ThinTESLA - A low-overhead temporal assertions framework

Daniele Vettorel
St. Catharine's College

UNIVERSITY OF CAMBRIDGE

A dissertation submitted to the University of Cambridge in partial fulfilment of the requirements for the degree of Master of Philosophy in Advanced Computer Science

University of Cambridge
Department of Computer Science and Technology
William Gates Building
15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom

Email: dv300@cam.ac.uk

June 7, 2018

Declaration

I Daniele Vettorel of St. Catharine's College, being a candidate for the M.Phil in Advanced 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: 14,911

Signed:

Date:

Abstract

Systems developers use a variety of debugging techniques, including source-level assertions. However, the usual kind of assertions lacks expressiveness and is local to a specific scope. TESLA is a framework that adds temporal semantics to assertions for the C programming language, allowing them to span arbitrary scopes and periods of time without requiring the developers to manually carry state between procedures. This dissertation presents ThinTESLA, a new and improved version of the framework that addresses its predecessor's shortcomings and performance concerns.

ThinTESLA's design is based on a new lightweight, efficient representation of temporal assertions' automata and novel algorithms for automata verification at runtime. New temporal semantics are introduced that overcome the expressiveness limitations of the original framework, and a number of optimizations to the handling of temporal bounds are employed as to further minimize overhead in kernel-space. Furthermore, this dissertation introduces a prototype for a static analysis algorithm that can help optimize temporal assertions at the source level.

The new framework has been evaluated in a series of micro and macrobenchmarks, and is shown to achieve better performance over TESLA by more than an order of magnitude in user-space while reducing the overhead of in-kernel assertions by up to 45% when the FreeBSD kernel is instrumented with temporal assertions. The overhead incurred by the use of ThinTESLA is significantly lower than that of commonly-used kernel debugging utilities such as WITNESS and INVARIANTS, making the use of temporal assertions during everyday debugging much more realistic.

Acknowledgements

I would like to thank my supervisor, Robert N. M. Watson, for his guidance through these very intense 9 months, and for having helped me write a much better dissertation with his useful insights.

I would like to thank all TESLA's authors: Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Brooks Davis and Ilias Marinos. In particular, I would like to thank Jonathan Anderson for helping me getting acquainted with TESLA, help which has been crucial at the beginning of the project. I'm very grateful to Khilan Gudka for the time he has spent setting up the testing machine for the evaluation of ThinTESLA. Similarly, I would like to thank Nuno Lopes from Microsoft Research, who has given me many ideas on how to improve my project.

This dissertation has been carried out while I was applying to my PhD. I would like to thank my referees Åge Kvalnes, Francesco Ricci, Miguel Castro and Werner Nutt for their time and patience.

Lastly, I would like to thank my friends, which are now all around the world, and my family, for all their support which I deeply appreciate, and for having made this year much less stressful.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A useful technique systems developers employ to make sure program invariants are not violated is to write source-level assertions that are later checked at run time. While this helps avoid some kinds of bugs such as access violations and invalid arguments, assertions are often not sufficient if an invariant spans more than a single function. Temporal assertions aim to solve this issue by allowing developers to express sequences of temporal events that should be respected by the running program. Temporal semantics allow programmers to refer to more than just the local scope where a normal, "instantaneous" assertions would be placed. For example, an assertion in the filesystem component may refer to a function found in the virtual memory subsystem (e.g. to check that `mprotect` was called before memory mapping a file).

TESLA [1] is a framework for C and Objective-C that enhances source-level assertions with temporal semantics. However, the performance overhead that it introduces and the difficulty to integrate it with existing codebases make it less usable and hard for developers to adopt it. In fact, TESLA is a piece of research that has not seen wide deployment. This dissertation focuses on a new, specialized version of TESLA for temporal assertions where events are C function calls. The new version addresses the performance concerns of the original version and makes it easy to integrate.

The shortcomings of TESLA have been analysed and improved upon in a new re-designed implementation that employs asymptotically and micro-architecturally faster algorithms for assertion verification. Furthermore, a prototype of a novel static analysis pass that deduces temporal properties of programs has been developed as a static aid to the runtime library. Special effort has been put in maintaining the original semantics while also extending them to address important limitations, and bugs that were found from the analysis of the original version have been addressed.

This dissertation makes the following contributions:

- A temporal automaton representation, used to encode temporal assertions, which drastically reduces the number of explicit transitions.
- An algorithm based on *temporal tagging* for verifying non-static temporal automata (derived from TESLA temporal assertions) in low-memory environments.
- A cache-efficient algorithm for the linearisation of temporal histories and verification of non-static temporal automata in the context of temporal assertions.
- A static analysis algorithm that can infer *temporal properties* from the program and use them to optimize TESLA temporal assertions.
- An experimental evaluation of the algorithmic improvements through a prototype implementation and performance-related experiments.

ThinTESLA has been evaluated in a series of micro and macrobenchmarks, including some of the original ones used for the evaluation of TESLA such as the FreeBSD kernel. Evaluation results show that ThinTESLA can perform more than an order of magnitude better than TESLA in many userspace scenarios, and can reduce the overhead of temporal assertions in the kernel by up to 3 times with respect to TESLA. The improvements can be attributed to both the efficiency of the verification algorithms, which prove to be much less computationally expensive than the general one employed by TESLA, and to an implementation focused on reducing the footprint on micro-architectural

aspects such as the cache.

Furthermore, ThinTESLA's overhead is shown to be much more bearable than that of kernel debugging utilities such as WITNESS and INVARIANTS, allowing the use of temporal assertions during debugging without unrealistically compromising on performance.

Chapter 2

Background

ThinTESLA is a re-design of TESLA, an experimental tool for writing temporal assertions that enables developers to express and verify complex temporal properties of C and Objective-C programs[1]. This chapter focuses on introducing TESLA, its semantics and implementation, and its shortcomings that prevent it from being widely adopted. A discussion on related work can be found in Chapter 6.

2.1 TESLA: Temporally enhanced system logic assertions

TESLA is a tool that allows programmers to dynamically validate temporal properties of their C and Objective-C programs, such as verifying that a certain function is always called before another with some specific arguments, or that a particular sequence of function calls always happens in a pre-determined order. Additionally, TESLA is able to verify that structs' fields are modified in a specified order and with a specified value.

TESLA aims to enhance the concept of assertion, previously only confined to a single temporal point, extending it to span a period of time. A non-

```
TESLA_WITHIN(systemcall, TSEQUENCE(
        has_permissions(fd) == true,
        TESLA_ASSERTION_SITE,
        call(open_file(fd))))
```

Assertion 2.1: An example of a TESLA assertion

temporal assertion is limited in scope to its enclosing function and likely to only check variables (e.g. `assert(ptr != NULL)` or expressions that are limited by the syntax of the C language. A TESLA assertion, on the other hand, can be much more expressive and span more than one function or scope, as shown by the example Assertion 2.1. In the example, multiple functions are referenced in a temporal sequence that the target program must respect.

