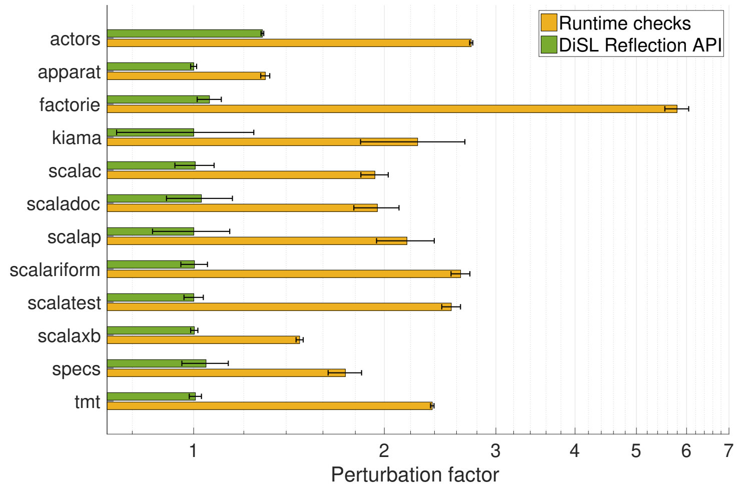
computations inserted may alter the metrics collected, leading to results that may not be representative of the original application behavior. Among all metrics profiled by tgp, reference cycles (used as a measure of task granularity) are particularly susceptible to perturbations caused by the profiling 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 to which extent the DiSL Reflection API helps reduce the 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 in three settings: 1) without using tgp, 2) using tgp with the DiSL Reflection API, and 3) using tgp resorting to runtime checks. 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, as they are not related to the execution of application code.

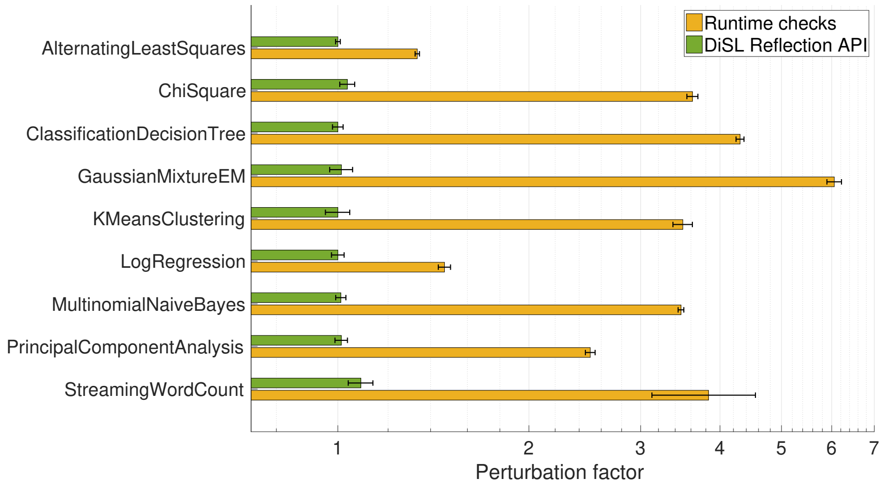
To relate the amount of extra cycles introduced by tgp to the cycles originally elapsed in the target application, we introduce a new metric, the *perturbation factor*, defined as the amount of reference cycles elapsed with tgp enabled, divided by the amount of cycles elapsed with tgp disabled. Note that the purpose of such metric is not to quantify exactly the (relative) amount of extra cycles inserted by tgp in the collected task-granularity profiles. An high perturbation factor indicates that a significant amount of extra cycles have been introduced by the profiling logic; however, such cycles might not be accounted 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 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. In summary, we use perturbation factors only as indication of the quality of the collected profiles. High perturbation factors indicate that the collected metrics are likely to be biased. Low perturbation factors increase our confidence in the results, although they do not proof the absence of significant measurement perturbations.



(a) DaCapo [15].



(b) ScalaBench [123].



(c) Spark Perf [25].

Figure 4.12. Perturbation factors using the DiSL Reflection API vs. resorting to runtime checks.

Figure 4.12 summarizes our results. For each workload, we report the average perturbation factor obtained on 20 steady-state runs. Error bars indicate 95% confidence intervals. In general, the additional cycles introduced are proportional to the profiling overhead (Figure 4.11). This fact highlights that a low profiling overhead is key to conduct an accurate task-granularity analysis. As shown in the figure, the DiSL Reflection API is fundamental to reduce measurement perturbations and enable accurate measures on task granularity. Overall, the perturbation factor related to the DiSL Reflection API is very low for most of the workloads. In all DaCapo benchmarks (Figure 4.12(a)), the perturbation factor caused by our technique is close to 1×, with the highest factor observed in luindex (1.03×). In ScalaBench (Figure 4.12(b)), the only workload 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 5.1.2), each of them introducing extra cycles elapsed due to profiling code. Apart from actors, the perturbation factor observed in all other workloads is below 1.05×. Finally, the only Spark Perf workload (Figure 4.12(c)) with a significant perturbation factor is StreamingWordCount (1.09×), while in all other benchmarks the factor is no higher than 1.03×.

On the other hand, the perturbation factors related to runtime checks are significant, and higher than the ones introduced by the DiSL Reflection API in all workloads. In DaCapo, the highest perturbation factor is observed in jython (2.30×), followed by fop (1.46×). In ScalaBench, the minimum perturbation factor observed is in apparat (1.30×), the maximum is in factorie (5.80×), while several benchmarks incur perturbation factors above 2.00× (actors, kiama, scalap, scalariform, scalatest and tmt). Similarly, runtime checks cause very high perturbation factors in Spark Perf, where the observed factors are above 2.00× in 7 benchmarks (out of 9) and above 3.00× in 6 benchmarks. The minimum perturbation factor observed in the suite is 1.33× (AlternatingLeastSquares) while the maximum is in GaussianMixtureEM (6.06×).

On average, profiling task granularity resorting to the DiSL Reflection API introduces a very low perturbation factor, equal to 1.01× (DaCapo), 1.03× (ScalaBench), and 1.02× (Spark Perf), while runtime checks introduce many extra cycles in the observed application, resulting in perturbation factors in the order of 1.26×, 2.41×, and 3.34× in DaCapo, ScalaBench, and Spark Perf, respectively. Such high factors stem from the need of executing expensive runtime checks in many methods (in particular, in the constructors of each created object). The DiSL Reflection API removes the need for such checks, executing profiling code only in tasks, drastically reducing the amount of extra cycles elapsed, and resulting in collected task-granularity profiles that can more closely represent the original application behavior.

4.7 Discussion

In this section, we report additional features offered by our approach and discuss its limitations.

4.7.1 Reclamation of Classloader Namespaces

In the observed application, `ClassLoader` or `Class` objects may be reclaimed by the garbage collector. When this occurs, our approach ensures that the corresponding `DiSLClassLoader` or `DiSLClass` instances in the DiSL server can be reclaimed as well, to avoid a memory leak in the server. To this end, the JVMTI agent attached to the observed JVM is notified when a tagged `Classloader` object has been freed by the garbage collector (via the JVMTI `ObjectFree` event). As our agent only tags `ClassLoader` objects, this event can only be sent upon the reclamation of a classloader. When the event occurs, the agent signals to the DiSL server the need for invalidating the namespace of the just freed classloader. In turn, the server removes any reference to the corresponding `DiSLClassLoader` and the `DiSLClass` instances composing the namespace, such that they will eventually be reclaimed by the garbage collector of the instrumentation-server JVM.

Freeing up a `DiSLClass` along with the corresponding `DiSLClassLoader` is fully compliant with the Java language specification [100], stating that a class can be reclaimed if and only if its defining classloader may be reclaimed (because classloaders maintain strong references to the loaded classes). As a result, our agent does not need to tag each `Class` object and notify the server upon the reclamation of each of them. While classloader reclamation does not strictly imply that classes in its namespace have been already reclaimed, they are by no means reachable by application code and will eventually be reclaimed as well; hence, it is safe to reclaim the corresponding `DiSLClass` instances in the DiSL server.

4.7.2 Preprocessing Classes Outside `java.*`

Our framework can safely preprocess the Java core classes (i.e., classes inside the `java.*` package) because they are always defined by the bootstrap classloader [101]. To enable complete RSI, our framework must also preprocess the supertypes of each Java core class; however, some of them fall outside `java.*` (for example, we observed the presence of supertypes inside `javax.*`, `sun.*`, or `com.sun.*`). Our framework ensures a correct preprocessing of such classes, even if their defining classloader is (in principle) not regulated by the JVM specification. Indeed, not-yet-loaded supertypes would be loaded during the resolution of the subtype using the same defining classloader of the subtype. Since the subtype will always be loaded by the bootstrap classloader (that cannot delegate the loading), the supertypes will always be loaded by the bootstrap classloader, too. Consequently, preprocessing supertypes of the Java core classes is compliant with the JVM specification, even if they are declared outside `java.*`.

4.7.3 Instrumentation State

The forced loading of supertypes employed in our approach ensures that a subtype will be instrumented after all its supertypes (except for the classes loaded during the primordial phase). Thus, custom *instrumentation state* can be passed from the instrumentation of a supertype to the instrumentation of a subtype. Our framework enables developers to attach custom instrumentation state to `DiSLClass` instances. Instrumentation state eases the propagation of instrumentation-time information to subtypes, such as properties of a class or bytecode-level statistics. Note that propagation of the instrumentation state cannot be guaranteed for classes loaded during the primordial phase, when subtypes may be instrumented before supertypes (see Section 4.4.3).

4.7.4 Checking Classloader IDs

Our approach allows instrumenting the subtypes of a class based not only on its fully qualified name, but also on its defining classloader, by setting up guard checks against classloaders. In this respect, our approach offers more flexibility than the AspectJ *subtype pattern* (the '+' operator) that does not allow specifying the defining classloader.

4.7.5 Limitations

Other analyses making use of the JVMTI heap tagging feature may interfere with our framework. In particular, such analyses could modify the order in which classloaders are tagged or could modify the tag of an already-tagged classloader (JVMTI supports only a single tag per object). Moreover, they could interfere with the reclamation of classloader namespaces, because the agent may receive

`ObjectFree` events even for objects that are not classloaders. Such interference can be avoided by resorting to a weak-key hash map[13] in native code to map classloaders to IDs, instead of relying on JVMTI heap tagging. This solution would guarantee a correct assignment of IDs to classloaders. Moreover, it would still allow namespace reclamation by periodically checking whether a weak reference to a classloader points to a freed object (via the JNI method `IsSameObject`), notifying the server in such a case. On the other hand, this solution is more complex to implement than resorting to heap tagging, and requires an additional channel between the agent and the DiSL server (to notify reclaimed classloaders).

While our framework enables the instrumentation of a method `m` based on complete RSI of the class where `m` is defined, it does not guarantee the availability of RSI for the arguments of `m`. Similarly, for call sites, RSI for the receiver and method arguments is not guaranteed available. This limitation arises from the fact that class loading is forced only for the supertypes of a class being loaded, but not for types that may occur as method arguments or as receivers of method calls. Enabling forced loading in the latter cases would likely cause the loading of more classes than those that would be normally loaded, and may lead to a huge memory footprint and slow startup due to excessive class loading.

