Renaissance: Benchmarking Suite for Parallel Applications on the JVM

Aleksandar Prokopec
Oracle Labs
Switzerland
aleksandar.prokopec@oracle.com

Andrea Rosà
Università della Svizzera italiana
Switzerland
andrea.rosa@usi.ch

David Leopoldseder
Johannes Kepler Universität Linz
Austria
david.leopoldseder@jku.at

Gilles Duboscq
Oracle Labs
Switzerland
gilles.m.duboscq@oracle.com

Petr Tůma
Charles University
Czech Republic
petr.tuma@d3s.mff.cuni.cz

Martin Studener
Johannes Kepler Universität Linz
Austria
martinstudener@gmail.com

Lubomír Bulej
Charles University
Czech Republic
bulej@d3s.mff.cuni.cz

Yudi Zheng
Oracle Labs
Switzerland
yudi.zheng@oracle.com

Alex Villazón
Universidad Privada Boliviana
Bolivia
avillazon@upb.edu

Doug Simon
Oracle Labs
Switzerland
doug.simon@oracle.com

Thomas Würthinger
Oracle Labs
Switzerland
thomas.wuerthinger@oracle.com

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

Abstract

Established benchmark suites for the Java Virtual Machine (JVM), such as DaCapo, ScalaBench, and SPECjvm2008, lack workloads that take advantage of the parallel programming abstractions and concurrency primitives offered by the JVM and the Java Class Library. However, such workloads are fundamental for understanding the way in which modern applications and data-processing frameworks use the JVM's concurrency features, and for validating new just-in-time (JIT) compiler optimizations that enable more efficient execution of such workloads. We present *Renaissance*, a new benchmark suite composed of modern, real-world, concurrent, and object-oriented workloads that exercise various concurrency primitives of the JVM. We show that the use of concurrency primitives in these workloads reveals optimization opportunities that were not visible with the existing workloads. We use Renaissance to compare performance of two state-of-the-art, production-quality JIT compilers (HotSpot C2 and Graal), and show that the performance differences are more significant than on existing suites such as DaCapo and SPECjvm2008. We also use Renaissance to expose four new compiler optimizations, and we analyze the behavior of several existing ones. Evaluating these optimizations using four benchmark suites shows a more prominent impact on the Renaissance workloads than on those of other suites.

CCS Concepts • **General and reference** → **Empirical studies**; **Evaluation**; *Metrics*; • **Software and its engineering** → **Just-in-time compilers**; **Dynamic compilers**; **Runtime environments**; *Object oriented languages*; *Functional languages*; • **Computing methodologies** → *Parallel programming languages*; *Concurrent programming languages.*

Keywords benchmarks, JIT compilation, parallelism, concurrency, JVM, object-oriented programming benchmarks, functional programming benchmarks, Big Data benchmarks

ACM Reference Format:
Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. 2019. Renaissance: Benchmarking Suite for Parallel Applications on the JVM. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19), June 22–26, 2019, Phoenix, AZ, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3314221.3314637

PLDI '19, June 22–26, 2019, Phoenix, AZ, USA

ACM ISBN 978-1-4503-6712-7/19/06…$15.00
https://doi.org/10.1145/3314221.3314637

1 Introduction

To drive innovation in managed runtimes and just-in-time (JIT) compilers, researchers and practitioners rely on benchmarks suites that capture representative patterns of application behavior. This allows developing and validating new optimizations and memory management techniques. Even though an improvement demonstrated on a benchmark is not always a sufficient condition, it is usually a necessary condition for a new optimization or technique to be considered useful. The Java Virtual Machine (JVM) is at the forefront of managed runtime research and development, and due to its popularity and widespread use, many benchmarking suites targeting the JVM have been developed. Of those most established, the DaCapo benchmark suite introduced a set of workloads to help understand memory behavior of complex Java applications [21]. The ScalaBench suite [100] focused on Scala programs, which exhibit a significantly different behavior compared to Java programs [99], despite Scala being compiled to Java bytecode and executed on the JVM. The SPECjvm2008 focused on computationally intensive workloads [1], while the more recent SPECjbb2015 benchmark simulates the IT infrastructure of an online supermarket [4].

Still, as we argue in this paper, neither of the existing benchmark suites for the JVM was specifically focused on concurrency and parallelism. Moreover, the JVM has evolved considerably over the years, and gained support for atomic operations [54], lock-free concurrent data structures [51], and non-blocking I/O [19], as well as new language and JVM features such as Java Lambdas [36], `invokedynamic` and method handles [97], to name a few. At the same time, the arrival of multicore processors prompted a plethora of new programming models and frameworks, such as software transactional memory [24, 41, 101], fork-join [50], data-parallel collections [33, 81, 111], and actors [5, 39, 105]. These workloads potentially present new optimization opportunities for compilers and virtual machines.

This paper proposes a new representative set of benchmarks that cover modern JVM concurrency and parallelism paradigms. While we do not argue that the proposed benchmark suite is exhaustive in that it represents all relevant workloads or optimization opportunities, we have found these benchmarks useful, since they helped us identify, implement and assess new compiler optimizations. As we show in the paper, three other established benchmark suites did not exhibit similar characteristics.

The contributions in this work are as follows:

- We propose a new benchmark suite, which consists of 21 parallelism- and concurrency-oriented benchmarks for the JVM, and relies on multiple existing state-of-the-art Java and Scala frameworks (Section 2).
- We identify a set of metrics that reflect the usage of the basic concurrency primitives on the JVM, along with a normalization technique (Section 3).
- We show that the proposed benchmarks are diverse in the sense that they span the space of concurrency metrics better than the existing suites (Section 4).
- We demonstrate that the proposed benchmarks reveal new opportunities for JIT compilers by implementing four new optimizations in the Graal JIT compiler [32] (Section 5).
- We evaluate the impact of the four new optimizations, plus three pre-existing optimizations, on Renaissance, DaCapo, ScalaBench and SPECjvm2008 suites, showing that all 7 optimizations exhibit over 5% impact on Renaissance, compared to only 2 on ScalaBench, 1 on DaCapo, and 3 on SPECjvm2008 (Section 6).
- We show that the proposed benchmark suite is as complex as DaCapo and ScalaBench, and much more complex than SPECjvm2008, by evaluating six Chidamber and Kemerer metrics, and by inspecting the compiled code size and the hot method count of all the benchmarks (Section 7),

We present related work in Section 8, and we summarize our observations in Section 9.

2 Motivation and Benchmark Selection

Our work has several motivating factors. First, we realized that the state-of-the-art JIT compilers, such as C2 and Graal, have very comparable performance, e.g., on the DaCapo suite, and that it has become extremely difficult to demonstrate significant performance gains on the existing workloads when exploring new compiler optimizations. Second, we determined that many modern programming abstractions, such as Java Lambdas [36], Streams [33], or Futures [53], were not represented in the existing benchmark suites. Moreover, we found that the optimizations in existing JIT compilers rarely focused on concurrency-related primitives.

2.1 Selection Methodology

To identify a core set of benchmarks which would reflect the use of modern concurrency-related abstractions and programming paradigms, we manually collected a large number of candidate workloads using popular Java and Scala frameworks. In addition to manual filtering, we also developed a tool, inspired by the AutoBench toolchain [113], to scan the online corpus of open-source projects on GitHub[1] for candidate workloads. When selecting benchmarks for the Renaissance suite, our main goals were as follows.

Modern Concurrency. Benchmarks in the suite should rely on different styles of concurrent and parallel programming. These include data-parallelism, task-parallelism, streaming

[1] We included GitHub projects with updates in the last 3 years (2015–2017) and higher number of recent commits, pull requests, and number of stars.

and pipelined parallelism, message-based concurrency, software transactional memory, lock-based and lock-free concurrent data structures and in-memory databases, as well as asynchronous programming and network communication.

Realistic Workloads. Benchmarks should reflect real-world applications. To this end, we focused on workloads using popular Java and Scala frameworks, such as Java Stream API [33], Apache Spark [111], the Java Reactive Extensions [15, 16], the Java Fork/Join framework [50], ScalaSTM [22], and Twitter Finagle [12]. We reused existing workloads where possible, and we adopted several standalone benchmarks that target specific paradigms, such as STMBench7 [38], J. Paumard's Scrabble benchmark [69], and the CloudSuite MovieLens benchmark for Apache Spark [35]. Overall, we collected roughly 100 benchmarks since starting the effort in 2016.

Workload Diversity. Benchmarks should be sufficiently diverse so that they exercise different concurrency-related features of the JVM, such as atomic instructions, Java `synchronized` blocks, thread-park operations, or guarded blocks (i.e., the `wait-notify` pattern). At the same time, the benchmarks should be object-oriented to exercise the parts of the JVM responsible for efficient execution of code patterns commonly associated with abstraction in object-oriented programs, i.e., frequent object allocation and virtual dispatch. To this end, we analyzed candidate projects by running their testing code and recording various concurrency-related metrics (details in Section 3) to ensure adequate coverage of the metric space. This served as the basis for the diversity analysis in Section 4.

Deterministic Execution. Benchmarks should run deterministically, even though it is not possible to achieve full determinism in concurrent benchmarks due to non-determinism inherent to thread scheduling. However, apart from that, the control flow of a benchmark should not include non-deterministic choices on data produced, e.g., by a random generator seeded by the current time instead of a constant.

Open-Source Availability. Benchmarks should be open-source, and rely only on open-source frameworks, whenever possible. This goal is important for several reasons. First of all, it enables inspection of the workloads by the community and allows multiple parties to collaborate on improving and maintaining the suite so that it can evolve along with the target platform ecosystem. Moreover, it enables source-code level analysis of the benchmarks, and allows evaluating the actionability of profiler results. And last, but not least, open-source software is less likely to cause licensing issues.

Avoiding Known Pitfalls. The benchmarks should avoid pitfalls that were identified in other benchmark suites over time and which hinder adoption in a broader community. The lack of benchmark source code is one such pitfall. Another pitfall is the use of timeouts, which make it difficult to analyze the benchmark execution using dynamic analysis tools that require heavy-weight instrumentation, because the timeouts keep triggering due to high overhead. Yet another pitfall is benchmark correctness, avoiding benchmarks with resource leaks and, especially in the case of concurrent benchmarks, data races and deadlocks.

2.2 Renaissance Benchmarks and Harness

The set of benchmarks that we ultimately decided to include in the Renaissance suite is shown in Table 1, along with short descriptions. In total, 14 out of 21 benchmarks were adapted from existing standalone benchmarks (*akka-uct*, *als*, *chi-square*, *dec-tree*, *fj-kmeans*, *log-regression*, *movie-lens*, *naive-bays*, *philosophers*, *reactors-savina*, *rx-scrabble*, *scrabble*, *stm-bench7* and *streams-mnemonics*), and the rest were gathered from production usages of different companies, and usages in existing applications.