2.2 TESLA semantics

TESLA assertions are expressed through temporal specifications inserted in the codebase by the programmer. Every assertion must specify the temporal bounds of the assertion (usually a function's entry and exit) and the temporal sequence of events to be verified. For example, in Assertion 2.1, the temporal bounds consist of the function `systemcall`'s prologue and epilogue, and within those temporal bounds a call to `open_file` must be preceded by a call to `has_permissions` that must return `true`. Both calls refer to the same file descriptor. The special event **TESLA_ASSERTION_SITE** is an event that is fired whenever the assertion site (e.g. the point in the code in which the assertion is defined) is reached. Both constants and variables are allowed to be used in TESLA assertions, as long as the variables are in scope when the assertion site is reached.

The assertion specification is translated into a non-deterministic finite automaton, of which a different instance is created each time the temporal bounds are entered at runtime. TESLA ensures that the automaton's in-

stance only takes valid transitions and causes the assertion to fail if that is not the case. If the assertion site is not reached by a specific instance, the assertion is ignored for that instance, unless the programmer specifies otherwise. Figure 2.1 shows the NFA generated by TESLA for Assertion 2.1.

Various *modifiers* are available to the programmer to tune assertions: functions can be instrumented either in the caller or callee context, some calls may be allowed to repeat up to a certain number of times, or different event sequences can be joined together via a logical OR expression. Specific automata can be expressed out-of-source and reused by different inline assertions as sub-automata, and struct field assignments can be verified as well as function calls.

Additionally, TESLA allows the programmer to specify whether an automaton's instance should be spawned for each thread entering the temporal bounds or if there should be a single, global automaton instance. In the latter case, TESLA should be responsible for the synchronization of the single automaton's instance. However, this does not seem to be the case in the original implementation, probably due to a bug (see §2.4).

Finally, in the FreeBSD kernel implementation of TESLA, facilities are provided to instantiate DTrace probes so that TESLA assertions can be traced and monitored from userspace.

2.3 TESLA implementation

TESLA's implementation consists of two major components, the instrumentation library and the runtime verification library (*libtesla*). The first operates at compile-time and is responsible of instrumenting the events referenced by each assertion found in the source code, inserting the appropriate calls to *libtesla*. At runtime, *libtesla* is responsible for verifying that the assertions are respected and reporting an error in case they are not.

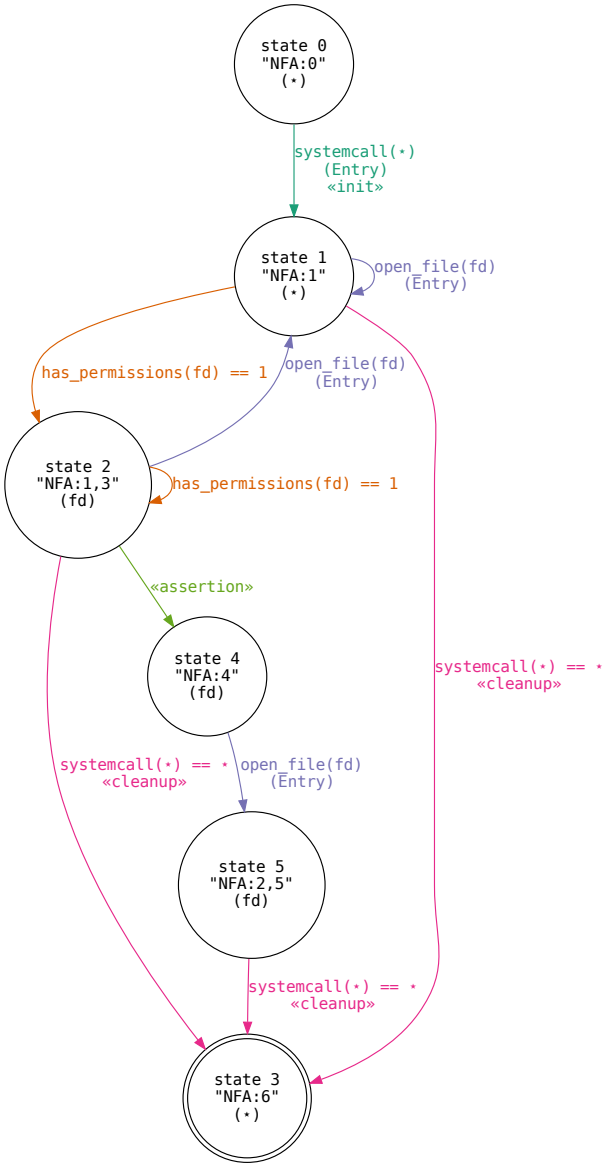


Figure 2.1: A TESLA state machine derived from an assertion specification (Assertion 2.1)

2.3.1 Instrumentation library

The instrumentation library is implemented as an LLVM [2] module which instruments the code base after every source file containing TESLA assertions has been analysed by a custom Clang LibTooling module. The Clang module analyses a source file in search of TESLA assertions and outputs a description of them in a format defined by a Protobuf [3] manifest file. Each source file is analysed independently and then each output is concatenated together in a final TESLA manifest.

The LLVM module reads the TESLA manifest, produces an NFA for every assertion, and inserts the correct instrumentation for every event that appears in the automaton. Specifically, a call to *libtesla* is inserted at every point in which an automaton's instance needs to be updated, including the temporal bounds which create and destroy the instance. All of the static data structures required by TESLA, such as the automaton description and each event's information, are inserted as global variables in the data section of the final executable. When an event contains static values to match (e.g. a function call with a constant as an argument), the call to *libtesla* is inserted only after a check on those static values, in order to minimize the performance overhead incurred by calling *libtesla*.

2.3.2 The *libtesla* runtime library

The existing runtime library, *libtesla*, is written in C and is responsible for managing automata's instances, verifying whether assertions pass or fail, and reporting messages to the user. The instrumented program is linked to *libtesla* and the user can control the level of information that is produced and reported through an environment variable without the need of re-linking the library. Per-thread state is maintained with the help of `pthreads` thread-local storage APIs, while global assertions are synchronized via pthread mutexes (however, the implementation does not seem to be always producing correct results, as explained in §2.4).

Automata instances are created at the first temporal bound and may be forked, if necessary, into more specific instances when more information becomes available, e.g. when the assertion site is reached or when an event relying on runtime variables matches specific values. A set of handlers define the behaviour when certain conditions become true, for example when an automaton's instance fails to find a correct transition. A handlers may report the failure to the user via the console, while another handler may instead stop the execution of the program. Handlers are provided as an easy way to augment TESLA's behaviour and make it more flexible for programmers' needs.

A slightly different version of *libtesla* is available for the FreeBSD kernel, with changes to the per-thread logic, as the `pthreads` API is not available in the kernel. Per-thread state is instead kept in a heap-allocated structure pointed to by the kernel's thread data structure.

The library reserves some static space to use for bookkeeping and storage of data such as events' runtime values, and falls back to the heap whenever the former is used up in its entirety. The kernel version of *libtesla* is not always able to rely on heap allocation (due to locking, reentrancy and ordering concerns), and therefore simply drops automata instances in situations of high memory pressure. The only exception is when the per-thread state is allocated, since the kernel is guaranteed to be able to use the heap allocator at thread creation time without causing a deadlock or any other unwanted effect.