4.8 Summary

In this chapter, we have presented a novel technique to reify the class hierarchy of an instrumented application within a separate instrumentation process. Our technique ensures that accurate and complete RSI is available for each class to be instrumented. RSI is guaranteed available even for classes inside the Java class library. Thanks to our approach, the instrumentation process can instrument only the classes that are in the focus of a type-specific analysis, inserting more efficient instrumentation code. Moreover, our technique exposes classloader namespaces to the instrumentation process, allowing the instrumentation framework to handle correctly homonym classes defined by different classloaders.

We have implemented our technique in an extension of the dynamic analysis framework DiSL. Evaluation results conducted on benchmarks from the DaCapo, Scalabench, and Spark Bench suites show that our approach significantly speeds up task-granularity profiling with tgp up to a factor of 6.24× wrt. resorting to runtime checks. Moreover, our work is fundamental to keep the overall overhead of tgp low (i.e., below 1.04× in most of the analyzed benchmarks) as well as to reduce the perturbations of the collected task-granularity profiles, resulting in

[13]Weak keys can be implemented by creating weak global references in JNI.

limited perturbation factors (i.e., not exceeding 1.03×–1.05×) in almost every observed workload. Our approach is also beneficial for other type-specific dynamic analyses on the JVM.

Overall, our technique enables complete, accurate, and efficient task-granularity profiling. In the next chapter, we use our profiler (using the DiSL Reflection API) for characterizing and optimizing task granularity in many task-parallel workloads running on a JVM in a shared-memory multicore.

Chapter 5

Task-Granularity Analysis and Optimization

In this chapter, we use our profiling methodology implemented in tgp to analyze and optimize the task granularity of various task-parallel applications running on the JVM. The chapter is organized as follows. Section 5.1 details our empirical task-granularity analysis, showing the presence of many fine- or coarse-grained tasks in several workloads and highlighting opportunities for optimizations. Section 5.2 presents our approach to optimize granularity in several tasks that impair performance, and quantifies the speedup obtained. Section 5.3 discusses the limitations of our approach. Finally, Section 5.4 summarizes our findings and the achievements presented in this chapter.

5.1 Analysis

In this section, we use tgp to characterize task granularity in multiple task-parallel benchmarks. We first outline the methodology used for conducting our analysis (Section 5.1.1); then, we detail our findings on fine-grained tasks (Section 5.1.2) and on coarse-grained tasks (Section 5.1.3).

5.1.1 Methodology

Our analysis targets task-parallel benchmarks of the latest DaCapo [15], Scala-Bench [123], and Spark Perf [25] suites. We analyze the same workloads used for the evaluation of the DiSL Reflection API (Section 4.6); a comprehensive list is presented in Tables 4.3, 4.4, and 4.5. The experimental setup and our evaluation methodology are the same presented in Section 4.6.1, including the version of the

Table 5.1. Benchmarks spawning fine-grained (FG) or coarse-grained (CG) tasks.

Benchmark DaCapo [15]	**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 [123]	**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$2`	16 184	FG
Benchmark Spark Perf [25]	**Problematic Task Class**	**# tasks spawned**	**CG/FC**
ChiSquare	`org.apache.spark.executor.Executor$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`	1852	FG

workloads and the frameworks considered, the benchmark input size, the number of warm-up iterations, the GC used, and the environment (i.e., the machine, its configuration, and the deployment setting) on which the analysis has been obtained. We use our extended DiSL framework with reification of complete supertype information enabled (presented in Chapter 4), and the implementation of guards in tgp resorts to the DiSL Reflection API.

Table 5.1 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 workloads not appearing in the table are either explicitly labeled as single-threaded in the benchmark documentation (factorie, kiama, scalap, scalariform and scalaxb) or they have neither coarse- nor fine-grained tasks (AlternatingLeastSquares, batik, ClassificationDecisionTree, fop, jython, luindex, scalac, scaladoc, scalatest, specs and xalan).

5.1.2 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.

Regarding DaCapo, eclipse runs a series of performance tests for the well-known Eclipse integrated development environment (IDE) [139]. As part of the workload, a `ReadManager`[1] creates many `Thread` instances to read the content (i.e., source code) of different 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 [131] 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 [52] online stock trading benchmark on the H2 in-memory database [44]. 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 [137] 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*[2] to execute computations in parallel. Actors are a form of lightweight parallel tasks [3]; 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 [140] 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

[1]We omit package declaration and outer classes to improve readability. Table 5.1 reports the fully qualified class names.

[2]In the *actor model* [50], *actors* are the atomic elements of a concurrent application. Actors wait continuously for messages incoming in their mailbox, processing one message at a time. Depending on the type of the received message, actors can execute different actions, such as carry on computations, change their state or their defined behavior, create other actors, or send messages to other actors.

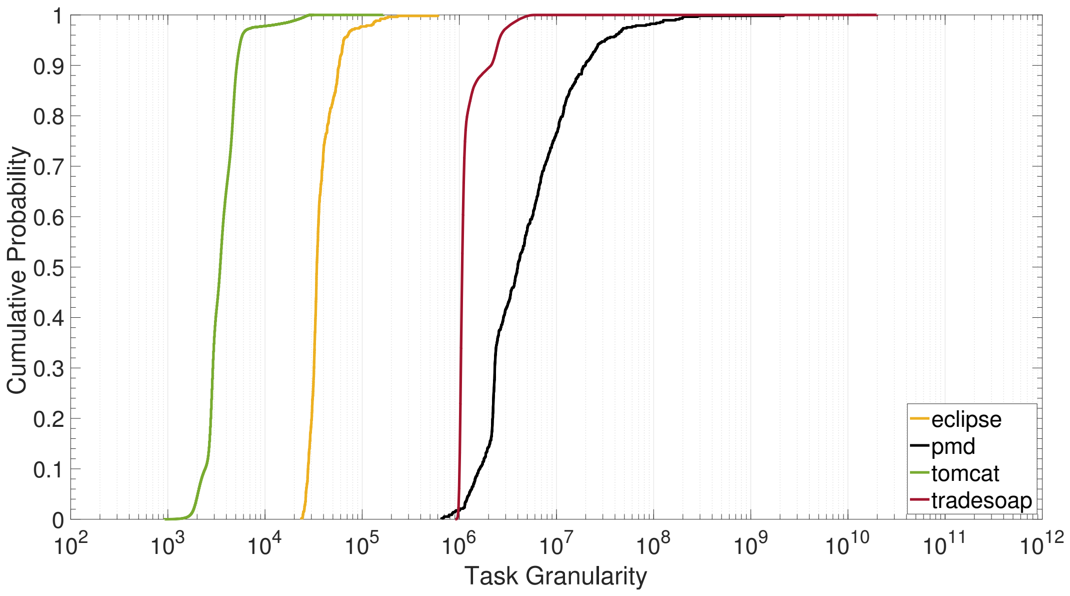
large number of tasks, i.e., 5 197 993 and 122 626, respectively, while tmt resorts to 16 184 instances of `ThreadRunner`.

Finally, StreamingWordCount (from Spark Perf) executes performance tests on the streaming library of Spark [134] 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 5.2.3), 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.

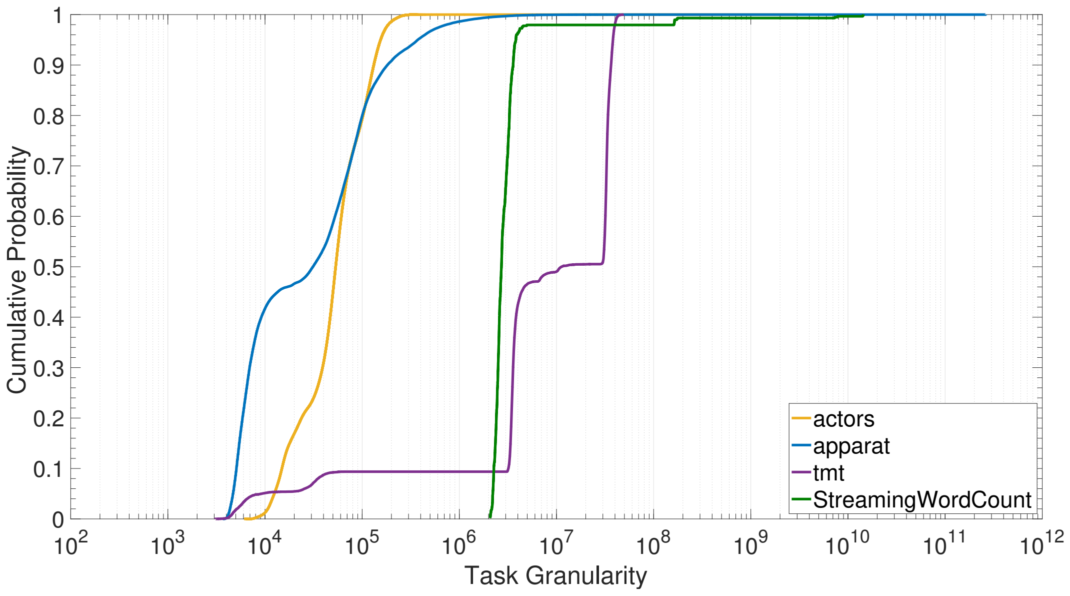
Figure 5.1 shows the Cumulative Distribution Function (CDF) of the granularity of tasks outlined above. Among the DaCapo benchmarks (Figure 5.1(a)), the lowest granularities can be observed in tomcat and eclipse, where ∼98% of the tasks spawned do not execute more than 10^4 cycles and 10^5 cycles, respectively. 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 5.1(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, sign that most actors executes 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.

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 5.2 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



(a) DaCapo [15].



(b) ScalaBench [123] and Spark Perf [25].

Figure 5.1. Cumulative Distribution Function (CDF) of the task granularity for benchmarks spawning fine-grained tasks. The figure shows only the granularities of fine-grained tasks.