All benchmarks run within a single JVM process, and only rely on the JDK as an external dependency. Some of the benchmarks use network communication, and they are encoded as multiple threads that exercise the network stack within a single process (using the loopback interface). We realize that this does not reflect realistic network conditions, but it does exercise code responsible for spanning multiple address spaces, which is common for data-parallel computational frameworks. The default execution time of each benchmark is tuned to take several seconds.

We provide a harness that allows to run the benchmarks and collect the results, and also allows to easily add new benchmarks. The harness also provides an interface for custom measurement plugins, which can latch onto benchmark execution events to perform additional operations. For example, most of the metrics in Section 3 were collected using custom plugins. In addition, the harness allows running the benchmarks with JMH [102] (a standard microbenchmarking tool for the JVM) as a frontend to avoid common measurement pitfalls associated with benchmarking.

3 Characterizing Metrics

To meet the goals outlined in Section 2.1, we define a set of metrics to characterize the usage of concurrency primitives, basic primitives of object-oriented programming, and modern programming primitives introduced in JDK 7 or later. Notably, our goal is not to define an exhaustive set of metrics to cover all of these features. Instead, we aim at profiling metrics that 1) can reasonably characterize the use of these features, 2) can be collected with simple instrumentation, and 3) can be easily understood.

In this section, we present the metrics used in benchmark selection, and the details of how we collect them. Subsequent sections compare these metrics across different benchmarks and relate them with compiler optimizations.

3.1 Description of the Metrics

Table 2 presents the metrics used during benchmark selection. The initial several metrics in the list reflect the usage of some basic concurrency primitives on the JVM [94]:

Table 1. Summary of benchmarks included in Renaissance.

Benchmark	Description	Focus
akka-uct	Unbalanced Cobwebbed Tree computation using Akka [5].	actors, message-passing
als	Alternating Least Squares algorithm using Spark.	data-parallel, compute-bound
chi-square	Computes a Chi-Square Test in parallel using Spark ML [62].	data-parallel, machine learning
db-shootout	Parallel shootout test on Java in-memory databases.	query-processing, data structures
dec-tree	Classification decision tree algorithm using Spark ML [62].	data-parallel, machine learning
dotty	Compiles a Scala codebase using the Dotty compiler for Scala.	data-structures, synchronization
finagle-chirper	Simulates a microblogging service using Twitter Finagle [12].	network stack, futures, atomics
finagle-http	Simulates a high server load with Twitter Finagle [12] and Netty [11].	network stack, message-passing
fj-kmeans	K-means algorithm using the Fork/Join framework [50].	task-parallel, concurrent data structures
future-genetic	Genetic algorithm function optimization using Jenetics [9].	task-parallel, contention
log-regression	Performs the logistic regression algorithm on a large dataset.	data-parallel, machine learning
movie-lens	Recommender for the MovieLens dataset using Spark ML [62].	data-parallel, compute-bound
naive-bayes	Multinomial Naive Bayes algorithm using Spark ML [62].	data-parallel, machine learning
neo4j-analytics	Analytical queries and transactions on the Neo4J database [10].	query processing, transactions
page-rank	PageRank using the Apache Spark framework [111].	data-parallel, atomics
philosophers	Dining philosophers using the ScalaSTM framework [22].	STM, atomics, guarded blocks
reactors	A set of message-passing workloads encoded in the Reactors framework [90].	actors, message-passing, critical sections
rx-scrabble	Solves the Scrabble puzzle [69] using the RxJava framework.	streaming
scrabble	Solves the Scrabble puzzle [69] using Java 8 Streams.	data-parallel, memory-bound
stm-bench7	STMBench7 workload [38] using the ScalaSTM framework [22].	STM, atomics
streams-mnemonics	Computes phone mnemonics [64] using Java 8 Streams.	data-parallel, memory-bound

Table 2. Metrics considered during benchmark selection.

Name	Description
synch	`synchronized` methods and blocks executed.
wait	Invocations of `Object.wait()`.
notify	Invocations of `Object.notify()` and `Object.notifyAll()` .
atomic	Atomic operations executed.
park	Park operations.
cpu	Average CPU utilization (user and kernel).
cachemiss	Cache misses, including L1 cache (instruction and data), last-layer cache (LLC), and translation lookaside buffer (TLB; instruction and data).
object	Objects allocated.
array	Arrays allocated.
method	Methods invoked with `invokevirtual`, `invokeinterface` or `invokedynamic` bytecodes.
idynamic	`invokedynamic` bytecodes executed.

(1) `synchronized` blocks and methods (i.e. critical sections), (2) `wait`/`notify` calls (i.e. guarded blocks), (3) atomic memory-access operations (e.g., compare-and-swap), (4) thread parking and unparking. Since most high-level concurrency abstractions are implemented from these basic primitives, their usage rates estimate the use of concurrency abstractions during execution. We therefore collect the dynamic invocation counts of these primitives[2].

Next, we define a metric for the average CPU utilization, and a metric for the number of cache misses during execution. Both these metrics are indirectly related to concurrency – for example, unless several CPUs or cores are utilized by the benchmark, it is unlikely that the benchmark will exhibit interesting synchronization behavior. Similarly, a high cache-miss rate may indicate contention between threads.

Then, we track the object-allocation rate, the array-allocation rate, and the dynamic-dispatch rate (i.e., the execution counts of the `invokedynamic`, `invokeinterface`, or `invokevirtual` bytecodes). These metrics estimate the usage of the basic primitives in object-oriented programming, and usually correlate with the complexity of the code[3].

Finally, we track the execution rate of the `invokedynamic` bytecode (introduced in JDK 7 [97]), which supports dynamic languages on the JVM [66], and is also used to implement Lambdas [36]. Java Lambdas are used in various data-processing libraries – for example, Java Streams [33] or Reactive Extensions [15] expose operations such as `map` and `reduce` that typically take Lambda values as arguments. Therefore, counting the `invokedynamic` occurrences allows estimating the usage rate of high-level operations.

[2]We omitted unparks, since they correlated with parks in all benchmarks.

[3]Some data-parallel and streaming frameworks allocate intermediate arrays.

3.2 Normalization

Absolute values of the metrics from Table 2 are not very indicative, since a high value may reflect a long execution rather than a frequent usage of a given primitive, which is our focus. To this end, we use rates instead of absolute values, and we normalize each metric (except cpu) by the number of *references cycles* executed by the benchmark, defined as machine cycles measured at a constant nominal frequency, even if the real CPU frequency scales up or down [46].

We use reference cycles as a measure of the amount of computations executed by an application. Using reference cycles allows comparing the metrics between different benchmarks, even if they are executed under different or fluctuating CPU frequencies. Moreover, reference cycles represent runtime conditions more accurately, since they account for instruction complexity (as complex instructions take more cycles) and for latencies due to e.g. cache misses.

3.3 Metric Collection

To collect the metrics, we rely on several tools. To profile the usage of concurrency primitives (synch, wait, notify, atomic, park), allocations (object, array), invocations (method), and `invokedynamic` executions (idynamic), we developed a profiler based on bytecode instrumentation, built on top of the open-source dynamic program-analysis framework DiSL[4] [61]. The profiler exploits the full bytecode coverage offered by DiSL (i.e., it can instrument every method that has a bytecode representation), so the metrics can be collected in every loaded class, including the JDK classes, as well as the classes that are dynamically generated or fetched from a remote location. The profiler uses a deployment setting of DiSL called Shadow VM [60] to enforce isolation between analysis and application code, preventing certain well-known problems that may be introduced by less isolated approaches [49]. We profile atomic and park by intercepting the respective method calls in `sun.misc.Unsafe`. CPU utilization is sampled every ~150ms using top [57]. Finally, we use perf [70] to collect cache misses and reference cycles (used for normalization). While instrumentation may influence thread interleaving and bias the collection, we made sure to keep the perturbations low with minimal instrumentation that avoids heap allocations, and by keeping the profiling data structures in a separate process, on a separate NUMA node. Additional details of the experimental setup are in Section B of the supplemental material.

4 Diversity

Having collected the metrics described in Section 3, we used these metrics to narrow down the list of benchmarks, and to compare Renaissance to the established benchmark suites. In this section, we show that Renaissance represents the selected concurrency primitives better than the existing suites, that it is comparable to suites such as DaCapo and ScalaBench in terms of object-allocation rates and dynamic dispatch, and that it exercises `invokedynamic` more often.

4.1 Benchmark Suites

We compare Renaissance with benchmarks included in several established and widely-used benchmark suites for the JVM: DaCapo [21], ScalaBench [100], and SPECjvm2008 [1][5]. These suites can be executed with different input sizes. In DaCapo and ScalaBench, we use the largest input defined for each benchmark, while in SPECjvm2008 we use the "lagom" workload (a fixed-size workload, intended for research purposes [2]). Table 6 in Section A of the supplemental material lists all benchmarks that we considered.

All benchmarks have a *warm-up* phase, which takes either a number of iterations (Renaissance, DaCapo, ScalaBench) or some execution time (SPECjvm2008). Execution after the warmup is classified as *steady-state*, and we always measure steady-state execution in this paper.

4.2 Methodology

To study the differences between Renaissance and other suites, we first collect the metrics defined in Section 3 for all workloads;[6] then, we visually compare them by means of scatter plots. Rather than analyzing benchmarks on a coordinate system composed of 11 dimensions (one dimension for each collected metric) we resort to *principal component analysis (PCA)* [48] to reduce data dimensionality, comparing benchmarks on the resulting coordinate system. PCA is a statistical technique that transforms a K-dimensional space into another space of linearly uncorrelated variables, called *principal components* (PCs). PCA applies linear transformations such that the first PC has the largest possible variance, while each subsequent component has the highest possible variance, under the constraint of being orthogonal to the preceeding components. Considering variance as a measure of information, PCA allows one to reduce the dimensionality of the original dataset by considering only the first components (i.e., those retaining most of the variance) while preserving as much information as possible from the original dataset.

We apply PCA as follows. Matrix $\mathbf{X}$ contains the set of collected metrics. Each component $x_{ij} \in X$ ($i \in [1, N]$; $j \in [1, K]$; where N is the number of benchmarks analyzed and K the number of collected metrics) is the normalized value of metric j obtained on benchmark i. The metrics are profiled during a single steady-state benchmark execution, as discussed in Section B of the supplemental material.