2.4 Shortcomings of TESLA

The original TESLA implementation based on LLVM 3.5 is freely available and open source on an online Git repository [1], and I have updated it to support LLVM 6.0 [2]. TESLA's runtime library *libtesla* has not been modified

[1] `https://github.com/cadets/tesla`
[2] `https://github.com/banex19/thin-tesla`

from the original version, and only the instrumentation library has been slightly modified to accommodate the API changes in the recent version of LLVM.

2.4.1 Bugs

While performing some tests both on userspace programs and on the FreeBSD kernel, a number of inconsistencies, probably due to bugs, were observed:

1. Boolean OR expressions in assertions do not always yield the same semantic result. In some cases they are treated as a logical OR as expected, while in others they are treated as an XOR operation. This appears to depend on seemingly irrelevant factors such as whether the same event appears twice or more times in the assertion or if the OR expression is at the top-level of the assertion.

2. Some events are not correctly handled and cause assertion failures even when the assertion should pass instead. This mainly pertains function calls instrumented to pass runtime values to *libtesla*.

3. Assertions that are global and should be synchronised by *libtesla* do not appear to always be. This always results in unintended behaviour and often also in a crash of the instrumented program.

4. The corpus of assertion used to evaluate TESLA in the FreeBSD kernel contains a large number of failing assertions, mainly due to the same assertion path being run twice in succession and triggering a transition failure. While this was observed on FreeBSD 11.1 and the original evaluation was on a development version of FreeBSD 10, many code paths, including the failing ones, are the same.

2.4.2 Design trade-offs

TESLA was designed with a large number of features to support in mind, including various assertion modifiers and the instrumentation of functions, struct fields, and Objective-C messages as possible assertion events. The infrastructure needed to generalize such an extensive event space is not trivial and does not allow for specific optimizations and assumptions that may make the system considerably faster. For example, a function call may be assumed to be much less frequent than the assignment of a structure's field, and is likely to require less storage space as a consequence.

The need to support various modifiers such as the number of repetitions and OR expression increases the overall complexity of the system, and many of the observed bugs are related to the usage of many of those features at the same time.

2.4.3 Effort required to integrate TESLA

The developer is required to augment the build workflow to accommodate the new analysis and instrumentation passes introduced by TESLA. Doing so is often cumbersome and likely to interfere with the build system that is in place. This is particularly true for complex build systems such as the one used in the FreeBSD kernel, which requires non-trivial expertise to modify. Incremental builds are not supported, making the possibility of leaving TESLA enabled during development quite unrealistic as the time required for a full build can be prohibitive.

This is a common problem among frameworks and tools based on instrumentation that relies on information shared by multiple translation units. An example of a tool that shares the same integration overhead as TESLA is SOAAP [4], a framework for application compartmentalization based on source-code annotations.

2.4.4 Performance considerations

TESLA imposes a considerable performance overhead on the instrumented programs, depending on how many assertions are enabled. The original evaluation shows a slowdown of more that 20% in macrobenchmarks such as the Sysbench OLTP benchmark when TESLA assertions are enabled in the FreeBSD kernel. Microbenchmarks show an even higher overhead, with TESLA making a system call up to 3 times slower than when no assertions are enabled.

At the core of the performance overhead lies a verification algorithm that is made sub-optimal by its need to support a vast range of scenarios (as discussed in §2.4.2). As it will be shown by the evaluation effort undertaken as part of this project (Chapter 5), performance degradation caused by TESLA is particularly noticeable when userspace programs are instrumented.

Chapter 3

Design

This chapter presents the design of ThinTESLA along with the core algorithms it relies on. When applicable, the design decisions of ThinTESLA are compared to the original design of TESLA, to identify where the two diverge and why.

3.1 Automaton representation

Each ThinTESLA assertion is represented by an *automaton*, which describes the sequence of events that, if not respected by the program, causes the assertion to fail. This section presents an overview of the automata and events representation, and introduces several definitions that will be referenced throughout this and later chapters.

3.1.1 Types of events

In ThinTESLA, an event is always a function call or a function return, and each event may specify further details such as the values of the parameters with which the function is called or the expected return value. A special event, the assertion site event, represents the program reaching the point in

which the assertion is defined. Additionally, the first and last event represent the *temporal bounds* of the assertion. An instance of the automaton is activated and disabled when the first and second temporal bounds are reached respectively. An instance that is disabled does not respond to events and is in an invalid state.

An event can be optional, meaning it is not an error if it does not occur, or can be conjuncted with other events through a logical OR operation. The relative order in an OR expression is not important.

ThinTESLA vs. TESLA

The decision of allowing only function invocations and returns as events is very different from the one taken by TESLA to support struct field assignments and Objective-C messages as well. However, focusing on function events makes it easier to specialize ThinTESLA and enable a range of optimizations that can rely on this design decision.

3.1.2 Static and non-static events

An event can be either static, e.g. a function call, or depend on runtime values, e.g. a function called with a specific value only known at runtime. For example:

- **call**`(foo)` is static.
- **call**`(foo(x))` is non-static as it is dependent on a runtime value `x` which is not known at compile time.
- **call**`(foo(`**ANY**`(int)) == y)` is also non-static as it is dependent on a runtime value `y`. The parameter is still static as it matches all possible values and does not need runtime information.

Runtime values need to become known at the assertion site. All non-static events must reference variables that are in scope when the assertion site is reached. For example, **call**`(foo(x))` matches any call to `foo` where the

parameter value equals the value of a variable `x` that must be known when the assertion site is reached.

An automaton is static if all its events are static. The temporal bounds of an automaton must be static, and the special assertion site event is always static. An automaton is non-static if one or more of its events are non-static.

A static event may not appear multiple times in the same assertion, nor may an non-static event whose arguments or return value will ultimately be the same at runtime.

ThinTESLA vs. TESLA

The original TESLA design supports the same event appearing multiple times in the same assertion, even though the implementation reveals some bugs that can be re-conducted to this feature. The decision of disallowing such scenarios is to streamline the verification algorithm and reduce the number and complexity of cases to take into account.

3.1.3 Event successors

Each event E has a list of potential *successors*, that is events to which the automaton, if currently in state E, can transition to if they occur. The successors of an event depend on the type of event:

- An non-optional, non OR-ed event in a sequence will have the following event in the sequence as the single successor.
- An event preceding a series of optional events will have all the optional events in the series and the first non-optional event as its successors.
- An event preceding a number of events OR-ed together will have all the OR-ed events as its successors.
- An optional or an OR-ed event will have the following optional or OR-ed events and the first non-optional, non OR-ed event as its successors.

- The last event (the second temporal bound) has no successors.

An event can only have successors that represent forward progress in the assertion sequence. Events in an OR block are not linked together explicitly (specifically, no event will have as successor a previous event in the same block, even though it is a valid transition), and are handled by the verification algorithm as a special case.

