Renaissance: Benchmarking Suite for Parallel Applications on the JVM

Aleksandar Prokopec[1], Andrea Rosà[2], David Leopoldseder[3], Gilles Duboscq[1], Petr Tůma[4], Martin Studener[3], Lubomír Bulej[4], Yudi Zheng[1], Alex Villazón[5], Doug Simon[1], Thomas Würthinger[1], Walter Binder[2]

[1]Oracle Labs, Switzerland, [2]Università della Svizzera italiana, Switzerland, [3]Johannes Kepler Universität Linz, Austria, [4]Charles University, Czech Republic, [5]Universidad Privada Boliviana, Bolivia

ORACLE Labs

Università della Svizzera italiana

JOHANNES KEPLER UNIVERSITY LINZ

CHARLES UNIVERSITY

UPB Garantía de Excelencia

Renaissance Suite

- A modern, open, and diversified benchmark suite for the JVM
- Focused on **concurrency** and **parallelism**
- Contains modern workloads and popular systems, frameworks and applications
- Exercises many programming paradigms: concurrent, parallel, functional and object-oriented programming
- Aimed at testing JIT compilers, garbage collectors, profilers, analyzers, …

- Main publication:
  - A. Prokopec et al., "Renaissance: Benchmarking Suite for Parallel Applications on the JVM". **PLDI 2019**.

Motivation

- JVM is evolving
  - New JVM features:
    - concurrency, lambdas, method-handles, non-blocking I/O, …
  - New paradigms:
    - data-parallelism, fork-join, streams, asynchronous programming with futures, transactional memory, ...
  - New frameworks:
    - Big Data (Spark), stream-processing (JDK8 Streams, Rx), actors (Akka), network communication (Netty), …
- … many of them not represented by existing benchmark suites

Motivation

- JIT compilers show similar performance on existing suites
  - Need for new workloads to demonstrate performance gains when exploring compiler optimizations
- JIT optimizations rarely focus on concurrency-related primitives
  - Need for new workloads including concurrency- and parallelism-related constructs

Selection Criteria

- Use of modern concurrency primitives:
  - data-parallelism, task-parallelism, streaming and pipelined parallelism, message-based concurrency, software transactional memory, lock-based and lock-free concurrent data structures, in-memory databases, asynchronous programming, network communication
- Realistic workloads using popular frameworks:
  - Java Streams, Apache Spark, Java Reactive Extensions, Java Fork/Join framework, ScalaSTM, Twitter Finagle

Selection Criteria

- Workload diversity
  - Exercise different concurrency-related features while making use of object-oriented abstractions
- Deterministic execution
- Open-source availability
  - Enable inspection by the community, source-code level analysis, evaluation of actionability of profiler results
- Avoid known pitfalls of other suites
  - Lack of benchmark source code, timeouts, resource leaks, data races, deadlocks

Selection Criteria

- Gathered around 100 candidate workloads
  - Manual search
  - Automatic search with NAB [1]
- Selected 25 benchmarks from them

[1] A. Villazón et al. "Automated Large-scale Multi-language Dynamic Program Analysis in the Wild". ECOOP 2019.

Benchmark List (v 0.10)

Benchmark	Description	Focus
akka-uct	Unbalanced Cobwebbed Tree computation using Akka.	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.	data-parallel, machine learning
db-shootout	Parallel shootout test on Java in-memory databases.	query-processing, data structs.
dec-tree	Classification decision tree algorithm using Spark ML.	data-parallel, machine learning
dotty	Compiles a Scala codebase using the Dotty compiler.	data-structures, synchronization
finagle-chirper	Simulates a microblogging service using Twitter Finagle.	network stack, futures, atomics
finagle-http	Simulates a high server load with Twitter Finagle and Netty.	network stack, message-passing
fj-kmeans	K-means algorithm using the Fork/Join framework.	task-parallel, conc. data structs.
future-genetic	Genetic algorithm function optimization using Jenetics.	task-parallel, contention
gauss-mix	Computes a Gaussian mixture model.	machine learning
log-regression	Performs logistic regression on a large dataset.	data-parallel, machine learning
mnemonics	Solves the phone mnemonics problem using JDK streams.	streaming
movie-lens	Recommender for the MovieLens dataset using Spark ML.	data-parallel, compute-bound
naive-bayes	Multinomial Naive Bayes algorithm using Spark ML.	data-parallel, machine learning
neo4j-analytics	Analytical queries and transactions on the Neo4J database.	query processing, transactions
page-rank	PageRank using the Apache Spark framework.	data-parallel, atomics
par-mnemonics	Solves the phone mnemonics problem using parallel streams.	parallel streaming
philosophers	Dining philosophers using the ScalaSTM framework.	STM, atomics, guarded blocks
reactors	A set of message-passing workloads encoded in Reactors.	actors, msg-passing, critical sect.
rx-scrabble	Solves the Scrabble puzzle using the RxJava framework.	streaming
scala-kmeans	Runs the K-Means algorithm using Scala collections.	machine learning
scala-stm-bench7	STMBench7 workload using the ScalaSTM framework.	STM, atomics
scrabble	Solves the Scrabble puzzle using Java 8 Streams.	data-parallel, memory-bound
streams-mnemonics	Computes phone mnemonics using Java 8 Streams.	data-parallel, memory-bound

