Statically Checked Assertions for TESLA

Bruce Collie

Part III Computer Science
Trinity Hall
June 27, 2017

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for Part III of the Computer Science Tripos

Declaration

I, Bruce Collie of Trinity Hall, being a candidate for the Part III in Computer Science, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose.

Total word count: 11,993

Signed:

Date:

Abstract

TESLA [2] is a compiler-based tool that allows temporal run-time assertions to be written about C programs, expressing richer safety properties than traditional tools can. However, adding temporal assertions to a program incurs a significant runtime performance overhead. As a result, the usefulness of TESLA is limited to debugging scenarios where this overhead is acceptable.

I formalise the assertion language used by TESLA, then use the formalism to develop a *model checker* (TMC) for a subset of its assertions, demonstrating its usefulness by applying it to a TESLA-instrumented implementation of mutual exclusion locks. I analyse the usefulness of TESLA with respect to a large, widely-used open source library, finding that common source-code idioms inhibit the useful application of TESLA. Based on these results, I improve on previous work by describing a general method for applying TESLA to library interfaces, then apply the method successfully to an existing server application. To evaluate TMC, I analyse the performance overhead of TESLA instrumentation in this server, finding a 40% overhead for as few as five assertions. Applying TMC to prove these assertions at compile time eliminates the run-time overhead completely while retaining the asserted safety properties.

Contents

1 Introduction

The tools available for programmers to assert the correctness of their code are almost always *instantaneous*—assertions can be made about the current state of the program, but not about previous or future states. As a result, *temporal* properties of programs are often checked informally using manual instrumentation, or not at all.

TESLA [2] provides C-language systems programmers with a way of mechanically checking temporal properties of their code by using an augmented compilation proces. This approach proved successful—a number of bugs in open-source libraries were identified and fixed with the help of temporal assertions. However, using TESLA imposes significant run-time performance overhead on a program (up to 7× on some workloads). As a result, TESLA has only been useful as a debugging tool in research contexts.

In this dissertation I propose the use of *static analysis* for optimisation of TESLA assertions—if an assertion can be proved correct at compile time, then its instrumentation code can be omitted from the program. The program is likely to be smaller and faster than if the instrumentation were included, and potential counterexamples to assertions can be used by the programmer as a useful debugging tool in their own right.

To demonstrate the utility of static analysis, I provide an implementation of TESLA-instrumented mutual exclusion locks along with benchmarks that show performance overhead of more than 10% for a single assertion. I then contribute a translation of TESLA assertions to finite state automata that formalises components of the original work, and use this translation to implement a model checker for a subset of TESLA assertions: TMC (TESLA Model Checker).

I investigate how TESLA can be applied to the internal implementation of TCP in LWIP [19], a widely-used library implementing a complete, portable IP protocol stack. Then, motivated by the difficulties encountered during this process, I describe a general framework for using TESLA to instrument library interface code. This represents a more general approach to using TESLA than earlier work.

To evaluate, I apply this technique to an existing application written using LWIP, adding TESLA assertions to validate its usage of TCP protocol code. Architectural and microarchitectural benchmarks show performance overhead of up to 40% from five TESLA assertions. This overhead can be completely eliminated by TMC, demonstrating that it is a valuable contribution towards greater applicability of TESLA.

2 Background

In this chapter I give a summary of previous work related to TESLA on which this project builds, a short overview of the practical issues associated with using TESLA and an introduction to the TESLA assertion language. I also provide a guide to TESLA-specific terminology.

2.1 Temporal Assertions

Before describing TESLA in detail, it is worth giving a motivating example of why it is useful. A simple explanation is that it allows the programmer to make assertions about events that occur in the past and future, rather than just about the current program state.

Figure 2.1 shows C functions for acquiring and releasing a mutual exclusion lock.[1] For a program to make progress, it should eventually release the lock after it has been acquired. However, within `lock_acquire`, there is no way of asserting this property using standard C constructs—the call that releases the lock could be logically separated from the call that acquires it (for example, a library function could acquire the lock then depend on user code releasing it).

TESLA allows for *temporal* properties to be expressed. Figure 2.2 shows the same lock acquisition function, but with a TESLA assertion enforcing the safety property. The property is defined along with the function it applies to, and is independent of where calls to `lock_acquire` and `lock_release` are made. The remainder of this chapter describes TESLA assertions and the associated tooling in detail.

2.2 Summary of Existing Work

> TESLA is a description, analysis, and validation tool that allows systems programmers to describe expected temporal behaviour in low-level languages such as C. [2, p. 1]

[1] Modulo a suitable type `struct lock_t` and a correct implementation of atomic compare-and-swap (`CAS`).

```
void lock_acquire(struct lock_t *lock) {
  return CAS(&lock->locked, false, true);
}

void lock_release(struct lock_t *lock) {
  lock->locked = false;
}
```

Figure 2.1: Lock operations without progress property enforced

```
void lock_acquire(struct lock_t *lock) {
  TESLA_WITHIN(main, eventually(
    lock_release(lock)
  ));
  return CAS(&lock->locked, false, true);
}

void lock_release(struct lock_t *lock) {
  TESLA_WITHIN(main, previously(
    lock_acquire(lock) == 1;
  ));
  lock->locked = false;
}
```

Figure 2.2: Lock operations with progress property enforced using TESLA

Anderson et al. [2] introduce TESLA as a tool for validating safety[2] properties of systems code. These properties are written inline with the program they describe, and are checked at run time by instrumentation code added during an extra compilation phase.

The authors detail their experiences using TESLA to perform complex debugging on large, well-known systems software. Their efforts were successful—the discovery of a known security vulnerability in OpenSSL was reproduced using TESLA, and an elusive bug in the GNUStep graphics library was diagnosed and fixed. Additionally, they detail the overhead associated with using TESLA (at compile and run time) and identify static analysis as a possible future direction of research.

At the time I began work on this project, TESLA comprised a parser for assertions written in C programs, a compiler-based instrumentation tool, and a runtime library (`libtesla`). The primary contribution of my work is a model checker for TESLA assertions (TMC). Figure 2.3 shows how these components interact with each other. Existing

[2] A *safety* property asserts that "bad things" do not happen during the execution of a program, while a *liveness* property asserts that "good things" do eventually happen [1, 26].

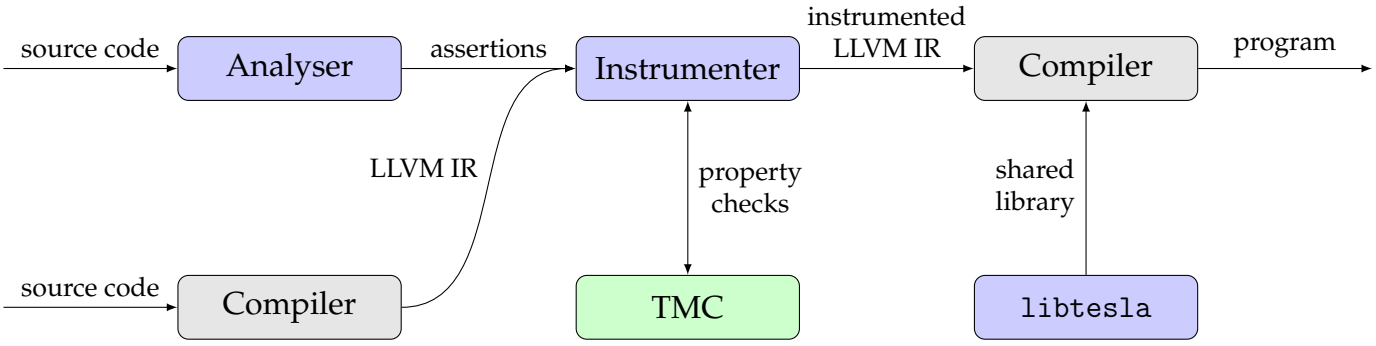

Figure 2.3: TESLA system components

components are highlighted in blue, and my contribution in green. As well as TMC, I have contributed several bug fixes and improvements to the existing TESLA components.

2.3 Programming with TESLA

In this section I give a brief overview of how TESLA is used in practice to instrument programs.

2.3.1 Terminology

Today, programmers may add *assertions* to their code to ensure its correctness—these are logical statements predicated on data in the current scope. TESLA assertions express temporal relations between *program events*. Such assertions require a *bounding interval*—a pair of start and end events that limit the scope of the assertion. In Figure 2.2, the bounding interval is from each call to `main` to the corresponding return.

Each TESLA assertion defines an *automaton*, and a collection of these automata is referred to as a *manifest* when serialised to disk. An *assertion site* is the source location where an assertion was originally written, and a *function event* is a call to or return from a function.

2.3.2 Build Process

Using TESLA to instrument a program requires that its build process is modified to produce TESLA-specific intermediate products. Figure 2.4 shows the traditional compilation model for C programs; Figure 2.5 shows the additional steps required by TESLA.

A `.bc` file contains LLVM intermediate code, and a `.tesla` file contains a binary or textual representation of the TESLA assertion manifest.

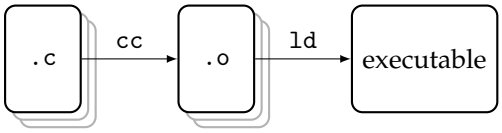

Figure 2.4: The traditional C compilation model

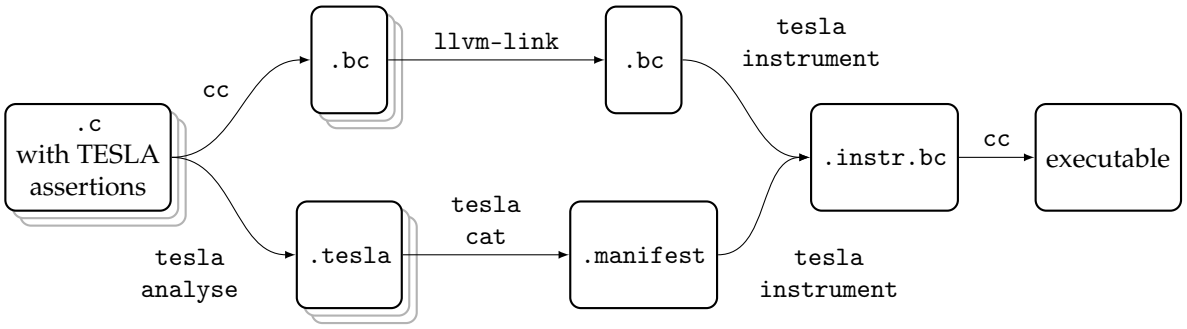

Figure 2.5: The C compilation model with TESLA

The TESLA toolchain is used together with the Clang / LLVM compiler infrastructure to generate these intermediate artifacts. A brief summary of the individual TESLA tools used is:

`analyze` parses TESLA assertions from a C source file and outputs them to a `.tesla` manifest file.

`instrument` adds instrumentation code to a program in LLVM IR format based on the data in a TESLA manifest file.

`cat` combines several TESLA assertion manifests together, checking for consistency and eliminating redundant definitions.

These tools can be easily integrated with an automatic build system. An issue specific to TESLA is that assertions written in one compilation unit can affect instrumentation code in all other compilation units. This means that changing one source file can cause the entire project to be reinstrumented, increasing build times; incremental builds are worst affected by this issue.

2.3.3 Writing Assertions

TESLA assertions are written using a set of preprocessor macros that expand to calls to stand-in functions. These functions have no definition, and are only used as a way to store information in the IR. Calls to them are removed by the instrumenter.