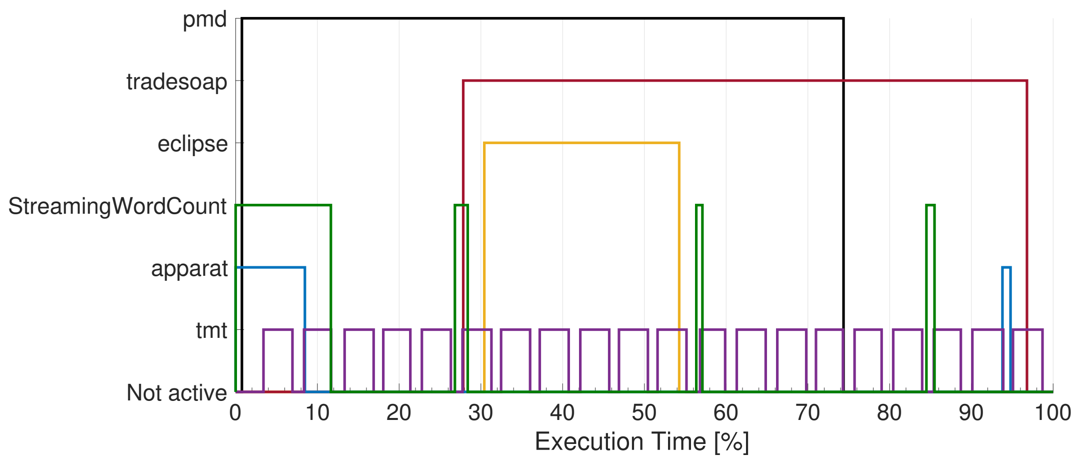
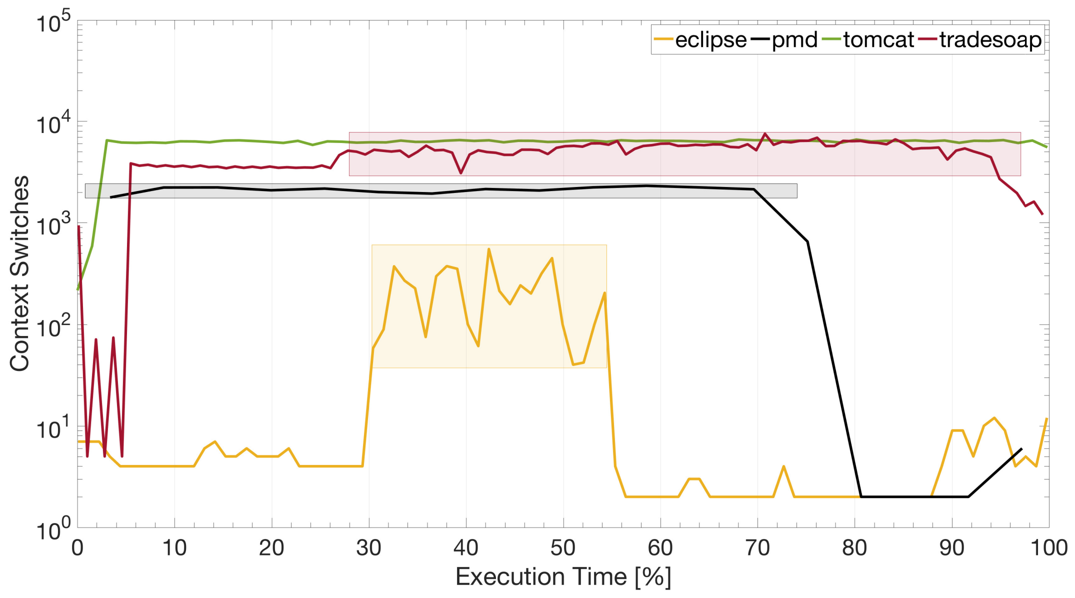


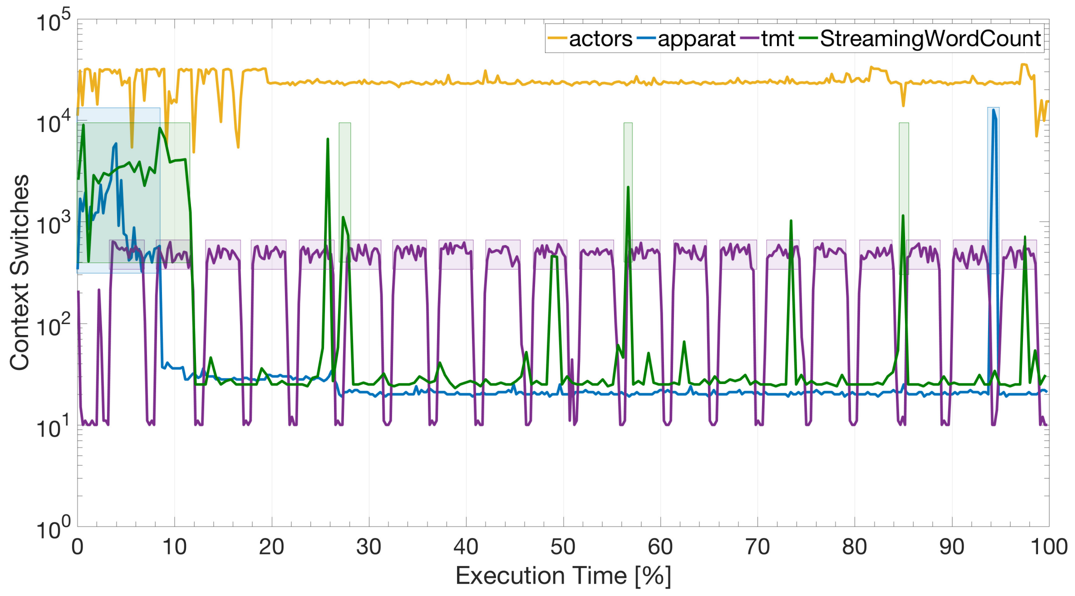
Figure 5.2. Portion of workloads where fine-grained tasks are executed. Tasks are always in execution in benchmarks not reported here (actor and tomcat).

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. Similarly, apparat spawns most of the tasks (85%) in the first 8% of the workload, while others are created in a short spike towards 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.

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 5.3. 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 5.2) with colored regions superimposed to the trends shown in the figure. A region



(a) DaCapo [15]. Data on eclipse, tomcat, and tradesoap has been downsampled to 20%.



(b) ScalaBench [123] and Spark Perf [25]. Data on actors and apparat has been downsampled to 10% and 40%, respectively.

Figure 5.3. Context switches over execution time in benchmarks spawning fine-grained tasks. Measurements sampled during GC have been removed. The colored regions superimposed to the lines represent the time intervals where fine-grained tasks are executed (shown in Figure 5.2). Tasks are executed for the whole duration of the workload in benchmarks with no colored regions.

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),[3] 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 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.

In summary, 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 workloads, it would be sufficient to assemble data in groups

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

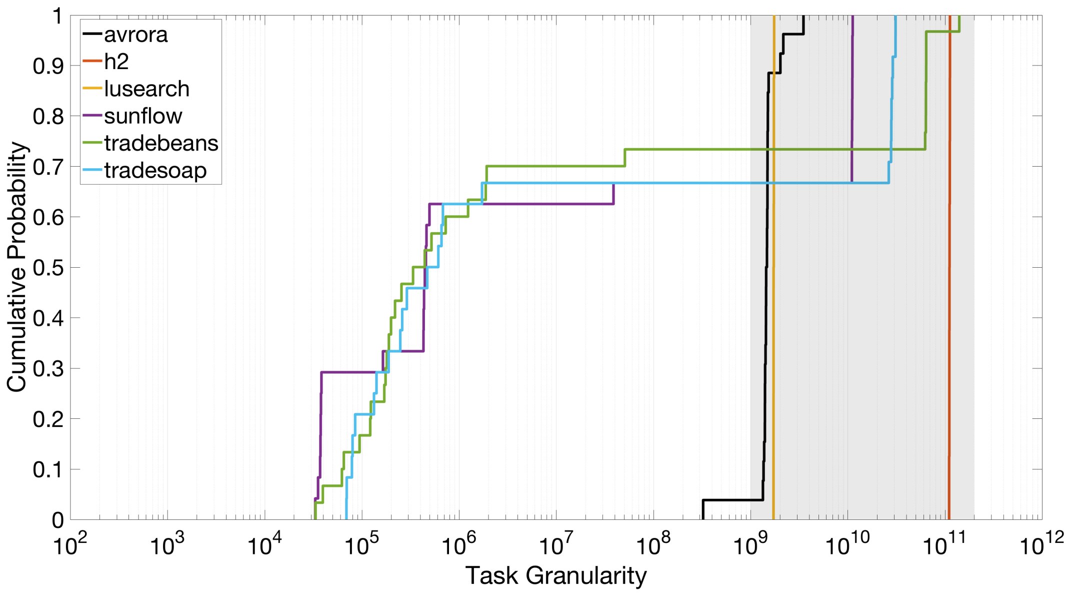
(e.g., in case of pmd and tmt), modifying each task to process a group rather than a single element (in tmt, this optimization could be applied to tasks within each batch), or to increase the size of the dataset processed by a single task (e.g., in case of eclipse and StreamingWordCount). Section 5.2 confirms the benefit of such an optimization on pmd and StreamingWordCount. On the other hand, while also fine-grained tasks in tradesoap interfere with each other, decreasing such interference may not significantly decrease the contention experienced by the benchmark, which is caused only partially by fine-grained tasks.

5.1.3 Coarse-Grained Tasks

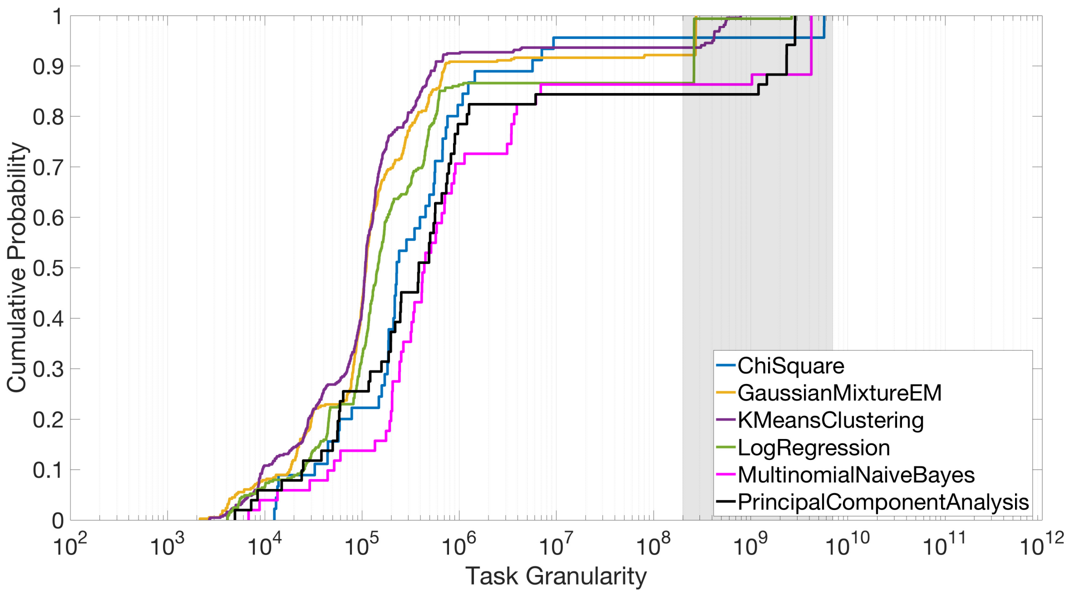
We observe the presence of coarse-grained tasks in six benchmarks of the DaCapo suite (avrora, h2, lusearch, sunflow, tradebeans, and tradesoap) and six applications from Spark Perf (ChiSquare, GaussianMixtureEM, KMeansClustering, LogRegression, MultinomialNaiveBayes, and PrincipalComponentAnalysis). Figure 5.4 shows the CDF of their task granularity. To better see the presence of coarse-grained tasks in tradesoap, we remove fine-grained tasks from its CDF.

In all DaCapo benchmarks (Figure 5.4(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 5.4(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).

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 26 programs on a grid of AVR microcontrollers. The benchmark runs each simulation in parallel in a separate `SimulatorThread`. h2 executes the TPC-C [142] on-line 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 [132] 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



(a) DaCapo [15]. The CDF of tradesoap does not include fine-grained tasks.



