Selected Challenges in Concurrent and Distributed Programming

Philipp Haller

KTH Royal Institute of Technology
Stockholm, Sweden

Dagstuhl Seminar 19442 on Programming Languages for Distributed Systems and Distributed Data Management

Philipp Haller: Background

- Associate professor at KTH (2014–2018 assistant professor)
  - PhD 2010 EPFL, Switzerland
- 2005–2014 ***Scala language team***
  - 2012–2014 Typesafe, Inc. (now Lightbend, Inc.)
- Co-author Scala language specification
- Focus on ***asynchronous, concurrent and distributed programming***
  - Creator of Scala actors, co-author of Scala's futures and Scala Async
  - *Topics:* programming languages, concurrent and distributed programming, type systems, semantics, static analysis

Goals

- Programming languages for distributed systems that provide high scalability, reliability, and availability
- Prevent bugs in distributed systems

Challenge 1: Ensuring Fault-Tolerance Properties

- Specific fault-tolerance mechanism:
  Lineage-based fault recovery
  - Lineage records dataset identifier plus transformations
  - Maintaining lineage information in available, replicated storage enables recovering from replica failures
- ***A widely-used fault-recovery mechanism*** (e.g., Apache Spark)

How to *statically* ensure fault-tolerance properties for languages based on lineage-based fault recovery?

Lineage-based Fault Recovery: Results

- Proof establishing the ***preservation of lineage mobility***
- Proof of ***finite materialization of remote, lineage-based data***
- P. Haller, H. Miller, N. Müller: **A programming model and foundation for lineage-based distributed computation**
  J. Funct. Program. 28: e7 (2018)

Challenge 2: Data Consistency

- In order to satisfy latency, availability, and performance requirements of distributed systems, developers use ***variety of data consistency models***
  - Theoretical limit given by CAP theorem[1]
- There is no one-size-fits-all consistency model

How to *safely* use both consistent and available (but inconsistent) data within the same application?

[1] Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33(2), 51-59 (2002)

Consistency Types: Idea

To satisfy a range of performance, scalability, and consistency requirements, provide two different kinds of replicated data types

1. ***Consistent data types***:
   - Serialize updates in a global total order: ***sequential consistency***
   - ***Do not provide availability*** (in favor of partition tolerance)
2. ***Available data types***:
   - Guarantee ***availability and performance*** (and partition tolerance)
   - ***Weaken consistency***: strong eventual consistency

Consistency Types in LCD

First-class functions

Replicated data types

LCD:

- A higher-order language with distributed references and ***consistency types***
- Values and types annotated with ***labels indicating their consistency***

$$
\begin{array}{lcl}
\ell & ::= & \cdot \mid \mathsf{con} \mid \mathsf{ava} \\
t & ::= & v \mid t \bigoplus t \mid t \ \mathsf{op} \ t \mid t \ t \mid \mathtt{if} \ x \ \mathtt{then} \\
 & & \mid \mathsf{ref}_\ell \ t \mid !t \mid t := t \\
r & ::= & \mathsf{d} \mid \mathsf{true} \mid \mathsf{false} \mid (\lambda^\ell x : \tau .\ t) \mid \mathsf{unit} \\
v & ::= & r_\ell \mid x \\
\tau & ::= & \mathsf{Bool}_\ell \mid \mathsf{Unit}_\ell \mid \mathsf{Lat}_\ell \mid \mathsf{Ref}_\ell \ \tau \mid \tau \xrightarrow{\ell}_\ell \tau \\
\bigoplus & ::= & \vee \mid \wedge \\
\mathsf{op} & ::= & \preceq \mid \prec
\end{array}
$$

- Typed lambda-calculus
- ML-style references
- Labeled values and types

Consistency Types: Results

LCD: a higher-order language with replicated types and consistency labels

- Consistency types enable ***safe use*** of both strongly consistent and available (weakly consistent) data within the same application
- Proofs of ***type soundness*** and ***noninterference***
- Noninterference:
  Cannot observe mutations of available data via consistent data
- Paper to appear in proceedings of LCPC 2019 (Zhao & Haller 2019)

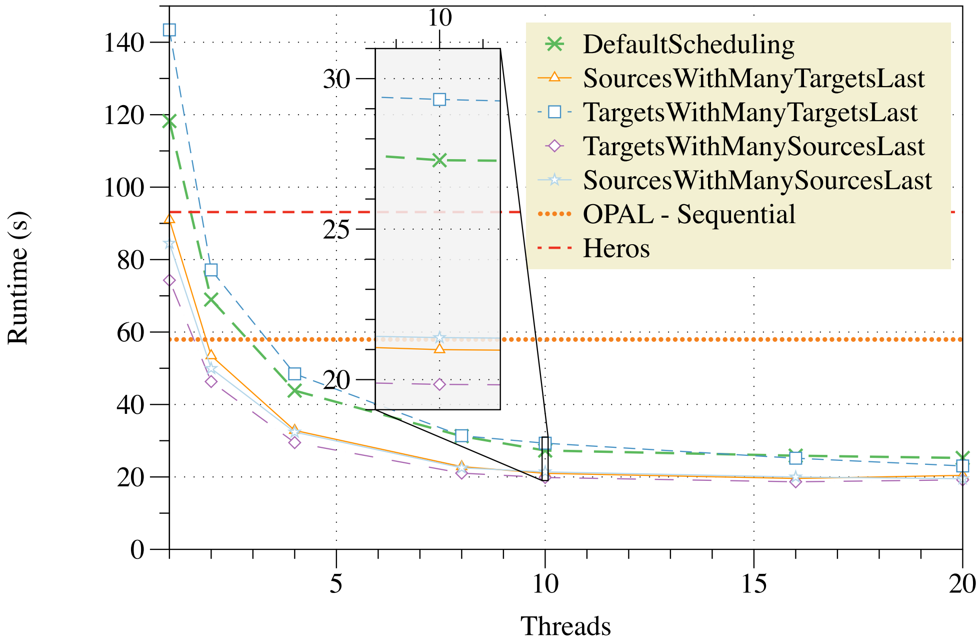
Challenge 3: Parallel Programming

- Increasing importance of static analysis
  - Bug finding, security analysis, taint tracking, etc.
- Precise and powerful analyses have ***long running times***
  - Infeasible to integrate into nightly builds, CI, IDE, …
  - ***Parallelization difficult:*** advanced static analyses not data-parallel
- Scaling static analyses to ever-growing software systems requires ***maximizing utilization of multi-core CPUs***

The Approach

- Novel ***concurrent programming model***
  - Generalization of futures/promises
  - Guarantees deterministic outcomes *(if used correctly)*
- Implemented in Scala
  - Statically-typed, integrates functional and object-oriented programming
  - Supported backends: JVM, JavaScript (+ experimental native backend)
- Integrated with OPAL, a state-of-the-art ***JVM bytecode analysis framework***

Parallel Static Analysis: Results



Analysis executed on Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz (10 cores) using 16 GB RAM running Ubuntu 18.04.3 and OpenJDK 1.8_212

Conclusion

- Challenge: building distributed systems providing high scalability, reliability, and availability
  - System builders use various ***unsafe techniques*** to achieve these properties
  - How can we support system builders and prevent bugs?
- Thesis:
  Programming language techniques can help!
  - ***Language constructs, abstractions***
    - for composing systems modularly
    - for exploiting parallelism, replication, etc.
  - ***Type systems and static analysis*** for preventing hard-to-reproduce bugs