tgp: A Task-Granularity Profiler for the Java Virtual Machine

Conference Paper · December 2017

DOI: 10.1109/APSEC.2017.67

3 authors:

Edgar Eduardo Rosales Rosero
Los Andes University (Colombia)
27 PUBLICATIONS 87 CITATIONS
SEE PROFILE

Andrea Rosà
University of Lugano
53 PUBLICATIONS 304 CITATIONS
SEE PROFILE

Walter Binder
University of Lugano
314 PUBLICATIONS 3,299 CITATIONS
SEE PROFILE

tgp: a Task-Granularity Profiler for the Java Virtual Machine

(Short Paper)

Eduardo Rosales, Andrea Rosà, and Walter Binder
Faculty of Informatics, Università della Svizzera italiana (USI), Lugano, Switzerland
Email: {rosale, andrea.rosa, walter.binder}@usi.ch

***Abstract*—The analysis of task granularity in parallel applications (i.e., the amount of work to be performed by parallel tasks) is essential to unveil performance problems and to optimize task-parallel applications. Too small task granularities may result in high parallelization overheads, while too large task granularities may indicate missed parallelization opportunities. Despite the importance of task granularity, this metric is not considered by existing profilers for parallel applications on the Java Virtual Machine (JVM). In this paper we present tgp, a novel task-granularity profiler for multi-threaded applications on the JVM. tgp collects bytecode- and hardware-level metrics to characterize task granularity, assisting the developer in diagnosing and locating parallelization shortcomings.**

I. Introduction

In the multicore era, developers are increasingly writing multi-threaded applications that execute workloads in parallel to utilize the available computing resources and to achieve significant speedups. Still, understanding parallelization problems preventing applications to fully leverage the hardware resources remains a major challenge. Here, we tackle this issue for multi-threaded, task-parallel applications running in a single Java Virtual Machine (JVM) on a shared-memory multicore machine. In this context, a deep comprehension of *task granularity*, i.e., the amount of work to be performed by parallel tasks, is crucial in locating performance issues.

To illustrate this concept, consider a multi-threaded application performing some fixed amount of work by assigning parts of it to several tasks running in parallel. On the one hand, if the work is divided into too fine-grained tasks (i.e., tasks of small size, each of them performing only few computations), the application may suffer from parallelization overheads caused by significant inter-thread communication, synchronization, and task scheduling. On the other hand, if the work is distributed into only a few too coarse-grained tasks (i.e., tasks of large size, each of them executing substantial sequential computations), the application may suffer from load imbalance and low CPU utilization, thus missing parallelization opportunities. Locating too fine- and coarse-grained tasks is therefore fundamental to optimize task-parallel applications.

In spite of the importance of task granularity in understanding performance issues in parallel applications, we are not aware of profilers targeting the JVM and specifically considering this metric. To support the understanding and optimization of task-parallel applications, in this paper we introduce tgp, a novel task-granularity profiler for the JVM. tgp focuses specifically on the tasks spawned by an application, reconciling high-level metrics (i.e., metrics obtained at the JVM-level such as bytecode count) and low-level metrics (i.e., metrics obtained at the hardware-level such as machine-instructions and reference-cycles counts) to characterize tasks as fine- or coarse-grained. tgp also collects thread-level information for each spawned task (i.e., the thread responsible for its initialization and its execution). Lastly, for each task, tgp can collect the calling contexts where the task is created and where its execution method is invoked. This information guides the developer towards specific classes and methods where optimizations are needed.

Our work makes the following contributions:

1) We present a task-granularity profiler for the JVM. To the best of our knowledge, our profiler is the first task-granularity profiler for task-parallel applications on the JVM.
2) We characterize a benchmark from the DaCapo suite in terms of fine- and coarse-grained tasks, pinpointing classes and methods suffering from suboptimal task granularities.

The remainder of this paper is organized as follows. Section II describes the scope of our profiler and the metrics it uses. Section III presents a high-level architectural view of our tool. Section IV shows the evaluation of tgp. Section V discusses related work to our approach. Finally, section VI concludes.

II. Profiler Overview

This section details the scope of our profiler and the metrics it uses when profiling tasks.