An example of the TESLA macros being written inline with a program is given in Figure 2.6.

```
int main(void)
{
  TESLA_WITHIN(main,
    eventually(
      call(some_function(ANY(ptr))),
      other_function(ANY(int)) == 0
    )
  );

  int x;
  some_function(&x);

  return other_function(x);
}
```

Figure 2.6: Example of TESLA macros being used to write an assertion

```
automaton(name_of_auto, struct arg_type *s) {
  some_function(s) == 0;
  call(function1) || call(function2);
  s->field = 4;
  tesla_done;
}
```

Figure 2.7: Example of an explicit TESLA automaton

2.3.4 The TESLA Assertion Language

Assertions specify temporal relations between *program events* as defined in subsection 2.3.1 (function calls, assertion sites etc.).

The basic relationship expressible in the assertion language is sequencing—an assertion that events occur in a particular order. Disjunction of assertions is also expressible. Assertions must include a reference to an assertion site event—this reference is implicit in Figure 2.6 through the `eventually` macro, which states that the sequence of events named takes place after the assertion site.

Automata can be written *explicitly*, allowing for composition and code reuse. Figure 2.7 shows an example of this style.

2.4 LLVM

In this section I give a brief overview of LLVM [27], the compiler framework used to implement the TESLA toolchain.

```
declare void @bar(i32)

define i32 @foo(i32 %x) {
entry:
  %mul = mul i32 %x, 3
  %cmp = icmp sgt i32 %mul, 100
  br i1 %cmp, label %if.then, label %if.else

if.then:
  call void @bar(i32 %mul)
  ret i32 0

if.else:
  ret i32 %mul
}
```

Figure 2.8: Example of textual LLVM IR code

The primary attraction of implementing a compiler-based tool using LLVM is its *intermediate representation*, which can be easily manipulated in-memory using a C++ API or manipulated as human-readable text in an assembly-like format. LLVM IR is "a Static Single Assignment (SSA) based representation that provides type safety, low-level operations [and] flexibility" [29].

Figure 2.8 gives an example of textual IR that illustrates some of the important features of LLVM:

Functions LLVM IR can represent functions that are conceptually similar to C functions, though with basic blocks and jumps rather than loops and conditional statements. Similarly to C, functions can be declared but not defined.

Values A key design decision taken by LLVM is that instructions produce named SSA values. When manipulating LLVM IR programatically, an instruction is synonymous with the value it produces. In textual IR, values are identified by `%` or `@` preceding their name (for local and global values respectively).

Types LLVM IR retains type information from its source program—all values are typed, and casts between types must be made explicitly using intrinsic instructions.

The TESLA instrumenter is implemented using the in-memory LLVM IR manipulation libraries. It replaces known instrumentation sites in the IR with instrumentation code, yielding code that can be compiled to an executable.

3 Related Work

In this chapter I provide an overview of how TMC contributes new ideas and developments compared to previous work, with particular emphasis on the areas of bounded model checking and SMT methods.

3.1 Program Verification

In this section I give a summary of important work in the area of program analysis and verification that has influenced my development of TMC.

3.1.1 Bounded Model Checking

Biere et al. [7] introduce the concept of bounded model checking, building on the earlier idea of *symbolic* model checking due to McMillan [31]. In this section I give a summary of previous work in the area, with particular emphasis on where it has been applied to check C programs. I also note key differences between these previous works and TMC—the style of assertion supported, and the use of non-symbolic checking.

Key to model checking is the idea of counterexample generation—if a state satisfies the negation of a formula, then that state is a counterexample for the original formula. Bounded model checking extends this idea by searching for counterexamples with an upper bound on their allowable size. This means that counterexamples are discovered faster and are minimally sized.

While the implementation strategies and analysis goals of BMC [7] (the first practical bounded model checker) and TMC are different (BMC is a symbolic model checker for an explicit state description language; TMC is non-symbolic and checks TESLA assertions in C programs), the strategy of searching for counterexamples using an iterative-deepening method is derived from the original work on bounded model checking. Another difference between TMC and BMC is their approaches to soundness—BMC aimed to be provably sound, while TMC aims for *soundiness* [28] (see subsection 6.1.4).

CBMC

Clarke, Kroening, and Yorav's CBMC [13] applied bounded model checking to C programs by translating them into instances of the SAT decision problem. CBMC allowed

for C programs to be written as executable specifications for Verilog hardware designs without prohibiting the use of any C language constructs.

Although CBMC is an example of model checking being applied to C programs, the assertions that can be checked are not temporal. The goal of TESLA assertions is different to that of CBMC—TESLA assertions are written to verify the behaviour of a program, while CBMC uses a C program with assertions to verify that a hardware implementation is behaviourally equivalent to that program.

LLBMC

CBMC operates at the level of C source code using syntactic transformations. Merz, Falke, and Sinz [32] identify this as a potential avenue for improvement—they contribute LLBMC, a bounded model checker that operates on LLVM [27] IR. The benefits of this approach include broader language support, assistance from compiler optimisations for code simplification, and an improved memory model. Empirically, LLBMC represents an improvement over previous work it is compared to. The use of LLVM IR to simplify model checking is applied in TESLA and TMC.

Context-bounded LTL Checking

Recent work by Morse et al. [33, 34] uses bounded model checking to check LTL assertions against C programs. Their approach translates an LTL formula into a Büchi automaton, which is then itself translated to C code and woven into the program to be checked.

This approach is similar to TESLA instrumentation, with some key differences. The automaton code here has its own thread of execution rather than being inserted inline, and instead of producing a modified executable that exhibits runtime failures on assertion violations, the combined program is checked statically using ESMBC [16].

Although the development of this work was partly contemporaneous with initial work on TESLA, and the implementation strategies similar in some respects, the expressivity of this system is more similar to CBMC than to TESLA.

3.1.2 Other Approaches

While there has been a great deal of work derived from BMC on verifying systems software, there are also other approaches that do not share the same lineage.

Bessey et al. [6] describe the lessons learned when commercialising their static analysis research—many of these lessons are applicable to static analysis tools in general, particularly with regard to the *perceived usability* of a tool and how it is used in a non-research environment.

MOPS

Anderson et al. [2] identify MOPS [10] as being similar to TESLA in concept. The primary goal of MOPS is to discover potential vulnerabilities of C programs operating

in a Unix environment (where security properties may have whole papers dedicated to explaining their subtleties, as is the case with `setuid` [11]). The authors used MOPS to discover a number of vulnerabilities in well-known open-source software.

Properties in MOPS are expressed as finite state automata, with accepting states representing an execution on which an unsafe event has occurred. The expression of TESLA assertions is similar in concept to this, but with extensions to check function arguments and return values. MOPS assertions express slightly different properties to TESLA assertions (and vice versa). MOPS can assert properties such as "a call to `f` is immediately followed by a call to `g`", while TESLA can assert that "if execution reaches this program point, `f` was previously called". Broadly, however, the concepts are similar.

The primary advantage of TMC compared to MOPS is the inclusion of assertion site events for considering control flow only on certain paths, and the ability to check a subset of the program's data flow.

KLEE

KLEE [9] is a system for symbolic execution of programs in order to automatically generate tests or prove assertions. It differs from other approaches described here as it does not perform model checking—instead, it generates constraints that must hold for a program point to be reachable, then solves the constraints to generate test inputs to the program.

The primary motivation for this implementation style is to increase the source-level coverage of a program's test suite. This means exploring every possible execution path, an approach that TESLA sought explicitly to avoid (by using automata bounds and assertion site events). However, some of the ideas present in KLEE are relevant to TMC—like the return value inference algorithm I describe in subsection 4.3.1, KLEE uses an SMT translation of LLVM IR to solve constraint systems.

3.2 SMT

In this section I give a brief overview of the theory of SMT methods, as well as a summary of important work related to program analysis using SMT methods.

3.2.1 Background

The study of satisfiability modulo theories (SMT) is founded in boolean satisfiability. The decision problem SAT was the first to be proven to be NP-complete [15]—its statement can be given concisely as "Does there exist a consistent assignment of truth values to the variables in a boolean formula such that the formula is satisfied?". SAT exhibits the useful property of self-reducibility, meaning that any algorithm that solves the decision problem can be used to find a satisfying assigment.

Many problems can be easily reduced to SAT (formally, any problem in NP can be reduced to SAT in polynomial time, and informally, its structure makes it a good choice for encoding some domain-specific problems). However, many other problems are stated with respect to a background theory such as integer arithmetic or finite arrays. The key idea of SMT problems is to allow for a *background theory* to be combined with a satisfiability problem.

Biere et al. [8, ch. 12] provide a formal definition of SMT problems, as well as several commonly-used background theories. For the purposes of this report, only a basic definition is required.

Terminology

A *background theory* is a collection of axioms that allow for interpretation of the symbols in a formula. For example, the background theory of integer arithmetic provides the standard interpretations of symbols such as $+, -, \times, 0$ etc. A different background theory such as that of finite bit-vectors may interpret these symbols differently (for example, $+$ could be defined to wrap on overflow). Background theories are used because it is often either tedious or impossible to encode these axioms in propositional logic for a SAT solver.

Uninterpreted functions are the building blocks of SMT instances. No meaning is associated with these functions when an SMT problem is specified, only that they have a particular *sort*[1]. An SMT solver may assign interpretations (definitions) to these functions in order to satisfy the instance constraints.

Tools and Standards

A great deal of research and engineering work is invested in the use of SMT tools. Two of the most commonly used solver implementations are Z3 [18] and CVC4 [5], but there are numerous others with individual strengths and weaknesses. There exist standards such as SMT-LIB [3] that specify textual input and output formats for SMT solvers. Standardisation in this way allows for competitive benchmarking of solver performance (the primary venue for this is SMT-COMP [14]).

3.2.2 Related Work

SMT solvers are a low-level tool—using them to solve a domain-specific problem involves translating the problem into a formulae in a particular theory. Because of this flexibility, SMT solvers have been used to solve a large number of different problems. For example, Microsoft list 58 publications related to Z3 [36] that span areas as diverse as cloud computing, real-time systems, and functional programming. Broadly, most applications of SMT methods involve some form of *program analysis*.

[1] Informally, sorts can be understood as "types" in a particular problem. Examples are the sorts of integers, booleans, and functions with particular domain and range.

Dahlweid et al. [17] provide VCC, a tool for proving partial correctness of C programs using annotations that describe invariants on data structures. These annotations are converted to an intermediate representation, then to an SMT problem to be verified. The annotations supported by VCC are somewhat different to TESLA assertions—they specify invariant rather than temporal properties. VCC was used successfully to verify the implementation of the Microsoft Hyper-V kernel.

PAGAI [21] uses SMT solvers to implement analyses based on abstract interpretation. For example, it can be used to discover invariants that hold at points in the control flow graphs, and to prove properties based on assertion reachability. The methods used in PAGAI to map LLVM IR onto an SMT problem are more sophisticated that those I describe in subsection 4.3.1—for example, PAGAI implements an arithmetic simplification method based on parallel assignments that allows for stronger invariants to be proved.

4 Static Analysis

In this chapter I present the design and implementation of static analysis mechanisms for TESLA. First, I motivate this work by implementing a mutual exclusion lock instrumented with TESLA and demonstrating that performance improvements are possible by removing TESLA instrumentation code. I then fully describe TESLA assertions as finite-state automata. From this description, I implement TMC, a model checker for TESLA assertions.

4.1 Modelling Locks with TESLA

> TESLA exposes run-time behaviour using program instrumentation, illuminating coverage of complex state machines and detecting violations of specifications. [2, p. 1]

Anderson et al. [2] draw attention to the suitability of TESLA for modelling and verifying *state machines* within a program. A simple state machine used in many programs is the mutual exclusion lock—in this section, I develop TESLA assertions for the usage of these locks and show the possible benefit of static analysis with respect to runtime performance. Figure 4.1 shows the state machine for a spin-lock implemented using a mutex with non-blocking acquire and release operations.

While the number of states and operations associated with this state machine is small, asserting correct usage involves temporal properties over both control- and data-flow. It will therefore be a useful running example throughout the rest of this chapter.