(b) Spark Perf [25].

Figure 5.4. Cumulative Distribution Function (CDF) of the task granularity for benchmarks spawning coarse-grained tasks. The shaded region marks the presence of coarse-grained tasks.

Table 5.2. Average CPU utilization for benchmarks spawning coarse-grained tasks, with 95% confidence intervals. 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 GC are not considered.

Benchmark DaCapo [15]	**CPU Util. [%] (± 95% conf.)**	**Benchmark Spark Perf [25]**	**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

is the same of tradesoap (explained in Section 5.1.2), 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 [130]). All coarse-grained tasks in these benchmarks are of class `TaskRunner`. As discussed in Section 5.1.2, 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 5.1. Coarse-grained tasks are used during the whole execution of each Spark Perf benchmark.

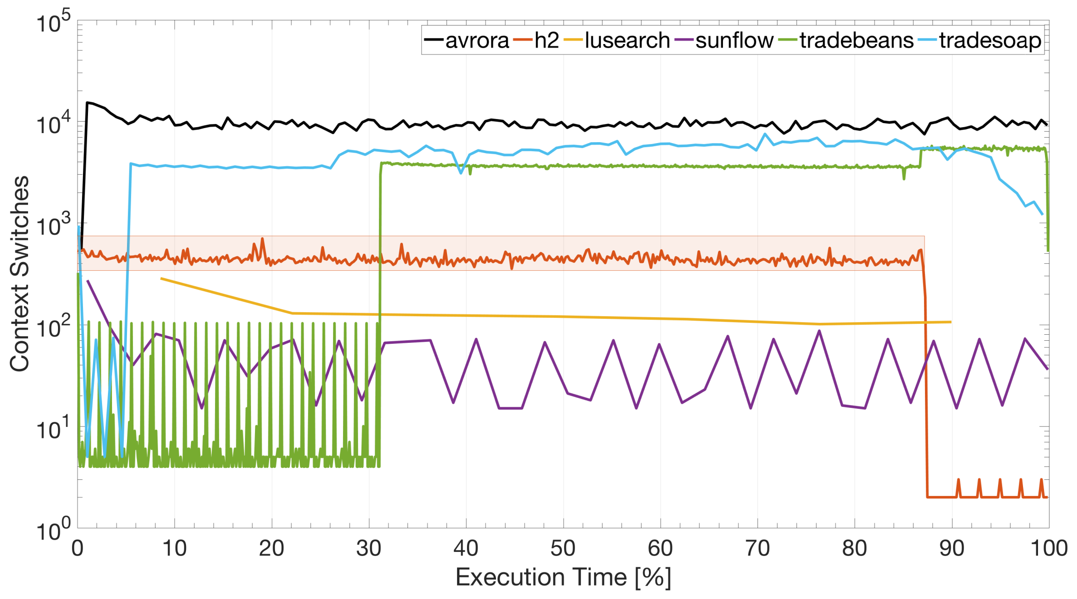
Table 5.2 depicts the average CPU utilization of each benchmark. In most of them, the CPU is far from being fully utilized, especially in avrora, GaussianMixtureEM, and h2 where the average CPU utilization is 9%, 18%, and 19%, respectively. The highest average CPU utilization occurs in sunflow (96%), followed by MultinomialNaiveBayes (76%). All other benchmarks 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. With the goal of better utilizing the available CPU cores, such coarse-grained tasks could be split into multiple tasks with smaller granularity. However, the need for synchronization among tasks may severely limit the benefits of 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 5.5 reports our results. Regarding the DaCapo benchmarks (Figure 5.5(a)), the workloads 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.

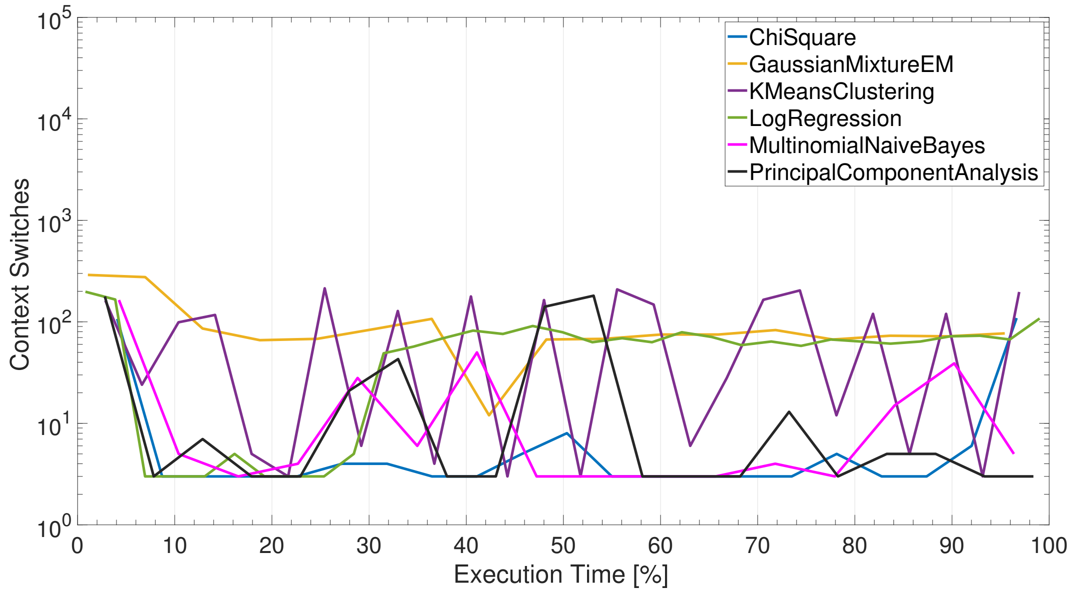
Regarding tradebeans and tradesoap, these benchmarks are little subjected to context switches in the first part of the workload (i.e., the first ∼31% of execution in tradebeans, and the first ∼5% in tradesoap), but experience a noticeable increase in the trend in the second part, resulting in an average 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 synchronization 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 workloads (Figure 5.5(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.

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 performance. The optimization is eased by the fact



(a) DaCapo [15]. Data on tradesoap has been downsampled to 20%. The colored region superimposed to the trend of h2 represents the time interval where coarse-grained tasks are executed.



(b) Spark Perf [25].

Figure 5.5. Context switches over execution time in benchmarks spawning coarse-grained tasks. Measurements sampled during GC have been removed.

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 apply such an optimization in Section 5.2. 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 workload with a higher number of tasks.

5.2 Optimization

Here, we demonstrate the benefits of optimizing task granularity. In the first part of the section, we show how tgp eases the optimization of task granularity, focusing on one benchmark where several fine-grained tasks interfere with each other (pmd; Section 5.2.1) and on another benchmark where the presence of coarse-grained tasks limits the utilization of the available computing resources (lusearch; Section 5.2.2). Then, we describe our approach to optimize task granularity in Spark Perf (Section 5.2.3). Finally, we discuss the speedup enabled by our optimizations (Section 5.2.4).

5.2.1 pmd

As discussed in Section 5.1.2, pmd determines errors and bad coding practices on a set of 570 source-code files, each processed by a `PmdRunnable`. Each `PmdRunnable` (despite its name, the class implements `Callable`) receives a single source-code file to process as an argument to its constructor, producing a report containing the problems found as object returned from its `call` method. 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. Figure 5.6(a) reports the creation calling context of `PmdRunnable`, which shows that such tasks are created in the method `processFiles` of the class `PMD` (line 14). The creation calling context

```
Harness.main
 java.lang.reflect.Method.invoke
  sun.reflect.DelegatingMethodAccessorImpl.invoke
   sun.reflect.NativeMethodAccessorImpl.invoke
    org.dacapo.harness.TestHarness.main
     org.dacapo.harness.TestHarness.runBenchmark
      org.dacapo.harness.Benchmark.run
       org.dacapo.harness.Pmd.iterate
        java.lang.reflect.Method.invoke
         sun.reflect.DelegatingMethodAccessorImpl.invoke
          sun.reflect.GeneratedMethodAccessor11.invoke
           net.sourceforge.pmd.PMD.main
            net.sourceforge.pmd.PMD.doPMD
             net.sourceforge.pmd.PMD.processFiles
              net.sourceforge.pmd.PMD$PmdRunnable.<init>
```

(a) `PmdRunnable` (in pmd).

```
Harness.main
 java.lang.reflect.Method.invoke
  sun.reflect.DelegatingMethodAccessorImpl.invoke
   sun.reflect.NativeMethodAccessorImpl.invoke
    org.dacapo.harness.TestHarness.main
     org.dacapo.harness.TestHarness.runBenchmark
      org.dacapo.harness.Benchmark.run
       org.dacapo.harness.Lusearch.iterate
        java.lang.reflect.Method.invoke
         sun.reflect.DelegatingMethodAccessorImpl.invoke
          sun.reflect.GeneratedMethodAccessor1.invoke
           org.dacapo.lusearch.Search.main
            org.dacapo.lusearch.Search$QueryThread.<init>
```

(b) `QueryThread` (in lusearch).

Figure 5.6. Creation calling contexts. `<init>` represents a constructor. We do not report input arguments for clarity.

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 contexts of all PmdRunnable objects are 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, involve the modification of only 32 lines of code, and do not require any specific knowledge of the benchmark.

5.2.2 lusearch

As discussed in Section 5.1.3, lusearch performs a set of queries over a corpus of data to locate keywords. The benchmark carries out a total of 128 queries. Such queries are divided 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 (shown in Figure 5.6(b)). 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. Both thread creation and start occur in method main of class Search (line 12).

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 lighter-weight 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 our 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 involve only 13 lines of code.

5.2.3 Spark Perf Benchmarks

Before describing our optimizations to Spark Perf workloads, we introduce the scheme used by Spark to divide computations into tasks, which is followed by all analyzed Spark Perf benchmaks [129; 119; 120].

In Spark, each application performs computations on *resilient distributed datasets* (RDDs) [149]. An RDD is an immutable collection of elements that can be processed in parallel. Regardless of the source of the input set (e.g., local filesystem, distributed file system, databases, etc.), data must be inserted into an RDD before being processed. Internally, RDDs are divided into *partitions*, representing the portions of data that can be processed in parallel. A partition is composed of several *records*, each representing a single data unit.

A Spark application can be seen as a sequence of operations over RDDs. Operations can either be *transformations* (which create a new RDD from an existing one) or *actions* (which return a value after running a computation on the RDD). Transformations create a new RDD by transforming partitions of the input RDD into modified partitions of a new output RDD. If a new partition (in the output RDD) can be created from records residing in a single partition of the input RDD, the transformation is classified as *narrow*. In contrast, transformations that need to access records scattered in different partitions of the input RDD to create a single partition of the output RDD are known as *wide*.

Transformations in Spark are *lazy*, i.e., they are executed only when invoking an action. At this time, Spark creates a new *job*, i.e., a computation whose goal is to calculate the result requested by the action. To compute the result, Spark analyzes the transformations on RDDs required to obtain such result, and formulates a workflow of computations consisting of a variable number of *stages*, each composed of a collection of *tasks* executing the same code (expressing transformations) on different partitions of the RDDs. Additional stages are introduced by Spark when there is a need for repartitioning an RDD. If a job is composed only of narrow transformations (which do not require repartitioning), the job will be composed of a single stage. On the other hand, wide transformations require a repartition of the input RDD, such that all data required to create a new partition in the output RDD is contained in a single partition of the input RDD. This repartitioning is called *shuffle*. Each wide transformation introduces a new shuffle; thus, a new stage in the job.