A. Tasks

tgp focuses on multi-threaded applications executing in a single JVM on a shared-memory multicore machine. tgp is flexible, allowing the developer to provide a customized task definition. In its default configuration, a task is defined as an 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` (which logically extends `Runnable`, allowing tasks to declare a non-void return value), and the abstract class `java.util.concurrent.`

`ForkJoinTask` (which defines tasks running within a fork-join pool)[1]. Accordingly, tgp profiles `Runnable` objects executed by Java threads, as well as tasks submitted to task execution frameworks[2]. Lastly, our tool profiles Java threads, since `java.lang.Thread` implements `java.lang.Runnable`.

B. *Task Granularity*

Task granularity refers to the amount of work performed by a task, subsuming all computations in the dynamic extent of the methods `Runnable.run`, `Callable.call`, and `ForkJoinTask.exec` (i.e., the methods defining the actions to be performed by a task)[3]. tgp represents this work using three different task-granularity metrics.

C. *Task-Granularity Metrics*

The main purpose of tgp is to accurately quantify the granularity of each task. To this end, our tool provides both high- and low-level metrics, each having different strengths and limitations. This plurality allows developers to choose a single metric or a combination of them (only one can be profiled at the same time) to profile their target applications.

Regarding high-level metrics, tgp can quantify task granularity in terms of *bytecode count*, i.e., the number of bytecodes executed by a task. On the one hand, this metric is little perturbed by the inserted instrumentation code (because the metric disregards the inserted bytecodes), is platform independent (as long as the same Java library is used), and does not require any hardware support [1], [2]. On the other hand, bytecode count cannot track native code executed by tasks or internal JVM functions, thus possibly underestimating the work performed by a task. Moreover, bytecode count represents work of different complexity with the same unit, and the inserted instrumentation code may perturb the optimizations performed by the JVM's dynamic compiler [3]. In summary, bytecode count is valuable when portability among different architectures is a main concern or when tgp is run on a platform not supporting Hardware Performance Counters (HPCs). However, results based on bytecode count may not well represent the real execution on a specific platform.

In terms of low-level metrics, tgp profiles the number of executed *machine instructions* and the *reference cycles*[4] spent by a task. Both metrics cover the execution of native code, resulting in an accurate estimation of the work performed at the task-level. In addition, counting reference cycles enables one to distinguish between simple and complex operations, and reference cycles can be used to estimate CPU time if the nominal clock frequency is known. tgp collects such metrics thanks to HPC support available in most modern processors. However, the portability of such metrics is limited since they depend on the JVM used, the optimizations performed by the dynamic compiler, and hardware details such as the instruction set architecture (ISA).

[1]For more information, we refer the reader to the documentation of the interfaces/classes `Executor`, `ExecutorService`, `ForkJoinTask` and `ThreadPoolExecutor` in package `java.util.concurrent`.

[2]We use the term *task submission* to refer to the submission of a task to a task execution framework. We denote as *task execution framework* any instance of class `Executor`. For example, `ThreadPoolExecutor` and `ForkJoinPool` are task execution frameworks.

[3]In this paper, we will refer to such methods as *task execution methods*.

[4]Reference cycles measure the elapsed cycles at a constant reference clock rate, independently from the actual clock rate of the core.

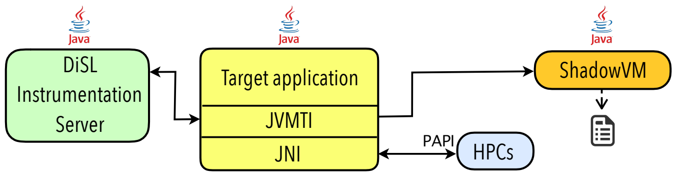


Fig. 1. Architecture of tgp.

D. *Other Information*

Apart from metrics focusing on task granularity, tgp collects complementary information to assist the user in the performance analysis. First, the tool provides information about task initialization. It may help locate expensive task initializations, as well as tasks that are created but not executed. Second, tgp provides information about the creating and executing thread of each spawned task, as well as the starting and ending execution timestamps of the task. Such information may be used during offline analysis to investigate task-to-thread distribution and load balancing. Finally, tgp provides the calling contexts of task initialization and execution. Such calling contexts help the user locate code where optimizations may be beneficial.

III. Architecture

In this section we present a high-level architectural overview of our tool.

tgp profiles every task spawned by the target application. That is, it detects the creation and execution of all tasks and precisely accounts the selected metric in the dynamic extent of task execution methods. tgp operates in two modes, the first mode enables accounting bytecode, machine instructions or reference cycles while the second mode allows profiling calling contexts.