4.1.1 Lock Implementation

A possible implementation of a mutual exclusion lock using the C11 atomics library is given in Figure 4.2. The only operations permitted by the lock are non-blocking acquisition[1] and release. Using an atomic member variable with a compare-and-swap function ensures thread-safety.

Figure 4.3 gives a possible implementation of a spin-lock as shown in Figure 4.1. Correct usage of such a lock can be summarised informally by a set of invariants:

[1] Returns immediately with `true` if the lock was acquired, and `false` if it was not.

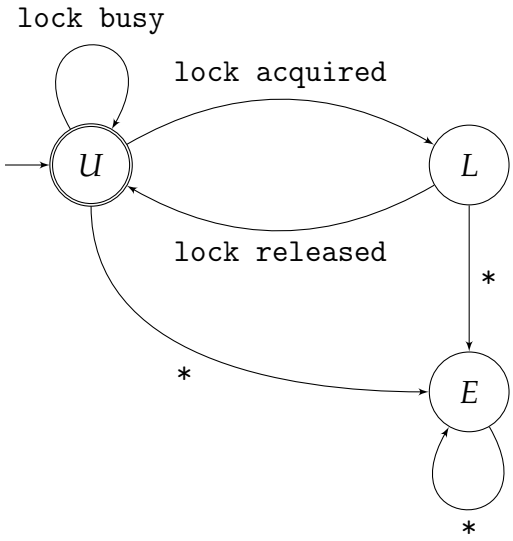

Figure 4.1: State diagram for a mutual exclusion spin-lock, with states *unlocked* (*U*), *locked* (*L*) and *error* (*E*).

```
struct lock_t {
  _Atomic(bool) locked
};

void lock_init(struct lock_t *lock) {
  atomic_init(&(lock->locked), false);
}

bool lock_acquire(struct lock_t *lock) {
  bool f = false;
  return atomic_compare_exchange_strong(
          &(lock->locked), &f, true);
}

void lock_release(struct lock_t *lock) {
  lock->locked = false;
}
```

Figure 4.2: Implementation of a mutual exclusion lock with C11 atomics

```
void lock_spin(struct lock_t *lock) {
  while(!lock_acquire(lock)) {}
}
```

Figure 4.3: Implementation of a spin-lock using non-blocking acquire

```
automaton(acq_rel, lock_t *lock) {
  acquire(lock);
  release(lock);
  tesla_done;
}

automaton(acquire, lock_t *lock) {
  ATLEAST(0, lock_acquire(lock) == false);
  lock_acquire(lock) == true;
  tesla_done;
}

automaton(release, lock_t *lock) {
  returnfrom(lock_release(lock));
  tesla_done;
}
```

Figure 4.4: Mutex lock properties expressed using TESLA

- Consumers can fail to acquire the lock any number of times
- Once the lock is acquired, no more attempts to acquire can be made
- The lock is released exactly once after being successfully acquired
- The lock is not released before it is acquired

4.1.2 TESLA Assertions

The properties described previously are well-suited to being expressed as TESLA assertions—they express temporal relationships between program events (calls to the functions `lock_acquire` and `lock_release`). Figure 4.4 shows a TESLA expression of the spin-lock usage properties using explicit TESLA automata.

I implemented a test suite of programs instrumented using these assertions. The test suite contained both correct and incorrect uses of the lock (with respect to the assertions in Figure 4.4), and was used to check that the invariants described previously were in fact properly checked by these assertions.

4.1.3 Performance Overhead

The lock assertions can be used to experimentally demonstrate the performance overhead of using TESLA instrumentation, motivating the removal of safe TESLA assertion code using static analysis.

Experimental Setup

The benchmark code used in this experiment created a number of threads, each of which attempts to sort a randomly chosen interval of a large shared array in a loop. Threads accessed the array under mutual exclusion, protected by a lock as described in Figure 4.2—this created contention on the lock, dependent on the number of executing threads.

A single TESLA assertion was added to the benchmark to assert the correct usage of the lock. Two versions of the program were compiled—one with the TESLA instrumentation added, and the other without. Both versions were compiled using release build settings.

Both programs were run with the same parameters (threads sort an interval of size 15,000 from a larger array of size 500,000, and the number of threads was varied from 8 to 40), with results averaged over 5 runs of the program. The benchmarks were run on a dedicated server (Intel Xeon E5-1620 3.6 GHz, 8 cores, 64GB of RAM) running FreeBSD 11.

Results

The uninstrumented binary is 25% smaller than the instrumented binary (19.1 KiB vs. 25.3 KiB).

The results from running the two benchmark programs as described above are shown in Figure 4.5. At low levels of contention there is little difference between the programs—this is because the TESLA instrumentation code is only executed during acquisitions and releases of the lock. However, at higher levels of contention more time is spent in the TESLA instrumentation code (because each call to `lock_acquire` is more likely to fail, more calls are made), and the instrumented version becomes slower relative to the uninstrumented version.

By manually removing assertions that lie on a frequently executed code path, a run time performance improvement is observed. A decrease in binary size is also observed. This result motivates the use of static analysis on TESLA-instrumented programs—if instrumentation code can be removed automatically, then the same performance improvements should be attainable.

4.2 Formalising TESLA Assertions

To statically analyse the correctness of TESLA assertions, their semantics must be defined. In particular, we are interested in the "bad things" that cause an assertion to fail at run time, and how these can be detected at compile time.

TESLA assertions "have a natural expression as finite-state automata that can be mechanically woven into a program" [2, p. 3]. However, the exact manner in which these automata are constructed is not given in full in the paper. In this section, I define the semantics of TESLA assertions by providing the full translation into finite-state automata. These automata consume strings of *program events*, accepting a sequence

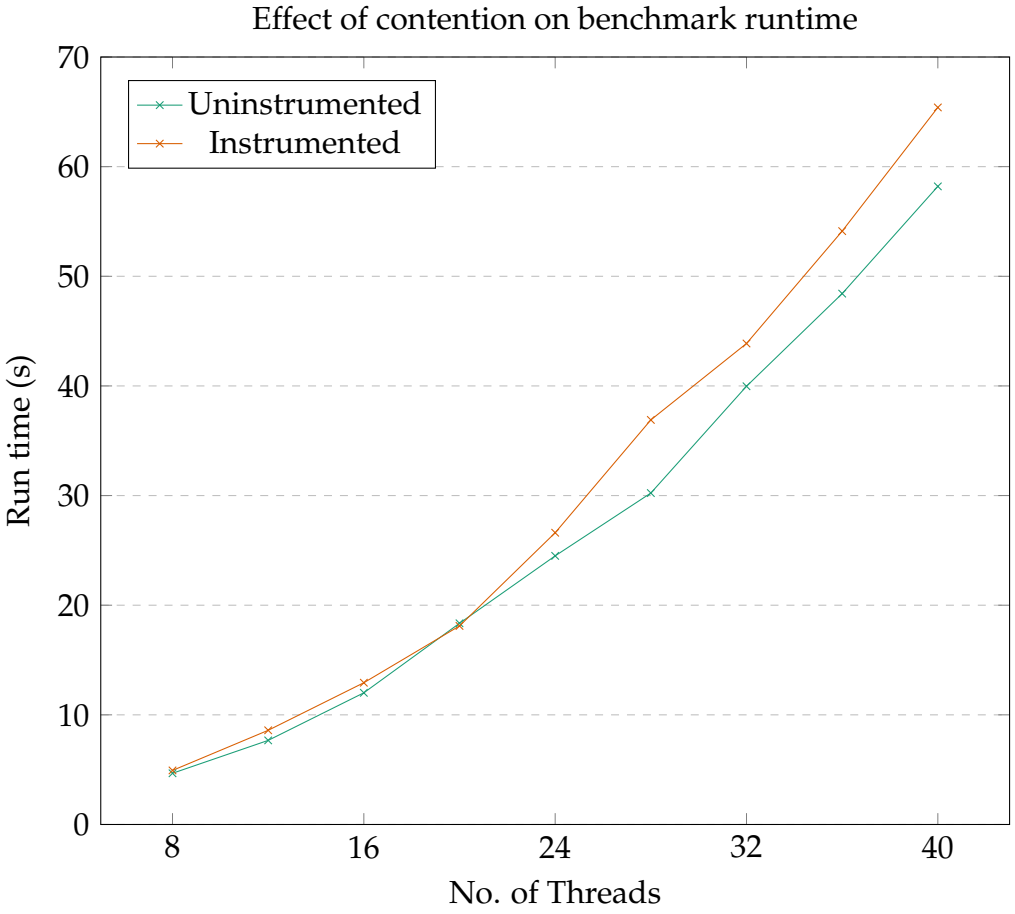

Figure 4.5: Runtime of instrumented and uninstrumented benchmarks at varying levels of lock contention

if it is valid with respect to an assertion, and rejecting it otherwise. A definition of program events is given below.

The automata constructions given in this section are nondeterministic with ε-transitions. It is worth noting the well-known result that a non-deterministic automaton with n states can always be converted to an equivalent deterministic automaton with up to 2^n states [37]. However, in practice the automata constructed using the methods described in this section do not experience an exponential increase in size.

A TESLA-specific issue is that there may be many instances of the same automaton "in-flight" at the same time (for example, if multiple locks are allocated on the heap and assertions are made of them)—the constructions I describe represent a single automaton instance.

4.2.1 Program Events

Single program events have no recursive structure (they define only an event category and associated metadata). As a result, the automata they define are very simple. Fig-

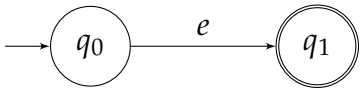

Figure 4.6: Program event automaton

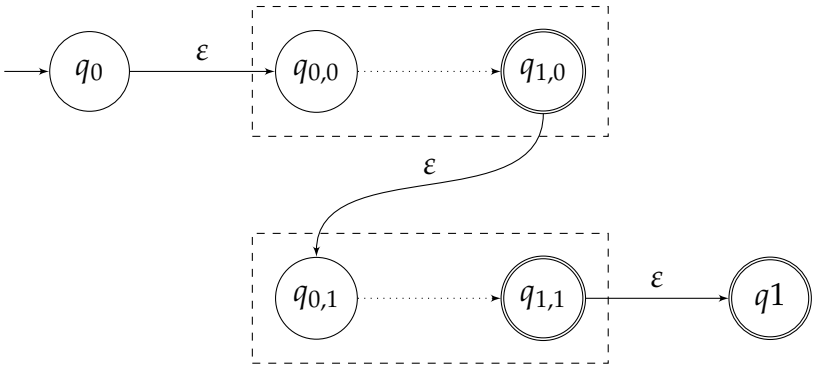

Figure 4.7: Single repetition sequence automaton

ure 4.6 shows the constructed automaton for an arbitrary program event e—it has a single transition from the initial state to the accepting state, labelled by the event e.

The structure of this automaton is the same no matter what event e it was constructed for, and it captures all the metadata associated with e. In section 4.3 I give a full description of how these properties are used to check properties of a program.

4.2.2 Composition

There are two ways in which TESLA automata may be composed—sequential ordering and disjunction. These assertions have recursive structure (i.e. they contain other assertions), and so their constructed automata are defined as compositions of other automata.

By convention, sub-automata are shown inside dashed boxes. Accepting states inside these boxes are the accepting states of the sub-automaton, and dotted lines indicate transitions that are internal to the sub-automaton.

Sequential Ordering

The primary temporal relationship TESLA can express is sequential ordering. Sequence assertions specify an arbitrary list of events that must happen in order, along with an upper and lower bound on the number of times the sequence may be repeated (the upper bound may be infinite).

A sequence that occurs exactly once simply links each sub-automaton's accepting state to the next's initial state with an ε-transition. Figure 4.7 shows this construction for two sub-automata.