The number of tasks in a stage is the same as the number of partitions in the last RDD in the stage. In general, the number of partitions in an RDD is the same as the number of partitions in the RDD on which it depends (i.e., the RDD passed as input to the transformation that produced the RDD).[4] The number of partitions in the RDDs containing the initial data sets (i.e., those created for containing input data at the beginning of the application) depends on the input format of the data and the original data source, unless a given partition number is specified by the user upon RDD creation.

Overall, the final number of tasks into which a Spark application is divided depends on several variables, such as the number of actions invoked by the application, the number of wide transformations (hence, stages) a job requires, and the number of partitions inside each RDD. To optimize task granularity in all Spark Perf applications (apart from StreamingWordCount), we modify the number of partitions inside the initial RDD, created at the beginning of the application. In such benchmarks the initial RDD is created from a textual file stored in the local filesystem, via the `textFile` method (defined in `SparkContext` [135]), 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. Since a different number of partitions result in a different number of tasks spawned, our approach enables us to decrease task granularity by increasing the number of partitions of the RDD (recall that all Spark Perf benchmarks apart from StreamingWordCount suffer from coarse-grained tasks). While other means of altering the number of tasks created are available in Spark (such as repartitioning the RDDs after a transformation, via `repartition`), they may introduce additional stages that are not present in the original application, and may increase the execution time of the application. Our approach enables us to control the number of tasks spawned

[4]Transformations such as `coalesce`, `union`, or `cartesian` cause exceptions to this behavior, as reported in the RDD API [129].

as well as their granularity with minimal modifications to the target applications, since it requires the modification of a single line of code in each benchmark (to explicitly set the partition number of the initial RDD).

Differently from all other Spark Perf benchmarks considered in this dissertation, in StreamingWordCount the input data is received in the form of a continuous stream of textual data from a socket, leveraging the Spark Streaming API [134]. Before being processed, data in the stream is assembled in RDDs as follows. Data received in a given temporal interval t_{bl} is grouped in a *block*. In turn, all blocks received in a temporal interval t_{ba} (which must be $\geq t_{bl}$) are assembled into a *batch*. Spark creates a new RDD for each received batch, and converts each block inside a batch into a partition of the RDD. Transformations and actions specified by the application are applied to all RDDs generated, following the scheme for converting operations to tasks outlined above. StreamingWordCount creates a new batch (hence, a new RDD) every 100ms, resulting in a high number of fine-grained tasks to process data inside each partition of the RDDs.

To increase task granularity in StreamingWordCount, we modify the number of partitions of the RDDs created after each shuffle. Each wide transformation (such as `reduceByKey` or `join`) 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. All wide transformations in StreamingWordCount do not specify any explicit partition number. Moreover, the application does not set `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 [133]). We modify the number of partitions after each shuffle 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 workloads 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 (i.e., a single line of code) on the target application to be implemented.

5.2.4 Evaluation

We execute the modified benchmarks on different environments. Apart from the machine used for task granularity analysis (Section 5.1), where applications are set up to use 8 cores on a single NUMA node as described in Section 4.6, 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 additional machine, equipped with an Intel i7-4710MQ (2.5 GHz) processor with 4 physical cores, 8 GB 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 4.6.1.

In each environment, we run the modified workloads in different settings, each with a different number of tasks spawned. In pmd and lusearch, we start from the same number of tasks used in the original workloads (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 for each iteration. 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).

In all Spark Perf benchmarks (excluding StreamingWordCount), 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 runs, 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 presence of wide transformations or multiple actions in the application can cause a much larger number of tasks spawned, as explained in the previous section. 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).[5]

The original StreamingWordCount repartitions RDDs at each shuffle in as many partitions as available cores. We increase task granularity as follows. We start by

[5]We do not apply the proposed optimization to MultinomialNaiveBayes because the benchmark well utilizes CPU (as shown in Section 5.1.3).

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.

Tables 5.3, 5.4 and 5.5 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 wrt. the original (unmodified) workload. Speedup is presented in terms of *speedup factors*, defined as the execution time of the original workload divided by the execution time of the modified workload. The values shown are the average over 20 runs. 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).

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 synchronization overheads between tasks increases, jeopardizing 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. Increasing the number of available cores makes the benefit of optimizing task granularity more evident, 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-

Table 5.3. Speedup (including 95% confidence intervals) resulting from task granularity optimization on a 8-core machine. The best result for each benchmark is highlighted.

pmd (`PmdRunnable`)		**lusearch** (`QueryThread`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
570	Baseline	8	Baseline
285	1.2577 ± 0.0085	16	1.0500 ± 0.0097
143	1.3548 ± 0.0136	32	1.0552 ± 0.0172
72	1.3597 ± 0.0142	64	1.0613 ± 0.0174
36	1.3644 ± 0.0089	128	1.0438 ± 0.0117
18	1.3736 ± 0.0117		
9	1.3799 ± 0.0094		
ChiSquare (`TaskRunner`)		**GaussianMixtureEM** (`TaskRunner`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
5	Baseline	35	Baseline
9	1.9644 ± 0.0047	69	1.7934 ± 0.0048
17	3.7961 ± 0.0694	137	2.2173 ± 0.1083
33	3.2505 ± 0.0938	273	2.1599 ± 0.0229
65	2.9895 ± 0.0954	545	2.1397 ± 0.0340
129	2.8045 ± 0.0350	1089	1.9778 ± 0.0273
257	2.2582 ± 0.0185	2177	1.6650 ± 0.0199
513	1.5004 ± 0.0066		
KMeansClustering (`TaskRunner`)		**LogRegression** (`TaskRunner`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
32	Baseline	45	Baseline
64	1.8196 ± 0.0040	89	1.8236 ± 0.0046
128	2.7234 ± 0.0844	220	2.4127 ± 0.0928
256	2.4808 ± 0.0536	440	2.1737 ± 0.0279
512	2.0943 ± 0.0226	814	2.0375 ± 0.0168
1024	1.7579 ± 0.0171	1584	1.7709 ± 0.0147
2048	1.3598 ± 0.0069	3037	1.4438 ± 0.0108
PrincipalComponentAnalysis (`TaskRunner`)		**StreamingWordCount** (`TaskRunner`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
9	Baseline	1852	Baseline
22	2.0475 ± 0.0305	976	1.2176 ± 0.0113
41	2.0101 ± 0.0476	543	1.0527 ± 0.0098
75	1.9527 ± 0.0384		
145	1.9472 ± 0.0465		
277	1.8387 ± 0.0502		
545	1.7668 ± 0.0136		
1069	1.6269 ± 0.0073		
2113	1.2963 ± 0.0135		

Table 5.4. Speedup (including 95% confidence intervals) resulting from task granularity optimization on a 16-core machine. The best result for each benchmark is highlighted.

pmd (`PmdRunnable`)		**lusearch** (`QueryThread`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
570	Baseline	16	Baseline
285	1.1623 ± 0.0286	32	1.3326 ± 0.0484
143	1.2340 ± 0.0372	64	1.2805 ± 0.0438
72	1.2591 ± 0.0309	128	1.2682 ± 0.0374
36	1.2777 ± 0.0325		
18	1.2955 ± 0.0386		

ChiSquare (`TaskRunner`)		**GaussianMixtureEM** (`TaskRunner`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
5	Baseline	35	Baseline
9	1.8014 ± 0.0150	69	1.7791 ± 0.0158
17	3.8854 ± 0.0462	137	2.8545 ± 0.0336
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	1089	2.8445 ± 0.0410
257	3.1957 ± 0.0589	2177	2.5317 ± 0.0253
513	2.3752 ± 0.0380	4353	1.7381 ± 0.0147
1025	1.5354 ± 0.0216		

KMeansClustering (`TaskRunner`)		**LogRegression** (`TaskRunner`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
32	Baseline	45	Baseline
64	1.7906 ± 0.0115	89	1.8107 ± 0.0106
128	2.8504 ± 0.0374	220	2.7036 ± 0.0233
256	3.3416 ± 0.0439	440	3.4413 ± 0.0833
512	3.1176 ± 0.0422	814	2.9703 ± 0.0364
1024	2.9112 ± 0.0276	1584	2.7148 ± 0.0223
2048	2.3676 ± 0.0161	3037	2.3428 ± 0.0174
4096	1.7848 ± 0.0089	5984	1.7437 ± 0.0188

PrincipalComponentAnalysis (`TaskRunner`)		**StreamingWordCount** (`TaskRunner`)	
# tasks	**Speedup (± 95% conf.)**	**# tasks**	**Speedup (± 95% conf.)**
9	Baseline	3661	Baseline
22	2.1849 ± 0.0109	1833	1.0102 ± 0.0050
41	3.5418 ± 0.1045	986	1.0274 ± 0.0060
75	3.3867 ± 0.0995	576	1.0703 ± 0.0057
145	3.2746 ± 0.0857	380	1.1345 ± 0.0063
277	3.2475 ± 0.0874		
545	3.2371 ± 0.0246		
1069	2.8603 ± 0.0198		
2113	2.2715 ± 0.0168		
4185	1.7741 ± 0.0122		

Table 5.5. Speedup (including 95% confidence intervals) resulting from task granularity optimization on a 4-core machine. The best result for each benchmark is highlighted.

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

granularity optimization leads to improved application performance also when a lower number of cores is available, as indicated by Table 5.5. 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.

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.

5.3 Discussion

In this section, we discuss the limitations of our approach.

5.3.1 Platform-dependent Results

The workloads analyzed in this chapter may exhibit different behavior on 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 where a higher number of cores is available, more tasks will be spawned, which may result in a reduced task granularity. On the contrary, task granularity may increase on machines with fewer computing 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 task-granularity optimizations, as indicated by the higher speedups obtained on the 16-core machine (Table 5.4). On the other hand, fewer CPU cores may limit the benefits of our approach. While we observe such an effect in our evaluation (Table 5.5), the presence of significant speedups in all settings remarks that all the considered workloads suffer from a suboptimal task granularity in all the environments used for the evaluation.

5.3.2 Optimization of DaCapo and ScalaBench

Our task-granularity characterization has revealed the presence of optimizable coarse- or fine-grained tasks in multiple benchmarks of the DaCapo and ScalaBench suites. We optimized task granularity in all benchmarks for which our characterization pinpointed significant optimization opportunities, provided that benchmark source code is publicly available and can correctly be built. Unfortunately, the source code of ScalaBench applications is either not available or has missing dependencies that impede a correct build, preventing us from applying optimizations to actors, apparat and tmt. Moreover, despite the release of a recent new version of DaCapo (January 2018), which is supposed to ensure that the source code of all benchmarks compiles correctly,[6] several benchmarks still cannot be built at time of writing. For this reason, we could not validate our findings on optimizable tasks in tomcat, tradebeans, and tradesoap. While we reported the issue to the DaCapo developers, no official fix has been proposed or released to date.