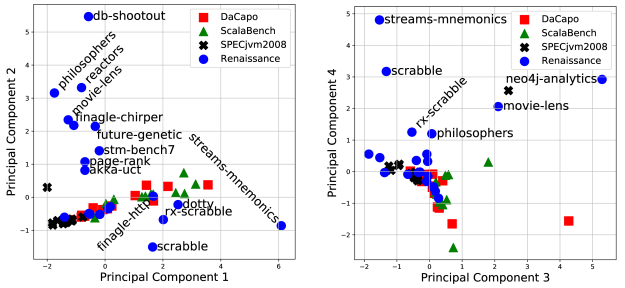
[4] https://disl.ow2.org/

[5] We use the latest released versions of the suites at the time of writing: DaCapo v.9.12-MR1 (dated Jan. 2018), ScalaBench v.0.1.0 (dated Feb. 2012) and SPECjvm2008 v.1.01 (dated May 2009). According to the guidelines on the DaCapo website, we use the *lusearch-fix* benchmark instead of *lusearch*.

[6] Table 7 in Section D of the supplemental material reports the collected unnormalized metrics for all analyzed benchmarks.

Table 3. Loadings of metrics (defined in Table 2) on the first four principal components (PCs), sorted by absolute value (descending order).

PC1		PC2		PC3		PC4	
Metric	**load.**	**Metric**	**load.**	**Metric**	**load.**	**Metric**	**load.**
object	+0.50	atomic	+0.67	cachemiss	+0.58	idynamic	+0.56
cpu	-0.49	park	+0.65	notify	+0.50	array	+0.42
method	+0.44	method	+0.20	wait	+0.41	notify	+0.42
array	+0.40	notify	+0.18	cpu	-0.28	method	-0.35
idynamic	+0.27	idynamic	-0.17	synch	-0.25	cachemiss	+0.28
synch	-0.17	cpu	-0.16	park	-0.20	cpu	+0.22
notify	-0.13	cachemiss	-0.08	idynamic	-0.18	atomic	+0.18
atomic	-0.13	object	+0.05	array	-0.13	wait	-0.15
cachemiss	-0.07	array	-0.03	method	+0.10	object	+0.13
park	-0.06	synch	-0.02	object	-0.04	synch	+0.11
wait	-0.02	wait	-0.00	atomic	-0.03	park	+0.05



(a) PC1 vs PC2. (b) PC3 vs PC4.

Figure 1. Scatter plots of benchmark scores over the first four principal components (PCs).

Each metric vector X_j is standardized to zero mean and unit variance, yielding a new vector $Y_j = \frac{X_j - \overline{x_j}}{s_j} \in \mathbf{Y}$ (where $\overline{x_j}$ and s_j are the mean and variance of X_j, respectively). According to established practices, we apply PCA on the standardized matrix $\mathbf{Y}$ rather than on $\mathbf{X}$. PCA produces a new matrix $\mathbf{S} = \mathbf{YL}$, where each row vector in $\mathbf{S}$ is the projection of the corresponding row vector in $\mathbf{Y}$ on the new coordinate system formed by the PCs (called *score*), while $l_{ij} \in \mathbf{L}$ ($l_{ij} \in [-1, 1]; i, j \in [1, K]$) is the *loading* of metric i on the j-th PC (henceforth called PCj). The absolute value of a loading quantifies the correlation of a metric to a given PC, while its sign determines if the correlation is positive or negative.

4.3 Analysis

Table 3 reports the loading of each metric on the first four PCs, ranked by their absolute values in descending order, while in Figure 1, we plot the scores of the considered benchmarks against the four PCs. [7] The first four components account for ~60% of the variance present in the data. [8]

As shown in Table 3, object allocation, dynamic dispatch, and array allocation correlate positively with PC1, and CPU utilization correlates negatively. Figure 1(a) shows that the SPECjvm2008 benchmarks are clustered in the bottom-left portion of the figure, while the benchmarks of the other suites are well distributed along PC1. This is a sign that Renaissance, DaCapo and ScalaBench contain applications that nicely exercise object allocation and dynamic dispatch, which is often a sign of using abstractions in those workloads.

In contrast, PC2 exhibits a strong positive correlation with two concurrency primitives, i.e., atomic and park. As demonstrated by the wide distribution of Renaissance benchmarks along PC2, as shown in Figure 1(a), Renaissance benchmarks extensively use these primitives. On the other hand, benchmarks from the other suites span a limited space along PC2. Similarly to PC2, PC3 is well correlated with metrics estimating concurrency, particularly cache misses and the usage of `wait/notify`. The benchmark distribution along PC3 is similar to the one along PC2 (as shown in Figure 1(b)), which suggests that generally, the use of concurrency primitives in DaCapo, ScalaBench, and SPECjvm2008 is limited.

Finally, idynamic contributes most to PC4, with a strong positive correlation. The presence of numerous Renaissance benchmarks in the top part of Figure 1(b) demonstrates the frequent execution of `invokedynamic` bytecodes in the workloads (indeed, 10 out of 21 benchmarks execute this bytecode at runtime), which in turn may represent the use of functional-style operations commonly found in parallelism frameworks. This is expected, as other suites were released before the introduction of `invokedynamic` in JDK 7 [97] and of Lambas and Streams in JDK 8 [36] (while a recent DaCapo version was released in Jan. 2018, there was no major change to the constituent benchmarks, originally released in 2009).

5 Optimization Opportunities

Once we identified a sufficiently diverse set of benchmarks, as explained previously in Sections 2, 3 and 4, we wanted to check if these benchmarks help us in identifying new optimization opportunities. We therefore investigated if we can develop new JIT-compiler optimizations that are useful for the code patterns that appear in Renaissance benchmarks.

In this section, we analyze several compiler optimizations for which we noticed a significant impact on Renaissance benchmarks, and we explain the reason for the impact. For the purposes of the analysis, we used the Graal JIT compiler to implement 4 new optimizations[9] (escape analysis with atomic operations, loop-wide lock coarsening, atomic-operation coalescing, and method-handle simplification), and to study 3 existing optimizations (speculative guard movement, loop vectorization, and dominance-based duplication simulation). Notably, we do not claim that these are the

[7] A larger version of Figure 1 can be found in Section E of the supplemental material.

[8] As discussed in Section B of the supplemental material, we excluded benchmarks *tradebeans*, *actors* and *scimark.monte_carlo* from the PCA.

[9] Graal's compliance test suite is available at GitHub [6], and we ran it to check that our implementations are correct.

only optimizations that are important for the Renaissance benchmarks – we only discuss those parallelism-related optimizations that we were able to identify. Also, we focused on high-level optimizations for the existing backends of the Graal compiler. In particular, we did not study fence optimizations although they are interesting for platforms such as PowerPC or AArch64[10].

In the rest of the section, we briefly explain each optimization and discuss its impact on the benchmarks. We also establish a relationship to the metrics from Section 3 and illustrate performance with data cited from Section 6. We present a minimal example for each optimization – while such examples are rare in the source code, they do appear in the compiler after transformations such as inlining.

In sections where optimization soundness is not apparent, we reason as follows. Each program P has some set R of possible results, where a result is a sequence of external side-effects that the program's threads make. An external side-effect is an output of the program that the user can observe (for example, the exit status or the I/O). Program's executions can produce different results because the program-threads' execution schedule is non-deterministic. When writing a concurrent program, users consider the program correct if its executions produce some result from the set of allowed results R. However, users are not allowed to assume any probability distribution of the program's results across executions. Thus, for an optimization to be valid, it must transform an original program P with a set of possible results R into a program P' with a set of possible results R', such that $R' \subseteq R$. For each optimization, we informally reason why the set of results of the transformed program is either equal to that of the original program or its strict subset.

5.1 Escape Analysis with Atomic Operations

Partial Escape Analysis (PEA) [106] is an optimization that reduces the amount of object allocations by postponing heap allocation of an object until it escapes and can be seen by other threads. PEA does this by analyzing the reads and writes to the allocated objects. So far, the PEA in Graal was limited to regular reads and writes, but it did not consider atomic operations such as Compare-And-Swap (CAS).

Consider the example on the right: if the CASes are not accounted for during PEA, then the A and B objects must be allocated before the first CAS.

```
1 o = new A(v);
2 CAS(o.x, v, new B(v2));
3 CAS(o.x.y, v2, v3)
4 return o.x;
```

A CAS operation involves up to three objects: the object containing the field, the expected value and the new value. If the object containing the field has not yet escaped, the CAS can simply update the state of the scalar-replaced object. This might replace the two other objects with scalars. This

[10] Graal has no PowerPC backend that we know of and its AArch64 backend has not yet been optimized for fine-grained use of fences.

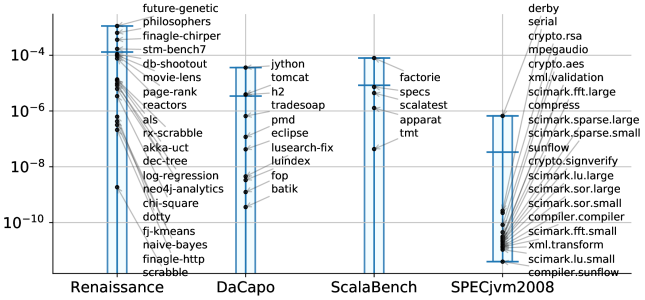


Figure 2. Number of atomic operations executed, normalized by reference cycles.

also helps when the objects escape later, because the CAS operation can be removed and the objects can be directly initialized in their mutated state before being published. In particular, this initialization can be performed with—potentially cheaper—regular writes rather than with atomic instructions.

For example, in the previous code, assuming the constructors

```
return new B(v3);
```

don't let the objects escape, the allocation of the A object can be completely eliminated and the value of the field in the B object can directly be set to its updated value (v3).

We observed this pattern in usages of `java.util.Random`, `com.twitter.util.Promise`, and `java.util.concurrent.atomic.AtomicReference`. These classes change their internal state using a CAS operation. An application can create an instance of these classes and change its internal state a few times before discarding it (full escape analysis), or publishing it to the rest of the program (partial escape analysis).

This optimization has a 24% impact on *finagle-chirper*. As seen in Figure 2, this benchmark exhibits a higher value for the atomic metric than any of the benchmarks from the existing suites. This is coherent, since the optimization targets an operation counted by the atomic metric.

Soundness. Objects that have not yet escaped cannot be observed or mutated by other threads by definition. Consequently, any schedule of the original program P will yield the same outcome for the CAS operation: the transformed program P' emulates this behavior on the thread which executed the CAS in P. If the object containing the memory location subject to CAS can be entirely escape-analyzed away in P', no other thread can observe the effects of this CAS in P. In this case, P' does not contain any external side-effect for this CAS and any result of P' is a possible result of P. If the object escapes at a later point in the program, its fully initialized state containing the emulated effects of the CAS is safely published during the side-effect that causes the object to escape. In this case any schedule of P' can be mapped to a schedule of P where all side-effects due to the initialization of this object happened before the escape point.