ThinTESLA vs. TESLA

TESLA stores all transitions explicitly, while ThinTESLA only keeps information on successors that represent forward progress in the overall sequence. As it will be shown in the evaluation, the data cache footprint is greatly reduced thanks to this optimization.

3.1.4 Conditional, strict and guideline automata

An automaton is said to be conditional if it does not need to be satisfied when the program does not reach the assertion site. A strict automaton requires the program to go through all of its events whenever an instance is activated at its first temporal bound. A conditional automaton's instance reaching the second temporal bound without having reached the assertion site does not cause the assertion to fail. A strict automaton's instance reaching the second temporal bound without having gone through all of its states in the exact order required by the automaton (including having reached the assertion site) causes the assertion to fail.

ThinTESLA introduces the concept of *guideline* automata. A guideline automaton's instance is disabled whenever a *final* event is reached, where a final event is any event which has the second temporal bound as one of its successors. This means an assertion is only checked up to once for every automaton's instance in the temporal bounds, and after it passes (or fails) it will not be checked again until the temporal bounds are entered again.

ThinTESLA vs. TESLA

The main reason many of the FreeBSD kernel assertions fail with the original TESLA is because the assertion path is reached multiple times within the temporal bounds causing a transition failure even though the temporal sequence is correct. Guideline mode addresses this issue and makes assertions more intuitive for the programmer.

3.1.5 Top-level OR-ed automata

In case of assertions that employ top-level OR-ed automata, i.e. assertions that have the form **TSEQUENCE**`(...) ||` **TSEQUENCE**`(...)`, ThinTESLA splits them into multiple automata. Each automaton is verified independently of the other ones, but a single failing automaton may not mean the assertion has failed. ThinTESLA causes the assertion to fail only if no automaton has succeeded when the second temporal bound is reached, when all OR-ed automata are re-grouped and checked at the same time.

ThinTESLA vs. TESLA

TESLA handles OR-ed automata as a single, large automaton, resulting in a very large state machine, which is one reason for performance degradation. Furthermore, TESLA's implementation of OR-ed automata is faulty and rarely produces correct results (§2.4).

3.1.6 Global and per-thread automata

An automaton can be either global, meaning it responds to events occurring in any thread of the instrumented program, or per-thread, meaning a different instance of it will be activated for each thread entering the temporal bounds.

Synchronization of a global automaton is left to the developer. Therefore, multiple events happening at the same time result in undefined behaviour.

The reasoning behind this rationale is that since temporal assertions rely on the total ordering between a sequence of events, the ordering of multiple events occurring in different threads during a very short time frame would not be consistent even with synchronization techniques such as locking. Relying on a particular sequence of events in multi-threaded environments without any synchronization is bad practice and compromises the program's correctness ([5, 6]). ThinTESLA introducing synchronization would exacerbate the problem and possibly make it more difficult to debug.

ThinTESLA vs. TESLA

TESLA guarantees synchronization for global automata. However, the implementation does not seem to respect this guarantee and leads to crashes in many multi-threaded scenarios which are not already properly synchronized. ThinTESLA eliminates the burden of unnecessary synchronization for global automata.

3.1.7 An example assertion

An example of a non-static automaton is represented by the following sequence of events, derived from Assertion 3.1:

- 0. **call**(`syscall`)
- 1 - 2. `firstFunction() == 0 ||` **call**(`secondFunction`)
- 3. **TESLA_ASSERTION_SITE**
- 4. **optional**(`lastFunction(x) == y`))
- 5. **returnfrom**(`syscall`)

In this example, the function `syscall` represents the temporal bounds of the assertion. All events are static except for event 4, which is non-static as it references two values only known at runtime (`x` and `y`). The events' properties and successors are shown in Table 3.1.

```
TESLA_WITHIN(syscall, TSEQUENCE(
    firstFunction() == 0 || call(secondFunction)
    TESLA_ASSERTION_SITE,
    optional(lastFunction(x) == y))))
```

Assertion 3.1: An example of a ThinTESLA assertion

Event	Final	Static	OR	Optional	Successors					
					0	1	2	3	4	5
0		✓				✓	✓			
1		✓	✓				✓	✓		
2		✓	✓					✓		
3	✓	✓							✓	✓
4	✓			✓						✓
5		✓								

Table 3.1: Events' properties and successors of assertion 3.1

3.2 Runtime assertion verification

As the program executes, ThinTESLA updates the automata's state and verifies if an assertion succeeds or fails. The verification logic is different depending on whether the assertion's automaton is static or non-static.

ThinTESLA vs. TESLA

TESLA's original design does not make a distinction between the two cases, and does not take the opportunity to specialize the runtime verification component based on the specific case. ThinTESLA reinforces the idea that specialization can result in significant performance gains.

3.2.1 Verification of static automata

Verifying whether a static automaton's sequence of events is respected by a running program is inexpensive relatively to when the automaton is non-static, as all required information is known a priori. Therefore, the assertion logic for static automata merely needs to keep track of the automata's state and to verify that the transitions taken by the program are allowed. This verification mechanism has constant space complexity ($O(1)$) and the total space required can be determined before run time.

An automaton is activated whenever its first temporal bound is reached. From that point onwards, until it is deactivated again, the automaton responds to events and potentially transitions from state to state. Whenever an event E occurs, ThinTESLA checks whether the automaton's current state lists E as a potential event to transition to. If the event is part of an OR expression, not only the successors but also the predecessors in the OR block are considered as candidates, as the relative order of events in an OR expression is not important. If a valid transition exists, the transition is taken and the automaton's state is updated. Otherwise, if E is not an event that immediately succeeds the last event that the automaton transitioned to, ThinTESLA performs the following operations depending on the automaton's mode:

- an automaton in conditional mode is reset to the first event, therefore discarding all transitions taken so far;

- an automaton in strict mode that receives an event which does not cause a transition causes the assertion to immediately fail.

The last event (the second temporal bound) resets and deactivates the automaton.

3.2.2 Verification of non-static automata

Non-static automata require runtime information in order for their correctness to be verified. A non static event may or may not interest the automaton depending on specific runtime values, which become known only when the assertion site is reached. Therefore, the automaton needs to store information on the temporal timeline of events, along with the values they took on. When the assertion site is reached, the timeline of those events that match the correct values needs to be inspected for correctness. There are several ways to store such temporal information and use it to verify the correctness of a temporal sequence, two of which have been investigated in this dissertation:

- Per-event temporal tagging.
- Per-automaton linearised temporal timeline.

These two techniques concern themselves with checking temporal sequences up to the assertion site event. Section 3.2.5 explains why this is enough for correctness purposes and shows how non-static automata can be reduced to static-like automata when the assertion site is reached.

3.2.3 Per-event temporal tagging

This technique employs a compact representation for the temporal history and embeds it in a per-event structure. The overall temporal timeline is then reconstructed at assertion time by collecting each event's history and putting them together.

Temporal tag

A *temporal tag* is a bit string which provides a compact temporal representation able to keep track of when events happen along with their relative ordering. Each bit in the bit string represents a temporal *epoch*, an arbitrary unit of time, where a value of 1 means the event was observed in that epoch.