In the first mode, if collecting bytecode count, complete bytecode coverage is particularly important because instrumentation code must be inserted in the Java class library.

As shown in Figure 1, tgp builds on DiSL [4], a dynamic program analysis framework based on bytecode instrumentation. DiSL guarantees full bytecode coverage, ensuring that the analysis results represent the overall execution of the target application. DiSL instruments the target application in a separate JVM process, the *DiSL instrumentation server*. To enable communication between this component and the target application, a JVMTI agent[5] is attached to the latter. The agent intercepts class loading, sending every loaded class to the instrumentation server. When the server receives a class, it determines which methods are to be instrumented (if any)

[5]https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/

to collect the desired metrics at runtime. Subsequently, the (potentially) instrumented class is returned to the JVM running the target application.

Additionally, to guarantee that the collected profiles are accurate, tgp must not significantly perturb the target application. To reduce profiling overhead and hence measurement perturbation, tgp minimizes the inserted instrumentation code and avoids allocating any large profiling data structure in the observed JVM. Moreover, tgp makes use of a deployment setting offered by DiSL that isolates the analysis from the instrumented application, executing analysis code in a separate process, the *ShadowVM* [5].

During task creation or execution, the inserted instrumentation code collects the desired metrics. When the constructor or the execution method of a task completes, the collected metrics are sent to the ShadowVM, which processes and stores them. Thanks to ShadowVM, tgp avoids sharing states between the execution of the main logic in the profiler and the target application, hence reducing measurement perturbation and avoiding known issues inherent to non-isolated approaches [6]. Communication with the ShadowVM relies on another JVMTI agent attached to the target application.

When collecting machine instructions or reference cycles, tgp resorts to HPCs able to query them with low overhead. HPCs are accessed through PAPI [7], a third-party C library providing a common interface to access HPCs on different processors. tgp sets up, activates, and reads such counters from the instrumented code in the target application. A JNI agent[6] attached to the target application enables managing PAPI's C API from the instrumented Java code.

In the second mode, tgp collects the calling contexts of task creation and execution. This feature requires instrumentation of all method entries and exits to keep a calling-context representation in a stack allocated on the heap of the target application. Therefore, when profiling calling contexts, the profiling of machine instructions and reference cycles are disabled since such measurements are very sensitive to perturbations introduced by the instrumentation. That is, calling-context profiling may only be activated in conjunction with bytecode profiling, where perturbations are much less of a concern as the inserted instrumentation code is disregarded when calculating the bytecode metric.

Finally, the analysis within ShadowVM dumps the collected metrics in traces containing the following data per task: the task ID, the ID of the threads creating and executing the task (each ID is obtained by combining the class name and the identity hash code of the object), the starting and ending execution timestamps, a flag indicating whether the task is also a thread, the collected metric (bytecodes, machine instructions, or reference cycles) for both task initialization and execution, and lastly, when activated, the initialization and execution calling contexts.

The traces are further processed via offline analysis to calculate aggregate statistics such as the number of (non-)executed

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

tasks or threads, or the average, standard deviation, and percentiles of the distribution of task granularities. This analysis is crucial to further investigate task granularity, since its interpretation helps the developer in locating fine- or coarse-grained tasks.

IV. Evaluation

In this section, we show how tgp can help the developer in locating fine- and coarse-grained tasks thanks to the collected traces. Moreover, we show the profiling overhead introduced by our tool.

A. Experimental Setup

Our evaluation is conducted on a server-class machine equipped with two NUMA nodes, each with an Intel Xeon E5-2680 (2.7 GHz) processor with 8 cores and 64 GB of RAM, running under Ubuntu 14.04 LTS. Turbo Boost and Hyper-threading are disabled, and the CPU governor is set to "performance". We use Java 1.8.0_131 and the Java HotSpot 64-Bit Server VM (build 25.131-b11).

The workloads used for the evaluation come from DaCapo [8], a well-known Java benchmark suite including several multi-threaded, task-parallel applications. We profile all the benchmarks included in the suite except tomcat, which failed on Java 8 for any input size, even without profiling. The number of warmup iterations per benchmark is 20; we profile the 21st iteration to measure steady-state performance.