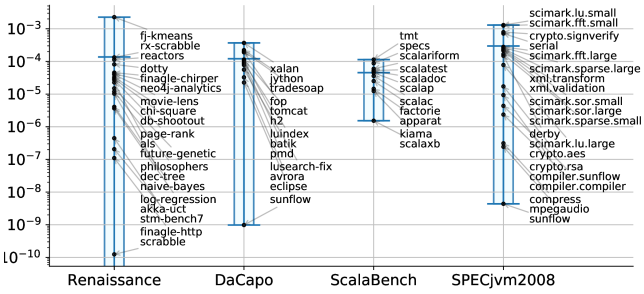


Figure 3. Number of synchronized methods or blocks executed, normalized by reference cycles.

5.2 Loop-Wide Lock Coarsening

Consider the Java loop shown in the snippet on the right enclosing a `synchronized` statement, which protects a critical region. The time spent entering and exiting the critical region can critically affect performance if the loop is frequently executed.

```
for (; condition;)
  synchronized(x) {
    ... region ...
  }
```

```
int b = 0;
for (; condition;)
  synchronized(x) {
    for(; condition;) {
      ... region ...
      b++;
      if (b % T == 0)
        break; } }
```

This synchronization pattern occurs when accessing synchronized collections such as the JDK `java.util.Vector` and `SynchronizedList` (whose operations use locks [13, 14]) inside a loop.

To reduce the cost of locking and unlocking within the `synchronized` block, we extended Graal with an optimization that coarsens the lock [30] by tiling the iteration space into C-sized chunks, as seen in the second snippet. The lock is held in the entire nested loop, so the total number of monitor operations decreases. The necessary condition is that the code motion on `condition` and `region` is legal (e.g., `condition` must not obtain another lock).

Existing lock optimizations in HotSpot's C2 [68] compiler can optimize loops that have a statically known number of iterations, by completely unrolling and coarsening the lock across the unrolled iterations. By contrast, our optimization tiles the iteration space of any loop to coarsen the lock, without pulling the lock out of the loop completely. Lock coarsening has an effect on fairness, but since `synchronized` regions in Java are inherently not fair, this optimization does not change the program semantics – applications relying on fairness must anyway use more elaborate locking mechanisms, such as the `ReentrantLock` class from the JDK.

Loop-wide lock coarsening improves *fj-kmeans* by about 71%. We found that a chunk size of $C = 32$ works well for this benchmark. Generally, the impact depends on the size of the critical region, as well as the number of loop iterations. Figure 3 shows the synchronization counts across the suites, normalized by the CPU reference cycles. Note that *fj-kmeans* uses the `synchronized` primitive considerably more often, which made it possible to identify this optimization.

Soundness. Any schedule s' of the transformed program P' can be mapped to a schedule s of the original program P, such that the respective external side-effects r' and r are equal. Consider a schedule s' of P' in which some thread T is about to execute a loop that contains a synchronized region, and which has external side-effects r_0 and the memory state m_0 before executing that loop. There exists some execution schedule s of P in which all other threads pause during the execution of the C iterations of the loop. That execution schedule has the same external side-effects r_0 and the memory state m_0 before executing that loop. Since `condition` does not acquire any locks, it is easy to show that both s and s' have the same set of external side-effects and the same memory state after executing C iterations of the loop. By induction, the two execution schedules have the same external side-effects $r' = r$, so the transformation is correct.

5.3 Atomic-Operation Coalescing

A typical retry-loop in lock-free algorithms reads the value at a memory address, and uses a CAS instruction to atomically replace the old value with the newly computed value. If the CAS fails due to a concurrent write by another thread, the loop repeats, or otherwise terminates. The code snippet on the right shows two such consecutive retry-loops. This pattern rarely appears directly in user programs, but it does occur after inlining high-level operations, such as random-number generators or concurrent counters.

```
1 do {
2   v = READ(x)
3   nv = f1(v)
4 } while(!CAS(x,v,nv))
5 do {
6   v = READ(x)
7   nv = f2(v)
8 } while(!CAS(x,v,nv))
```

Consider an execution schedule in which no other thread makes progress between the `READ` in line 2 up to the `CAS` in line 8, In this execution schedule, the intermediate `CAS` in line 4 is not observed by any other thread. According to the Java Memory Model [94], threads are not guaranteed to observe other execution schedules of this snippet, so user programs should not assume that they will ever see `f1(v)`.

```
do {
  v = READ(x)
  nv = f2(f1(v))
} while(!CAS(x,v,nv))
```

Consequently, if the functions `f1` and `f2` are referentially transparent, the new value `nv` of the first `CAS` can be directly replaced with the new value `nv` of the second `CAS`, and the second `CAS` can be removed; the resulting snippet is on the left.

We observed this pattern in `java.util.Random` usages, where the method `nextDouble` consecutively executes a `CAS` twice. This optimization mostly affects *future-genetic*, improving it by ≈24%. The high atomic operations rate in *future-genetic*, shown previously in Figure 2, is due to the shared use of a pseudo-random generator, and this optimization reduces the overall contention by eliminating some `CAS`es.

Soundness. Call the original program P, and the transformed program P'. Let S be the subset of all the execution schedules

of P in which some thread T executes the CAS instructions in lines 4-8, during which all other threads are paused. We will show that any schedule of P' can be mapped to $s \in S$.

First, consider some schedule s' of P' in which the CAS in line 4 succeeds. There must exist an execution schedule $s \in S$ of P that is exactly the same as s' until the lines 4-8. Before T executes the CAS in line 4 of P', the set of external side-effects must have been r_0 in both s and s', and the state of the memory must have been $m_0 \cup \{x \rightarrow v\}$. After T executes the CAS in line 4 of P', external side-effects remain r_0 because f1 and f2 are referentially transparent, and memory state becomes $m_0 \cup \{x \rightarrow f2(f1(v))\}$. Since the CAS in line 4 of P' succeeds in s', then CAS in line 4 of P succeeds in s. Since $s \in S$, the CAS in line 8 must also succeed. Consequently, after T executes the lines 4-8 of P, the set of external side-effects is r_0, and $m_0 \cup \{x \rightarrow f2(f1(v))\}$ is the state of the memory.

Therefore, in the schedule $s \in S$, after the other threads resume, P has the same memory state as P' in s', so the remaining external side-effects r_1 must be the same in both s and s', and the execution has the same result $r' = r = r_0 \cdot r_1$. The proof is similar when the CAS in line 4 of P' fails.

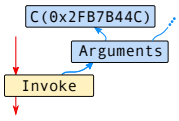
In conclusion, any schedule s' of P' can be mapped to a schedule s in P that has the same result $r = r'$. By the previous definition, $R' \subseteq R$ and the transformation is correct.

5.4 Method-Handle Simplification

Java 8 Lambdas are used to express declarative operations in multiple frameworks such as Java Streams [33], futures [12, 51, 53, 54], and Rx [15]. In the bytecode, the lambda-value creation is encoded with an invokedynamic instruction [97]. The first time this instruction is executed, a special *bootstrap method* generates an anonymous class for the lambda, and returns a *method handle*. The method handle's invoke method is then used to call the lambda.

```
...
invokedynamic #14, 0
aload 11
invokevirtual #15, 2
...
```

Consider the sequence of bytecodes shown on the left. By the time this code gets compiled, the invokedynamic gets replaced with the address of the generated method-handle object.



The figure on the right shows the corresponding program segment in the intermediate representation used by Graal [32]. The Invoke node represents the second invoke, while Arguments encapsulates the argument list. Importantly, the first argument C is a constant that represents the address of the method-handle in memory.

In frameworks that parameterize operations with lambda values, inlining the body of the lambda typically triggers other optimizations. Unfortunately, the method handle's invoke method is polymorphic, which prevents the inlining. Even though the method-handle object has a constant address, the compiler does not know *the actual code of the JVM method that the method-handle object refers to*. We therefore used the existing JVM compiler interface [96] to resolve the method-handle-object address to the JVM method that is encapsulated within the method handle. This allowed us to replace method-handle calls with direct calls, which can be inlined in the subsequent phases in the compiler.

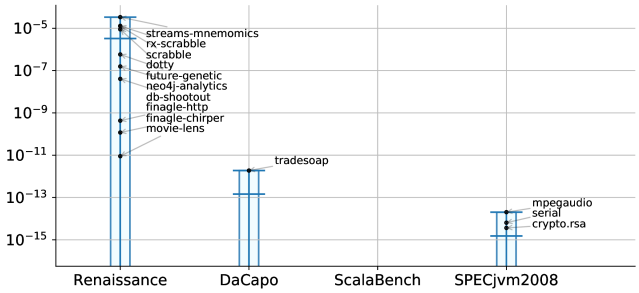


Figure 4. Number of invokedynamic bytecodes executed, normalized by reference cycles.

We found that this generally improves performance across multiple methods. For example, using Oracle Developer Studio 12.6, we recorded the hottest methods in *scrabble*, along with the execution times with and without this optimization.

with (ms)	w/o (ms)	Compilation unit
303.4	350.0	<Total>
25.9	29.6	JavaScrabble.lambdarun2
23.7	23.1	java.util.Spliterator$OfInt.forEachRemaining
22.4	28.8	java.util.stream.ReferencePipeline$3$1.accept
18.1	20.4	JavaScrabble.lambdarun5

In the preceding table, method-handle simplification reduces the execution time for most methods, but not for the forEachRemainining method of the Spliterator interface. The reason is as follows: parallel Stream operations split collections into subsets, pass some subsets to worker threads, and some to the calling thread. The invokedynamic instruction that creates the method-handle is executed on the calling thread. Thus, the method-handle type can be statically determined on the calling thread, but not on the worker threads, which invoke the forEachRemaining method. This indicates additional opportunities for optimizations in the future.

```
1 histogram = word ->
2   word.chars().boxed()
3     .collect(groupingBy(
4       identity(),
5       counting()));
```

Method lambdarun2 corresponds to the Java code on the left: a lambda that produces a histogram of characters for a given string. Using Graal, we inspected the intermediate representation of this method, both with and without the optimization. The additional inlining induced by method-handle simplification triggers other optimizations that reduce the number of callsites from 19 to 1, remove 37 out of 61 dynamic type- and null-checks, and reduce the total number of IR nodes in that method from 696 to 490.

Overall, method-handle simplification impacts the running time of *finagle-chirper* by 4%, *scrabble* by 22%, *streams-mnemonics* by 7%, and *rx-scrabble* by 1%. These results correlate with the `invokedynamic` counts normalized by CPU reference cycles, shown in Figure 4. On DaCapo, ScalaBench, and SPECjvm2008 (which precede `invokedynamic`), the calls to `invokedynamic` usually occur only indirectly through the Java class library.

