Optimization Coaching for Fork/Join Applications on the Java Virtual Machine

Extended abstract

Eduardo Rosales
Università della Svizzera italiana
Lugano, Switzerland
rosale@usi.ch

Walter Binder*
Università della Svizzera italiana
Lugano, Switzerland
walter.binder@usi.ch

ABSTRACT

In the multicore era, developers are increasingly writing parallel applications to leverage the available computing resources and to achieve speedups. Still, developing and tuning parallel applications to exploit the hardware resources remains a major challenge. We tackle this issue for *fork/join applications* running in a single Java Virtual Machine (JVM) on a shared-memory multicore.

Developing fork/join applications is often challenging since failing in balancing the tradeoff between maximizing parallelism and minimizing overheads, along with maximizing locality and minimizing contention, may lead to applications suffering from several performance problems (e.g., excessive object creation/reclaiming, load imbalance, inappropriate synchronization).

In contrast to the manual experimentation commonly required to tune fork/join parallelism on the JVM, we propose coaching developers towards optimizing fork/join applications by automatically diagnosing performance issues on such applications and further suggest concrete code modifications to fix them.

CCS CONCEPTS

• **General and reference → Metrics**; • **Software and its engineering → Software performance**;

KEYWORDS

Performance analysis, fork/join applications, optimization coaching, software optimization, Java Virtual Machine

ACM Reference Format:
Eduardo Rosales and Walter Binder. 2018. Optimization Coaching for Fork/Join Applications on the Java Virtual Machine: Extended abstract. In . ACM, New York, NY, USA, 4 pages.

1 INTRODUCTION

Recent hardware advances have brought shared-memory multicores to the mainstream, increasingly encouraging developers to write parallel applications able to utilize the available computing resources. Still, developing and tuning parallel applications to exploit the hardware resources remains challenging.

We tackle this issue for *fork/join applications*, i.e., parallel versions of *divide-and-conquer* algorithms recursively splitting (*fork*) a work into tasks that are executed in parallel, waiting for them to complete, and then typically merging (*join*) results; running in a single Java Virtual Machine (JVM) on a shared-memory multicore.

The *Java fork/join framework* [27] is the implementation enabling fork/join applications on the JVM. It uses the *work-stealing* [57] scheduling strategy for parallel programs, using an implementation where each worker thread has a deque (i.e., a double-ended queue) of tasks and processes them one by one until the deque is empty, in which case the thread *"steals"* by taking out tasks from deques belonging to other active worker threads. Since the release of Java 8, the Java fork/join framework has been supporting the Java library in the `java.util.Array` [37] class on methods performing parallel sorting, in the `java.util.streams` package [40], which provides a sequence of elements supporting sequential and parallel aggregate operations, and in the `java.util.concurrent.-CompletableFuture<T>` [38] class which extends the functionalities provided by Java's Future API [39] to develop asynchronous programs in Java. Moreover, the Java fork/join framework supports thread management for other JVM languages [4, 16, 46] and has been increasingly used in supporting fork/join applications based on Actors [28, 42] and MapReduce [5, 20, 53], to mention some.

Despite the features offered by the Java fork/join framework to developers targeting the JVM, writing efficient fork/join applications remains challenging. While design patterns have been proposed [26] to guide developers in declaring fork/join parallelism, there is no unique optimal implementation that best resolves the tradeoff between maximizing parallelism and minimizing overheads, along with maximizing locality and minimizing contention. Failing in balancing such conflicting goals in the development process may lead to fork/join applications suffering from several performance issues (e.g., excessive deque accesses and object creation/reclaiming, load imbalance, inappropriate synchronization) that may overshadow the performance gained by parallel execution.

Inspired by *Optimization Coaching* [51, 52], a technique focused on automatically providing the developer with information to facilitate both detecting issues and achieving potential optimizations on a program, we propose coaching developers towards optimizing fork/join applications by automatically diagnosing performance issues on such applications and further suggest concrete code refactoring to solve them. To this end, we devise a tool generating recommendations that should only require source-level knowledge (i.e., knowledge of the high-level programming language in which the target application is implemented) to be straightforwardly interpreted and implemented by the developer.