5.3.3 CPU Utilization

Our profiler collects CPU utilization through periodic sampling (every ∼150ms). Hence, we may not detect short peaks of high CPU utilization. This limitation is intrinsic to the metric collected, as CPU utilization is an instantaneous property of the system. Nonetheless, we profiled each benchmark multiple times, without experiencing any noticeable difference in terms of CPU utilization among different benchmark runs.

5.4 Summary

In this chapter, we have characterized the task granularity of several task-parallel applications running on the JVM. We applied our task-granularity profiler tgp (Chapter 3) making use of our new technique to reify complete reflective supertype information at instrumentation-time (Chapter 4) to workloads from the well-known DaCapo and ScalaBench suites, as well as to benchmarks for the widely-used Apache Spark big-data analytics framework, contained in a suite of performance tests (Spark Perf) maintained by the Spark developers.

The accurate profiles collected by tgp allow us to identify, in many workloads, the presence of coarse-grained tasks that result in idle CPU cores, as well as fine-

[6]As reported at `http://dacapobench.org`.

grained tasks that incur significant contention. We locate coarse-grained tasks that can be split into smaller ones to better utilize CPU, as well as fine-grained tasks that can be merged to reduce parallelization overheads.

We demonstrate that the profiles are actionable by optimizing task granularity in pmd and lusearch, modifying the classes and methods indicated by tgp through calling-context profiling. In both cases, only a few lines of code need to be changed. In addition, we optimize the granularity of tasks causing performance drawbacks in Spark Perf, modifying only a single line of code in all applications. Our approach enables significant speedups on multiple diverse environments, up to a factor of 5.90×. Overall, our evaluation results demonstrate the importance of analyzing and optimizing task granularity on the JVM using a complete, accurate, and efficient profiling methodology that enables the collection of actionable profiles.

Chapter 6

Conclusion

Nowadays, dividing work into parallel tasks is a fundamental strategy to exploit the available computing cores and speed up application execution. Our work focuses on the JVM, where task-parallel applications are widespread. Task granularity is a fundamental attribute of such applications and may significantly affect their performance. Hence, understanding task granularity is very important to assess and improve the performance of task-parallel applications.

This dissertation 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. We present a novel methodology and tool to accurately profile the granularity of every executed task, which help developers locate performance problems and guide them towards useful optimizations. We also introduce a technique that significantly lowers the overhead of task-granularity profiling, enabling the collection of less perturbed profiles with low overhead. We analyze task granularity in numerous task-parallel workloads, revealing inefficiencies related to fine-grained and coarse-grained tasks in many applications. We optimize the granularity of tasks causing performance drawbacks by modifying the classes and methods indicated by our tool. Our approach enables significant speedups in numerous workloads suffering from coarse- and fine-grained tasks in different environments. Overall, our work pinpoints the importance of analyzing and optimizing task granularity on the JVM.

6.1 Summary of Contributions

Below, we summarize the contributions of this dissertation.

Task-Granularity Profiling We present a new methodology to profile the granularity of all tasks spawned in task-parallel applications running on a JVM. Our approach enables an accurate collection of task-granularity profiles even for tasks showing complex patterns, such as nested tasks, tasks executed multiple times, and tasks with recursive operations, which may cause inaccurate results if not handled with care. Our technique resorts to vertical profiling to collect carefully selected metrics from the whole system stack, aligning them via offline analysis. The collected metrics allow one to analyze task granularity and its impact on application performance. Moreover, our approach collects the calling context upon the creation, submission, and execution of tasks causing performance drawbacks, yielding 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. To the best of our knowledge, tgp is the first task-granularity profiler for the JVM. We use efficient data structures to reduce profiling overhead, and apply novel and efficient instrumentation and profiling techniques to collect complete and accurate task-granularity profiles.

Reification of Complete Supertype Information We propose a novel approach to optimize type-specific analyses on the JVM. When applied to task-granularity profiling, our approach reduces the overhead of task detection and the perturbation of the collected task-granularity profiles. Our technique reifies the class hierarchy of an instrumented application within a separate instrumentation process, ensuring that accurate and complete reflective supertype information is available for each class to be instrumented, including those in the Java class library. Our approach allows the instrumentation process to instrument only classes falling in the scope of a type-specific analysis, inserting more efficient instrumentation code that avoids the execution of many runtime checks, which typically introduce significant runtime overhead and increase the perturbation of the collected metrics. Moreover, our technique exposes classloader namespaces to the instrumentation process, allowing the instrumentation framework to correctly handle homonym classes defined by different classloaders.

We implement our technique in an extension of the dynamic analysis framework DiSL, and design a new API to allow accessing reflective supertype information and classloader namespaces from an instrumentation process. We use our technique to optimize task-granularity profiling on benchmarks from the DaCapo, ScalaBench, and Spark Perf suites. Evaluation results demonstrate that our approach can significantly speed up task-granularity profiling with tgp, up to a factor of 6.24× wrt. resorting to runtime checks. Moreover, our technique is fundamental to keep the overall profiling overhead low (i.e., below 1.04×) and so to reduce perturbations of the collected task-granularity profiles (i.e., perturbation factors not exceeding 1.03×–1.05×) in most of the analyzed applications.

Task-Granularity Analysis and Optimization We apply our optimized profiling technique to workloads from the DaCapo, ScalaBench, and Spark Perf suites, characterizing their task granularity thanks to little perturbed metrics obtained with tgp. To the best of our knowledge, we provide the first analysis on task granularity for task-parallel applications running on the JVM, revealing previously unknown performance drawbacks of the analyzed applications. In many workloads, we identify the presence of a small number of coarse-grained tasks that underutilize CPU and result in idle core, and of many fine-grained tasks suffering from noticeable contention and leading to parallelization overheads. We identify coarse-grained tasks that can be split into several smaller ones to better leverage idle CPU, and fine-grained tasks that can be merged to reduce synchronization and contention.

We use the actionable profiles collected by tgp to optimize task granularity in numerous workloads. We modify the classes and methods indicated by our tool through calling-context profiling, 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 workloads suffering from coarse- and fine-grained tasks (up to a factor of 5.90×) in different environments. Overall, our evaluation results demonstrate the importance of analyzing and optimizing task granularity on the JVM using a complete, accurate, and efficient profiling methodology that enables the collection of actionable profiles.

6.2 Future Work

The work presented in this dissertation opens several future research directions. Below, we give an overview of new major research opportunities enabled by our work.

Large-Scale Task-Granularity Analysis and Optimization The optimization results presented in this dissertation indicate that our work can benefit various task-parallel applications in different fields, including increasingly popular ones where high performance is crucial, such as machine learning or big-data processing. A possible future direction could be to extend the results presented here by conducting a large-scale characterization of task granularity, performing related optimizations on a broad range of applications. To this end, one could integrate tgp into existing frameworks for large-scale dynamic analysis (such as AutoBench [151]), to automatically collect task-granularity profiles from many publicly available open-source workloads. Moreover, task-granularity analysis could be performed on multiple environments, including Cloud-based ones. In this case, identifying metrics suitable to characterize task granularity in the Cloud—where the applicability of platform-specific metrics (e.g., those collected with HPCs) may be limited—would be crucial. From the collected data, one could extract common anti-patterns leading to suboptimal task granularity in different workloads, deriving guidelines for developers to prevent such anti-patterns in future releases and in new software.

Automatic Task-Granularity Analysis and Optimization Coaching The characterization and optimization of task granularity presented in this dissertation is based on manual in-depth analysis of the collected profiles. Future research could automate this process, reducing the effort for analyzing numerous task-granularity profiles, especially when applying tgp to a large scale of workloads (as outlined in the previous paragraph). Thanks to automated task-granularity analysis, one could derive patterns and anti-patterns to classify task granularity as good or bad 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 (i.e., good or bad) in new open-source workloads. Thanks to this approach, one could notify developers about the discovered optimization opportunities, coaching them to achieve the optimizations by means of the actionable profiles collected by tgp, suggesting them which code portions should be inspected and modified to optimize task granularity.

New Task-Parallel Benchmark Suite for the JVM Our work demonstrates that task granularity is an important performance attribute of task-parallel applications, representing work carried out by each parallel task. Benchmark suites are typically used in academia and industry to evaluate the performance of new tools or techniques. To conduct a comprehensive evaluation, it is important that applications inside a benchmark suite exhibit enough diversity along several characteristics. Unfortunately, existing suites including parallel applications running on the JVM have not been designed to represent workloads showing enough diversity in terms of tasks spawned and their granularity. To fill this gap, one could assemble a new benchmark suite of task-parallel workloads that exhibit a high diversity of task granularities, selecting benchmark candidates through task-granularity profiling and analysis with tgp. Moreover, one could compare the task granularity of the selected benchmarks with the one of established benchmark suites, including DaCapo, ScalaBench, and Spark Perf, pinpointing the higher diversity of the task granularity in the chosen workloads.

Bibliography

[1] U. A. Acar, A. Charguéraud, and M. Rainey. Oracle Scheduling: Controlling Granularity in Implicitly Parallel Languages. In *OOPSLA*, pages 499–518, 2011.

[2] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent. HPCTOOLKIT: Tools for Performance Analysis of Optimized Parallel Programs. *Concurr. Comput.: Pract. Exper.*, 22(6):685–701, 2010.

[3] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[4] D. Akhmetova, G. Kestor, R. Gioiosa, S. Markidis, and E. Laure. On the Application Task Granularity and the Interplay with the Scheduling Overhead in Many-Core Shared Memory Systems. In *CLUSTER*, pages 428–437, 2015.

[5] G. Ammons, T. Ball, and J. R. Larus. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI*, pages 85–96, 1997.

[6] D. Ansaloni, W. Binder, A. Heydarnoori, and L. Y. Chen. Deferred Methods: Accelerating Dynamic Program Analysis on Multicores. In *CGO*, pages 242–251, 2012.

[7] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel Dynamic Analysis on Multicores with Aspect-Oriented Programming. In *AOSD*, pages 1–12, 2010.

[8] P. Avgustinov, A. S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. abc: An Extensible AspectJ Compiler. In *AOSD*, pages 87–98, 2005.

[9] S. Banerjee, E. Bozorgzadeh, and N. Dutt. PARLGRAN: parallelism granularity selection for scheduling task chains on dynamically reconfigurable architectures. In *ASPDAC*, pages 1–6, 2006.

[10] Barcelona Supercomputing Center. OmpSs Specification. `https://pm.bsc.es/ompss-docs/spec/`, 2018.

[11] J. Bi, X. Liao, Y. Zhang, C. Ye, H. Jin, and L. T. Yang. An Adaptive Task Granularity Based Scheduling for Task-centric Parallelism. In *HPCC*, pages 165–172, 2014.

[12] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[13] W. Binder, J. Hulaas, and P. Moret. Advanced Java Bytecode Instrumentation. In *PPPJ*, pages 135–144, 2007.

[14] W. Binder, P. Moret, É. Tanter, and D. Ansaloni. Polymorphic Bytecode Instrumentation. *Software: Practice and Experience*, 46(10):1351–1380, 2015.

[15] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*, pages 169–190, 2006.

[16] E. Bodden, L. Hendren, P. Lam, O. Lhoták, and N. A. Naeem. Collaborative Runtime Verification with Tracematches. In *RV*, pages 22–37, 2007.