5.5 Speculative Guard Motion

JIT compilers for the JVM often use speculative optimizations. Some of these optimizations require the insertion of *guards*, which ensure the speculation is correct at runtime. If the speculation is not correct, these guards divert the execution to an environment that does not use speculations (i.e., an interpreter, or code compiled by a baseline compiler). New profiling then takes place and a re-compilation that does not use the respective speculation is triggered. In order to avoid useless re-compilations, guards are not typically hoisted out of the branch where they are needed.

```
for (i=0; i<N; i++) {
  if (...) {
    guard(0<=i<L);
    ...
  }
}
```

```
guard(0<L && 0<=N-1<L);
for (i=0; i<N; i++) {
  if (...) {
    ...
  }
}
```

However, for some loops, it is beneficial to speculatively hoist guards out of loops even if the control-flow in the loop does not always lead to that guard. This is beneficial because the hoisted guards are executed less often. To capture typical cases such as bounds checks, comparisons of induction variables can be rewritten to loop-invariant versions. Speculative Guard Motion is described in more details by Duboscq [31]. This optimization has a 15% impact on the *log-regression* benchmark and an 8% impact on the *dec-tree* benchmark. To better understand this difference, we measured the number of guards executed with and without speculative guard motion for the *log-regression* benchmark.

Executions		Guard type
Without		
57 487 131	1%	Others
529 958 424	10%	UnreachedCode
1 639 903 998	32%	NullCheckException
2 917 610 059	57%	BoundsCheckException
5 144 959 612	100%	Total
With		
55 660 483	7%	Others
18 314 637	2%	*Speculative* NullCheckException
22 056 283	3%	*Speculative* UnreachedCode
28 570 247	3%	*Speculative* BoundsCheckException
62 046 637	7%	BoundsCheckException
166 341 946	20%	NullCheckException
499 933 160	59%	UnreachedCode
852 923 393	100%	Total

First, note that the total number of executed guards is reduced by 83%. Moreover, note the entry for *speculative* guard variants, which accounts for the hoisted guards. In particular, we can see the large effect on `BoundsCheckException` and `NullCheckException` guards where most occurrences have been replaced by the lower-frequency speculative versions.

In some cases, removing the guards enables additional optimizations. Loop vectorization, discussed next, is one such example – we found that by disabling speculative guard motion, loop vectorization almost never triggers.

Soundness. Hoisting guards of loop-invariant conditions is sound since executing unnecessary guards is always sound: guards have no external side-effects, they only run the risk of reducing performance. In this case performance is conserved by not doing this transformation again if a deoptimization already happened for this loop. The rewrite of loop-variant inequalities ($<$, $>$, $\leq$, $\geq$, signed and unsigned) involving loop induction variables requires the induction variable to change monotonically which can be statically or dynamically checked above the loop. Once this is established, if the inequality holds at both bounds, it holds over the whole range. As a result, the hoisted guard implies the original one, ensuring the transformed program will deoptimize in at least as many cases as the original.

5.6 Loop Vectorization

Modern CPUs include vector instruction sets that ensure better performance by operating on multiple memory addresses at once [3]. Most languages do not have dedicated abstractions for these instructions, so the compiler transforms parts of the program to use vector instructions, when possible.

Consider a simple Java loop shown on the right. When separate loop iterations have no data dependencies, a compiler can aggregate arithmetic operations from several consecutive iterations into a single vector operation. However, on the JVM, the code must also perform a bounds check on each array access, which introduces additional guards into the loop, and normally prevents vectorization. Speculative guard motion makes loop vectorization possible, since it moves the bounds-check guards outside of the loop.

```
for(i=0; i<N; i++) {
  c[i] = a[i] + b[i];
}
```

We found that combining loop vectorization with speculative guard motion results in a ≈10% impact on *als*, and a ≈3% impact on *dec-tree*.

5.7 Dominance-Based Duplication Simulation

Dominance-based duplication simulation (DBDS) [56] is an optimization that simplifies control flow by duplicating the code after control-flow merges. Consider the pattern on the right, which consists of two subsequent `instanceof`-checks on the same class `C`.

```
if (x instanceof C) a()
else b()
if (x instanceof C) c()
```

After duplicating the second `instanceof`-check within both preceding branches, this second check becomes dominated by the first check. It can therefore be eliminated, resulting in the code shown on the left.

```
1 if (x instanceof C) {
2   a(); c()
3 } else b()
```

Since tail-duplication optimizations typically enable other optimizations by devirtualizing the callsites, we decided to include DBDS in the impact analysis. We found that this optimization mostly affects *streams-mnemonics*, with a performance impact of around 22%.

6 Performance Evaluation

To investigate how the new optimization opportunities in Renaissance compare to the other suites, we provide a comprehensive performance evaluation that identifies the impact of each optimization discussed in Section 5 on each individual benchmark. As a consistent measure of impact across different optimizations and benchmarks, we define the impact of an optimization as the change in the benchmark execution time observed when the optimization is selectively disabled. The measure thus accounts for the cumulative impact of an optimization, including its possible (enabling or disabling) effects on other optimizations performed by the compiler.

The results are summarized in Figure 5. Evaluation results shown in this section have been obtained using the G1 collector. More details on the experimental setup used for collecting the measurements are described in Section C of the supplemental material. The figure supports our claim that the optimizations outlined in Section 5 benefit Renaissance more than other benchmark suites – at significance level $\alpha = 0.01$, all (7 of 7) evaluated optimizations have an impact of at least 5% on some Renaissance benchmark, compared with 2 of 7 for ScalaBench, 1 of 7 for DaCapo, and 3 of 7 for SPECjvm2008. At the same significance level, the median impact is 6.4% for Renaissance, compared with 2.8% for ScalaBench, 1.8% for DaCapo, and 3.9% for SPECjvm2008. In Section G of the supplemental material, we also provide some information on how each optimization impacts the overall compilation time.

The architecture of Graal enables rapid development and deployment of new optimizations, making it an obvious choice for our experiments with optimization opportunities in modern workloads. We found it relatively straightforward to implement new optimizations in the open-source Graal compiler. However, if Graal were a poorly performing compiler, the impact of each optimization relative to baseline might appear magnified. To prove that Graal can serve as a reasonable baseline, we also compared performance of the four benchmark suites on two variants of the HotSpot JVM, one equipped with the standard C2 compiler, other with Graal, both in the role of the final-tier compiler. The results, relative to the baseline HotSpot JVM using the C2 compiler, are shown in Figure 6.

Graal provides better performance than C2 in 51 out of 68 benchmarks, the opposite is true in 10 benchmarks, for the remaining 7 benchmarks the difference is not statistically significant as the 99% confidence interval straddles 1.0. On benchmarks where Graal is better than C2, the median speedup is 20%. On benchmarks where C2 is better than Graal, the median slowdown is 4%. This justifies our choice of Graal as the experimental evaluation platform.

7 Code Complexity

In this section, we compare the software complexity and the compiled code size of Renaissance against the one of DaCapo, ScalaBench and SPECjvm2008.

7.1 Software Complexity Metrics

Here, we compare Renaissance against the other benchmark suites by computing the Chidamber and Kemerer (CK) metrics [27], which were previously used to evaluate software complexity of benchmarks from the DaCapo and CD$_x$ [47] suites. We used the *ckjm* [104] tool to calculate the metrics. To analyze only the classes used in each benchmark (and not all the classes in the classpath), we also developed a JVMTI native agent [65] for the HotSpot JVM that receives class-loading events from the VM, and forwards the loaded classes to *ckjm*, which then calculates the metrics.

We use the following CK metrics:

1. *Weighted methods per class* (**WMC**), which is calculated as the number of declared methods per loaded class.
2. *Depth of the inheritance tree* (**DIT**), defined as the depth of the inheritance tree along all loaded classes.
3. *Number of children* (**NOC**) is computed as the number of immediate subclasses of a class.
4. *Coupling between object classes* (**CBO**) counts the number of classes coupled to a given class, either by method calls, field accesses, inheritance, method signatures or exceptions.
5. *Response for a class* (**RFC**) counts the number of different methods that are potentially (transitively) executed when a single method is invoked.
6. *Lack of cohesion* (**LCOM**) counts how many methods of a class are not related by accessing the same field of that class.

The summarized results of the analysis are shown in Table 4. For each benchmark suite, we report a geometric mean, the minimum, and the maximum for the sum and for the average (arithmetic mean) of each metric across all benchmarks of the suite. When looking at the sums, we overall observed the highest values on five out of six metrics on Renaissance, while the LCOM metric was the highest on ScalaBench. With respect to the average complexity across all benchmarks of the suite, we can see that ScalaBench has higher values than Renaissance in 5 cases. In general we cannot say whether Renaissance is more complex than ScalaBench. However,

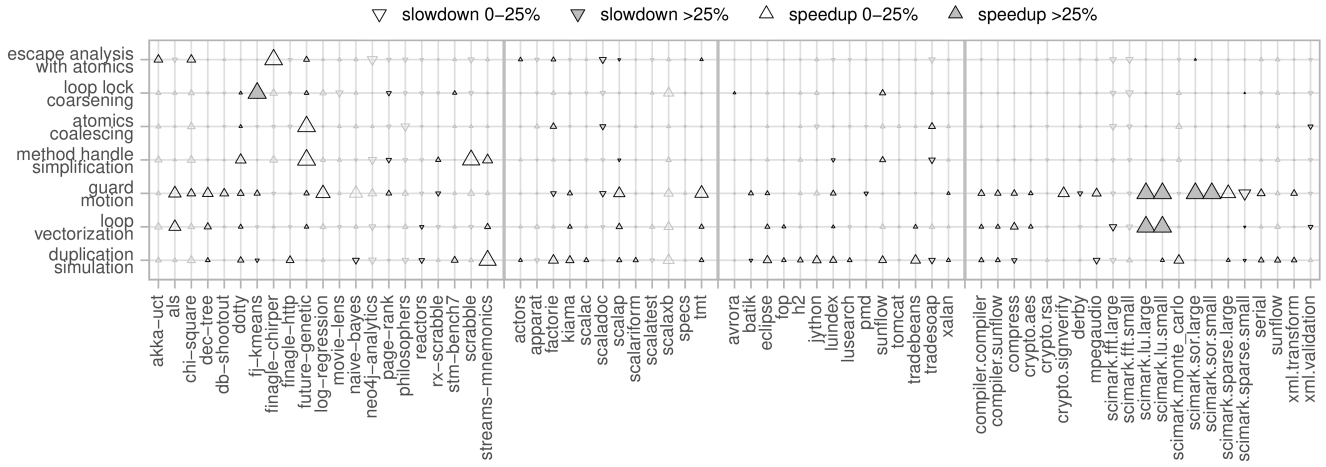


Figure 5. Optimization impact on individual benchmarks. Results with black outline significant at $\alpha = 0.01$.