Although several works and tools address the assisted optimization of sequential applications [15, 51, 52], parallelism discovery [14, 17, 19, 25, 45, 49, 50], parallelism profiling [1, 2, 11, 23, 36, 54, 56, 59],

*PhD advisor

EuroDW'18, April 2018, Porto, Portugal
.

or have analyzed the use of concurrency on the JVM [29–31, 44], with some even targeting fork/join parallelism [9, 43], we are not aware of previous works on coaching a developer towards optimizing a fork/join application running in the JVM.

The remainder of this proposal is organized as follows. In Section 2, we discuss our research directions. In Section 3, we analyze the feasibility of our proposal by reviewing related work.

2 OPTIMIZATION COACHING FOR FORK/JOIN APPLICATIONS ON THE JVM

We propose coaching developers towards the optimization of fork/-join applications on the JVM. To achieve our goal it is imperative to address the following research directions.

Diagnosing performance issues. We plan to build a methodology aimed at diagnosing issues and understanding its impact on the performance of fork/join applications on the JVM. To this end, first we plan to define a new model to characterize fork/join tasks.

Second, we plan to determine the entities and metrics worth to consider to automatically detect performance drawbacks on fork/join applications. We plan to use static analysis to inspect the source code in the search of fork/join anti patterns [9, 43]. Complementary, we plan to use dynamic analysis to deal with polymorphism and reflection, along with collecting performance metrics enabling diagnosing performance issues noticeable at runtime (e.g., excessive deque accesses and object creation/reclaiming, load imbalance, inappropriate synchronization). To this end, we devise a vertical profiler [18] collecting information from the full system stack, including metrics at the application level (e.g., tasks and threads), Java fork/join framework level (e.g., task submission, the number of already queued tasks), JVM level (e.g., garbage collections, allocations in Java Heap), operating system level (e.g., CPU usage, memory load) and hardware level (e.g., reference cycles, machine instructions, context switches, cache misses).

Third, we plan to profile and characterize all tasks spawned by a fork/join application to reveal performance drawbacks related to suboptimal *task granularity*, a key factor when implementing fork/join parallelism on the JVM as pointed out by Lea [26]. This is challenging since recursion, fine-grained parallelism, and scheduling affecting the life-cycle of a fork/join task (e.g., the *steal* operation), complicate task granularity profiling and may lead to incorrect measurements when not handled properly. Our profiler will rely on DiSL [33] to ensure full coverage of the tasks spawned by a fork/join application, including those in the Java class library.

Finally, the metrics collected can be susceptible to perturbations caused by the inserted instrumentation code, which may bias the analysis. Therefore, we plan to lower the perturbation of dynamic metrics by using efficient profiling data structures avoiding heap allocations in the target application [32].

In previous studies [47, 48] we analyzed the tradeoff between maximizing parallelism and minimizing overheads, mainly focusing on the size of the tasks spawned by task-parallel applications on the JVM. We introduced an early model to define a task as every instance of the Java interfaces `java.lang.Runnable` and `java.util.-concurrent.Callable`, focusing on the study of *Java thread pools* [41]. Furthermore, we presented *tgp* [48], a task-granularity profiler for the JVM. Relying on bytecode instrumentation [33] and resorting to Hardware Performance Counters (HPC) [22], *tgp* allowed us finding suboptimal task granularities causing performance issues in the DaCapo [6] benchmark suite. We plan to extend this work to study suboptimal task granularity on fork/join applications on the JVM.

Suggesting optimizations. We plan to develop a methodology for automatically generating (and reporting to the developer) specific recommendations aimed at fixing detected performance issues on fork/join applications. Inspired by the works of De Wael et al. [9] and Pinto et al. [43], we plan to develop a tool able to recognize fork/join anti-patterns, but it is also of paramount importance in our aim to automatically match such anti-patterns to concrete recommendations that a developer can use to avoid them. We plan to use *calling context profiling* [3] to pinpoint the developer to application code where tasks classified as problematic were created, submitted, or executed, guiding optimizations by means of suggesting the use of fork/join design patterns [26] and concrete code modifications to enable potential missed parallelization opportunities. This is challenging, since automatically matching performance issues with concrete code suggestions to fix them will require the use of advance data analysis techniques. In a previous study [47] we used calling contexts to locate tasks previously identified as of suboptimal granularity in DaCapo, guiding optimizations through code refactoring that led to noticeable speedups. Based on this work, we plan to automate and specialize such techniques to specifically coach developers in optimizing fork/join applications on the JVM.