From a single repetition, an automaton that can recognise an infinite number is

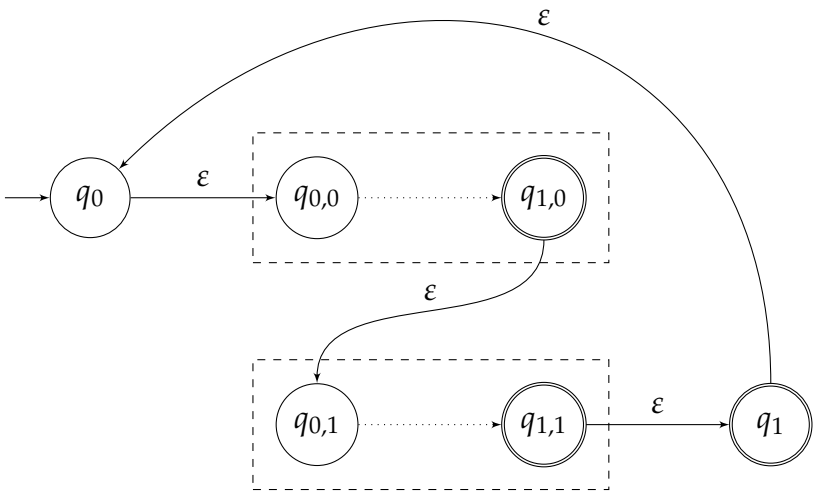

Figure 4.8: Infinite repetition sequence automaton

obtained by adding an ε-transition back from the accepting state to the initial state (as shown in Figure 4.8).

If the upper bound is finite, copies of the entire sequence are chained together to form the overall automaton. The copies in the accepting range[2] have an ε-transition to the final accepting state. Writing automata with a large but finite number of repetitions has a direct effect on program size as a consequence of this construction.

Sequential orderings of events must include a reference to an assertion site (the source location where an assertion is made). The logical property expressed by a sequential ordering is then "on an execution path that includes this assertion site, all the events named in the sequence occur exactly in order". There is no restriction on where in the program's execution the events occur.

Disjunction

TESLA can also express *inclusive* and *exclusive* disjunction of sub-automata. Anderson et al. [2] specify a cross-product based construction for the inclusive case, where `a || b` means that either or both of `a` and `b` can occur. The exclusive case (`a xor b`) where only one of `a` or `b` may occur is simpler to construct, and an example is shown in Figure 4.9.

Inclusive-or expresses the property that at least one of the events named occurs, but that it is not an error for the other events to occur as well. Exclusive-or expresses the property that exactly one of the named events can occur, and it would be an error for any of the others to occur as well.

[2] i.e. those where the number of repetitions is greater than or equal to the lower bound and less than or equal to the upper bound.

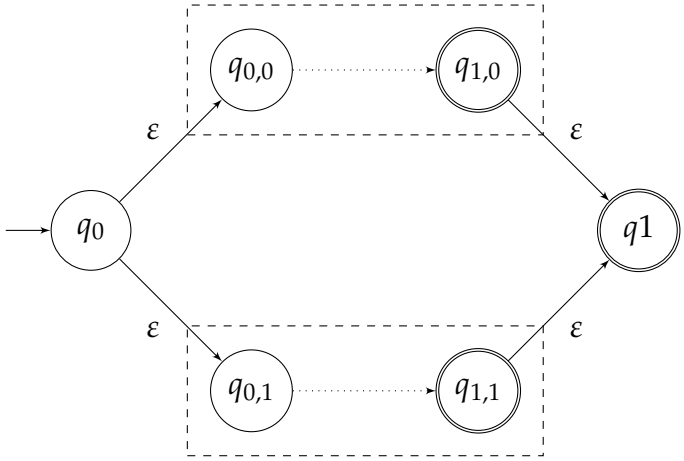

Figure 4.9: Inclusive-or automaton

4.3 TMC: a TESLA Model Checker

In section 4.2 I gave a translation from TESLA assertions to nondeterministic finite automata with transitions labeled with program event assertions. These automata act as *specifications* for a program. Checking whether a program is correct with regard to its specifications is a model checking problem in the style of Clarke and Emerson: "[to] mechanically determine if the system meets a specification expressed in propositional temporal logic." [12, p. 2]

In this section I describe the methods used to check a program against its specifications as derived from TESLA assertions. I first describe an SMT-based algorithm for computing data-flow information for an individual program execution. Using this analysis together with a form of bounded model checking, I implement a model checker (TMC) for TESLA assertions. Finally, I discuss the limitations and potential improvements that could be made to this approach.

4.3.1 Data-flow Inference

TESLA allows for properties of data flow to be asserted—in particular, the value of arguments passed to functions and the value returned by a function call. Checking arguments passed to functions can be performed with only minor modifications to existing TESLA code, and so in this section I describe only the algorithm used to check function return values.

Stated formally, the property we are interested in checking is "on a particular execution path, can we infer that a function call returned a fixed value?". In this section, I describe an algorithm for performing this inference by translating LLVM functions to SMT problems.

Encoding LLVM IR as SMT Formulae

In section 3.2 I gave an overview of SMT methods, their history and relevant terminology. Building on this background, in this section I describe how an LLVM function can be translated to an SMT problem that can be used to solve the return value inference problem described above.

A subset of LLVM IR can be easily translated to an SMT formula.[3] For example, the LLVM instruction `%3 = add i32 %1, %2` can be seen as a formulae stating that $\%_3 = \%_1 + \%_2$, whatever interpretation is given to the values $\%_1$ and $\%_2$. This encoding relies on a particular background theory (for example, the theory of finite bit vectors) to supply the semantics of $+$ and other operations. All of the LLVM binary and comparison operators can be translated to interpreted SMT functions using this method.[4]

Function calls and loads from memory are translated to uninterpreted functions—we have no *a priori* knowledge of the value they will take at run time, but they can be assigned values in a satisfying model for the problem.

A single execution path through a function is a finite sequence of basic blocks starting at the entry block. The blocks executed in the sequence constrain the values of *branch conditions* (the LLVM values on which conditional branches are predicated). These constraints are then added to the SMT problem as assertions.

Applying this translation to a single execution path yields an SMT problem, the solution to which produces an assignment of values to LLVM variables. It is possible that some variables are left unconstrained (if no conditional branches depend on their value), or that multiple values are satisfying assignments. In order to check whether a valuation is unique for a given variable, we construct an augmented problem that contradicts the original model. If this problem is unsatisfiable, the original solution was unique.

Finally, after translating a function execution and finding a unique model, we are left with a mapping from call sites to fixed return values (on this execution path). If a call site has a value in the mapping for an execution path, we know that that value *must* be observed on the execution. Otherwise, we assume no knowledge at all of the value returned at a call site.[5]

Implementation

The algorithm described above is an intraprocedural one; all the LLVM values in the SMT translation need to exist in the same function. Implementing it therefore requires several transformations to be made to a function before it can be translated to an SMT formula. First, the function must be completely inlined. This is achieved using existing

[3] At least for the properties relevant to this problem—a full translation of the LLVM semantics would be more complex.

[4] Except floating point operations, which are unsupported by TESLA in general.

[5] This lets us divide call sites cleanly between those we know about, and those we know we don't know about.

LLVM library code (with the caveat that recursion can only be inlined up to a finite depth).

When a function is inlined, all calls to it are removed—this means we lose information about arguments passed and values returned. This problem is resolved by adding calls to *stub functions* before and after each call site to be inlined. Each stub function receives the same arguments as the call being made, and the return value of the call is replaced by the return value of the "after" stub. This means that relevant call graph information is preserved in the inlined function. The inlined function can become very large for wide assertion bounds (for example, `main` or a system call).

Execution sequences are generated from the inlined function by walking its control flow graph to a specified finite depth—this finiteness condition is addressed further in subsection 4.3.2. Repeated basic blocks from loops in the graph are handled by simply duplicating the blocks. Figure 4.10 shows a single trace taken from one of the programs in the mutex test suite from subsection 4.1.2 after these transformations have been applied.[6]

Figure 4.10 illustrates the steps performed here—the LLVM basic blocks `%lock_acquire.exit` and `%do_work.exit` are inlined from different functions, and function calls have been replaced with calls to entry and return stub functions.

The Z3 SMT solver [18] is used to construct and solve the SMT problem corresponding to an execution sequence. Z3 was chosen as a production-quality SMT implementation with a well-documented C++ API, but the techniques used apply equally well to any SMT solver implementation. Figure 4.11 shows the SMT translation of the LLVM IR in Figure 4.10, given in SMT-LIB standard syntax [4].

4.3.2 Model Checking Algorithm

Given the SMT-based mechanism for determining function return values described above, a model checking algorithm for TESLA assertions can now be defined. This algorithm is defined in terms of execution traces over an LLVM function (as defined previously), along with the finite-state automaton translation of an assertion given in section 4.2.

Execution Traces

TESLA assertions are bounded by beginning and end events. In TMC, for ease of implementation we consider only assertions bounded by entry and exit to the same function. Other specifications of a bounding interval are possible, but in practice most TESLA assertions tend to be bounded by a function in this way.[7] Possible executions are then sequences of basic blocks that begin at the function's entry block.

TMC generates all possible executions through the inlined bounding function up to a given length maximum length. On each of these generated executions, we compute

[6] In this example, `__tesla_sink` is a basic block added to preserve control flow information.

[7] The FreeBSD assertions written by Anderson et al. [2] are all bounded by a system call and corresponding return, for example.

```
define i32 @trace_main_5() {
  call void (...)* @__tesla_inline_assertion(...)
  call void @__entry_stub_do_work()
  %2 = call i1 @__entry_stub_lock_acquire(...)
  ; locking code omitted
  br i1 %5, label %__tesla_sink, label %lock_acquire.exit

lock_acquire.exit: ; preds = %6
  %7 = call i1 @__return_stub_lock_acquire(...)
  %8 = xor i1 %7, true
  br i1 %8, label %__tesla_sink, label %do_work.exit

do_work.exit: ; preds = %lock_acquire.exit
  call void @__entry_stub_lock_release(...)
  call void @__return_stub_lock_release(...)
  call void @__return_stub_do_work()
  ret i32 0

__tesla_sink: ; preds = %lock_acquire.exit, %1
  unreachable
}
```

Figure 4.10: LLVM IR to be translated into an SMT problem

```
(declare-fun |%5| () Bool)
(declare-fun |%7| () Bool)
(define-fun |%8| () Bool (xor |%7| true))
(assert (not |%5|))
(assert (not |%8|))
```

Figure 4.11: SMT translation of LLVM IR

function return values using Z3 as described previously. Then, the execution is checked for acceptance against the finite-state automaton obtained from the original assertion.

If all possible executions are accepted by the automaton, then the assertion cannot fail at runtime. Its instrumentation code can therefore be removed safely from the program. If an execution is not accepted by the automaton, then a *counterexample* has been found—we can demonstrate an execution path on which the assertion may not hold, and the reasons why it may not hold. Figure 4.12 shows a counterexample generated from one of the mutex test programs (the full DOT output for the finite-state automaton has been omitted for brevity). All executions are either accepted or not accepted—there is no "don't know" outcome.

```
Unexpected event in state s3:
  return from lock_acquire (return value == 1)

FSM:
  digraph {
    ...
  }

Call stack:
  call: lock_init
  return: lock_init
  call: do_work
  call: lock_acquire
  return: lock_acquire
  call: lock_acquire
```

Figure 4.12: Counterexample trace generated from a mutex test program

Checking Execution Traces

In this section I describe how an execution trace can be checked against a specification automaton. I first describe how individual state transitions are checked, and from there define a notion of acceptance for an entire execution trace.