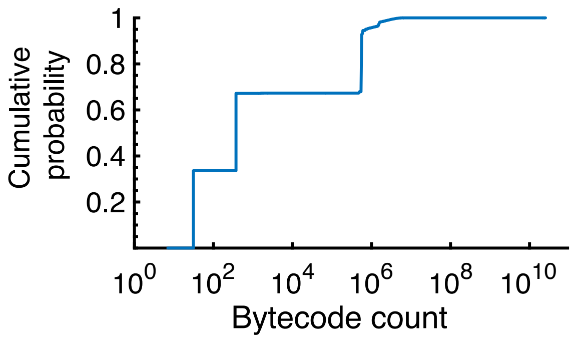
Lastly, the target application is pinned to a different NUMA node from the one executing the DiSL instrumentation server and ShadowVM. This deployment setting reduces the perturbation generated by the instrumentation, since the target application exclusively utilizes the cores and the memory of its dedicated underlying NUMA node.

B. Task-Granularity Analysis

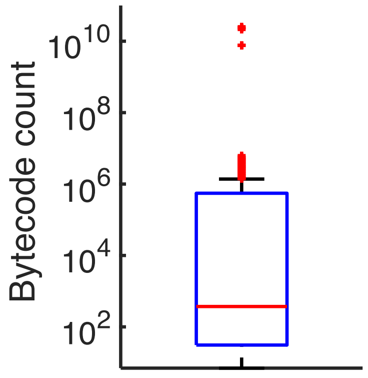
Here, we demonstrate how tgp helps the user in the characterization of task granularity. Due to space constraints, we show the results obtained for a single benchmark of the DaCapo suite, fully characterizing task granularity in terms of bytecode count. For additional results on the characterization of task granularity (measured in bytecodes, machine instructions, and reference cycles) of the DaCapo and ScalaBench [9] suites, we refer the reader to www.inf.usi.ch/phd/rosaa/tgp-additional-results/.

The target application of this analysis is tradesoap [10] executed with its largest input size ("huge"). As shown thereafter, this benchmark exhibits both too fine- and too coarse-grained tasks. The workload is composed of a total of 276 355 executed tasks.

In Figure 2(a) we present the Cumulative Distribution Function (CDF) of the task granularities of tradesoap. The x-axis reports bytecode count on a logarithmic scale, while the y-axis shows the cumulative probability. In the figure, we can observe three values of bytecode count where the CDF features a long vertical trend. The trend represents three large groups of tasks with the same granularity, two of which contain very



(a) Cumulative Distribution Function (CDF).



(b) Box plot.

Fig. 2. Distribution of task granularities for tradesoap (276 355 tasks in total).

fine-grained tasks. The first of such groups relates to tasks executing exactly 31 bytecodes and represents approximately 31% of all tasks spawned by tradesoap. The second group is composed of tasks executing only 372 bytecodes, representing approximately 36% of all executed tasks.

An analysis of the produced traces reveals that all tasks within a group belong to the same class, i.e., `org.apache.geronimo.pool.ThreadPool$1` for the 372-bytecode group and `org.apache.geronimo.pool.ThreadPool$ContextClassLoaderRunnable` for the 31-bytecode group. We inspected the source code of both classes and determined that such tasks act as wrappers to other tasks, delegating the work to other `Runnables`. Their purpose is to set or clean up some data (`ThreadPool$1`) or handling exception (`ThreadPool$ContextClassLoaderRunnable`) before/after calling another `Runnable`. Most instances of such classes execute the same control flow, which explains the presence of a large number of tasks executing the same number of bytecodes. Both classes could be further inspected to determine whether fine-grained tasks may be merged into larger ones. This optimization could lead to lower parallelization overheads, caused by the potential initialization, communication, synchronization and scheduling of too fine-grained tasks.

In Figure 2(b) we present the box plot of the distribution of task granularities for tradesoap. The y-axis represents the bytecode count on a logarithmic scale. The red line represents the median (372), the lower and upper blue line represent the first quartile (31), and the third quartile (553 041), respectively. The two black lines extending vertically from the box cover 99.3% of the data, assuming it is normally distributed (the lower black line overlaps with the x-axis). Values outside this area (i.e., the outliers) are plotted individually with a red cross.