Validating optimizations. As previously pointed out by De Wael et al. [9], there is a lack of standard benchmarks suitable to perform empirical studies focused on fork/join parallelism on the JVM. Indeed, our early work mentioned above relied on DaCapo to study suboptimal task granularities mainly in Java thread pools, however, this suite does not have any fork/join application. In this direction, we plan to discover workloads exhibiting high diversity in their task-parallel behavior by plugging *tgp* to *AutoBench* [60], a toolchain combining massive code repository crawling, pluggable hybrid analyses, and workload characterization techniques to discover candidate workloads satisfying the needs of domain-specific benchmarking. By fully integrating *tgp* and *AutoBench*, we plan to discover workloads exhibiting diverse fork/join parallelism, suitable for validating both aforementioned methodologies.

3 RELATED WORK

This section first introduces related studies on assisted optimization of applications. Next, we review previous work analyzing the use of concurrency on the JVM. Then, we focus on parallelism discovers. Finally, we review profilers for parallel applications.

Assisted optimization of applications. Several studies have focused on providing the developer with feedback to assist the optimization of several aspects of an application. Here, we relate three of them pioneering an approach that is very close to our goal in providing recommendations which require only source-level knowledge to be implemented by a developer.

St-Amour et al. [51, 52] first coined the term *Optimization Coaching*, introducing a technique modifying the compiler's optimizer to inform the developer which optimizations it performs, which optimizations it misses, and suggesting specific changes to the source code that could trigger additional optimizations. The goal is reporting to the developer precise recommendations which interpretation do not require expertise about the compiler's internals and which implementation enables a missed or failed optimization attempted by the compiler. The technique was first prototyped in a work [52] modifying Racket's [12] simple ahead-of-time byte-compiler which performs basic optimizations on Racket applications. In a second work [51], the authors extended this approach to consider dynamic object oriented just-in-time compiled programming languages, prototyping assisted optimizations in the SpiderMonkey JavaScript [34] engine.

Gong et al. [15] present JITProf, a prototype profiling framework that detects code patterns prohibiting optimizations on JavaScript Just-In-Time (JIT) compilers and reports their occurrences to the developer in the goal of achieving optimizations. JITProf profiles the target application at runtime implementing different strategies to detect seven *JIT-unfriendly* patterns. The goal is reporting to the developer JIT-unfriendly locations ranked by their relevancy on preventing optimizations to facilitate the developer fixing them.

The above works shred light in the goal of mentoring developers in locating performance drawbacks and in providing concrete recommendations to optimize an application. Nonetheless, they notably differ from our goal because the related tools were not designed for diagnosing and further recommending modifications specifically aimed at optimizing parallel applications. Indeed, the techniques proposed focus on feedback generated by specific compiler's optimizers, falling short in considering other metrics from the full system stack which may enable revealing issues specific to parallel applications, including fork/join applications on the JVM.

Analyses of the use of concurrency on the JVM. Numerous studies have focused on the use of concurrency on the JVM. Here, we highlight two works closely related to our aim.

De Wael et al. [9] analyzed 120 open-source Java projects to study the use of the Java fork/join framework, confirming manually the frequent use of four best coding practices and three recurring anti-patterns that potentially limit parallel performance. Their aim is providing practical conclusions to guide language or framework designers to propose new or improved abstractions that steer programmers toward using the right patterns and away from using the anti-patterns in developing fork/join applications for the JVM.

Pinto et al. [43] study parallelism bottlenecks in fork/join applications on the JVM, with a unique focus on how they interact with underlying system-level features, such as work-stealing scheduling and memory management. They identify six bottlenecks impacting performance and energy efficiency on the AKKA [28] actor framework and conduct an in-depth refactoring on it to improve its core messaging engine. They also present FJDETECTOR, a plugin for the Eclipse IDE [55] prototyping the detection of some bottlenecks identified by the authors by means of code inspection.