Each execution trace is a finite sequence of LLVM instructions. For each instruction in the sequence, the following checks are made:

- Is the instruction a function call? (all checkable program events are expressed as function calls in the IR—either to entry or return stubs, or to the internal TESLA assertion site function).
- If it is, is it accepted by any of the edges from the current state?
- If none of the edges accept the function call, does there exist an edge elsewhere in the automaton that would accept it?

Whether or not an individual edge accepts a program event is defined in terms of the event's type and metadata—if the event's metadata matches the edge's, then the event is accepted. For edges that specify a function return value, the check also verifies that the mapping generated from the SMT problem includes a matching entry.

Assertion failures occur when an event is not accepted by the current state, but could be by another state in the automaton. Events that are not accepted by any state are not relevant to the automaton, and so can be ignored.

It is now possible to define when an execution trace is accepted by the specification automaton. As noted previously, traces do not have to be complete—they may

not reach a terminating block for the function.[8] Separate definitions for complete and incomplete traces must therefore be given.

A complete trace ends in a basic block with no successors, and is accepted if the state reached after all instructions have been checked is an accepting state. Incomplete traces are accepted if an accepting state is reachable from the final state reached. In both cases if an unexpected event occurs, the execution is not accepted. The difference in semantics between complete and incomplete traces is discussed in some depth by Eisner et al. [20]—in particular, their description of a "weak view" corresponds to the checking criteria for incomplete traces described previously.[9]

4.3.3 Results

TMC was able to correctly check all the examples from the mutex test suite described in subsection 4.1.2. Correctness here means that a program checked by TMC should behave identically to one with the default TESLA instrumentation—all the test programs produced identical behaviour when run after having their instrumentation removed by TMC.

Field Assignments

As noted previously, TMC cannot check any assertion that contains a structure field assignment. This limitation arises because field assignments are inherently *value-dependent*—locating the IR instruction that assigns to a structure field is easily done, but computing the value assigned is not easy (in the general case).

A TESLA-specific solution to this issue would be to devise a way of splitting assertions such that statically provable components are proved where possible, leaving behind components of the assertion that must be instrumented dynamically. This approach would require major changes to some TESLA internals, and was found not to be feasible within the scope of this project. The SMT methods for function return value inference may also be applicable (with some adaptation).

[8] Incomplete traces are required to deal with code that may enter an infinite loop or sequence of recursive calls.

[9] The weak view asserts that "nothing has yet gone wrong" [20, p. 29], while the strong view asserts that an assertion is "already" satisifed on a truncated execution path.

5 Applications

In this chapter I discuss potential applications of statically checked TESLA to practical software engineering scenarios. I provide an analysis of how coding style can make writing TESLA assertions for a system more difficult, with reference to a large open-source library. Then, motivated by this difficulty, I describe a general method for applying TESLA to library interfaces with minimal modification to client code. Finally, I demonstrate a practical application of this technique by adding TESLA instrumentation to the interface of a LWIP, a widely-used network protocol library.

5.1 LWIP

An initial goal of the project was to investigate how TESLA might be applied to verify the behaviour of a larger state machine such as that of TCP. However, this verification proved to be more difficult than anticipated due to a number of C programming idioms and design choices present in LWIP.

In this section I investigate the application of TESLA to LWIP [19], a widely used, portable implementation of the IP protocol stack. I describe difficulties encountered in this process with reference to the LWIP source, as well as an analysis of how code written from scratch with TESLA instrumentation in mind could mitigate these issues.

5.1.1 Structure

LWIP is distributed as a configurable library so that it can be built on virtually any platform with a C compiler—interfaces to network buffers, timers and other platform-specific code are abstracted so that their implementation can be supplied by users. Configurations for widely used operating systems (generic Unix, Windows etc.) are distributed as a secondary library together with example applications.

The core networking code of LWIP is around 57K lines of C.[1] This code includes implementations of IPV4, IPV6, TCP, UDP and several application-layer protocols. The secondary library has around 13K lines of C, mostly contained in implementations of executable server applications (HTTP, Telnet, SMTP etc.).

[1] Not including header files or tests.

```
err_t tcp_bind(struct tcp_pcb *pcb, const ip_addr_t *ipaddr, u16_t port);

err_t tcp_close(struct tcp_pcb *pcb);

struct tcp_pcb * tcp_listen_with_backlog(struct tcp_pcb *pcb, u8_t
 ↪  backlog)
```

Figure 5.1: Function declarations from the LWIP TCP implementation.

5.1.2 Investigation

The goal of my investigation into LWIP was to instrument the core TCP implementation with useful TESLA assertions, then to demonstrate that performance improvements were attainable by applying static analysis to this instrumentation.

Before any investigation could be performed, a version of LWIP built using TESLA was required. Each of the LWIP-based server applications is built using a Makefile that compiles the core library separately, then links the application-specific code with the core library. Modifying this build system to use the TESLA infrastructure was not difficult, as much of the setup (flags, includes, linking etc.) was in place already—the only changes needed were to add the extra TESLA-specific rules and to compile to bitcode instead of object files. The changes made totalled 49 lines of Makefile code.

The end result of this modification was that any of the example applications distributed with LWIP could be instrumented and built using TESLA. Based on this modified build of LWIP, I investigated how TESLA could be applied to internal library code. The remainder of this section describes the features found in the source code that made applying TESLA (statically analysed or otherwise) more difficult.

TCP State Implementation

At the core of the TCP protocol implementation is a structure representing a single TCP connection (`struct tcp_pcb`). Almost all of the TCP protocol implementation is expressed in terms of these structures—Figure 5.1 contains some function declarations taken from the source code that use the structure.

Such heavy reliance on structure fields is less than ideal because TMC cannot check assertions that reference structure field assignments, though checking assignments to structure fields is within the capabilities of runtime TESLA instrumentation. A further complication is that the implementation is not consistent in its use of PCB structures—some functions modify a structure instance passed to them (`tc_pbind`, `tcp_close`), while others return an entirely new instance (`tcp_listen_with_backlog`). The latter style of function is more difficult to instrument effectively in TESLA because the object being asserted about may change midway through an assertion, which is difficult to express using the TESLA assertion language.

Macro Usage

In order for LWIP to be universally portable, it makes heavy use of the C preprocessor for a number of reasons. For example:

Platform-specific implementations The implementation of some functions can vary from system to system (e.g. endianness conversion functions). Macros are used to select the correct implementation of these functions without the overhead of a function call. This means that any TESLA assertions added to these functions would become platform-specific, and duplicated between implementations.

Conditional Compilation Almost every feature of LWIP can be enabled, disabled or modified at compile-time by setting the correct preprocessor definitions (this is what allows LWIP to be used so effectively on systems with limited resources). This feature is used in places to conditionally change the fields contained in a structure—any assertions written about that field must then be aware of the required `#ifdef` context.

Inlined Functions Some simple "functions" in LWIP are expressed using macros to guarantee that there is no function call overhead, rather than relying on the compiler to inline them. These function-like macros cannot be asserted about by TESLA[2], and are difficult to distinguish in source code.

Together, these uses of the macro system make TESLA instrumentation more difficult to add to the LWIP source.

Control Flow

TESLA assertions are most useful (especially when using static analysis) for asserting properties related to control flow events. However, the style in which LWIP code is written means that there is little explicit control flow within the protocol implementation itself—many functions perform complicated work on a PCB structure, then call only a single other function to send a packet.

In addition to the long functions and shallow call graph in the TCP implementation, applying TESLA becomes even more difficult because of the way users of the TCP implementation call into it—code that uses the TCP implementation must register a set of callback functions that are called at certain points in the protocol's execution. This means that control flow moves between user and library code through a *dynamic* interface that cannot be reasoned about easily with TESLA. Figure 5.2 shows an extract from the TCP echo server in which these callbacks are registered.

Registered callbacks are stored as members of a PCB structure. This behaviour defeats TESLA instrumentation (both static and dynamic)—there is currently no way to express "the function `pcb.member` is eventually called" in the assertion language. Unfortunately, these callback functions contain much of the behaviour that would be

[2]Because TESLA runs after preprocessing has been applied to the source code.

```
tcp_recv(newpcb, tcpecho_raw_recv);
tcp_err(newpcb, tcpecho_raw_error);
tcp_poll(newpcb, tcpecho_raw_poll, 0);
tcp_sent(newpcb, tcpecho_raw_sent);
```

Figure 5.2: Callback registration for a user of the LWIP TCP implementation

well-served by TESLA instrumentation. For example, the LWIP documentation describes the mandated behaviour of a particular callback function:

> When the application has processed the incoming data, it must call the `tcp_recved()` function to indicate that TCP can increase the receive window. [30]

Because of the callback interface, instrumenting the invariants of this function could only be done by the consumer of the library (rather than the author of the library). This means that the author of the library can do little beyond documentation to ensure correct usage of the API functions. In section 5.2 I show how this problem can be partially solved using TESLA.

Anderson et al. [2] did not encounter the problems related to callbacks that I describe here—the assertions they wrote are targeted at a complete system implementation where every function of interest is known ahead of time. This means that even if a function is called through a structure interface, it can still be instrumented. This is not the case when unknown user code is responsible for registering the callback functions, as is the case with LWIP.

5.1.3 Summary

The LWIP TCP library presents an interesting target for verification with TESLA. However, the style in which the library is written means that applying TESLA assertions to the internal code is both difficult and unlikely to yield any useful insight into the behaviour of the library. Further informal investigation into the FreeBSD TCP implementation yielded much the same conclusions, and it is likely that other similar libraries would suffer the same problems. Additionally, the use of a callback-based API for users of the library means that TESLA cannot be directly applied in the situation where it would be most useful (enforcing temporal assertions on consumer code). An interesting direction for future work would be to quantify the degree of modification required for library code to be usefully instrumented with TESLA.

5.2 Safer Library Interfaces with TESLA

In this section I describe the implementation of a mechanism by which a library can use TESLA assertions to verify correct usage of the library by user code. First, I relate

the problem to the difficulties encountered when attempting to apply TESLA to LWIP in section 5.1. Then, I describe the construction of such an interface using TESLA. Finally, I successfully apply the technique to an existing server application from the LWIP distribution.

5.2.1 Motivation

In section 5.1 I investigated how TESLA instrumentation could be applied to the internal implementation of LWIP, concluding that the most useful place for instrumentation is at the boundary between user and library code. However, the use of user-registered callbacks means that the library cannot use TESLA to make assertions on user behaviour.

A solution to this problem should allow users to consume the LWIP libraries as they would when using the callback API, while also allowing the library to add TESLA assertions about the behaviour of user code. This would mean the library is able to enforce temporal safety properties of user programs without any prior knowledge of the programs.

5.2.2 Implementation Strategy

LWIP distributes a number of example applications that consume the internal TCP API by using callbacks as described previously—the simplest of these is an implementation of the echo protocol [35]. In this section I describe a modified version of this application that includes TESLA assertions supplied by the library.

In normal usage, a program using the LWIP TCP library calls a library function with a function pointer argument to register their application-specific callbacks. This is a very flexible approach that gives the program fine-grained control over how it interacts with the library. Figure 5.3 shows how the program interacts with a library using this approach.

To add TESLA assertions, user programs must implement a static interface that can be linked with the library. The primary loss of flexibility with this approach is that users cannot change their callback functions at compile time. However, this is not yet a complete solution. Each TESLA assertion must be placed at a source location on the execution path it asserts over—intuitively, this would be within the user-supplied interface functions themselves. Because the definitions of these functions are not available to the library ahead of time, it must implement wrapper functions. These wrappers call through to the user-supplied interface functions, as well as containing the library TESLA assertions. Figure 5.4 shows how a program interacts with the library in this modified usage model.

Adapting the echo server to use this model required only that the wrapper functions and corresponding assertions were written—almost no modification of the application code was required beyond removing the callback registration calls.

This adaptation results in two versions of the server application–one where static analysis has been applied to the library assertions, and one where it has not been.

