Statically Checked Assertions for TESLA

A Part III project proposal

B. S. Collie (*bsc28*), Trinity Hall

Project Supervisor: Dr. Robert Watson

Abstract

Traditional software instrumentation methods are "instantaneous"—they only deal with program state at a particular point of the execution. TESLA [1] provides a library for dynamically checking temporal properties of C programs, allowing for more thorough assertions about program behaviour to be made. These assertions are made inline within a program through the use of a language extension implemented using Clang and LLVM, and are then compiled into automata that provide runtime checking and instrumentation. I propose an extension of the TESLA library to allow for static checking of such properties, with the aim of decreasing the runtime overhead of this kind of instrumentation.

1 Introduction, approach and outcomes

In their 2014 paper, Anderson et al. [1] present a method for dynamically checking the behaviour of C programs. TESLA ("*Temporally Enhanced System Logic Assertions*") allows for assertions about program behaviour in the past or the future to be made—for example, asserting that a particular function had been called successfully prior to reaching an assertion site. These assertions are written inline using an extension to the C language implemented with Clang and LLVM, and are evaluated purely at runtime.

In this project I propose to investigate static checking and analysis of TESLA assertions. In particular, I aim to characterise the subset of existing TESLA assertions that can be checked statically (and the accuracy / precision with which this characterisation can be made). To do this I will look primarily at how the constructed TESLA automata can be statically checked against representations of the code. The most feasible implementation strategy here is to write LLVM passes that analyse control and data flow in the IR, then remove or modify the generated TESLA automata code based on the static properties of the code. These passes will be written with the aim of analysing a particular subset of checks with respect to some property—a simple example is that removing checks that a function has been called in the past is possible if every path through the control flow graph includes a call to that function before the assertion site is reached. As well as the LLVM IR, similar analysis may well be possible on other representations (such as the Clang AST).

In the original TESLA paper, the authors provide several practical uses of TESLA along with performance analyses. As part of my research, I will aim to reproduce these experimental results so that the performance impact of my work can be properly evaluated. Doing this will involve updating the TESLA codebase to use current versions of the libraries used to implement it (primarily LLVM and Clang). I have verified with the original authors that this update work is feasible and will not consume

an undue amount of project time. A deliverable at this stage is a version of TESLA that can be built with up-to-date dependencies.

Performing runtime checks incurs a performance penalty—in a FreeBSD kernel instrumented using TESLA, slowdown from $1.1-7\times$ was experienced, with the worst performance on microbenchmarks. Once the dependencies have been updated, my next aim will be to instrument and verify properties about a piece of software that is currently infeasible to validate because of this runtime impact. Two potential areas of investigation here (identified in the original paper) are lock assertions and the TCP implementation in the FreeBSD kernel. Building such a verification may require extension of the TESLA syntax to better express the required behaviour (for example, more explicit description of a state machine). The deliverable at this stage will be this instrumented software together with a demonstration that static analysis entails a performance improvement.

Evaluating the performance impact of my modifications to TESLA will involve examining both the compile-time and runtime—to do this, suitable evaluation workloads will need to be chosen. For example, to evaluate the impact of TESLA on lock-checking overhead, a workload with high concurrent contention would be necessary. As the software being instrumented will have long compilation times (partly due to the size of the codebase, and partly due to TESLA overheads), evaluation will most likely be performed on dedicated server hardware, as was the case in the original paper.

2 Workplan

Block 1 (28 Nov - 4 Dec) Begin bringing TESLA up to date with the current versions of LLVM and Clang. Become familiar with TESLA assertion language and runtime by instrumenting small programs. Investigate reproduction of initial experimental performance results.

Block 2 (5 - 18 Dec) Finish gathering reproduced performance data from the previously instrumented libraries, identifying any problems with the updated version of TESLA if they arise. Begin investigating possible candidates for a new library to instrument.

Block 3 (19 Dec - 1 Jan) Identify where TESLA can be used to formalise internal assumptions made by the library identified in the previous block (e.g. implicit state machine code that can be made more explicit). Less work scheduled for this block due to the festive period.

Block 4 (2 - 15 Jan) Instrument library based on previous block's work, and gather performance data as before. Begin writing initial static analysis passes to optimise existing TESLA automata code where properties can be verified appropriately.

Block 5 (16 - 29 Jan) Continue investigating static analysis techniques, along with evaluation of their effect on execution and compilation performance of the libraries used so far for evaluation.

Block 6 (30 Jan - 12 Feb) Further work on more sophisticated static analysis techniques, along with the associated performance analyses. A potential challenge at this stage is investigating and dealing with false positives / negatives given by the static analysis.

Block 7 (13 - 26 Feb) Begin investigating how new TESLA syntax could be used to improve the instrumentation added so far. Look into how new assertion syntax can be added to TESLA, with a focus on fine-grained assertions that can be statically checked effectively (such as state machine code or lock checking).

Block 8 (27 Feb - 12 Mar) Continue implementation of new assertion syntax. Identify a piece of software that would have been infeasible to instrument previously (e.g. a TCP implementation), and demonstrate why this is the case by comparing instrumented with uninstrumented performance. As before, characterising false positives / negatives in the static analysis is important at this stage.

Block 9 (13 - 26 Mar) Develop examples showing performance gains through static analysis and new assertion syntax. Continue to work on instrumentation and performance analysis using the new assertions.

Block 10 (27 Mar - 9 Apr) Collate results from newly instrumented software (for example, any latent bugs that have been found or formalisms added to the code). Time for remaining development if necessary.

Block 11 (10 - 23 Apr) Finalise any remaining development and analysis work. Start to structure dissertation and gather results together.

Block 12 (24 Apr - 7 May) Writing up.

Block 13 (8 - 21 May) Writing up.

Block 14 (22 May - 4 June) *Contingency*

References

[1] Jonathan Anderson et al. “TESLA: Temporally Enhanced System Logic Assertions”. In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys ’14. New York, NY, USA: ACM, 2014, 19:1–19:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592801. URL: http://doi.acm.org/10.1145/2592798.2592801 (visited on 10/04/2016).