Overall, these works bring meaningful insight in methodologies focused on detecting bad coding practices and bottlenecks in real fork/join applications on the JVM. Complementary to such studies, we plan to focus on both performance metrics collected from the full system stack via dynamic analysis to automatically diagnose performance issues noticeable only at runtime and in characterizing all the tasks spawned by the fork/join application to study performance drawbacks related to suboptimal task granularity. Moreover, we specifically devise coaching developers towards optimizing fork/join applications by providing them with code refactoring suggestions which implementation may enable missed parallelization opportunities.

Parallelism discovers. A number of tools has been developed to discover parallelization opportunities. Most notably, tools focusing on parallelism discovery include Kremlin [14], which tracks loops and dependencies to pinpoint sequential parts of a program that can be parallelized. Kismet [25] builds on Kremlin and enhances the prediction of potential speedups after parallelization, given a target machine and runtime system. Similarly, Hammacher et al. [17], Rountev et al. [49], and Dig et al. [10] focus on refactoring sequential legacy Java programs identifying independent computation paths that could have been parallelized and recommend locations to the programmer with the highest potential for parallelization. Although the above works aim at guiding the developer towards modifications enabling parallelization, they only target the optimization of sequential applications.

In addition, several parallelism profilers based on the *work-span model* [7, 24] target parallel applications. Cilkview [19] builds upon this model to predict achievable speedup bounds for a Cilk [13] application when the number of used cores increases. CilkProf [50] extends Cilkview by measuring *work* and *span* on each call site to determine which of them constitutes a bottleneck towards parallelization. TaskProf [58] computes work, span, and the *asymptotic parallelism* of C++ applications using the Intel Threading Building Blocks (TBB) [45] task-parallel library, relying on causal profiling [8] techniques to estimate parallelism improvements. Despite these tools allow detecting bottlenecks on parallel applications and can estimate potential speedups, they do not focus on coaching a developer towards optimizing a fork/join application. Moreover, none of these tools supports the JVM.

Parallelism profilers. Researchers from industry and academia have developed several tools to analyze various parallel characteristics of an application. HPCToolkit [1] is a suite of tools using statistical sampling of timers and HPC collection to analyze the performance of an application, resource consumption, and inefficiency, attributing them to the full calling context in which they occur. THOR [54] is a tool for performance analysis of Java applications on multicore systems. THOR relies on vertical profiling to collect fine-grained events, such as OS context switches and Java lock contention from multiple layers of the execution stack. Each event is subsequently associated with a thread, providing a detailed graphical report on the behavior of threads spawned in the execution of a parallel Java application.

Finally, a number of tools exist to support the analysis of parallel Java applications through the collection of metrics at the application layer, most notably including VisualVM [56], IBM HealthCenter [21], Oracle Developer Studio Performance Analyzer [36], Java Mission Control [35], JProfiler [11], and YourKit [59], or at the hardware layer resorting to HPCs, including Intel VTune [23] and AMD CodeAnalyst [2], among others.

Overall, despite the aforementioned tools provide insight in characterizing processes or threads over time, none of them specializes on fork/join applications, and thus they fall short in diagnosing performance problems specific to this type of parallel applications.

ACKNOWLEDGMENTS

The research presented in this paper was supported by Oracle (ERO project 1332) and by the Swiss National Science Foundation (project 200021_153560).

REFERENCES

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

[2] AMD. 2017. CodeAnalyst for Linux. http://developer.amd.com/tools-and-sdks/archive/amd-codeanalyst-performance-analyzer-for-linux/. (2017).

[3] Glenn Ammons, Thomas Ball, and James R. Larus. 1997. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *PLDI '97*. 85–96.

[4] Apache Groovy project. 2018. Groovy. http://www.groovy-lang.org/. (2018).

[5] Colin Barrett, Christos Kotselidis, and Mikel Luján. 2016. Towards Co-designed Optimizations in Parallel Frameworks: A MapReduce Case Study. In *CF*. 172–179.

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

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.

[8] Charlie Curtsinger and Emery D. Berger. 2015. Coz: Finding Code That Counts with Causal Profiling. In *SOSP*. 184–197.