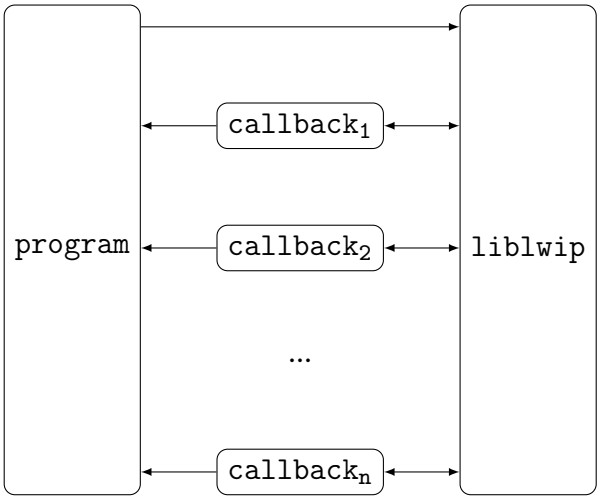

Figure 5.3: Default usage of the LWIP callback API

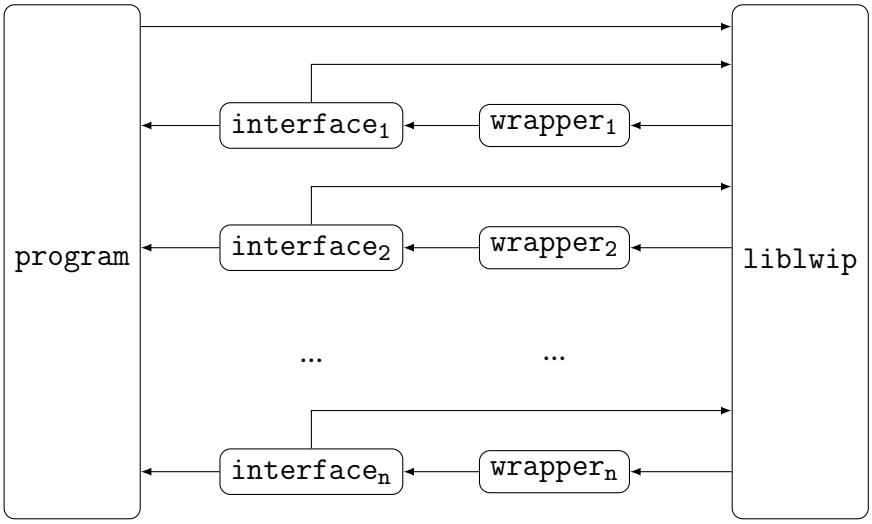

Figure 5.4: Usage of the LWIP callback API with TESLA instrumentation

5.2.3 Summary

In this section I have shown how TESLA can be used by library developers to apply temporal assertions to users of the library without prior knowledge of the user code. Additionally, I have demonstrated a method for adapting library interfaces not explicitly designed with TESLA instrumentation in mind. This technique was applied successfully to an existing application (with minimal modification to application code). In section 6.2 I give a detailed evaluation of the performance of the instrumented applications.

The methods developed have some shortcomings when compared to typical C library development:

Distribution A library developed using these methods must be distributed as LLVM bitcode together with the associated TESLA manifest—this means that users must install the TESLA toolchain and adapt their build process to use the library.

Safety TESLA assertions on a library interface can only usefully enforce properties of the functions in the interface—it is easy to construct user code that circumvents these assertions by executing unsafe code (for example, "casting data to `char *` and manipulating raw bytes" [2, p. 13]).

Performance If the user code is amenable to static analysis, then the performance impact of TESLA is minimal. However, this is dependent on how the user code is written.

Despite these issues, the interface adaptation technique is still a useful tool for library developers to prevent bugs in user code. Additionally, it represents a significant generalisation compared to previous applications of TESLA.

6 Evaluation

In this chapter I evaluate the success of the project with respect to its initial goals, and examine how future work could improve on what has been achieved.

6.1 Static Analysis of Assertions

The primary goal of this research project was to investigate to what extent TESLA assertions can be checked at compile time, with a view to performing optimisation of instrumented programs by omitting provable assertions. With respect to this goal, the project has been successful.

6.1.1 Contributions

My primary contribution with regard to static analysis of TESLA assertions is the TESLA model checker (TMC)—a significant addition to the existing TESLA toolchain. The assertion language used by TESLA is able to express temporal properties not expressible in comparable tools; static analysis of a subset of these assertions is therefore a useful and novel development in the field of program verification.

In subsection 6.2.2, I show that being able to omit provable automata using TMC leads to a performance increase of 64% on an example workload with only 5 assertions, a significant performance improvement. Additionally, counterexamples generated by TMC provide a valuable debugging tool to help developers understand the situations in which their assertions may fail—even if an assertion is not provable, being able to demonstrate situations in which it can fail is useful.

6.1.2 Regression Testing with TMC

To evaluate the usefulness of TMC as a development tool, I used it to perform retroactive regression testing on two applications distributed as part of LWIP (implementations of the echo and SMTP protocols). For each application, I identified a temporal property required of the application callbacks. For example, the SMTP callback for connection close events required a property similar to the one shown in Figure 6.1. The properties used were taken from the applications at their initial working commits.

TMC was able to prove the chosen property initially for the SMTP server, but not for the echo server because of an "exceptional" branch on which the assertion did not hold.

```
TESLA_WITHIN(smtp_close, eventually(
  call(tcp_arg),
  tcp_close(ANY(ptr)) == ERR_OK || call(tcp_arg)
));
```

Figure 6.1: Temporal property required by an SMTP implementation callback

I applied historical commits to the applications in sequence, checking the property at each commit. For the SMTP server, the property was proved by TMC at each commit. For the echo server, the property was provable after a commit refactored the relevant code.

An attempt was made to reproduce a known bug from the LWIP database, but this was not possible—the applications using the LWIP API did not have any bugs registered in the database[1]. A different choice of library may give better results with regard to bug-hunting. However, this is not an indictment of the usefulness of TESLA and TMC—Anderson et al. [2] conclude that TESLA is a useful bug-finding tool, and TMC is a strict improvement on the capabilities of TESLA.

6.1.3 Performance

Running TMC is computationally expensive—the number of execution traces examined depends on both the length bound and the complexity of the control flow graph. However, because traces are examined in order of length, a minimal counterexample will be produced as quickly as possible if one exists. In practical terms, this means that TMC will only run for a long time if an assertion has no counterexamples. For example, checking any incorrect test case from subsection 4.1.2 with a bound of 1000 basic blocks takes less than 0.1s, while the successful cases take approximately 125s.

This informal observation relies on the hypothesis that counterexamples are almost always minimal. I argue that this is the case for automata that do not place finite upper bounds on sequence repetition—if this holds, then the size of the automaton (and therefore of minimal counterexamples) is bounded by the number of separate statements in the automaton. A full analysis of automaton structure may be able to formalise this argument.

Model checking TESLA assertions is likely to remain an "offline" tool for developers—for development builds, the runtime overhead of TESLA instrumentation is likely to be acceptable, while running TMC for several minutes is not likely to be. For release builds, running TMC for a long period of time (by a continuous integration tool, for example) would be an acceptable trade-off to improve runtime performance.

Informal analysis indicates that the complexity of TMC is polynomial in the number

[1] All the bugs are instead relevant to the internal implementation.

of basic blocks bounding the counterexample length. Performance can be significantly improved in two primary ways:

- LLVM implements a control flow graph simplification pass that can potentially reduce the number of conditional branches. Running this pass after the inlining process led to a significant performance improvement.
- Narrower bounds for assertions will naturally lead to less complex control flow in the inlined function—this is the responsibility of the assertion author.

I estimate that the corpus of 84 TESLA assertions written by Anderson et al. [2] for the FreeBSD kernel could be checked by TMC during an overnight build of the kernel (8–10 hours). This time estimate relies on the fact that some of the assertions are not provably safe. Improvements could be made to this time by parallelising TMC, or by not rechecking assertions on unmodified code paths. A formal measurement of this time was not possible because the assertions were written against an older version of TESLA not supported by TMC.

6.1.4 Correctness

TMC implements a decision procedure ("is this assertion usage safe to remove?"). False negatives result in redundant instrumentation code being left in a program, while false positives result in dynamic checking being unsafely removed. As a result, TMC must be completely free of false positives in order to be correct. Livshits et al. [28] advocate for *soundiness*[2] as a goal of program analysis. In this spirit, I acknowledge that TMC is unsound in two important ways: trace length and inlining depth. If these two parameters are sufficiently large (for a program being analysed), then the analysis I present is sound.

During development, any false positives were treated as serious bugs. During my instrumentation of the LWIP library interface (and other work not included in this dissertation), I encountered only a small number of these, all of which were corrected.[3] Future work on testing TMC could use techniques from automatic test generation or fuzzing to ensure no false positives are found.

6.1.5 Future Work

Future work on TMC could improve it in a number of ways. The use of non-symbolic model checking is not optimal—adapting it to construct a symbolic representation would allow it to take advantage of wider improvements in the field of model checking.

The TESLA assertion language could be extended to allow richer properties of function return values to be expressed (e.g. "function `f` returns a value > 0 and ≤ 10").

[2] In short, soundiness is "soundness modulo openly-stated exceptions".

[3] Some "unsafe" behaviour remains because of bugs in the original TESLA implementation.

SMT methods are well suited to this type of expression. Another interesting extension would be to allow a non-sequential syntax for describing complex automata explicitly, though integrating this with the clang-based assertion parser would be difficult.

Kashyap and Garg [25] describe a method for discovering "crucial events" in traces—these events can be used to perform more intelligently directed search for counterexamples. Applying these techniques to TMC could improve checking performance when counterexamples are not the shortest traces.

6.2 Application to Real-World Code

A secondary goal of the project was to investigate how TESLA could be used to verify the behaviour of a network protocol. In particular, performance-sensitive code was of interest here in order to show the potential benefits of applying static analysis to TESLA assertions. While this goal was not directly successful, the investigation into how such code could be modelled led to an explicit analysis of source code features that make TESLA assertions less applicable, as well as to a general method for applying TESLA to library interface code.

6.2.1 Contributions

The approach taken by Anderson et al. [2] towards using TESLA placed it firmly in the category of *debugging tools*—that is, TESLA instrumentation could be added to a program in order to diagnose bugs, but it would be removed in a release build due to the associated runtime performance overhead. The use cases described in the original TESLA paper are all applications to individual programs or libraries where this debugging process has been used successfully. However, it is worth noting that the original uses of TESLA all involve instrumentation on system boundaries in much the same way as the general usage I describe (though this is not stated explicitly in the paper).[4]

I contribute a more general usage of TESLA—library developers can use TESLA to enforce usage properties of their code without having prior knowledge of the user's code. This method is applied successfully to the LWIP TCP callback library. While applying TESLA in this way is possible without my extensions for static analysis, the performance impact of doing so can be prohibitive. In subsection 6.2.2 I perform a detailed performance analysis of this performance impact, and the possible improvements that can be made.

6.2.2 Performance

In section 5.2 I described an adaptation of the LWIP TCP interface to allow for TESLA instrumentation, and demonstrated that it could be used to build existing applications

[4] In particular, the boundaries identified are the OpenSSL X509 verification API, the FreeBSD MAC framework and the Objective-C message dispatch mechanism.

with almost no modification of their code. However, adding TESLA instrumentation incurs a performance overhead for the application.

In this section I analyse the overhead of TESLA instrumentation in the context of an echo server written using the LWIP TCP library. I show that even with as few as five assertions, performance is degraded by 40%. Finally, I show that applying TMC to these assertions can reduce the overhead significantly, and in some cases remove it entirely.