[17] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-Java: The New Adventures of Old X10. In *PPPJ*, pages 51–61, 2011.

[18] F. Chen, T. F. Serbanuta, and G. Rosu. jPredictor: A Predictive Runtime Analysis Tool for Java. In *ICSE*, pages 221–230, 2008.

[19] K. Y. Chen, J. M. Chang, and T. W. Hou. Multithreading in Java: Performance and Scalability on Multicore Systems. *IEEE Transactions on Computers*, 60(11):1521–1534, 2011.

[20] W. Chen, R. F. D. Silva, E. Deelman, and R. Sakellariou. Balanced Task Clustering in Scientific Workflows. In *eScience*, pages 188–195, 2013.

[21] S. Chiba and M. Nishizawa. An Easy-to-use Toolkit for Efficient Java Bytecode Translators. In *GPCE*, pages 364–376, 2003.

[22] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen. Solving Large, Irregular Graph Problems Using Adaptive Work-Stealing. In *ICPP*, pages 536–545, 2008.

[23] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[24] Cray. The Chapel Parallel Programming Language. `https://chapel-lang.org`, 2018.

[25] Databricks. Spark Performance Tests. `https://github.com/databricks/spark-perf`, 2015.

[26] F. David, G. Thomas, J. Lawall, and G. Muller. Continuously Measuring Critical Section Pressure with the Free-Lunch Profiler. In *OOPSLA*, pages 291–307, 2014.

[27] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout. Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior. In *ISCA*, pages 511–522, 2013.

[28] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout. Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-Threaded Applications. In *OOPSLA*, pages 355–372, 2013.

[29] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge. Dynamic Metrics for Java. In *OOPSLA*, pages 149–168, 2003.

[30] B. Dufour, L. Hendren, and C. Verbrugge. *J: A Tool for Dynamic Analysis of Java Programs. In *OOPSLA Companion*, pages 306–307, 2003.

[31] A. Duran, J. Corbalan, and E. Ayguade. An adaptive cut-off for task parallelism. In *SC*, pages 1–11, 2008.

[32] ej-technologies. JProfiler. `https://www.ej-technologies.com/products/jprofiler/overview.html`, 2018.

[33] D. Eklov, N. Nikoleris, D. Black-Schaffer, and E. Hagersten. Cache Pirating: Measuring the Curse of the Shared Cache. In *ICPP*, pages 165–175, 2011.

[34] D. Eklov, N. Nikoleris, and E. Hagersten. A Software Based Profiling Method for Obtaining Speedup Stacks on Commodity Multi-Cores. In *ISPASS*, pages 148–157, 2014.

[35] S. Eyerman, K. D. Bois, and L. Eeckhout. Speedup Stacks: Identifying Scaling Bottlenecks in Multi-Threaded Applications. In *ISPASS*, pages 145–155, 2012.

[36] X. Fan, H. Jin, L. Zhu, X. Liao, C. Ye, and X. Tu. Function Flow: Making Synchronization Easier in Task Parallelism. In *PMAM*, pages 74–82, 2012.

[37] R. Ferreira da Silva, T. Glatard, and F. Desprez. Controlling fairness and task granularity in distributed, online, non-clairvoyant workflow executions. *Concurrency and Computation: Practice and Experience*, 26(14):2347–2366, 2014.

[38] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *PASTE*, pages 1–8, 2010.

[39] M. Fluet, M. Rainey, J. Reppy, and A. Shaw. Implicitly Threaded Parallelism in Manticore. *J. Funct. Program.*, 20(5-6):537–576, Nov. 2010.

[40] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*, pages 212–223, 1998.

[41] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor. Kremlin: Rethinking and Rebooting gprof for the Multicore Age. In *PLDI*, pages 458–469, 2011.

[42] B. R. Gaster and L. Howes. Can GPGPU Programming Be Liberated from the Data-Parallel Bottleneck? *IEEE Computer*, 45(8):42–52, August 2012.

[43] P. Grubel, H. Kaiser, J. Cook, and A. Serio. The Performance Implication of Task Size for Applications on the HPX Runtime System. In *CLUSTER*, pages 682–689, 2015.

[44] H2. `http://www.h2database.com`, 2018.

[45] R. H. Halstead, Jr. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst.*, 7(4):501–538, Oct. 1985.

[46] K. Hammond, H.-W. Loidl, and A. S. Partridge. Visualising Granularity in Parallel Programs: A Graphical Winnowing System for Haskell. In *HPFC*, pages 208–221, 1995.

[47] M. Hauswirth, P. F. Sweeney, A. Diwan, and M. Hind. Vertical Profiling: Understanding the Behavior of Object-Oriented Applications. In *OOPSLA*, pages 251–269, 2004.

[48] Y. He, C. E. Leiserson, and W. M. Leiserson. The Cilkview Scalability Analyzer. In *SPAA*, pages 145–156, 2010.

[49] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout. Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-Threaded Workloads. In *IISWC*, pages 38–49, 2011.

[50] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *IJCAI*, pages 235–245, 1973.

[51] L. Huelsbergen, J. R. Larus, and A. Aiken. Using the Run-time Sizes of Data Structures to Guide Parallel-Thread Creation. In *LSP*, pages 79–90, 1994.

[52] IBM. DayTrader. `https://www.ibm.com/support/knowledgecenter/en/linuxonibm/liaag/wascrypt/l0wscry00_daytrader.htm`, 2007.

[53] IBM. Health Center. `http://www.ibm.com/developerworks/java/jdk/tools/healthcenter/`, 2018.

[54] IBM. J9 Virtual Machine. `https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/user/java_jvm.html`, 2018.

[55] IBM. The X10 Parallel Programming Language. `http://x10-lang.org`, 2018.

[56] ICL. PAPI. `http://icl.utk.edu/papi/`, 2017.

[57] H. Inoue and T. Nakatani. How a Java VM Can Get More from a Hardware Performance Monitor. In *OOPSLA*, pages 137–154, 2009.

[58] Intel. Hyper-Threading Technology. `https://www.intel.com/content/www/us/en/architecture-and-technology/hyper-threading/hyper-threading-technology.html`, 2018.

[59] Intel. Turbo Boost Technology 2.0. `https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html`, 2018.

[60] Intel. VTune Amplifier. `https://software.intel.com/en-us/intel-vtune-amplifier-xe`, 2018.

[61] S. Iwasaki and K. Taura. Autotuning of a Cut-Off for Task Parallel Programs. In *MCSoC*, pages 353–360, 2016.

[62] J. JaJa. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[63] V. Janjic and K. Hammond. Granularity-Aware Work-Stealing for Computationally-Uniform Grids. In *CCGrid*, pages 123–134, 2010.

[64] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor. Kismet: Parallel Speedup Estimates for Serial Programs. In *OOPSLA*, pages 519–536, 2011.

[65] Y. Jiang, C. Xu, and X. Ma. DPAC: An Infrastructure for Dynamic Program Analysis of Concurrency Java Programs. In *MDS*, pages 2:1–2:6, 2013.

[66] D. Jin, P. O. Meredith, C. Lee, and G. Roşu. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE*, pages 1427–1430, 2012.

[67] T. Kalibera, M. Mole, R. Jones, and J. Vitek. A Black-box Approach to Understanding Concurrency in DaCapo. In *OOPSLA*, pages 335–354, 2012.

[68] M. Kambadur, K. Tang, and M. A. Kim. Harmony: Collection and Analysis of Parallel Block Vectors. In *ISCA*, pages 452–463, 2012.

[69] S. Kell, D. Ansaloni, W. Binder, and L. Marek. The JVM is Not Observable Enough (and What to Do About It). In *VMIL*, pages 33–38, 2012.

[70] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *ECOOP*, pages 327–353, 2001.

[71] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, pages 220–242, 1997.

[72] C. P. Kruskal and C. H. Smith. On the Notion of Granularity. *The Journal of Supercomputing*, 1(4):395–408, 1988.

[73] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Transactions on Computers*, 37(9):1088–1098, Sep 1988.

[74] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO*, pages 75–86, 2004.

[75] P. Lengauer, V. Bitto, H. Mössenböck, and M. Weninger. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *ICPE*, pages 3–14, 2017.

[76] J. Lifflander, S. Krishnamoorthy, and L. V. Kale. Optimizing Data Locality for Fork/Join Programs Using Constrained Work Stealing. In *SC*, pages 857–868, 2014.

[77] Linux man. top(1). `https://linux.die.net/man/1/top`, 2013.

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

[79] X. Liu and J. Mellor-Crummey. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *PPoPP*, pages 259–272, 2014.

[80] P. Lopez, M. Hermenegildo, and S. Debray. A Methodology for Granularity-Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation*, 21(4):715–734, 1996.

[81] L. Marek, S. Kell, Y. Zheng, L. Bulej, W. Binder, P. Tůma, D. Ansaloni, A. Sarimbekov, and A. Sewe. ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform. In *GPCE*, pages 105–114, 2013.

[82] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *AOSD*, pages 239–250, 2012.

[83] E. Mohr, D. Kranz, and J. Halstead, R.H. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):264–280, 1991.

[84] P. Moret, W. Binder, and A. Villazon. CCCP: Complete Calling Context Profiling in Virtual Execution Environments. In *PEPM*, pages 151–160, 2009.

[85] A. Muddukrishna, P. A. Jonsson, A. Podobas, and M. Brorsson. Grain Graphs: OpenMP Performance Analysis Made Easy. In *PPoPP*, pages 28:1–28:13, 2016.

[86] N. Muthuvelu, I. Chai, E. Chikkannan, and R. Buyya. *On-Line Task Granularity Adaptation for Dynamic Grid Applications*, pages 266–277. In *ICA3PP*, 2010.

[87] N. Muthuvelu, J. Liu, N. L. Soe, S. Venugopal, A. Sulistio, and R. Buyya. A Dynamic Job Grouping-based Scheduling for Deploying Applications with Fine-grained Tasks on Global Grids. In *ACSW Frontiers*, pages 41–48, 2005.

[88] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the Accuracy of Java Profilers. In *PLDI*, pages 187–197, 2010.

[89] A. Navarro, S. Mateo, J. M. Perez, V. Beltran, and E. Ayguadé. *Adaptive and Architecture-Independent Task Granularity for Recursive Applications*, pages 169–182. In *IWOMP*, 2017.

[90] A. Noll and T. Gross. Online Feedback-directed Optimizations for Parallel Java Code. In *OOPSLA*, pages 713–728, 2013.

[91] N. Nystrom, M. Clarkson, and A. C. Myers. Polyglot: An Extensible Compiler Framework for Java. In *Compiler Construction*, pages 138–152, 2003.

[92] OpenMP Architecture Review Board. OpenMP. `http://www.openmp.org`, 2018.

[93] J. Opsommer. A Taskgraph Clustering Algorithm based on an Attraction Metric between Tasks. In *CompEuro*, pages 77–82, 1992.

[94] Oracle. The Reflection API. `https://docs.oracle.com/javase/tutorial/reflect/`, 2015.

[95] Oracle. Documentation of `System.nanotime()`. `https://docs.oracle.com/javase/9/docs/api/java/lang/System.html`, 2017.