Diversity

- Goal: ensure that Renaissance
  - represents concurrency primitives better than existing suites
  - exercises object-oriented abstractions similarly to existing suites
- Comparison with DaCapo [1], ScalaBench [2], SPECjvm2008 [3]

[1] S. Blackburn et al., "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". SIGPLAN Not. 41, 10 (Oct. 2006).
[2] A. Sewe et al., "Da Capo Con Scala: Design and Analysis of a Scala Benchmark Suite for the Java Virtual Machine". OOPSLA 2011.
[3] SPECjvm2008. https://www.spec.org/jvm2008/

Diversity

- Metrics:

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.

Concurrency primitives

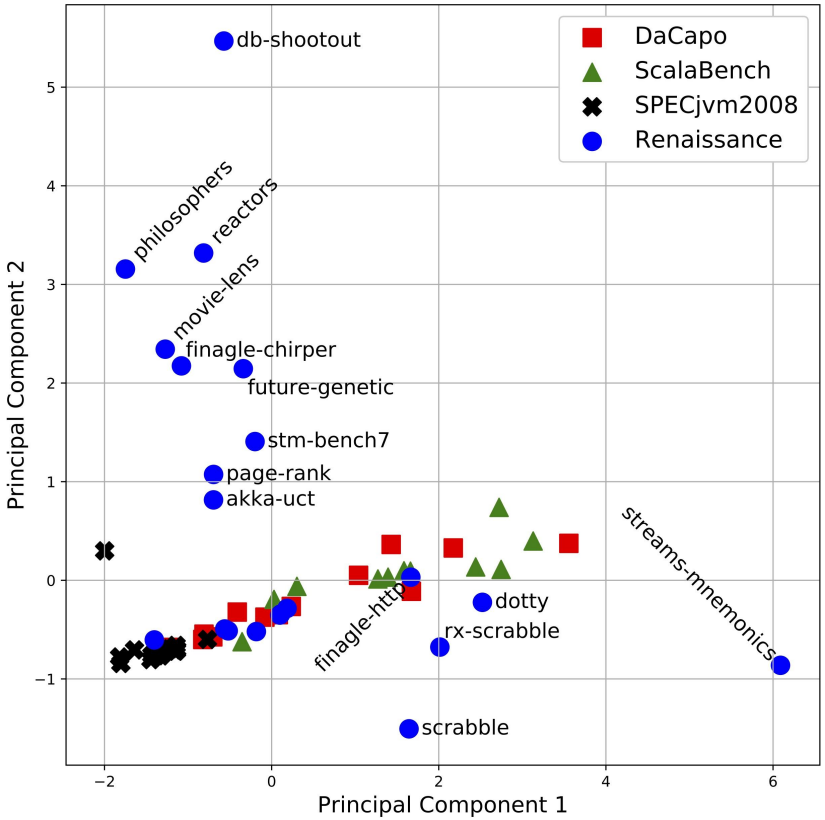
Object-oriented programming

New JVM features

Diversity

- Metric collection on all benchmark suites
- Normalization over reference cycles
- Standardization to a [-1, 1] range
- Principal Component Analysis (PCA)
  - Goal: visually demonstrate benchmark diversity