Experimental Setup

A useful benchmark for an echo server is to measure how many requests of a fixed size it can handle in a fixed time period. An existing tool by Hoyer [23] was used to perform this benchmark—the tool runs for a fixed length of time, sending as many messages to a server as it can (using a configurable number of threads).

Three versions of the echo server were compared in this experiment—an unmodified one compiled directly from the LWIP sources, one with TESLA instrumentation enabled, and one where TESLA instrumentation was removed by TMC (referred to as *unmodified*, *instrumented* and *static* respectively).

The benchmarks were run on a dedicated server (Intel Xeon E5-1620 3.6 GHz, 8 cores, 64GB of RAM) running FreeBSD 11. The benchmark was run for 60 s with a message size of 512 bytes in every case, and the number of sending threads was varied from 1 to 10.

For the instrumented and statically analysed versions, the library wrapper code contained five TESLA assertions covering the possible library calls that the user code could make. All of these were reported as safe by TMC.

Results

In all three version, throughput was saturated when sending on two or more threads—using more threads to send data had no effect on throughput. The mean throughput over 2–10 sending threads was therefore used as a measure of the maximum possible performance of each server (relative to the unmodified server implementation).

Figure 6.2 compares the relative throughput for each implementation. The version with runtime TESLA instrumentation achieves only 61% throughput when compared to the unmodified version, while the version with static analysis applied achieves 84% throughput.

Even though all TESLA automata were reported as safe by TMC, some TESLA code remains in the binary. Using Callgrind [38] (a simulation-based profiling tool) shows that the statically-analysed server spends a large portion of its execution time in the TESLA runtime library (performing redundant work).

Because the overhead of the remaining TESLA instrumentation is high, even when there are no automata, I extended the instrumenting tool with an optimisation to handle this case. The resulting server implementation achieved 99.4% throughput com-

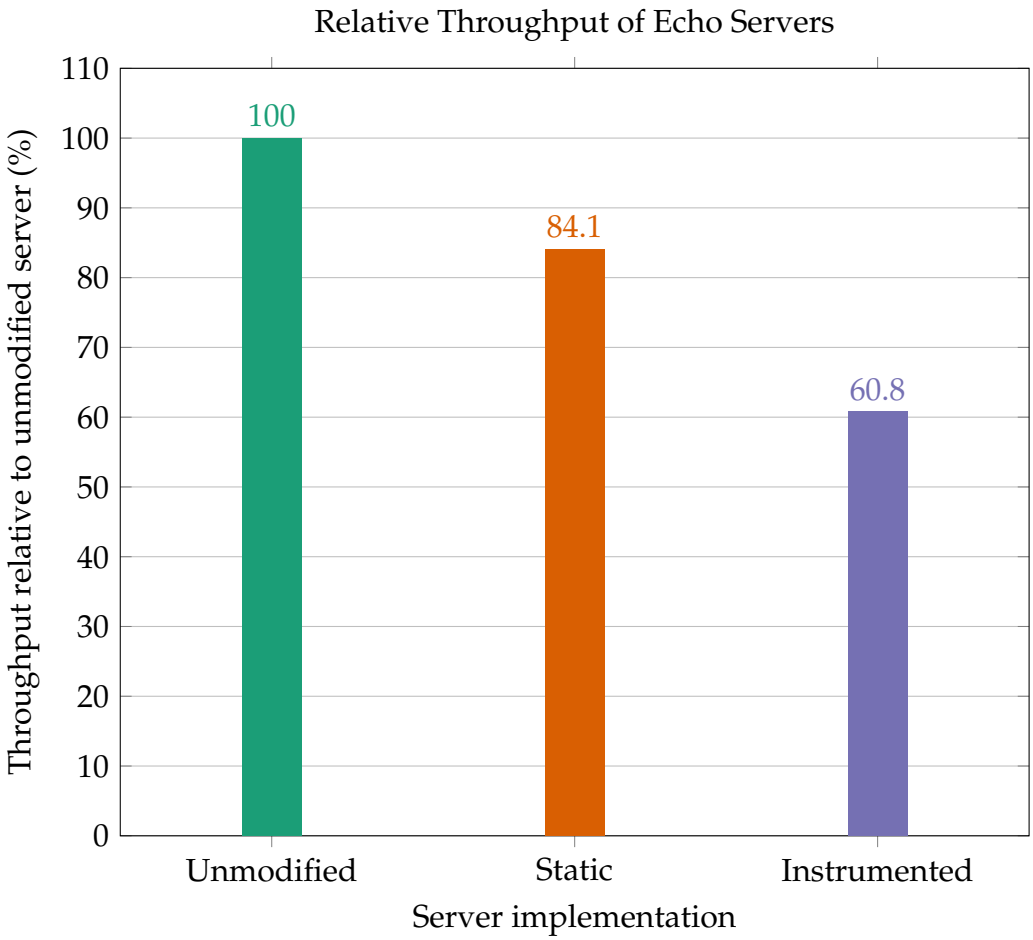

Figure 6.2: Effect of TESLA instrumentation and library interface adaptation on echo server throughput

pared to the unmodified server. While this performance result is appealing, it is worth noting that it will only be possible when every automaton is statically provable.

Analysis

It is clear from these results that TESLA instrumentation has a significant impact on performance. In this section I analyse the causes of this overhead by using the `hwpmc` [24] tools available in FreeBSD.

Hardware performance counters are a set of specialised registers available on modern processors that can be used to measure architectural and microarchitectural performance events. For example, the number of mispredicted branches or the number of cache line misses are commonly made available. The exact data available varies between architectures, and even between different models of processor on the same architecture. FreeBSD makes these performance counters available to user programs through the `hwpmc` driver and associated tooling.

Using these counters can have an effect on the performance characteristics of a program. In order to verify that this effect was acceptable, the experiment described above was repeated with five performance counters enabled. There was no significant change in relative performance when running with counters enabled—the maximum deviation from the results given above was 1.7%.[5]

The performance counters enabled were as follows: the number of instructions retired, the number of load instructions retired, and the number of loads from the L1, L2 and L3 caches respectively. Each counter was run in sampling mode, meaning that data was obtained by statistical sampling over all the events observed. Each server was sampled over the course of a fixed-size data transfer, with the size of the data used varying from 10–150MB.

An approximation to the amount of work performed by each version is the total number of instructions retired.[6] Figure 6.3 shows the number of instructions retired during these transfers for each server version.

The number of instructions retired is proportional to the quantity of data being transferred by the server. On average, the statically analysed server retired 1.37× the instructions of the unmodified server, while the instrumented version retired 2.49× as many. Further experiments with much larger data files showed that this relationship continues to hold.

The memory access patterns for each version are almost identical—for each one, 28% of the total instructions retired are load instructions, of which 94% are served by the L1 cache, and less than 0.2% by the L3 cache (although because the total number of instructions executed is greater, the total number of loads from the L3 cache and cache misses will increase).

A similar analysis was performed for instruction cache misses, with the same conclusion—TESLA instrumentation did not proportionately change the microarchitectural performance characteristics of the server implementations.

The sampled counter data can be used to show where instructions are retired during execution. Figure 6.4 shows a sampled excerpt from the instrumented server. From this data we can see that TESLA automaton operations account for 37% of the retired instructions, with 25% corresponding to state updates and 12% to lifetime management.[7]

The equivalent data for the statically analysed server (in Figure 6.5) shows where the remaining overhead arises. Calls to the automaton lifetime functions remain (with a similar number of associated instructions), while the state update functions have been removed. Calls to the lifetime functions can be removed only if there are no automata to be instrumented.

[5] Absolute throughput with counters enabled was approximately 90% of the throughput with no counters.

[6] An instruction is retired when it has been executed and its effects written back—in a superscalar architecture, instructions may be speculatively dispatched but not retired.

[7] The TESLA terminology for creating and destroying automata is “sunrise” and “sunset”.

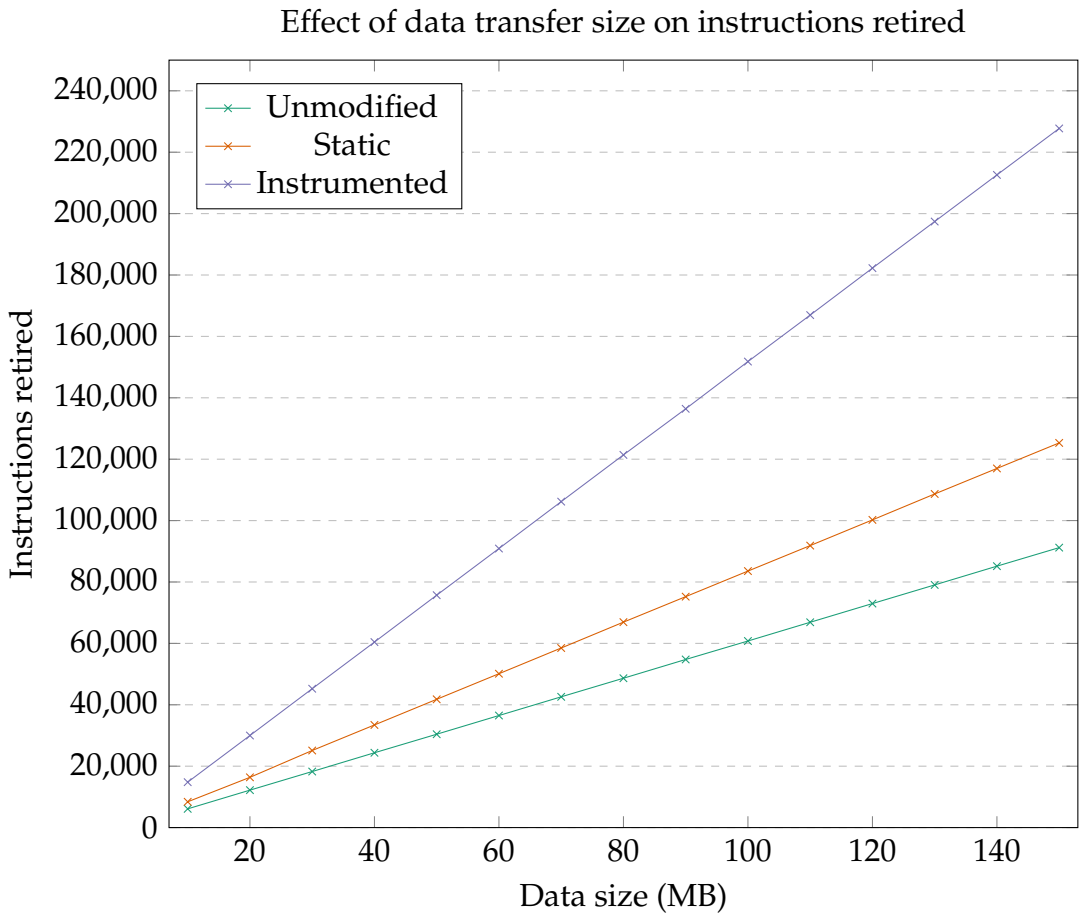

Figure 6.3: Number of instructions retired during a fixed-size data transfer

```
@ INSTR_RETIRED_ANY [227749 samples]

36.66%  [83489]    strncmp @ /lib/libc.so.7
  100.0%  [83489]     same_static_lifetime @ libtesla.so
    68.99%  [57601]      tesla_update_class_state @ libtesla.so
    16.00%  [13358]      tesla_sunset @ libtesla.so
    15.01%  [12530]      tesla_sunrise @ libtesla.so
06.73%  [15337]    inet_chksum_pseudo @ tesla-app.instr
  ...
```

Figure 6.4: Callchain output showing number of instructions retired with TESLA instrumentation enabled

```
@ INSTR_RETIRED_ANY [125375 samples]

22.71%  [28467]    strncmp @ /lib/libc.so.7
  100.0%  [28467]     same_static_lifetime @ libtesla.so
    51.17%  [14567]      tesla_sunrise @ libtesla.so
    48.83%  [13900]      tesla_sunset @ libtesla.so
12.74%  [15968]    inet_chksum_pseudo @ tesla-app.static
  ...
```

Figure 6.5: Callchain output showing number of instructions retired with TESLA instrumentation removed

6.3 Usability

While not an explicit goal of the project at the outset, the usability of TESLA as a programming tool is an issue that I encountered frequently during the course of the project. I was able to resolve a number of these issues, contributing to the usability of TESLA:

LLVM TESLA now builds against the latest stable version of LLVM, making it easier to install (a full source build of LLVM is no longer necessary on most operating systems). Future updates will also be made easier by this work.