The presence of a few outliers at the top of the figure reveals that some tasks execute an amount of bytecode significantly greater than most of the spawned tasks. The traces show a group of 8 threads executing more than 21 billion bytecodes, of class `org.apache.geronimo.samples.daytrader.dacapo.DaCapoTrader`. The largest observed granularity is around 25 billion bytecodes. Apart from them, the traces show 4 tasks executing around 7.7 billion bytecodes, of class `org.mortbay.jetty.nio.SelectChannelConnector$ConnectorEndPoint`. Both classes may benefit from further inspection to determine whether the work executed by such very coarse-grained tasks could be split into smaller tasks. This could lead to better load balancing and CPU utilization, and may result in speedups.

C. *Profiling Overhead*

Finally, we discuss the overhead introduced by tgp when profiling machine instructions and reference cycles, i.e., the metrics mostly affected from perturbations among those collected by our tool. We consider the workload executed by all benchmarks with the "default" input size (with the exception of eclipse, which only passes validation on "small" size on Java 8 even without profiling). Our results show that the overhead caused by machine-instruction and reference-cycle profiling is lower than 1% for xalan, batik, lusearch, and luindex. Moreover, it remains below 2% for sunflow, fop, h2, and jython, and lower than 5% for tradebeans, tradesoap, and eclipse. In contrast, pmd and avrora are the only benchmarks presenting overheads around 12% and 14%, respectively. Overall, our results show that tgp introduces only low overhead while profiling machine instructions and reference cycles. Consequently, such metrics can be considered representative of the task granularities of the target application.

V. Related Work

Here, we discuss work related to our approach along three dimensions. We first analyze related work based on the work-span model. Next, we focus on metrics for studying parallelism. Finally, we compare tgp with existing parallelism profilers.

A. *Work-Span Model*

The work-span model [11], [12] aims at finding the maximum theoretical parallelism of an application by dividing its work (i.e., the time a serial 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). Cilkview [13] builds upon this model to predict achievable speedup bounds for a

Cilk [14] application when increasing the number of processors it uses. CilkProf [15] extends Cilkview by measuring work and span on each site where a function is called or spawned to build an ordered list of call sites that constitute a bottleneck towards parallelization. TaskProf [16] computes work, span, and the asymptotic parallelism of Intel Threading Building Blocks (TBB) [17] applications and relies on causal profiling [18] to estimate parallelism improvements.

Overall, these works aim at detecting bottlenecks and predicting speedups by mainly focusing on the longest tasks of an application, paying little attention to the possible performance drawbacks caused by short tasks. In contrast, our work targets also short tasks, precisely enabling the location of too fine-grained task-level parallelism.

B. Metrics for Parallelism

Several researchers have introduced metrics to describe the parallel behavior of an application. Dufour et al. [19] propose dynamic metrics to analyze concurrency and synchronization of Java programs. Kalibera et al. [20] provide several platform-independent metrics on concurrency and scalability to enable a black-box understanding on the parallel behavior of Java applications. Kambadur et al. [21] introduce *parallel block vectors* to show which parts of a program belong to the serial and parallel phases of its execution.

In their study of scalability bottlenecks, Roth et al. [22] define *work*, *distribution*, and *delay* as the basis of a hierarchical set of metrics to identify scalability issues. Chen et al. [23] analyze scalability bottlenecks relying on cache- and pipeline-misses along with cache-to-cache transfers. Heirman et al. [24] and Eyerman et al. [25] introduce *speedup stacks* to quantify the impact of scalability bottlenecks on the speedup of parallel applications. Schörgenhumer et al. [26] introduce a sampling-based lock contention profiling technique for Java applications to collect detailed contention data for both *intrinsic locks* and for *explicit locks* in the `java.util.concurrent` library.

Regarding visualization of bottlenecks, Du Bois et al. [27] introduce *bottle graphs* to show application performance, relating work per thread with execution time, while *criticality stacks* [28] study the running time of a thread and the number of threads waiting on its termination. Finally, Noll et al. [29] describe the tradeoff between the overhead of parallel executions and the potential performance gain.

Unlike the scope of such works, we utilize high- and low-level metrics to specifically characterize the granularity of all tasks spawned by an application and to analyze their impact on the application performance. To this end, we implement an efficient profiler able to collect such metrics, introducing only low overhead.

Finally, note that some of the aforementioned works [20], [23], [26], [27] describe the parallel characteristics of the DaCapo [8] benchmarks, as we do. Our work is complementary to them, as we reveal benchmark features that were previously unknown (i.e., a full characterization of a benchmark in terms of its fine- and coarse-grained tasks).

