Analyzing Serializabilty of Cassandra Applications

Arthur Kurath
Supervisor: Lucas Brutschy

October 8, 2016

1 Introduction

It is often challenging to develop applications that use eventually consistent data stores due to the presence of unexpected behaviors that do not occur under strong consistency guarantees. Recent work [3] proposed a new serializability criterion for clients of eventually consistent data stores, and evaluated it using a dynamic analysis, meaning traces from concurrently running clients were collected and the criterion was applied in order to find out if the given execution was serializable. The goal of this masters project is to build a static analysis which applies the criterion to an abstraction of all possible traces of a given client, thus determining if all executions of such clients are serializable.

In this project, we target programs performing queries and updates on Apache Cassandra [1], a scalable open source database. In Cassandra, data is distributed and replicated across many nodes which can be placed in multiple data centers. Cassandra extends the concept of eventual consistency by offering tunable consistency, which means that the client application decides how consistent the requested data must be. For example, when writing or reading data, one can specify the number of replicas that need to respond to the request before the result of the operation is returned. Additionally, Cassandra offers a linearizable compare-and-set operation. This mechanism can be used if a value should only be written if a certain condition is met, for example to implement that a new user should only be inserted in the database if no other user with the same username is existing already. In addition to basic types like numbers, strings or dates, Cassandra also offers sets, maps and lists as conflict-free replicated data types. There also exists a counter type. [1, 2]

2 The Serializability Criterion

An eventually consistent schedule is characterized by visibility and arbitration. Informally, an update is *visible* to a query if it is included in the evaluation of the result of the query. The order in which update conflicts are eventually resolved by the system, is called *arbitration*. Given an eventually consistent schedule, one can find dependencies and anti-dependencies between these updates and queries. A query *depends* on an update if the returned data of the query would change if a visible update would not be visible. An update has an *anti-dependency* to a query if the result of the query would change if an invisible update would become visible to the query (i.e. making an update visible to a query would introduce a new dependency).

Given a schedule, one can determine dependencies and anti-dependencies taking commutativity and absorption into account. Two actions (updates / queries) *commute*, if the order in which the actions are executed does not change their result. An action u *absorbs* v, if the result of executing v and then u is the same as executing only u. Using these notions, a visible update u is a dependency of a query q unless commutativity and absorption can be applied as rewrite rules repeatedly to obtain a schedule where u is not visible to q. An invisible update u anti-depends on a query q if q would depend on u when u would be visible.

All existing dependencies and anti-dependencies together with arbitration and program order can be combined in a directed graph, known as the dependency serialization graph (DSG). If this graph does

[1] http://cassandra.apache.org/

not contain a cycle, the given execution is serializable. One can construct a serial schedule of the events by ordering them so that the partial orderings of the DSG are preserved. If the DSG contains at least one cycle, it is possible that the execution is not serializable.

3 Core Goals

In this project, a static analysis is developed that builds a graph, the static dependency serialization graph (SDSG), that abstracts dependencies and anti-dependencies of all possible executions of a given program meeting the following requirements:

- **Programming Language:** The program is written in Java.
- **Database Driver:** Access to Cassandra is done using the datastax driver [2].
- **Queries:** Queries are either built using plain CQL or using the QueryBuilder class of the datastax driver.

The analysis takes as an input the source of the program and a list of entry points (e.g. a process-request method in the case of a web application). Dependencies and anti-dependencies are then determined using the following information, which are defined beforehand or collected during the analysis using off-the-shelf techniques:

- Commutativity and absorption rules for the operations issued by the datastax driver
- Pointer and aliasing information
- Set of possible values for a given variable at a given program point

In order to build the static dependency serialization graph, it may be necessary to create stubs for library methods. If assumptions are made that introduce unsoundness, these should be clearly stated. Given the SDSG, cycles are detected and reported. These cycles indicate actions that may not be serializable when executed concurrently.
The analysis is then evaluated on several real programs found in public code repositories. It may be necessary to modify programs to use the datastax driver instead of another one in order to find a representative set of programs.

4 Extensions

Systematic Manual Inspection: It is likely that the static analysis reports cycles that do not reveal real bugs. In order to reduce these false positives, it is necessary to determine which (anti-)dependencies in the SDSG do not exist in the concrete schedules of the analyzed programs. Once these are known, the next step is to find out which information is necessary in order to prevent the creation of these edges in the SDSG.

Annotations: In order to improve the precision of the program, it may be necessary that information is needed that could be supplied easily by the programmer. For example, a constraint in the database is maintained by the program and can thus be assumed by the analysis. Or a variable in the program is unique over all running clients (for example in the case of a unique session identifier). There should be a set of annotations which can be used by the programmer to express such constraints in the database or in the program. These annotations should then increase the precision of the analysis.

Infer Annotations: Once the analysis supports a set of annotations, it should be examined if some of them can (partially) be inferred automatically. If this is the case for certain annotations, inference should be implemented where it is possible.

[2] https://github.com/datastax/java-driver

5 Schedule

Task	Date	Duration (months)
Search for example programs, explore related work	01.10.2016	0.5
Implement basic analysis	15.10.2016	1.5
First evaluation, imprecision analysis	01.12.2016	0.5
Improve precision, implement annotations	15.12.2016	1.5
Infer annotations	01.02.2017	1
Finish report	01.03.2017	1
End of project	01.04.2017	

References

[1] CQL for Apache Cassandra 2.0 & 2.1. http://docs.datastax.com/en/cql/3.1/.

[2] Documentation of Apache Cassandra 2.1. http://docs.datastax.com/en/cassandra/2.1/.

[3] Lucas Brutschy, Dimitar Dimitrov, Peter Mueller and Martin Vechev. Serializability for Eventual Consistency: Criterion, Analysis and Applications.