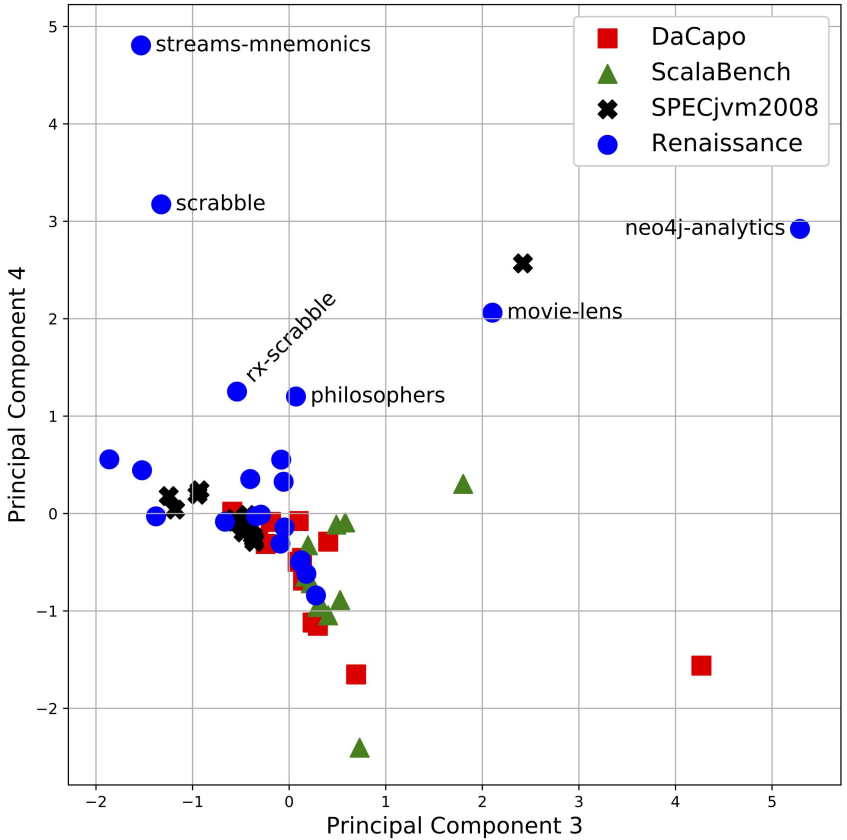
Principal Component Analysis



PC1		PC2	
Metric	**Loading**	**Metric**	**Loading**
object	**+0.50**	**atomic**	**+0.67**
cpu	**-0.49**	**park**	**+0.65**
method	**+0.44**	method	+0.20
array	**+0.40**	notify	+0.18
idynamic	+0.27	idynamic	-0.17
synch	-0.17	cpu	-0.16
notify	-0.13	cachemiss	-0.08
atomic	-0.13	object	+0.05
cachemiss	-0.07	array	-0.03
park	-0.06	synch	-0.02
wait	-0.02	wait	-0.00

- PC1: object-oriented programming
  - Renaissance similar to other suites
- PC2: concurrency primitives
  - Renaissance much more spread along PC2

Principal Component Analysis



PC3 Metric	PC3 Loading	PC4 Metric	PC4 Loading
cachemiss	**+0.58**	**idynamic**	**+0.56**
notify	**+0.50**	array	+0.42
wait	**+0.41**	notify	+0.42
cpu	-0.28	method	-0.35
synch	-0.25	cachemiss	+0.28
park	-0.20	cpu	+0.22
idynamic	-0.18	atomic	+0.18
array	-0.13	wait	-0.15
method	+0.10	object	+0.13
object	-0.04	synch	+0.11
atomic	-0.03	park	+0.05

- PC3: concurrency primitives
  - Renaissance more spread along PC3
- PC4: invokedynamic
  - Reflects functional-style operations

Principal Component Analysis

- Conclusion: Renaissance
  - is comparable to DaCapo and ScalaBench in terms of object-allocation rates and dynamic dispatch
  - represents concurrency primitives better than existing suites
  - exercises invokedynamic more often