C. Parallelism Profilers

Researchers from both industry and academia have developed several tools to analyze various parallel characteristics of an application.

The aforementioned tools based on the work-span model [13], [15], [16] are also parallelism profilers. Differently from Cilkview [13], tgp is not aimed to compute the whole program parallelism but it measures the granularity of each task spawned by the application. In contrast to CilkProf [15] and TaskProf [16], the use of tgp requires neither compiler support nor library modification. Furthermore, apart from the fact that none of these tools supports the JVM, our tool offers flexibility in allowing the developer to specify what constitutes a task of interest.

Tools focusing on parallelism discovery include Kremlin [30], which tracks loops and dependencies to pinpoint sequential parts of programs that can be parallelized. Kismet [31] builds on Kremlin and attempts to predict the speedup after parallelization, given a target machine and runtime system. Free-Lunch [32] computes the percentage of time spent by an application in acquiring locks to individually correlate the progress of threads on locks. Similarly, SyncProf [33] locates code portions where bottlenecks are caused by threads suffering from contention and synchronization issues.

Other tools and profilers are able to characterize threads and processes over time by focusing on JVM-level metrics (e.g., VisualVM [34], Java Mission Control [35], HPROF [36], JProfiler [37], and YourKit [38]) or by collecting hardware-level metrics (including HPCToolkit [39], Intel VTune [40] and AMD CodeAnalyst [41]).

Overall, these works consider processes or threads as the main computing entities, providing little information about individual tasks and their impact on the application performance. On the contrary, our work specifically addresses the profiling of fine- and coarse-grained tasks to diagnose performance shortcomings on task-parallel applications. Moreover, thanks to the collection of calling-context information per task, our tool facilitates the location of specific classes and code portions which can benefit from optimizations.

VI. Conclusions

In this paper, we presented tgp, a novel task-granularity profiler for the JVM. To the best of our knowledge, tgp is the first profiler centered on the analysis of task granularity on the JVM. Our evaluation results show that tgp is helpful in assisting the developer in the identification of fine- and coarse-grained tasks in task-parallel applications. This information is valuable to help the developer in locating specific classes and code portions that should be further inspected to achieve performance improvements.

More information on tgp can be found at www.inf.usi.ch/phd/rosaa/tgp, where users can find a demonstration version of the tool and proper support, including documentation, scripts, tutorials, and sample workloads.

As part of our future work, we plan to extend the analysis presented in this paper by expanding our analysis on other task-parallel applications and applying optimizations based on our findings. Finally, we plan to release tgp as open-source software.

REFERENCES

[1] W. Binder, J. G. Hulaas, and A. Villazón, "Portable Resource Control in Java," in *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '01, 2001, pp. 139–155.

[2] W. Binder, "A Portable and Customizable Profiling Framework for Java Based on Bytecode Instruction Counting," in *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings*, 2005, pp. 178–194.

[3] A. Rosà, L. Y. Chen, and W. Binder, "Profiling Actor Utilization and Communication in Akka," in *Proceedings of the 15th International Workshop on Erlang*, ser. Erlang 2016, 2016, pp. 24–32.

[4] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi, "DiSL: A Domain-specific Language for Bytecode Instrumentation," in *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, ser. AOSD '12, 2012, pp. 239–250.

[5] 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 *Proceedings of the 12th International Conference on Generative Programming: Concepts and Experiences*, ser. GPCE '13, 2013, pp. 105–114.

[6] S. Kell, D. Ansaloni, W. Binder, and L. Marek, "The JVM is Not Observable Enough (and What to Do About It)," in *Proceedings of the Sixth ACM Workshop on Virtual Machines and Intermediate Languages*, ser. VMIL '12, 2012, pp. 33–38.

[7] ICL, "PAPI," http://icl.utk.edu/papi/.

[8] 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 *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06, 2006, pp. 169–190.

[9] 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 *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11, 2011, pp. 657–676.

[10] DaCapo Project, http://dacapobench.org/daytrader.html.

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

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

[13] Y. He, C. E. Leiserson, and W. M. Leiserson, "The Cilkview Scalability Analyzer," in *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '10, 2010, pp. 145–156.

[14] M. Frigo, C. E. Leiserson, and K. H. Randall, "The Implementation of the Cilk-5 Multithreaded Language," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98, 1998, pp. 212–223.