The least significant bit represents the first epoch, while the most significant bit represents the maximum observable epoch. For example, the 8-bits string

00100010

can hold observations of up to 8 epochs and represents the fact that a certain event was observed both at epoch 2 and epoch 6.

Temporal store

Each event of an automaton holds a temporal store that stores information on when that event is observed and which values it took on in case of non-static events. The temporal store holds the following information:

- a single temporal tag for static events;
- ⟨*runtime values, temporal tag*⟩ tuples for non-static events.

Every automaton instance maintains a per-automaton temporal tag that is used to dictate the epochs the automaton spans during its lifetime. The automaton always keep track of the last event that occurred, regardless of whether it caused a transition or not. The temporal tag is then updated as events happen:

- At the first temporal bound, when the automaton is activated, the temporal tag is set to the first epoch.
- If an event occurs that is the successor of the last event that occurred, the temporal tag is left unchanged and the event's store is updated.
- If an event occurs that is the same as or precedes the last event that occurred, the temporal tag is advanced by one epoch and then the event's store is updated.
- If an event occurs that is after the last event that occurred but is not a direct successor of it, the event's store is updated and only then is the temporal tag advanced by one epoch.

```
1  input : an automaton instance that has reached the assertion site
2  output: true if this instance respects the assertion, false otherwise

4  function VerifyAutomatonTemporalTagging(automaton):
5   minEpoch = 0
6   maxEpoch = 0
7   for event in events_before_assertion(automaton)
8     tag = GetTemporalTag(event)
9     if tag == 0 then return false
10    if MostSignificantBitSet(tag) < maxEpoch then return false
11    maxEpoch = MostSignificantBitSet(tag)
12    if event_id(event) == 1 then minEpoch = maxEpoch
13    if (BitMaskBetween(minEpoch, maxEpoch) AND tag) != 0 then
          return false
14  for event in events_after_assertion(automaton)
15    tag = GetTemporalTag(event)
16    if MostSignificantBitSet(tag) >= minEpoch then return false
17  return true
```

Algorithm 3.1: Verification of a non-static automaton with per-event temporal tagging

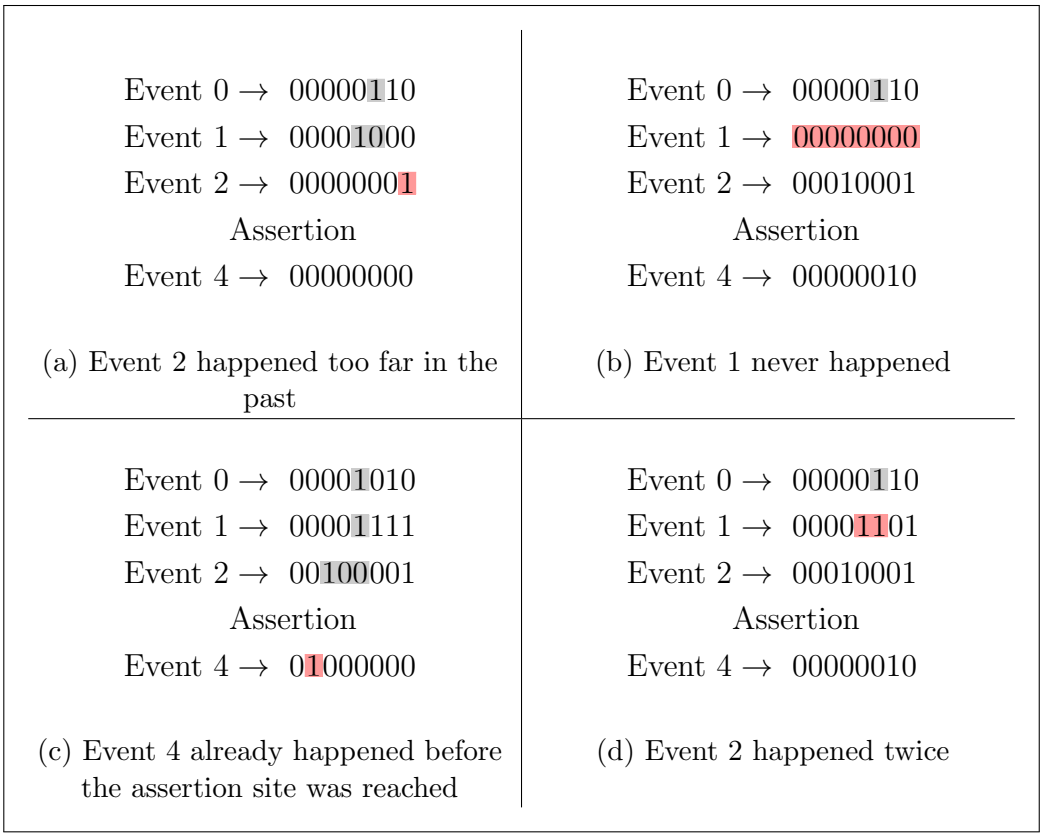
Verification algorithm

When the assertion site is reached, the verification algorithm has enough information to verify whether the automaton's instance respected the correct temporal sequence up to the assertion site. For each event in the sequence up to the assertion site event, the temporal tag stored in each event's store is retrieved. In case of non-static events, only the temporal tag associated to the relevant runtime values is retrieved. This effectively discards all other observations of non-static events with values that are not relevant, as their temporal history is not of importance. The algorithm used to update the temporal tag ensures that a total temporal order between events can be established by looking at every event's tag in succession. The verification algorithm (3.1) then verifies that the temporal order observed by the automaton's instance is as prescribed by the assertion specification.

The verification algorithm ensures that each required event before the assertion event:

1. has occurred (line 9);
2. has occurred after the previous event (line 10);
3. has only occurred once and has not occurred at any other time after the first event in the sequence occurred (line 13).

Additionally, it ensures that every event after the assertion event has not occurred prior to the assertion site being reached (line 16).



(a) Event 2 happened too far in the past

(b) Event 1 never happened

(c) Event 4 already happened before the assertion site was reached

(d) Event 2 happened twice

Figure 3.1: Example of failure scenarios

Events or sequences of events that happened before the last correct sequence are discarded in conditional mode, but are not allowed in strict mode. Furthermore, optional events are allowed not to happen, and OR-ed events are

allowed to occur in any order assuming that at least one happens. These trivial changes to the verification logic are not shown in Algorithm 3.1 for clarity.

An example of a temporal sequence that is correctly verified is the following:

Event 0 → 00000110
Event 1 → 00001010
Event 2 → 00001000
Assertion
Event 4 → 00000010

where the greyed bits represent the epochs spanned by the automaton's instance during its last iteration through the events.

Figure 3.1 shows a number of scenarios in which the verification algorithm rejects the temporal sequence and therefore causes the assertion to fail. The bits in red show which temporal epochs the algorithm uses to determine that the overall sequence is not correct.

3.2.4 Linearised temporal history

This technique works by maintaining a per-automaton linear temporal history of all events that occur within the temporal bounds, and then checking its validity at assertion time.