Compiler Optimizations

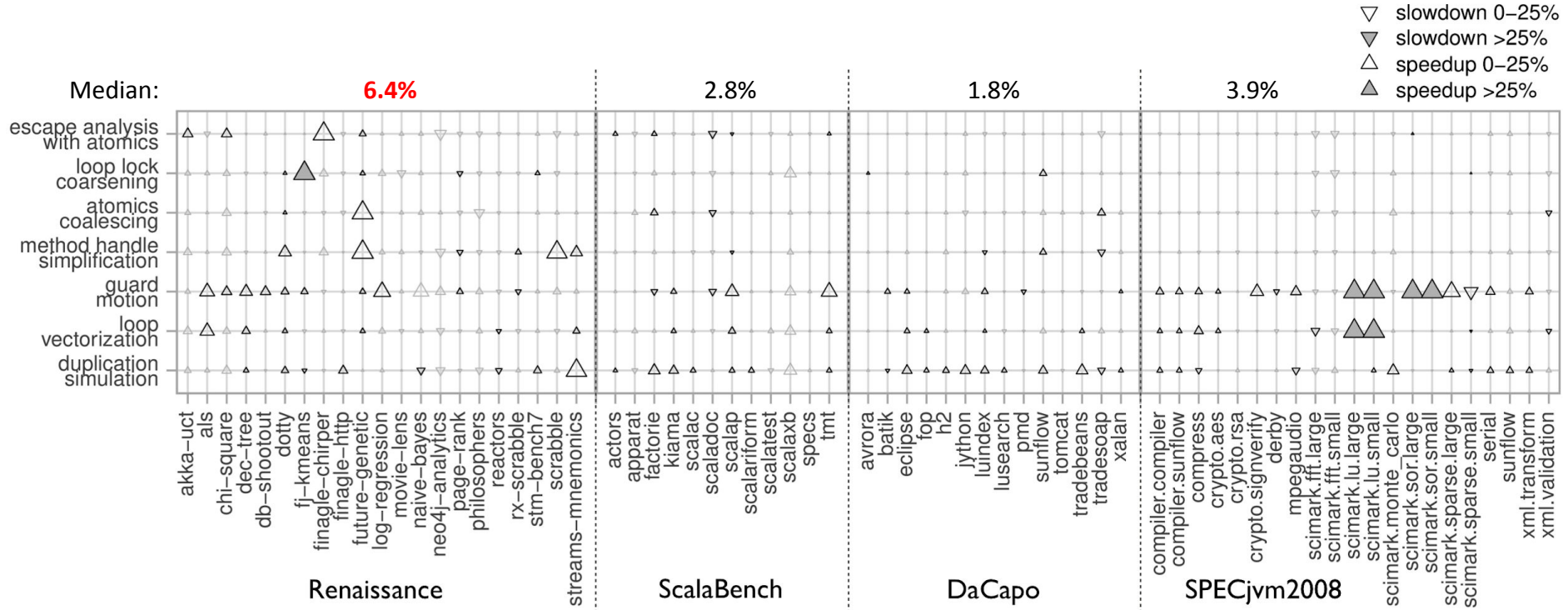
- Goal (1): prove the utility of Renaissance by implementing new compiler optimizations
  - Optimizations inspired by the code patterns found in Renaissance
- Result: 4 new compiler optimizations
  - Escape analysis with atomic operations
  - Loop-wide lock coarsening
  - Atomic-operation coalescing
  - Method-handle simplification
- Optimizations implemented in the Graal JIT compiler [1]

[1] G. Duboscq et al., "An Intermediate Representation for Speculative Optimizations in a Dynamic Compiler". VMIL 2013.

Compiler Optimizations

- Goal (2): show that Renaissance is more sensitive to existing compiler optimizations
  - Renaissance can be used to better evaluate JIT compilers
- 3 existing optimizations
  - Speculative guard movement
  - Loop vectorization
  - Dominance-based duplication simulation

Compiler Optimizations



Compiler Optimizations

- Conclusion:
  - The considered optimizations have a significant impact on Renaissance benchmarks, while other suites benefit less from the optimizations
- More details in the paper

Conclusions

- We presented Renaissance, a new modern, open, and diversified benchmark suite for the JVM focused on concurrency and parallelism
- Renaissance contains diversified workloads
- Renaissance inspired 4 new compiler optimizations
- Renaissance is more sensitive to 3 existing compiler optimizations than existing benchmark suites

Conclusions

- Renaissance can be downloaded at **https://renaissance.dev/**
- Renaissance is an open suite
  - Open source, contributions are welcome!
  - Community can propose new benchmarks
  - Committee votes on benchmarks for the next release
- Goal: keep improving and evolving the suite in an open manner

Conclusions

Thanks for your attention

- Contacts:

Walter Binder

walter.binder@usi.ch