[15] T. B. Schardl, B. C. Kuszmaul, I.-T. A. Lee, W. M. Leiserson, and C. E. Leiserson, "The Cilkprof Scalability Profiler," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, ser. SPAA '15, 2015, pp. 89–100.

[16] A. Yoga and S. Nagarakatte, "A Fast Causal Profiler for Task Parallel Programs," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, pp. 15–26.

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

[18] C. Curtsinger and E. D. Berger, "Coz: Finding Code That Counts with Causal Profiling," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15, 2015, pp. 184–197.

[19] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic Metrics for Java," in *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, ser. OOPSLA '03, 2003, pp. 149–168.

[20] T. Kalibera, M. Mole, R. Jones, and J. Vitek, "A Black-box Approach to Understanding Concurrency in DaCapo," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '12, 2012, pp. 335–354.

[21] M. Kambadur, K. Tang, and M. A. Kim, "Harmony: Collection and Analysis of Parallel Block Vectors," in *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ser. ISCA '12, 2012, pp. 452–463.

[22] M. Roth, M. J. Best, C. Mustard, and A. Fedorova, "Deconstructing The Overhead in Parallel Applications," in *2012 IEEE International Symposium on Workload Characterization (IISWC)*, 2012, pp. 59–68.

[23] K. Y. Chen, J. M. Chang, and T. W. Hou, "Multithreading in Java: Performance and Scalability on Multicore Systems," *IEEE Transactions on Computers*, vol. 60, no. 11, pp. 1521–1534, 2011.

[24] W. Heirman, T. E. Carlson, S. Che, K. Skadron, and L. Eeckhout, "Using Cycle Stacks to Understand Scaling Bottlenecks in Multi-threaded Workloads," in *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, ser. IISWC '11, 2011, pp. 38–49.

[25] S. Eyerman, K. D. Bois, and L. Eeckhout, "Speedup Stacks: Identifying Scaling Bottlenecks in Multi-threaded Applications," in *2012 IEEE International Symposium on Performance Analysis of Systems Software*, ser. ISPASS, 2012, pp. 145–155.

[26] A. Schörgenhumer, P. Hofer, D. Gnedt, and H. Mössenböck, "Efficient Sampling-based Lock Contention Profiling for Java," in *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ser. ICPE '17, 2017, pp. 331–334.

[27] K. Du Bois, J. B. Sartor, S. Eyerman, and L. Eeckhout, "Bottle Graphs: Visualizing Scalability Bottlenecks in Multi-threaded Applications," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13, 2013, pp. 355–372.

[28] K. Du Bois, S. Eyerman, J. B. Sartor, and L. Eeckhout, "Criticality Stacks: Identifying Critical Threads in Parallel Programs Using Synchronization Behavior," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13, 2013, pp. 511–522.

[29] A. Noll and T. Gross, "Online Feedback-directed Optimizations for Parallel Java Code," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '13, 2013, pp. 713–728.

[30] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and Rebooting Gprof for the Multicore Age," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, 2011, pp. 458–469.

[31] D. Jeon, S. Garcia, C. Louie, and M. B. Taylor, "Kismet: Parallel Speedup Estimates for Serial Programs," in *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '11, 2011, pp. 519–536.

[32] F. David, G. Thomas, J. Lawall, and G. Muller, "Continuously Measuring Critical Section Pressure with the Free-lunch Profiler," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '14, 2014, pp. 291–307.

[33] T. Yu and M. Pradel, "SyncProf: Detecting, Localizing, and Optimizing Synchronization Bottlenecks," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016, 2016, pp. 389–400.

[34] VisualVM, https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/intro.html.

[35] Oracle, "Java Mission Control," http://www.oracle.com/technetwork/java/javaseproducts/mission-control/index.html.

[36] ——, "HPROF: A Heap/CPU Profiling Tool," http://docs.oracle.com/javase/8/docs/technotes/samples/hprof.html.

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

[38] YourKit, https://www.yourkit.com.

[39] 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 Http://Hpctoolkit.Org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010.

[40] Intel, "Intel VTune Amplifier 2017," https://software.intel.com/en-us/intel-vtune-amplifier-xe.

[41] AMD, "CodeAnalyst for Linux," http://developer.amd.com/tools-and-sdks/archive/amd-codeanalyst-performance-analyzer-for-linux/.