Temporal history

A temporal history is a linear sequence of event observations $\{o_1, o_2, o_3, ..., o_n\}$ such that

$$\forall(o_x, o_y) \wedge x \neq y \implies \begin{cases} o_x \text{ was observed before } o_y & \text{if } x < y \\ o_x \text{ was observed after } o_y & \text{if } x > y \end{cases}$$

Recording the history

An automaton's instance is responsible for recording the events that occur while it is active, and building a linear history that is kept in a per-automaton store. As an optimization, it is not necessary for the automaton to record static events, since all static events either result in a valid transition or cause the automaton's instance to reset to the first event, making it possible to clear the temporal history as well.

The only exception are static events that are part of an OR block with other non-static events. In that case, the automaton's instance would need to keep track of which path it took in order to go over the OR block, and static events' information is required for that reason.

Verification algorithm

The verification algorithm is simple, and involves traversing the history from the last observation to the first, checking that the temporal sequence defined by the assertion is respected. The algorithm is shown in 3.2 and goes through the following steps for each event before the assertion:

1. Static events are not checked (line 9-11).
2. If no observation is available for an event, the observed temporal sequence is incorrect (line 12).

```
1  input : an automaton instance that has reached the assertion site
2  output: true if this instance respects the assertion, false therwise

4  function VerifyAutomatonTemporalHistory(automaton):
5   history = GetLinearHistory(automaton)
6   observation = GetLastObservation(history)
7   event = last_event_before_assertion(automaton)
8   while exists(event)
9       if is_static(event) then
10          event = previous(event)
11          continue
12      if not exists(observation) then return false
13      if not matches(observation, event) then return false
14      observation = previous(observation)
15      event = previous(event)
16 return true
```

Algorithm 3.2: Verification of a non-static automaton with per-automaton linearised temporal history

3. If an observation exists but doesn't match the event that is expected at that point, the temporal sequence is incorrect (line 13).

It is not necessary for the algorithm to check that events after the assertion did not happen before the assertion site was reached, since the temporal history is cleared whenever the automaton's instance is reset. Any observation of an extraneous event (such as an event expected after the assertion site) would be caught by the check of line 13.

As before, the trivial changes to support OR blocks and optional events are not shown in Algorithm 3.2 for clarity.

3.2.5 Verifying non-static automata after the assertion

The algorithms introduced in sections 3.2.3 and 3.2.4 only check non-static automata once, at the time the assertion site is reached, but are not invoked when further events happen. This optimization is possible because when the

assertion site is reached, the automaton's instance has all the information it needs and can be considered in the same way as a static automaton's.

A non-static automaton reaching a non-static event before the assertion site has not enough information to determine whether the transition caused is a valid one, and therefore needs to store temporal information that will later be analysed. When an automaton reaches the assertion site, non-static events can be promoted to static by adding a check to their runtime values, which become known at that point. Therefore, after the assertion site is reached, the automaton's instance can always determine whether a transition is valid or not whenever an event is observed.

By promoting non-static automata to static, the effort required for runtime verification is greatly reduced and the overall work required to verify non-static automata is minimized to the bare minimum.

3.3 Static analysis for automata minimization

ThinTESLA introduces a static analysis component that aims at inferring static temporal properties at compile time in order to simplify automata. The static analysis algorithm builds a knowledge base of temporal properties of an automaton's events and then uses it to determine whether an event can be omitted entirely or simplified. It can also detect cases when an assertion is guaranteed to be always true or always false.

3.3.1 Preconditions and limitations

The static analysis component of ThinTESLA is not to be considered a direct replacement of the runtime verification library, but rather an aid to it. The analysis algorithm on which it is based upon brings along some requirements that the code to be analysed has to respect. In particular, given an assertion where the temporal bounds are represented by a function f:

- The call graph having the function f as the root node cannot contain any cycles. Therefore, direct or indirect recursion is not supported.
- Indirect or external function calls can exist in f or any of its children in the call graph, but make the algorithm unsound unless assumptions about the nature of the external calls are made. An external function is any function whose definition is not accessible.

The second requirement is best addressed by running the algorithm when all the code is visible to the compiler. This can be achieved by enabling link-time optimizations (LTO) so that all translation units are visible at the same time. Unfortunately, not all codebases support LTO out of the box.

The algorithm only deals with static events outside of OR blocks, but can extended to support non-static events and OR blocks as well. Devising temporal properties for non-static events goes beyond the scope of this project, but is left as a possible direction for future work.

3.3.2 Temporal properties

The first step of the analysis algorithm is to infer temporal properties from the program being analysed. These properties will be the building blocks that, when considered together, allow to determine whether an automaton can be simplified.

A *temporal property* usually refers to an automaton's event, but can express relationships between different events as well. For example, the property $\mathit{DefinitelyHappens}(\mathit{bound}, \mathit{event})$ represents the fact that if function *bound* is called, function *event* will be definitely called before *bound* returns.

Many properties rely on certainty, as this allows for the algorithm to be sound (although not complete). That is, they express facts that are always true, no matter the runtime behaviour of the program. Some properties express possibility, but do not concern themselves with the conditions that make the property false or true.

The temporal properties that the algorithm can infer are shown in Figure 3.2.

DefinitelyHappens(bound, f)
Within the temporal bounds of function *bound*, function f is called at least once.

MayHappen(bound, f)
Within the temporal bounds of function *bound*, it is possible that function f may be called.

DefinitelyNeverHappens(bound, f)
Within the temporal bounds of function *bound*, it is not possible for function f to be called.

AtMostHappensOnce(bound, f)
Within the temporal bounds of function *bound*, it is not possible that function f may be called more than once.

DefinitelyHappensAfter(bound, f, g)
Within the temporal bounds of function *bound*, it is not possible that a call to g may happen before a call to f.

AreSequenced(bound, f, g, events)
Within the temporal bounds of function *bound*, and given a set of functions *events*, it is not possible for a call to any function in *events* to happen between a call to f and a call to g. Additionally, it is always the case that *DefinitelyHappensAfter(bound, f, g)* is true.

Figure 3.2: Temporal properties used by the static analysis algorithm

3.3.3 Automata optimization

By building a knowledge base of the temporal properties in Figure 3.2 for a specific automaton's temporal bounds and events, it is possible to determine facts that can lead to the simplification of the automaton. Only static events that are not part of an OR block are considered.

Every non-optional event should have the possibility to happen

For every event E in an automaton A, it should not be possible that E definitely does not occur, unless it is an optional event:

$$\begin{cases} DefinitelyNeverHappens(bounds(A), E) \\ \neg IsOptional(E) \end{cases} \implies A \text{ is unsatisfiable}$$

Consecutive events can be simplified

For every pair of events (E_1, E_2) in A where E_2 is a direct successor of E_1,

$$\begin{cases} AreSequenced(bounds(A), E_1, E_2, O_{E_{1,2}})) \\ DefinitelyHappens(bounds(A), E_1) \\ AtMostHappensOnce(bounds(A), E_1) \end{cases} \implies E_1 \text{ can be ignored}$$

where $O_{E_{1,2}}$ is the set of all events in A except E_1 and E_2.