[96] Oracle. Java Native Interface. `https://docs.oracle.com/javase/9/docs/specs/jni/index.html`, 2017.

[97] Oracle. Java Platform, Standard Edition & Java Development Kit Version 9 API Specification. `https://docs.oracle.com/javase/9/docs/api/`, 2017.

[98] Oracle. Java Virtual Machine Tool Interface (JVM TI). `https://docs.oracle.com/javase/9/docs/specs/jvmti.html`, 2017.

[99] Oracle. Package `java.lang.instrument`. `https://docs.oracle.com/javase/9/docs/api/java/lang/instrument/package-summary.html`, 2017.

[100] Oracle. The Java Language Specification. `https://docs.oracle.com/javase/specs/jls/se9/html/index.html`, 2017.

[101] Oracle. The Java Virtual Machine Specification. `https://docs.oracle.com/javase/specs/jvms/se9/html/index.html`, 2017.

[102] Oracle. The Parallel Collector. `https://docs.oracle.com/javase/9/gctuning/parallel-collector1.htm`, 2017.

[103] Oracle. `ExecutorService`. `https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ExecutorService.html`, 2017.

[104] Oracle. `ForkJoinPool`. `https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ForkJoinPool.html`, 2017.

[105] Oracle. `ThreadPoolExecutor`. `https://docs.oracle.com/javase/9/docs/api/java/util/concurrent/ThreadPoolExecutor.html`, 2017.

[106] Oracle. Java Mission Control. `http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html`, 2018.

[107] M. S. Orr, B. M. Beckmann, S. K. Reinhardt, and D. A. Wood. Fine-grain Task Aggregation and Coordination on GPUs. In *ISCA*, pages 181–192, 2014.

[108] OW2 Consortium. ASM. `http://asm.ow2.org/`, 2018.

[109] M. A. Palis, J.-C. Liou, and D. S. L. Wei. Task Clustering and Scheduling for Distributed Memory Parallel Architectures. *IEEE Transactions on Parallel and Distributed Systems*, 7(1):46–55, Jan 1996.

[110] A. K. Paul, W. Zhuang, L. Xu, M. Li, M. M. Rafique, and A. R. Butt. CHOPPER: Optimizing Data Partitioning for In-memory Data Analytics Frameworks. In *CLUSTER*, pages 110–119, 2016.

[111] R. Pawlak. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Systems Online*, 7(11), 2006.

[112] D. J. Pearce, M. Webster, R. Berry, and P. H. Kelly. Profiling with AspectJ. *Software: Practice and Experience*, 37(7):747–777, 2007.

[113] perf. Linux profiling with performance counters. `https://perf.wiki.kernel.org`, 2015.

[114] G. Reger, H. C. Cruz, and D. Rydeheard. MarQ: Monitoring at Runtime with QEA. In *TACAS*, pages 596–610, 2015.

[115] J. Reinders. *Intel Threading Building Blocks*. O'Reilly & Associates, Inc., 1st edition, 2007.

[116] M. C. Rinard and M. S. Lam. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems*, 20(3):483–545, 1998.

[117] A. Rosà, L. Y. Chen, and W. Binder. Actor Profiling in Virtual Execution Environments. In *GPCE*, pages 36–46, 2016.

[118] M. Roth, M. J. Best, C. Mustard, and A. Fedorova. Deconstructing the Overhead in Parallel Applications. In *IISWC*, pages 59–68, 2012.

[119] Sandy Ryza. How-to: Tune Your Apache Spark Jobs (Part 1). `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/`, 2015.

[120] Sandy Ryza. How-to: Tune Your Apache Spark Jobs (Part 2). `http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/`, 2015.

[121] A. Sarimbekov, A. Sewe, W. Binder, P. Moret, and M. Mezini. JP2: Call-site Aware Calling Context Profiling for the Java Virtual Machine. *Sci. Comput. Program.*, 79:146–157, Jan. 2014.

[122] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson. The Cilkprof Scalability Profiler. In *SPAA*, pages 89–100, 2015.

[123] A. Sewe, M. Mezini, A. Sarimbekov, and W. Binder. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *OOPSLA*, pages 657–676, 2011.

[124] Simon Kitching. OSGi Classloading. http://moi.vonos.net/java/osgi-classloaders/, 2013.

[125] G. Singh, M.-H. Su, K. Vahi, E. Deelman, B. Berriman, J. Good, D. S. Katz, and G. Mehta. Workflow Task Clustering for Best Effort Systems with Pegasus. In *MG*, pages 9:1–9:8, 2008.

[126] E. Tanter, P. Moret, W. Binder, and D. Ansaloni. Composition of Dynamic Analysis Aspects. In *GPCE*, pages 113–122, 2010.

[127] Q. M. Teng, H. C. Wang, Z. Xiao, P. F. Sweeney, and E. Duesterwald. THOR: A performance analysis tool for Java applications running on multicore systems. *IBM Journal of Research and Development*, 54(5):4:1–4:17, Sept 2010.

[128] The Apache Software Foundation. BCEL. http://commons.apache.org/bcel/, 2017.

[129] The Apache Software Foundation. Apache Spark - RDD Programming Guide. https://spark.apache.org/docs/latest/rdd-programming-guide.html, 2018.

[130] The Apache Software Foundation. Apache Spark MLlib. https://spark.apache.org/mllib/, 2018.

[131] The Apache Software Foundation. Apache Tomcat. http://tomcat.apache.org, 2018.

[132] The Apache Software Foundation. Lucene. https://lucene.apache.org, 2018.

[133] The Apache Software Foundation. Spark Configuration. https://spark.apache.org/docs/latest/configuration.html, 2018.

[134] The Apache Software Foundation. Spark Streaming. https://spark.apache.org/streaming/, 2018.

[135] The Apache Software Foundation. SparkContext API. https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/SparkContext.html, 2018.

[136] The AspectJ Team. AspectJ Quick Reference - Chapter 5. Pitfalls - Infinite Loops. http://www.eclipse.org/aspectj/doc/released/progguide/pitfalls-infiniteLoops.html, 2001.

[137] The Eclipse Foundation. Jetty. `http://www.eclipse.org/jetty/`, 2016.

[138] The Eclipse Foundation. `AjTypeSystem`. `https://eclipse.org/aspectj/doc/next/adk15notebook/reflection.html`, 2018.

[139] The Eclipse Foundation. Eclipse. `https://www.eclipse.org`, 2018.

[140] The Stanford Natural Language Processing Group. Stanford Topic Modeling Toolbox. `https://nlp.stanford.edu/software/tmt/tmt-0.4/`, 2010.

[141] P. Thoman, H. Jordan, and T. Fahringer. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning. In *Euro-Par*, pages 164–177, 2013.

[142] TPC. TPC-C. `http://www.tpc.org/tpcc/`, 2010.

[143] R. Vallée-Rai, E. Gagnon, L. J. Hendren, P. Lam, P. Pominville, and V. Sundaresan. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible? In *CC*, pages 18–34, 2000.

[144] L. Wang, H. Cui, Y. Duan, F. Lu, X. Feng, and P.-C. Yew. An Adaptive Task Creation Strategy for Work-Stealing Scheduling. In *CGO*, pages 266–277, 2010.

[145] G. Xirogiannis. Granularity Control for Distributed Execution of Logic Programs. In *ICDCS*, pages 230–237, 1998.

[146] A. Yoga and S. Nagarakatte. A Fast Causal Profiler for Task Parallel Programs. In *ESEC/FSE*, pages 15–26, 2017.

[147] YourKit. YourKit. `https://www.yourkit.com`, 2018.

[148] T. Yu and M. Pradel. SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks. In *ISSTA*, pages 389–400, 2016.

[149] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *NSDI*, pages 1–14, 2012.

[150] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar. Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs. In *PACT*, pages 169–180, 2010.

[151] Y. Zheng, A. Rosà, L. Salucci, Y. Li, H. Sun, O. Javed, L. Bulej, L. Y. Chen, Z. Qi, and W. Binder. AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses. In *SANER*, pages 639–643, 2016.

[152] G. M. Zoppetti, G. Agrawal, L. Pollock, J. N. Amaral, X. Tang, and G. Gao. Automatic Compiler Techniques for Thread Coarsening for Multithreaded Architectures. In *ICS*, pages 306–315, 2000.

Index

ϵ_T, 37
$\perp$, 34
$\perp_T$, 34

Actor, 91
    - model, 91
Agent, 25
AOP, 13
Aspect-oriented programming, 13
    advice, 13
    join point, 13

Class, 52
Classfile, 52
Classloader, 52
    application -, 53
    bootstrap -, 53
    defining -, 53
    namespace, 5, 53
    user-defined -, 53
Classpath, 53
Context
    - information, 26
    calling -, 4
Cycle
    - counter, 39
    reference -, 30

DaCapo, 74
DiSL, 26
    - server, 27
    annotation, 26
    code snippet, 26
    context information, 26
    guard, 26
    marker, 26
    scope, 26
    synthetic local variable, 26
    thread-local variable, 26

Empty element, 37

Factor
    overhead -, 77
    perturbation -, 81
    speedup -, 79, 109
Full bytecode coverage, 21
Fully qualified name, 53

Hardware performance counter, 32
Heap tagging, 26
HPC, 32

Instrumentation
    compile-time -, 55
    in-process -, 56
    load-time -, 56
    out-of-process -, 57
Iteration
    steady-state -, 74

warm-up -, 74

Java class library, 4
JNI, 26
JVMTI, 25

Method
submission -, 28
execution -, 28
Model
actor -, 91
task -, 27
work-span -, 14

Nested
- call, 37
- task, 28
Null value, 34

Overhead factor, 77

PAPI, 32
perf, 32
Perturbation factor, 81
Primordial phase, 26
Profile, 31
actionable -, 4
task -, 34
Profiling
calling context -, 33
vertical -, 4

Reflection API
DiSL -, 60
Java -, 54
Reflective
- information, 53
- supertype information, 5, 54
RSI, 5

ScalaBench, 74
Shadow stack, 36
empty element, 37
Shadow VM, 27
Spark, 74
action, 105
batch, 107
block, 107
job, 105
local mode, 77
partition, 105
record, 105
resilient distributed dataset, 105
shuffle, 106
stage, 105
task, 92, 105
transformation, 105
narrow -, 105
wide -, 105
Spark Perf, 74
Speedup factor, 79, 109
Stop-the-world GC, 30

Tag, 26
Task, 1, 27
- execution, 28
- execution framework, 28
- granularity, 2, 28
- interface, 28
- model, 27
- profile, 34
- submission, 28
- trace, 32
-parallel application, 1
-parallelism, 1
coarse-grained -, 2, 97
executed -, 28
fine-grained -, 2, 90
nested -, 28
outer -, 28

submitted -, 28
tgp, 4
Thread
- cycle counter, 39
creating -, 29
executing -, 29
top, 32
Trace, 31
CPU -, 32
CS -, 32
GC -, 32
task -, 32
Type-specific analysis, 5

Variable
synthetic local -, 26
thread-local -, 26
Vertical profiling, 4

Weaver, 21
Weaving, 21
Work-span model, 2, 14
Critical path, 14
Span, 2, 14
Work, 2, 14