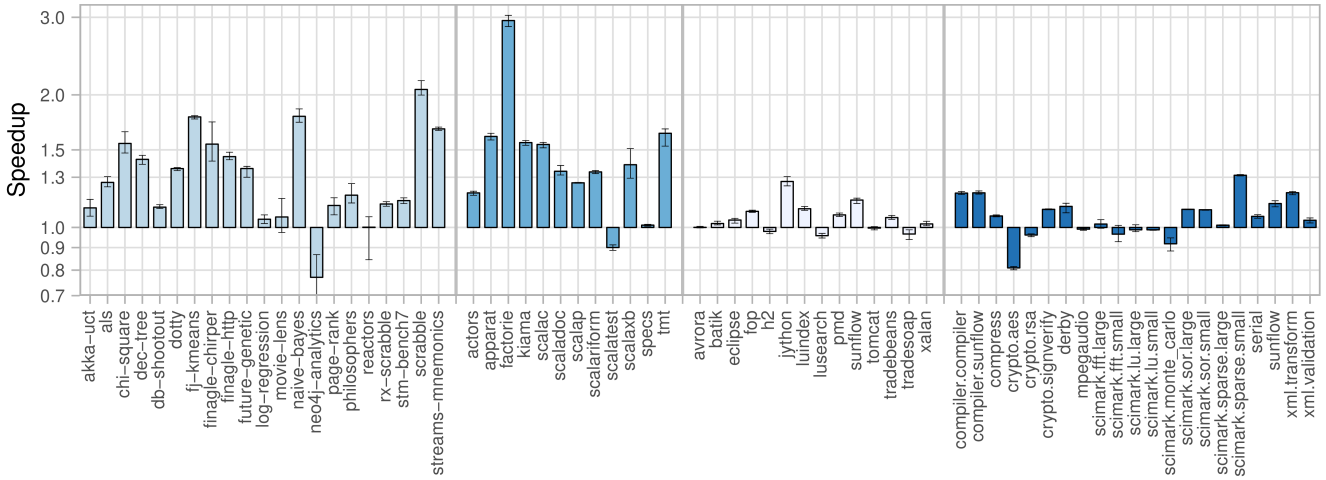


Figure 6. Graal compiler performance relative to HotSpot second tier JIT compiler.

we see that Renaissance is in the same ball park as ScalaBench and the other suites. The reason why Renaissance is more complex on some of the metrics is highly related to the number of loaded classes of the benchmark suites. We took a look at the sum of all loaded classes across each benchmark suite and the number of unique loaded classes for each benchmark suite in Table 5. We observed that Renaissance benchmarks on average load many classes, compared to the other benchmark suites.

7.2 Compiled Code Size

This section presents several metrics about the size of the compiled programs. Figure 7 shows the code size and compiled-method count distribution for the DaCapo, ScalaBench, SPECjvm2008, and Renaissance suites. Both the code size and the method count refer to the hot code, compiled with the second-tier optimizing compiler, and were measured using Oracle's Graal compiler [7] by letting the benchmark run for 120 seconds.

The JVM uses the method's invocation count and its loop iteration counts to decide if a method is hot (i.e. frequently executed). Hot methods are compiled using the second-tier optimizing compiler, so the total size and the method count of the hot code indicate how much of the program was deemed important for the overall performance.

The geomean of the compiled code size (across all benchmarks) is ≈6.87MB for Renaissance, which is slightly lower than ≈7.98MB for DaCapo, and ≈10.03MB for ScalaBench. The geomean of the total number of hot methods is ≈1 636 for Renaissance, which is slightly higher than ≈1 599 for DaCapo, and somewhat lower than ≈1 853 for Scalabench.

Table 4. CK Metrics Summary: Minimum, maximum and geometric mean across the sum and arithmetic mean of all loaded classes of a benchmark suite. Bold values indicate the highest value for a metric across all benchmark suites.

	WMC	DIT	CBO	NOC	RFC	LCOM
Renaissance						
min-sum	21830	3066	21757	1571	41799	455958
max-sum	206933	23936	184901	14029	369131	7360650
geomean-sum	**55533**	**7842**	**55147**	**4212**	**105105**	1358042
min-avg	11.07	1.79	12.57	0.97	20.78	141.02
max-avg	18.48	2.29	17.54	1.16	33.84	706.54
geomean-avg	13.58	**1.92**	13.49	1.03	25.71	332.19
DaCapo [21]						
min-sum	12466	1687	10640	857	22378	161333
max-sum	124191	5753	45597	6421	97757	616524
geomean-sum	32470	3377	23275	2160	48461	336192
min-avg	11.74	1.24	11.90	0.97	22.07	130.36
max-avg	42.43	2.13	14.45	2.19	32.27	310.86
geomean-avg	17.97	1.87	12.88	**1.20**	26.82	186.05
ScalaBench [100]						
min-sum	24693	2657	19713	1481	45364	1080228
max-sum	150895	7789	66036	7890	124240	2298594
geomean-sum	44505	4291	32810	2735	71840	**1489515**
min-avg	14.04	1.70	12.67	1.00	27.82	440.66
max-avg	42.60	1.97	16.17	2.49	34.26	836.09
geomean-avg	**18.85**	1.82	**13.90**	1.16	**30.43**	**631.02**
SPECjvm2008 [1]						
min-sum	30560	3832	29966	2190	65741	546396
max-sum	55044	5744	45745	3480	103373	1131251
geomean-sum	33195	4142	32187	2383	70280	578408
min-avg	13.55	1.72	13.48	0.99	28.69	206.19
max-avg	16.90	1.86	14.04	1.07	31.73	347.22
geomean-avg	14.00	1.75	13.58	1.01	29.65	244.03

Table 5. Loaded classes per benchmark suite.

Benchmark Suite	Sum All Loaded Classes	Sum Unique Loaded Classes
Renaissance	109 004	29 157
DaCapo	27 923	12 073
ScalaBench	27 911	12 207
SPECjvm2008	50 099	5 274

While DaCapo, ScalaBench and Renaissance are roughly in the same range, the SPECjvm2008 has a geomean of ≈486 hot methods, and a geomean code size of ≈1.17MB (for readability, the names of individual SPECjvm2008 benchmarks are not shown in Figure 7). This indicates that the SPECjvm2008 workloads are considerably smaller.

8 Related Work

By differentiating between the optimizations that are worth pursuing, benchmark suites were at the core of innovation throughout the JVM's history. Since its introduction in 2006, the DaCapo suite [21] has been a de facto standard for JVM benchmarking. While much of the original motivation for the DaCapo suite was to understand object and memory behavior in complex Java applications, this suite is still actively used to evaluate not only JVM components such as JIT compilers [34, 56, 83, 85, 106] and garbage collectors [17, 63],

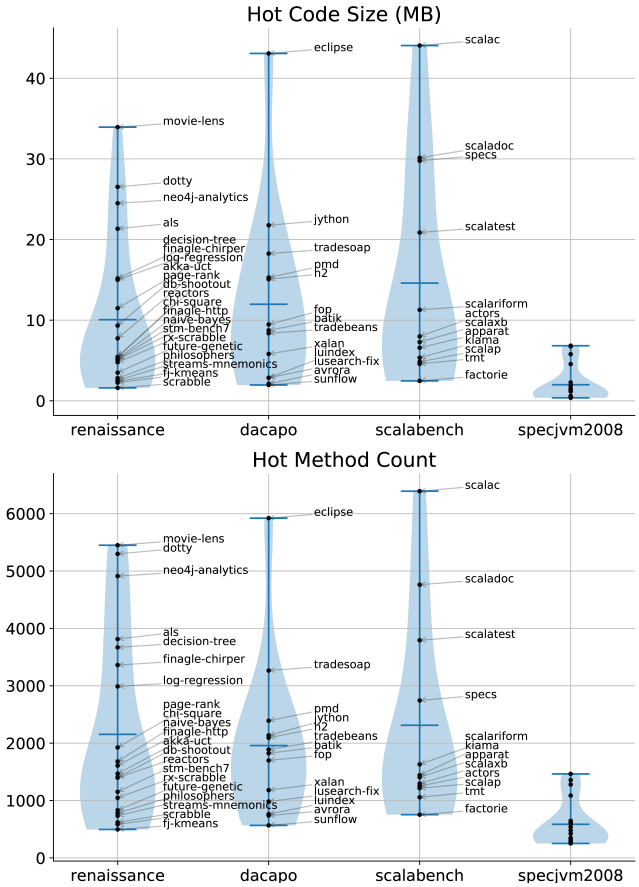


Figure 7. Code Size Metrics

but also tools such as profilers [25, 98], data-race detectors [20, 110], memory monitors and contention analyzers [42, 109], static analyzers [37, 107], and debuggers [58].

The subsequently proposed ScalaBench suite [99, 100] identified a range of typical Scala programs, and argued that Scala and Java programs have considerably different distributions of instructions, polymorphic calls, object allocations, and method sizes. This observation that benchmark suites tend to over-represent certain programming styles was also noticed in other languages, (e.g., JavaScript [95]). On the other hand, the SPECjvm2008 benchmark suite [1] focused more on the core Java functionality. Most of the SPECjvm2008 benchmarks are considerably smaller than the DaCapo and ScalaBench benchmarks, and do not use a lot of object-oriented abstractions – SPECjvm2008 exercises classic JIT compiler optimizations, such as instruction scheduling and loop optimizations [31].

We did not include SPECjbb2015 [4] in our analysis, because it consists of a single benchmark designed to run over 2 hours, typically executed on multiple JVM instances. Using SPECjbb2015 to detect small statistically significant changes is therefore relatively costly (needs long execution time).

A brief evaluation suggests that the combined effect of all optimizations considered here accounts for about 1.2% improvement in the reported peak throughput (max jOPS) in single JVM configuration.

Another older JVM benchmark suite is Java Grande [103], but it consists mostly of microbenchmarks.

The tuning of compilers such as C2 [68] and Graal [7, 32] was heavily influenced by the DaCapo, ScalaBench, and SPECjvm2008 suites. Given that these existing benchmark suites do not exercise many frameworks and language extensions that gained popularity in the recent years, we looked for workloads exercising frameworks such as Java Streams [33] and Parallel Collections [81, 92, 93], Reactive Extensions [15], Akka [5], Scala actors [39] and Reactors [72, 75, 84, 90], coroutines [8, 86, 87], Apache Spark [111], futures and promises [40], Netty [11], Twitter Finagle [12], and Neo4J [10]. Most of these frameworks either assist in structuring concurrent programs, or enable programmers to declaratively specify data-parallel processing tasks. In both cases, they achieve these goals by providing a higher level of abstraction – for example, Finagle supports functional-style composition of future values, while Apache Spark exposes data-processing combinators for distributed datasets. By inspecting the IR of the open-source Graal compiler (c.f. Section 5), we found that many of the benchmarks exercise the interaction between different types of JIT compiler optimizations: optimizations, such as inlining, duplication [56], and partial escape analysis [106], typically start by reducing the level of abstraction in these frameworks, and then trigger more low-level optimizations such as guard motion [31], vectorization, or atomic-operation coalescing. Aside from a challenge in dealing with high-level abstractions, the new concurrency primitives in modern benchmarks pose new optimization opportunities, such as contention elimination [59], application-specific work-stealing [89], NUMA-aware node replication [26], speculative spinning [73], access path caching [74, 76–78], or other traditional compiler optimizations applied to concurrent programs [108]. Many of these newer optimizations may be applicable to domains such as concurrent data structures, which have been extensively studied on the JVM [18, 23, 28, 52, 67, 71, 79, 80, 82, 88, 91].