If every event E in A can be ignored according to the above definition, then the assertion is always true:

$\forall E \in events(A) \wedge E$ can be ignored $\implies A$ is always satifised

Relative order should be respected

Finally, for every pair of non-optional events (E_1, E_2) in A where E_2 is either a direct or indirect successor of E_1, it should not be possible that E_1 can only happen after E_2:

$DefinitelyHappensAfter(bounds(A), E_2, E_1) \implies A$ is unsatisfiable

Chapter 4

Implementation

ThinTESLA's implementation aims to be as low-overhead as possible by taking into account micro-architectural aspects and by utilizing specialized instructions where possible for the most expensive components. ThinTESLA, just like TESLA, is divided into two main components: the instrumentation library, and the runtime library *libthintesla*. The complete implementation is available open-source as a Git repository [1].

4.1 Instrumentation library

The instrumentation library, an LLVM 6.0 module, has been re-implemented from scratch and made slimmer by taking advantage of the fact that only function calls are instrumented.

4.1.1 Automaton and event representation

The instrumentation library and *libthintesla* agree on a common structure to hold automata and event's static information. This includes for example the number of events of an automaton, the successors of an event, whether the

[1] `https://github.com/banex19/thin-tesla`

automaton has thread-local semantics, whether the event is optional, and so on. All of this information is available at compile-time and an instance of these structures is created as a global variable for each event and for each automaton.

Many of these structures also contains fields reserved for use by the runtime library and that will be modified during execution. For example, an automaton's structure contains a pointer to the last event observed, which will be updated as the program executes.

4.1.2 Function instrumentation

Functions are either instrumented at entry or at every exit, based on the semantics of the event that needs instrumentation (e.g. if it requires knowledge of the return value). The instrumentation consists in a call to a specific function of *libthintesla* depending on whether the event is static or not. Static and partially-static events result in more instrumentation as constant values are matched statically without going through the runtime library.

Whenever a non-static event references function arguments or return values, the values thereof are passed to *libthintesla* as a pointer to a stack-allocated array that contains the values referenced by the event. Given a function with arguments $\{a_1, a_2, ..., a_n\}$ and return value r, the instrumentation library builds an array on the stack that contains the values $\{a_x, a_y, ..., r\}$ where $\{x, y, ...\}$ are the indices of the non-constant arguments referenced by the assertion. The return value r is omitted from the array in case the event doesn't need it.

For example, considering the function and the event

```
int open_file(int fileDesc, void* dir, int openForWrite)
```

call`(open_file(fd,` **ANY**`(ptr), readWrite)) == 0`

the instrumentation code will build an array of two elements to store the first and third arguments (`fileDesc` and `openForWrite`) as they are the

only non-static arguments referenced by the event. The second argument is not matched as it is free to take on any value, and the return value is statically matched to be equal to `0`.

All arguments and return values are extended to the largest type supported by the underlying architecture's general registers, in order to provide alignment and consistency guarantees. Since C can only pass primitive types or pointers as function arguments, which are not larger than the architecture's register size, this is a safe operation. Non-constant return values are supported in ThinTESLA, while they were not in TESLA.

4.1.3 Build system integration

ThinTESLA provides a new tool that helps the integration of the framework in existing codebases, regardless of the build system in use. The tool is a Clang plugin and analyses each source file of the codebase in search of TESLA assertions. The code that parses assertions into Protobuf representations is the same used by TESLA and is the only piece of code that is shared by the two frameworks for compatibility reasons. It maintains a cache with information on source files, their modification timestamps, the assertions found in each file, and the names of the functions defined in each file. Whenever a change is detected, the tool only processes the file that have changed, and if an assertion has been modified, added or removed, the affected source files' timestamps are updated to signal the build system that they need to be rebuilt.

This is a summary-based approach loosely based on the concept put forward by ThinLTO [7], where only the strictly-necessary information to perform a specific operation is kept and nothing more. With the addition of this tool, ThinTESLA aims to solve a problem which is shared by many whole-program instrumentation tools such as TESLA or SOAAP [4], which otherwise require significant effort to be adopted (see §2.4.3).

4.2 The runtime library *libthintesla*

The runtime library is written in C and is linked to the instrumented program in order to produce the final executable. It implements both algorithms 3.1 and 3.2, and a number of optimizations configurable at compile time. It is also responsible of managing per-thread automata.

4.2.1 Per-thread support

Unlike TESLA, ThinTESLA does not employ thread-local storage in order to support per-thread semantics. Instead, a single global variable is created for each automaton and *libthintesla* is responsible for cloning it whenever a new thread enters the temporal bounds. In this sense, the global variable can be considered a "template" automaton that is used as a global reference to a particular automaton but is cloned to a specific instance for each thread and that instance is retrieved every time it needs to be modified.

The implementation is different for userspace, where the number of user threads is assumed to be relatively small utilized, and the FreeBSD kernel, where thousands of kernel threads can be active within an assertion's temporal bounds at any time. The first relies on a lock-free implementation, while the second extends the kernel thread structure to remove the need of synchronization.

Userspace implementation

Each automaton structure contains a thread key and a pointer to another automaton structure, therefore forming a linked list of the same automaton's structure. The automaton's global variable represents the *template* structure and is always the first in the chain. Whenever an automaton instance is activated for a new thread, *libthintesla* clones the template instance into a new automaton structure, and tries to append it to the chain with a compare-and-swap operation until it succeeds. The new instance contains an unique

thread key obtained through the *pthreads* API, but that is easily adaptable to different thread (and even fiber) libraries.

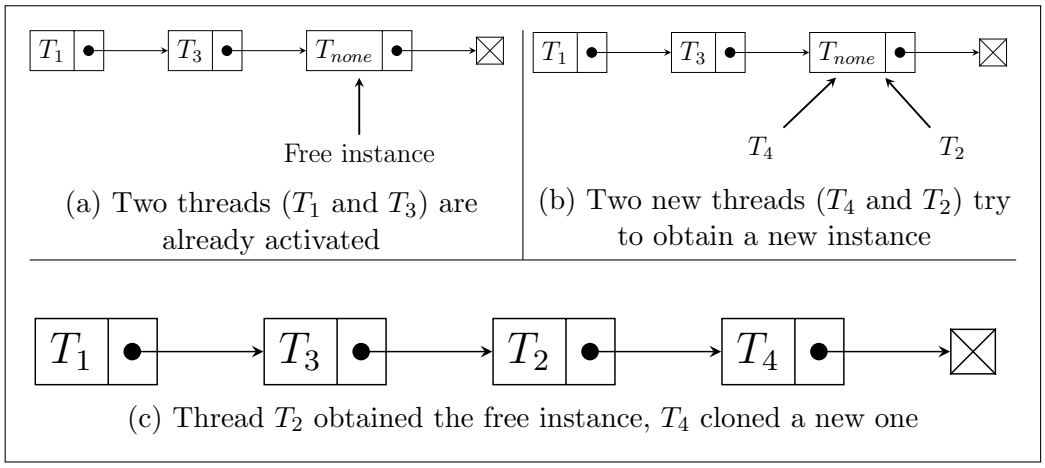


Figure 4.1: Creation and assignment of per-thread automaton instances

When the automaton's instance needs to be retrieved, for example for updating its internal state, the chain is traversed in search of the instance with the correct thread key. When the automaton's instance is disabled, the thread instance is not removed from the chain but only reset, along with the thread key field. This optimization enables new activations to avoid creating new instances and instead just find the first available instance in the chain that is not currently claimed by any thread. A thread that wants to gain possession of an instance in the chain will do so by comparing-and-swapping its thread key to the empty thread key field of the instance. If the operation fails because another thread was faster, the unsuccessful thread retries until it succeeds or is forced to create a new instance because all instances are taken up.