[9] Mattias De Wael, Stefan Marr, and Tom Van Cutsem. 2014. Fork/Join Parallelism in the Wild: Documenting Patterns and Anti-patterns in Java Programs Using the Fork/Join Framework. In *PPPJ*. 39–50.

[10] Danny Dig, John Marrero, and Michael D. Ernst. 2009. Refactoring Sequential Java Code for Concurrency via Concurrent Libraries. In *ICSE (ICSE)*. 397–407.

[11] ej-technologies. 2017. JProfiler. https://www.ej-technologies.com/products/jprofiler/overview.html. (2017).

[12] Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

[13] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI*. 212–223.

[14] Saturnino Garcia, Donghwan Jeon, Christopher M. Louie, and Michael Bedford Taylor. 2011. Kremlin: Rethinking and Rebooting Gprof for the Multicore Age. In *PLDI*. 458–469.

[15] Liang Gong, Michael Pradel, and Koushik Sen. 2015. JITProf: Pinpointing JIT-unfriendly JavaScript Code. In *ESEC/FSE 2015*. 357–368.

[16] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-based and Event-based Programming. *Theor. Comput. Sci.* 410, 2-3 (2009), 202–220.

[17] C. Hammacher, K. Streit, S. Hack, and A. Zeller. 2009. Profiling Java Programs for Parallelism. In *ICSE'09*. 49–55.

[18] Matthias Hauswirth, Peter F. Sweeney, Amer Diwan, and Michael Hind. 2004. Vertical Profiling: Understanding the Behavior of Object-priented Applications. In *OOPSLA*. 251–269.

[19] Yuxiong He, Charles E. Leiserson, and William M. Leiserson. 2010. The Cilkview Scalability Analyzer. In *SPAA*. 145–156.

[20] Zhang Hong, Wang Xiao-ming, Cao Jie, Ma Yan-hong, Guo Yi-rong, and Wang Min. 2016. An Optimized Model for MapReduce Based on Hadoop. *TELKOMNIKA* 14 (12 2016), 1552.

[21] IBM. [n. d.]. IBM HealthCenter. https://www.ibm.com/developerworks/java/jdk/tools/healthcenter/. ([n. d.]).

[22] ICL. 2017. PAPI. http://icl.utk.edu/papi/. (2017).

[23] Intel. 2017. Intel VTune Amplifier. https://software.intel.com/en-us/intel-vtune-amplifier-xe. (2017).

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

[25] Donghwan Jeon, Saturnino Garcia, Chris Louie, and Michael Bedford Taylor. 2011. Kismet: Parallel Speedup Estimates for Serial Programs. In *OOPSLA*. 519–536.

[26] Doug Lea. 1999. *Concurrent Programming in Java. Second Edition: Design Principles and Patterns* (2nd ed.). Addison-Wesley Professional.

[27] Doug Lea. 2000. A Java Fork/Join Framework. In *JAVA*. 36–43.

[28] Lightbend Inc. [n. d.]. Akka. http://akka.io. ([n. d.]).

[29] Y. Lin and D. Dig. 2013. CHECK-THEN-ACT Misuse of Java Concurrent Collections. In *ICST'13*. 164–173.

[30] Y. Lin, S. Okur, and D. Dig. 2015. Study and Refactoring of Android Asynchronous Programming (T). In *ASE*. 224–235.

[31] Yu Lin, Cosmin Radoi, and Danny Dig. 2014. Retrofitting Concurrency for Android Applications Through Refactoring. In *FSE*. 341–352.

[32] Lukáš Marek, Stephen Kell, Yudi Zheng, Lubomír Bulej, Walter Binder, Petr Tůma, Danilo Ansaloni, Aibek Sarimbekov, and Andreas Sewe. 2013. ShadowVM: Robust and Comprehensive Dynamic Program Analysis for the Java Platform. In *GPCE*. 105–114.

[33] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *AOSD*. 239–250.

[34] Mozilla. 2018. SpiderMonkey. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey. (2018).

[35] Oracle. [n. d.]. Java Mission Control. http://www.oracle.com/technetwork/java/javaseproducts/mission-control/java-mission-control-1998576.html. ([n. d.]).