Unlike some other suites whose goal was to simulate deployment in clusters and Cloud environments, such as CloudSuite [35], our design decision was to follow the philosophy of DaCapo and ScalaBench, in which benchmarks are executed within a single JVM instance, whose execution characteristics can be understood more easily. Still, we found some alternative suites useful: for example, we took the *movie-lens* benchmark for Apache Spark from CloudSuite, and we adapted it to use Spark's single-process mode.

The Unbalanced Cobwebbed Tree (UCT) benchmark [112] was designed for better actor-scheduler comparison on non-uniform workloads, and our *akka-uct* benchmark is the implementation of that benchmark in the Akka actor framework for the JVM [5]. A suite that specialized exclusively on benchmarks for the actor model is the Savina suite [43].

Several other benchmarks were either inspired by or adapted from existing workloads. The *naive-bayes*, *log-regression*, *als*, *dec-tree* and *chi-square* benchmarks directly work with several machine-learning algorithms from Apache Spark MLLib, and some of these benchmarks were inspired by the SparkPerf suite [29]. The *Shakespeare plays Scrabble* benchmark [69] was presented by José Paumard at the Virtual Technology Summit 2015 to demonstrate an advanced usage of Java Streams, and we directly adopted it as our *scrabble* benchmark. The *rx-scrabble* is a version of the *scrabble* benchmark that uses the Reactive Extensions framework instead of Java Streams. The *streams-mnemonics* benchmark is rewritten from the *Phone Mnemonics* benchmark that was originally used to demonstrate the usage of Scala collections [64]. The *stm-bench7* benchmark is STMBench7 [38] applied to ScalaSTM [22, 24], a software transactional memory implementation for Scala, while the *philosophers* benchmark is ScalaSTM's *Reality-Show Philosophers* usage example.

9 Conclusion

We presented Renaissance – a benchmark suite consisting of 21 benchmarks that represent modern concurrency and parallelism workloads, written in popular Java and Scala frameworks. To obtain these benchmarks, we gathered a large list of candidate workloads, both manually and by scanning an online corpus of GitHub projects. We then defined a set of basic metrics to filter potentially interesting workloads, and to ensure that the selection is sufficiently diverse. Our PCA analysis revealed that the set of benchmarks selected this way covers the metric space differently than the existing benchmark suites. To confirm that the thus-selected benchmarks are useful, we then analyzed them for performance-critical patterns, which lead us to implement four new optimizations in the Graal JIT compiler. These optimizations have a considerably smaller impact on existing suites, such as DaCapo, ScalaBench and SPECjvm2008, indicating that Renaissance helped in identifying new compiler optimizations. We also identified three existing optimizations whose performance impact is prominent. Furthermore, by comparing two production-quality JIT compilers, Graal and HotSpot C2, we determined that performance varies much more on Renaissance than on DaCapo and SPECjvm2008.

While we showed that Renaissance aids in identifying new compiler optimizations, we believe that the suite might be beneficial for other domains as well, such as garbage collectors, profilers, data-race detectors, and debuggers. We leave these and other investigations to future work.

References

[1] 2008. SPECjvm2008. https://www.spec.org/jvm2008/.

[2] 2008. SPECjvm2008 User's Guide. https://www.spec.org/jvm2008/docs/UserGuide.html

[3] 2013. AVX 512 Instructions. https://software.intel.com/en-us/blogs/2013/avx-512-instructions.

[4] 2015. SPECjbb2015. https://www.spec.org/jbb2015/.

[5] 2018. Akka Documentation. http://akka.io/docs/.

[6] 2018. Graal JTTTest Source Code. https://github.com/oracle/graal/blob/master/compiler/src/org.graalvm.compiler.jtt/src/org/graalvm/compiler/jtt/JTTTest.java.

[7] 2018. GraalVM Website. https://www.graalvm.org/downloads/

[8] 2018. Kotlin Coroutines. https://github.com/Kotlin/kotlinx.coroutines/blob/master/coroutines-guide.md. Accessed: 2018-11-15.

[9] 2018. Open-Source Jenetics Repository at GitHub. https://github.com/jenetics/jenetics.

[10] 2018. Open-Source Neo4J Repository at GitHub. https://github.com/neo4j/neo4j.

[11] 2018. Open-Source Netty Repository at GitHub. https://github.com/netty/netty.

[12] 2018. Open-Source Twitter Finagle Repository at GitHub. https://github.com/twitter/finagle.

[13] 2018. OpenJDK SynchronizedList Source Code. http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Collections.java.

[14] 2018. OpenJDK Vector Source Code. http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/687fd7c7986d/src/share/classes/java/util/Vector.java.

[15] 2018. ReactiveX project. http://reactivex.io/languages.html.

[16] 2018. RxJava repository. https://github.com/ReactiveX/RxJava.

[17] Shoaib Akram, Jennifer B. Sartor, Kathryn S. McKinley, and Lieven Eeckhout. 2018. Write-rationing Garbage Collection for Hybrid Memories. In *PLDI*. 62–77.

[18] Dmitry Basin, Edward Bortnikov, Anastasia Braginsky, Guy Golan-Gueta, Eshcar Hillel, Idit Keidar, and Moshe Sulamy. 2017. KiWi: A Key-Value Map for Scalable Real-Time Analytics. In *Proceedings of the 22Nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '17)*. ACM, New York, NY, USA, 357–369. https://doi.org/10.1145/3018743.3018761

[19] Alan Bateman and Doug Lea. 2011. Java Specification Request 203: More New I/O APIs for the JavaTM Platform ("NIO.2"). https://jcp.org/en/jsr/detail?id=203.

[20] Swarnendu Biswas, Man Cao, Minjia Zhang, Michael D. Bond, and Benjamin P. Wood. 2017. Lightweight Data Race Detection for Production Runs. In *CC*. 11–21.

[21] 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. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190.

[22] Nathan Bronson, Jonas Boner, Guy Korland, Aleksandar Prokopec, Krishna Sankar, Daniel Spiewak, and Peter Veentjer. 2011. ScalaSTM Expert Group. https://nbronson.github.io/scala-stm/expert_group.html.

[23] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A Practical Concurrent Binary Search Tree. *SIGPLAN Not.* 45, 5 (Jan. 2010), 257–268. https://doi.org/10.1145/1837853.1693488

[24] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. 2010. CCSTM: A library-based STM for Scala. In *The First Annual Scala Workshop at Scala Days*.

[25] Rodrigo Bruno and Paulo Ferreira. 2017. POLM2: Automatic Profiling for Object Lifetime-aware Memory Management for Hotspot Big Data Applications. In *Middleware*. 147–160.

[26] Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera. 2017. Black-box Concurrent Data Structures for NUMA Architectures. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 207–221. https://doi.org/10.1145/3037697.3037721

[27] Shyam R Chidamber and Chris F Kemerer. 1994. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering* 20, 6 (1994), 476–493.

[28] Cliff Click. 2007. Towards a Scalable Non-Blocking Coding Style. http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf

[29] Databricks. 2018. Spark Performance Tests. https://github.com/databricks/spark-perf.

[30] Pedro C. Diniz and Martin C. Rinard. 1998. Lock Coarsening: Eliminating Lock Overhead in Automatically Parallelized Object-based Programs. *J. Parallel and Distrib. Comput.* 49, 2 (1998), 218–244.

[31] Gilles Duboscq. 2016. *Combining Speculative Optimizations with Flexible Scheduling of Side-effects*. Ph.D. Dissertation. Johannes Kepler University, Linz.

[32] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. 2013. An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler. In *VMIL*. 1–10.

[33] Michael Duigou. 2011. Java Enhancement Proposal 107: Bulk Data Operations for Collections. http://openjdk.java.net/jeps/107.

[34] Josef Eisl, Matthias Grimmer, Doug Simon, Thomas Würthinger, and Hanspeter Mössenböck. 2016. Trace-based Register Allocation in a JIT Compiler. In *PPPJ*. 14:1–14:11.

[35] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (March 2012), 37–48.

[36] Rémi Forax, Vladimir Zakharov, Kevin Bourrillion, Dan Heidinga, Srikanth Sankaran, Andrey Breslav, Doug Lea, Bob Lee, Brian Goetz, Daniel Smith, Samuel Pullara, and David Lloyd. 2014. Java Specification Request 335: Lambda Expressions for the JavaTM Programming Language. https://jcp.org/en/jsr/detail?id=335.

[37] Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. 2018. Shooting from the Heap: Ultra-scalable Static Analysis with Heap Snapshots. In *ISSTA*. 198–208.

[38] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: A Benchmark for Software Transactional Memory. In *EuroSys*. 315–324.

[39] Philipp Haller and Martin Odersky. 2007. Actors That Unify Threads and Events. In *COORDINATION*. 171–190.

[40] Philipp Haller, Aleksandar Prokopec, Heather Miller, Viktor Klang, Roland Kuhn, and Vojin Jovanovic. 2012. Scala Improvement Proposal: Futures and Promises (SIP-14). http://docs.scala-lang.org/sips/pending/futures-promises.html

[41] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. 2005. Composable Memory Transactions. In *PPoPP*. 48–60.

[42] Peter Hofer, David Gnedt, Andreas Schörgenhumer, and Hanspeter Mössenböck. 2016. Efficient Tracing and Versatile Analysis of Lock Contention in Java Applications on the Virtual Machine Level. In *ICPE*. 263–274.

[43] Shams M. Imam and Vivek Sarkar. 2014. Savina - An Actor Benchmark Suite: Enabling Empirical Evaluation of Actor Libraries. In *AGERE!* 67–80.

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

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

[46] Intel. 2019. Intel 64 and IA-32 Architectures Developer's Manual, Section 18.18. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-manual-325462.html.

[47] Tomas Kalibera, Jeff Hagelberg, Petr Maj, Filip Pizlo, Ben Titzer, and Jan Vitek. 2010. A Family of Real-time Java Benchmarks. *Concurrency and Computation: Practice and Experience* 23, 14 (2010), 1679–1700.

[48] Karl Pearson. 1901. On lines and planes of closest fit to systems of points in space. *Philos. Mag.* 2, 11 (1901), 559–572.

[49] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. 2012. The JVM is Not Observable Enough (and What to Do About It). In *VMIL*. 33–38.