Installation I have simplified the TESLA build and installation process, allowing it to be installed using Homebrew [22] on macOS. Distributions for other package managers would also be possible.

Documentation The TESLA documentation was somewhat out of date when I began my work—I have produced a new set of documentation that covers basic usage of TESLA, and notes many of the subtle issues I have experienced during my work.

7 | Conclusion

In this report, I have investigated the use of static analysis techniques for optimising TESLA assertions. My implementation of a model checker for TESLA (TMC) is able to correctly check a useful subset of its assertions. Additionally, I have shown significant performance improvements that allow TESLA to be used in contexts it would not previously have been feasible to.

As well as my work on static analysis, I have contributed to the understanding and usage of TESLA in general—TMC can be used to produce counterexamples to a TESLA assertion, and I provide a collection of programs that demonstrate how TESLA can be used to model temporal properties of a data structure. Future work on TESLA (whether in the context of static analysis or otherwise) will benefit from these improvements, as well as from the improvements to the TESLA implementation and documentation I have contributed.

TMC itself constitutes an explicit formalisation of the underlying principles of TESLA, as well as demonstrating a novel SMT-based algorithm for proving properties of control flow that depend partially on data flow. I have identified ways in which future work could extend these techniques to prove stronger properties.

An initial goal of the project was to provide a TESLA model of a network protocol implementation. While this goal was not fully successful, the insights gained from the investigation itself are a useful contribution in their own right—I have described features of source code that are hostile to TESLA instrumentation and static analysis, and introduced a characterisation of scenarios in which TESLA can be usefully applied.

Motivated by this characterisation of the effective use of TESLA, I have demonstrated a framework by which library developers can more reliably enforce temporal properties on usages of their library code. This is a more general application of TESLA than previous work has demonstrated. The performance improvements made possible through the use of static analysis are an important part of this contribution—I have shown that even a small number of assertions can cause a large runtime performance overhead on instrumented programs, and that TMC is able to completely eliminate this overhead in real programs.

Bibliography

[1] Bowen Alpern and Fred B. Schneider. "Defining Liveness". In: *Information Processing Letters*. 21st ed. Vol. 4. Ithaca, NY, USA: Cornell University, 1984, pp. 181–185. URL: https://doi.org/10.1016/0020-0190(85)90056-0.

[2] Jonathan Anderson et al. "TESLA: Temporally Enhanced System Logic Assertions". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. ACM, 2014, 19:1–19:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592801. (Visited on 10/04/2016).

[3] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. 2016. URL: www.SMT-LIB.org.

[4] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*. Ed. by A. Gupta and D. Kroening. 2010. URL: http://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.0-r10.12.21.pdf.

[5] Clark Barrett et al. "CVC4". In: *Proceedings of the 23rd International Conference on Computer Aided Verification*. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177. ISBN: 978-3-642-22109-5. URL: http://dl.acm.org/citation.cfm?id=2032305.2032319 (visited on 04/27/2017).

[6] Al Bessey et al. "A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World". In: *Commun. ACM* 53.2 (Feb. 2010), pp. 66–75. ISSN: 0001-0782. DOI: 10.1145/1646353.1646374. (Visited on 05/23/2017).

[7] Armin Biere et al. "Symbolic Model Checking Without BDDs". In: *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*. TACAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 193–207. ISBN: 978-3-540-65703-3. URL: http://dl.acm.org/citation.cfm?id=646483.691738 (visited on 04/09/2017).

[8] Armin Biere et al. *Handbook of Satisfiability*. Vol. 185. Frontiers in Artificial Intelligence and Applications. Amsterdam, The Netherlands: IOS Press, 2009. ISBN: 978-1-58603-929-5.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224. URL: http://dl.acm.org/citation.cfm?id=1855741.1855756 (visited on 04/09/2017).

[10] Hao Chen and David Wagner. "MOPS: An Infrastructure for Examining Security Properties of Software". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. CCS '02. New York, NY, USA: ACM, 2002, pp. 235–244. ISBN: 978-1-58113-612-8. DOI: 10.1145/586110.586142. (Visited on 04/10/2017).

[11] Hao Chen, David Wagner, and Drew Dean. "Setuid Demystified". In: *Proceedings of the 11th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2002, pp. 171–190. ISBN: 978-1-931971-00-3. URL: http://dl.acm.org/citation.cfm?id=647253.720278 (visited on 04/12/2017).

[12] Edmund M. Clarke and E. Allen Emerson. "Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic". In: *Logic of Programs, Workshop*. London, UK, UK: Springer-Verlag, 1982, pp. 52–71. ISBN: 978-3-540-11212-9. URL: http://dl.acm.org/citation.cfm?id=648063.747438 (visited on 04/08/2017).

[13] Edmund Clarke, Daniel Kroening, and Karen Yorav. "Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking". In: *Proceedings of the 40th Annual Design Automation Conference*. DAC '03. New York, NY, USA: ACM, 2003, pp. 368–371. ISBN: 978-1-58113-688-3. DOI: 10.1145/775832.775928. (Visited on 04/10/2017).

[14] David R. Cok, David Déharbe, and Tjark Weber. "The 2014 SMT Competition". In: *Journal on Satisfiability, Boolean Modeling and Computation* 9 (2014), pp. 207–242. URL: https://satassociation.org/jsat/index.php/jsat/article/view/122.

[15] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. New York, NY, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047. (Visited on 04/27/2017).

[16] Lucas Cordeiro, Bernd Fischer, and Joao Marques-Silva. "SMT-Based Bounded Model Checking for Embedded ANSI-C Software". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. ASE '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 137–148. ISBN: 978-0-7695-3891-4. DOI: 10.1109/ASE.2009.63. (Visited on 04/11/2017).

[17] Markus Dahlweid et al. "VCC: Contract-Based Modular Verification of Concurrent C". In: *2009 31st International Conference on Software Engineering - Companion Volume*. 2009 31st International Conference on Software Engineering - Companion Volume. May 2009, pp. 429–430. DOI: 10.1109/ICSE-COMPANION.2009.5071046.

[18] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. ISBN: 978-3-540-78799-0. URL: http://dl.acm.org/citation.cfm?id=1792734.1792766 (visited on 04/25/2017).

[19] Adam Dunkels. *Design and Implementation of the lwIP TCP/IP Stack*. Swedish Institute of Computer Science, 2001. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.109.1795&rep=rep1&type=pdf.

[20] Cindy Eisner et al. "Reasoning with Temporal Logic on Truncated Paths". In: *Computer Aided Verification*. International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, July 8, 2003, pp. 27–39. DOI: 10.1007/978-3-540-45069-6_3. (Visited on 05/05/2017).

[21] Julien Henry, David Monniaux, and Matthieu Moy. "PAGAI: A Path Sensitive Static Analyser". In: *Electron. Notes Theor. Comput. Sci.* 289 (Dec. 2012), pp. 15–25. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2012.11.003. (Visited on 03/31/2017).

[22] *Homebrew — The Missing Package Manager for macOS*. URL: https://brew.sh/ (visited on 05/31/2017).

[23] Harald Hoyer. *Rust Echo Bench*. Version 8c116c6. Dec. 9, 2016. URL: https://github.com/haraldh/rust_echo_bench (visited on 04/18/2017).

[24] *hwpmc(4): FreeBSD Kernel Interfaces Manual*. Nov. 2012. URL: https://www.freebsd.org/cgi/man.cgi?query=hwpmc.

[25] Sujatha Kashyap and Vijay K. Garg. "Producing Short Counterexamples Using "Crucial Events"". In: *Computer Aided Verification*. International Conference on Computer Aided Verification. Springer, Berlin, Heidelberg, July 7, 2008, pp. 491–503. DOI: 10.1007/978-3-540-70545-1_47. (Visited on 05/06/2017).

[26] Leslie Lamport. "Proving the Correctness of Multiprocess Programs". In: *IEEE Trans. Softw. Eng.* 3.2 (Mar. 1977), pp. 125–143. ISSN: 0098-5589. DOI: 10.1109/TSE.1977.229904. (Visited on 04/06/2017).

[27] Chris Lattner. "LLVM: An Infrastructure for Multi-Stage Optimization". Computer Science Dept., University of Illinois at Urbana-Champaign, 2002. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.10.331 (visited on 04/06/2017).

[28] Benjamin Livshits et al. "In Defense of Soundiness: A Manifesto". In: *Commun. ACM* 58.2 (Jan. 2015), pp. 44–46. ISSN: 0001-0782. DOI: 10.1145/2644805. (Visited on 05/14/2017).

[29] *LLVM Language Reference Manual — LLVM 5 Documentation*. URL: http://llvm.org/docs/LangRef.html#abstract (visited on 05/31/2017).

[30] *lwIP Wiki: Raw TCP*. URL: http://lwip.wikia.com/wiki/Raw/TCP#Receiving_TCP_data (visited on 05/31/2017).

[31] Kenneth Lauchlin McMillan. "Symbolic Model Checking: An Approach to the State Explosion Problem". Pittsburgh, PA, USA: Carnegie Mellon University, 1992. URL: http://www.kenmcmil.com/pubs/thesis.pdf.

[32] Florian Merz, Stephan Falke, and Carsten Sinz. "LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR". In: *Verified Software: Theories, Tools, Experiments*. International Conference on Verified Software: Tools, Theories, Experiments. Springer, Berlin, Heidelberg, Jan. 28, 2012, pp. 146–161. DOI: 10.1007/978-3-642-27705-4_12. URL: http://link.springer.com/chapter/10.1007/978-3-642-27705-4_12 (visited on 02/07/2017).

[33] Jeremy Morse et al. "Context-Bounded Model Checking of LTL Properties for ANSI-C Software". In: *Proceedings of the 9th International Conference on Software Engineering and Formal Methods*. SEFM'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 302–317. ISBN: 978-3-642-24689-0. URL: http://dl.acm.org/citation.cfm?id=2075679.2075702 (visited on 04/11/2017).

[34] Jeremy Morse et al. "Model Checking LTL Properties over ANSI-C Programs with Bounded Traces". In: *Software & Systems Modeling* 14.1 (Feb. 1, 2015), pp. 65–81. ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-013-0366-0. (Visited on 02/07/2017).

[35] J. Postel. *Echo Protocol*. STD 20, RFC 862. May 1983. URL: http://dx.doi.org/10.17487/RFC0862.

[36] *Publications: Z3Prover/Z3 Wiki*. URL: https://github.com/Z3Prover/z3/wiki/Publications (visited on 05/31/2017).

[37] Michael O. Rabin and Dana Scott. "Finite Automata and Their Decision Problems". In: *IBM J. Res. Dev.* 3.2 (Apr. 1959), pp. 114–125. ISSN: 0018-8646. DOI: 10.1147/rd.32.0114. (Visited on 04/07/2017).

[38] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. "A Tool Suite for Simulation Based Analysis of Memory Access Behavior". In: *Computational Science - ICCS 2004*. International Conference on Computational Science. Springer, Berlin, Heidelberg, June 6, 2004, pp. 440–447. DOI: 10.1007/978-3-540-24688-6_58. (Visited on 04/19/2017).