[36] Oracle. [n. d.]. Solaris Studio Performance Analyzer. http://www.oracle.com/technetwork/server-storage/solarisstudio/features/performance-analyzer-2292312.html. ([n. d.]).

[37] Oracle. 2017. Class Arrays. https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html. (2017).

[38] Oracle. 2017. Class CompletableFuture<T>. https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html. (2017).

[39] Oracle. 2017. Interface Future<V>. https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Future.html. (2017).

[40] Oracle. 2017. Package java.util.stream. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html. (2017).

[41] Oracle. 2017. Thread Pools. https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html. (2017).

[42] Patrick Peschlow. 2012. The Fork/join Framework in Java 7. http://www.h-online.com/developer/features/The-fork-join-framework-in-Java-7-1762357.html. (2012).

[43] Gustavo Pinto, Anthony Canino, Fernando Castor, Guoqing Xu, and Yu David Liu. 2017. Understanding and Overcoming Parallelism Bottlenecks in ForkJoin Applications. In *ASE 2017*. 765–775.

[44] Gustavo Pinto, Weslley Torres, Benito Fernandes, Fernando Castor, and Roberto S.M. Barros. 2015. A Large-scale Study on the Usage of Java's Concurrent Programming Constructs. *Journal of Systems and Software* 106 (2015), 59 – 81.

[45] James Reinders. 2007. *Intel Threading Building Blocks* (1st ed.). O'Reilly & Associates, Inc.

[46] Rich Hickey. 2018. The Clojure Programming Language. https://clojure.org//. (2018).

[47] Andrea Rosà, Eduardo Rosales, and Walter Binder. 2018. Analyzing and Optimizing Task Granularity on the JVM. In *CGO*. 1–11.

[48] Eduardo Rosales, Andrea Rosà, and Walter Binder. 2017. tgp: a Task-Granularity Profiler for the Java Virtual Machine. In *APSEC*. 1–6.

[49] A. Rountev, K. Van Valkenburgh, Dacong Yan, and P. Sadayappan. 2010. Understanding Parallelism-Inhibiting Dependences in Sequential Java Programs. In *ICSM*. 1–9.

[50] Tao B. Schardl, Bradley C. Kuszmaul, I-Ting Angelina Lee, William M. Leiserson, and Charles E. Leiserson. 2015. The Cilkprof Scalability Profiler. In *SPAA*. 89–100.

[51] V. St-Amour and S. Guo. 2015. Optimization Coaching for Javascript. In *ECOOP*.

[52] Vincent St-Amour, Sam Tobin-Hochstadt, and Matthias Felleisen. 2012. Optimization Coaching: Optimizers Learn to Communicate with Programmers. In *OOPSLA*. 163–178.

[53] Robert Stewart and Jeremy Singer. 2018. Comparing Fork/Join and MapReduce. (2018).

[54] Q. M. Teng, H. C. Wang, Z. Xiao, P. F. Sweeney, and E. Duesterwald. 2010. THOR: a Performance Analysis Tool for Java Applications Running on Multicore Systems. *IBM Journal of Research and Development* 54, 5 (2010), 4:1–4:17.

[55] The Eclipse Foundation. 2016. Eclipse IDE. http://www.eclipse.org/ide. (2016).

[56] VisualVM. [n. d.]. https://visualvm.java.net. ([n. d.]).

[57] F Warren Burton and Michael Sleep. 1981. Executing Functional Programs on a Virtual Tree of Processors. *FPCA* (1981), 187–194.

[58] Adarsh Yoga and Santosh Nagarakatte. 2017. A Fast Causal Profiler for Task Parallel Programs. In *ESEC/FSE*. 15–26.

[59] YourKit. 2017. YourKit. https://www.yourkit.com. (2017).

[60] Yudi Zheng, Andrea Rosà, Luca Salucci, Yao Li, Haiyang Sun, Omar Javed, Lubomir Bulej, Lydia Y. Chen, Zhengwei Qi, and Walter Binder. 2016. AutoBench: Finding Workloads That You Need Using Pluggable Hybrid Analyses. In *SANER*. 639–643.