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

[51] Doug Lea. 2012. Java Enhancement Proposal 155: Concurrency Updates. http://openjdk.java.net/jeps/155.

[52] Doug Lea. 2014. Doug Lea's Workstation. http://g.oswego.edu/

[53] Doug Lea. 2015. Java Enhancement Proposal 266: More Concurrency Updates. http://openjdk.java.net/jeps/266.

[54] Doug Lea, Joshua Bloch, Sam Midkiff, David Holmes, Joseph Bowbeer, and Tim Peierls. 2004. Java Specification Request 166: Concurrency Utilities. https://jcp.org/ja/jsr/detail?id=166.

[55] Philipp Lengauer, Verena Bitto, Hanspeter Mössenböck, and Markus Weninger. 2017. A Comprehensive Java Benchmark Study on Memory and Garbage Collection Behavior of DaCapo, DaCapo Scala, and SPECjvm2008. In *ICPE*. 3–14.

[56] David Leopoldseder, Lukas Stadler, Thomas Würthinger, Josef Eisl, Doug Simon, and Hanspeter Mössenböck. 2018. Dominance-based Duplication Simulation (DBDS): Code Duplication to Enable Compiler Optimizations. In *CGO*. 126–137.

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

[58] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. In *PLDI*. 359–373.

[59] Honghui Lu, Alan L. Cox, and Willy Zwaenepoel. 2001. Contention Elimination by Replication of Sequential Sections in Distributed Shared Memory Programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming (PPoPP '01)*. ACM, New York, NY, USA, 53–61. https://doi.org/10.1145/379539.379568

[60] 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.

[61] 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.

[62] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17, 34 (2016), 1–7.

[63] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-performance Big-data-friendly Garbage Collector. In *OSDI*. 349–365.

[64] Martin Odersky. 2011. State of Scala. http://days2011.scala-lang.org/sites/days2011/files/01.%20Martin%20Odersky.pdf.

[65] Oracle. 2018. JVM Tool Interface. https://docs.oracle.com/javase/8/docs/platform/jvmti/jvmti.html.

[66] F. Ortin, P. Conde, D. Fernandez-Lanvin, and R. Izquierdo. 2014. The Runtime Performance of invokedynamic: An Evaluation with a Java Library. *IEEE Software* 31, 4 (July 2014), 82–90.

[67] Rotem Oshman and Nir Shavit. 2013. The SkipTrie: Low-depth Concurrent Search Without Rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing (PODC '13)*. ACM, New York, NY, USA, 23–32. https://doi.org/10.1145/2484239.2484270

[68] Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotspotTM Server Compiler. In *JVM*.

[69] José Paumard. 2018. JDK8 Stream/Rx Comparison. https://github.com/JosePaumard/jdk8-stream-rx-comparison.

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

[71] Aleksandar Prokopec. 2015. SnapQueue: Lock-free Queue with Constant Time Snapshots. In *Proceedings of the 6th ACM SIGPLAN Symposium on Scala (SCALA 2015)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2774975.2774976

[72] Aleksandar Prokopec. 2016. Pluggable Scheduling for the Reactor Programming Model. In *AGERE!* 41–50.

[73] Aleksandar Prokopec. 2017. *Accelerating by Idling: How Speculative Delays Improve Performance of Message-Oriented Systems*. Springer International Publishing, Cham, 177–191. https://doi.org/10.1007/978-3-319-64203-1_13

[74] Aleksandar Prokopec. 2017. Analysis of Concurrent Lock-Free Hash Tries with Constant-Time Operations. *ArXiv e-prints* (Dec. 2017). arXiv:cs.DS/1712.09636

[75] Aleksandar Prokopec. 2017. Encoding the Building Blocks of Communication. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2017)*. ACM, New York, NY, USA, 104–118. https://doi.org/10.1145/3133850.3133865

[76] Aleksandar Prokopec. 2018. Cache-tries: Concurrent Lock-free Hash Tries with Constant-time Operations. In *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '18)*. ACM, New York, NY, USA, 137–151. https://doi.org/10.1145/3178487.3178498

[77] Aleksandar Prokopec. 2018. *Efficient Lock-Free Removing and Compaction for the Cache-Trie Data Structure*. Springer International Publishing, Cham.

[78] Aleksandar Prokopec. 2018. Efficient Lock-Free Removing and Compaction for the Cache-Trie Data Structure. https://doi.org/10.6084/m9.figshare.6369134.

[79] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Cache-Aware Lock-Free Concurrent Hash Tries*. Technical Report.

[80] Aleksandar Prokopec, Phil Bagwell, and Martin Odersky. 2011. *Lock-Free Resizeable Concurrent Tries*. Springer Berlin Heidelberg, Berlin, Heidelberg, 156–170. https://doi.org/10.1007/978-3-642-36036-7_11

[81] Aleksandar Prokopec, Phil Bagwell, Tiark Rompf, and Martin Odersky. 2011. A Generic Parallel Collection Framework. In *Euro-Par*. 136–147.

[82] Aleksandar Prokopec, Nathan Grasso Bronson, Phil Bagwell, and Martin Odersky. 2012. Concurrent Tries with Efficient Non-blocking Snapshots. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '12)*. ACM, New York, NY, USA, 151–160. https://doi.org/10.1145/2145816.2145836

[83] Aleksandar Prokopec, Gilles Duboscq, David Leopoldseder, and Thomas Würthinger. 2019. An Optimization-driven Incremental Inline Substitution Algorithm for Just-in-time Compilers. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2019)*. IEEE Press, Piscataway, NJ, USA, 164–179. http://dl.acm.org/citation.cfm?id=3314872.3314893

[84] Aleksandar Prokopec, Philipp Haller, and Martin Odersky. 2014. Containers and Aggregates, Mutators and Isolates for Reactive Programming. In *Proceedings of the Fifth Annual Scala Workshop (SCALA '14)*. ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2637647.2637656

[85] Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. 2017. Making Collection Operations Optimal with Aggressive JIT Compilation. In *SCALA*. 29–40.

[86] Aleksandar Prokopec and Fengyun Liu. 2018. On the Soundness of Coroutines with Snapshots. *CoRR* abs/1806.01405 (2018). arXiv:1806.01405 https://arxiv.org/abs/1806.01405

[87] Aleksandar Prokopec and Fengyun Liu. 2018. Theory and Practice of Coroutines with Snapshots. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands*. 3:1–3:32. https://doi.org/10.4230/LIPIcs.ECOOP.2018.3

[88] Aleksandar Prokopec, Heather Miller, Tobias Schlatter, Philipp Haller, and Martin Odersky. 2012. FlowPools: A Lock-Free Deterministic Concurrent Dataflow Abstraction. In *LCPC*. 158–173.

[89] Aleksandar Prokopec and Martin Odersky. 2014. Near Optimal Work-Stealing Tree Scheduler for Highly Irregular Data-Parallel Workloads. In *Languages and Compilers for Parallel Computing*, Călin Cascaval and Pablo Montesinos (Eds.). Springer International Publishing, Cham, 55–86.

[90] Aleksandar Prokopec and Martin Odersky. 2015. Isolates, Channels, and Event Streams for Composable Distributed Programming. In *Onward!* 171–182.

[91] Aleksandar Prokopec and Martin Odersky. 2016. *Conc-Trees for Functional and Parallel Programming*. Springer International Publishing, Cham, 254–268. https://doi.org/10.1007/978-3-319-29778-1_16

[92] Aleksandar Prokopec, Dmitry Petrashko, and Martin Odersky. 2014. On Lock-Free Work-stealing Iterators for Parallel Data Structures. (2014), 10.

[93] A. Prokopec, D. Petrashko, and M. Odersky. 2015. Efficient Lock-Free Work-Stealing Iterators for Data-Parallel Collections. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. 248–252. https://doi.org/10.1109/PDP.2015.65

[94] William Pugh, Sarita Adve, and Doug Lea. 2004. Java Specification Request 133: JavaTM Memory Model and Thread Specification Revision. https://jcp.org/ja/jsr/detail?id=133.

[95] Paruj Ratanaworabhan, Benjamin Livshits, and Benjamin G. Zorn. 2010. JSMeter: Comparing the Behavior of JavaScript Benchmarks with Real Web Applications. In *WebApps*. 3–3.

[96] John Rose. 2014. Java Enhancement Proposal 243: Java-Level JVM Compiler Interface. http://openjdk.java.net/jeps/243.

[97] John Rose, Bini Ola, William Cook, Rémi Forax, Samuele Pedroni, and Jochen Theodorou. 2011. Java Specification Request 292: Supporting Dynamically Typed Languages on the JavaTM Platform. https://jcp.org/en/jsr/detail?id=292.

[98] Andreas Schörgenhumer, Peter Hofer, David Gnedt, and Hanspeter Mössenböck. 2017. Efficient Sampling-based Lock Contention Profiling for Java. In *ICPE*. 331–334.

[99] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, Danilo Ansaloni, Walter Binder, Nathan Ricci, and Samuel Z. Guyer. 2012. new Scala() instanceof Java: A Comparison of the Memory Behaviour of Java and Scala Programs. In *ISMM*. 97–108.

[100] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. 2011. Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine. In *OOPSLA*. 657–676.

[101] Nir Shavit and Dan Touitou. 1995. Software Transactional Memory. In *PODC*. 204–213.

[102] Aleksei Shipilev. 2018. Code Tools: jmh. http://openjdk.java.net/projects/code-tools/jmh/.

[103] L. A. Smith, J. M. Bull, and J. Obdrizalek. 2001. A Parallel Java Grande Benchmark Suite. In *SC*.

[104] Diomidis Spinellis. 2005. Tool Writing: A Forgotten Art? *IEEE Software* 4 (2005), 9–11.

[105] Sriram Srinivasan and Alan Mycroft. 2008. Kilim: Isolation-Typed Actors for Java. In *ECOOP*. 104–128.

[106] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. 2014. Partial Escape Analysis and Scalar Replacement for Java. In *CGO*. 165:165–165:174.

[107] Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *PLDI*. 263–277.

[108] Jaroslav Ševčík. 2011. Safe Optimisations for Shared-memory Concurrent Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 306–316. https://doi.org/10.1145/1993498.1993534

[109] Markus Weninger and Hanspeter Mössenböck. 2018. User-defined Classification and Multi-level Grouping of Objects in Memory Monitoring. In *ICPE*. 115–126.

[110] Benjamin P. Wood, Man Cao, Michael D. Bond, and Dan Grossman. 2017. Instrumentation Bias for Dynamic Data Race Detection. *Proc. ACM Program. Lang.* 1, OOPSLA (Oct. 2017), 69:1–69:31.

[111] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*. 10–10.

[112] Xinghui Zhao and Nadeem Jamali. 2013. Load Balancing Non-uniform Parallel Computations. In *AGERE!* 97–108.

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