Figure 4.1 shows an example scenario where two threads are activated and there is a free instance in the chain. Two threads enter the temporal bounds at the same time and when trying to obtain a new instance, one manages to claim the free instance of the chain, while the other resorts to cloning a new one and appending it to the chain.

Since all threads traverse the chain from the beginning until they find an

empty instance to possess, the active instances will be close to the beginning of the chain most of the time. The worst case complexity to find a thread's instance is $O(T)$ where T is the maximum number of threads active at any given time in the same temporal bounds.

FreeBSD kernel implementation

A lock-free synchronization mechanism for a linked list is of little value in the kernel where traversing a long chain of instances can easily result in extremely poor performance. Instead, the kernel thread structure (`struct thread`) has been extended to include a pointer to an array of automata. The array contains a cloned automaton instance of each different automaton that can be activated in the kernel (i.e. one for each ThinTESLA assertion in the kernel).

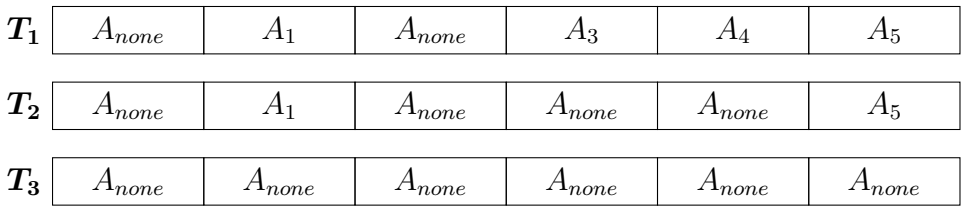


Figure 4.2: Automata instances array per kernel thread

The array is populated and individual instances are instantiated only if needed, in order to save time and space. Whenever an automaton is activated, the array is created if not existent, and the correct instance is retrieved (and initialized if it is retrieved for the first time). Since the number of different automata is known at compile-time, each automaton is assigned a zero-based, progressive ID that allows the correct instance to be retrieved in constant $O(1)$ time.

Figure 4.2 shows an example where 6 assertions have been added to the kernel. A kernel thread (T_1) has encountered automata 1, 3, 4 and 5 so far. Another kernel thread (T_2) has encountered only automata 1 and 5. Yet another kernel thread (T_3) has not encountered any automata yet, and therefore all instances are still not initialized.

As an optimization, since kernel threads are frequently reused, the space used for ThinTESLA per-thread instances is not freed at thread destruction time and re-allocated when needed again, but is instead always kept ready to be reused whenever the thread is reincarnated. This optimization can be turned off in low-memory environments.

4.2.2 Temporal store

Both runtime verification algorithms require some kind of storage in order to reconstruct the temporal history. Static automata are guaranteed to be $O(1)$ in terms of space complexity, while non-static automata need to deal with dynamic memory allocation.

Memory allocation

ThinTESLA reserves some static space (of configurable size) that will be used for all memory allocation requests made by *libthintesla*. This static space is allocated until it runs out, and subsequent memory requests will be re-directed to the heap allocator. The static memory allocator is synchronized with lock-free semantics and does not support memory being freed. Therefore, memory that is allocated from the static storage will always be retained by the requester. This is not a concern since storage structures get reused throughout execution and is in fact an optimization.

Per-event temporal tagging

Algorithm 3.1 relies on per-event storage of temporal tags, and in case of non-static events, it needs to store the runtime values that the event references as well. The current implementation uses 64-bit strings as temporal tags.

Every automaton's instance contains a pointer to a store structure for each event. For non-static events, the store is implemented with a hash table where the key is the hash of the runtime values and the value is the associated

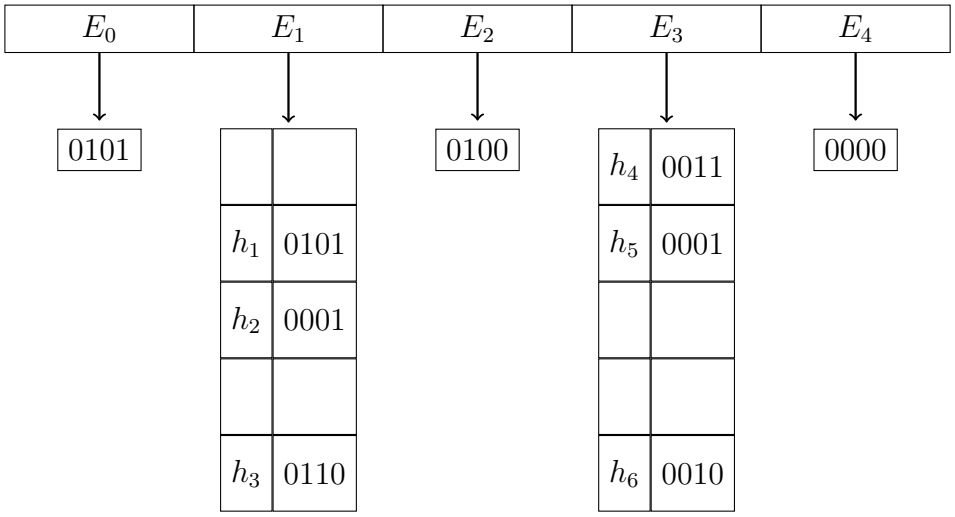


Figure 4.3: Per-event temporal store

temporal tag. In case on static events, the pointer to the store itself is reinterpreted as a bit string and used as a temporal tag.

Figure 4.3 shows an example automaton with 5 events. The first, third, and last event are static and require a single temporal tag, while the second and second-to-last events are non-static and employ hash tables to store temporal tags. 4-bits temporal tags are shown for convenience and the values h_n represent tuples of the form $\langle$*data, hash(data)*$\rangle$.

Per-automaton linearised temporal history

Algorithm 3.2 requires a linear representation of the temporal history to be kept in a per-automaton store.

This is implemented as a dynamic array which is populated with event observations as they happen during execution. Whenever the history is reset (e.g. when the automaton's state is reset to the first event or when the automaton is disabled), the array does not need to be zeroed out or shrunk down. Instead, its size is simply reset to zero to signal that it has become empty.

4.2.3 Verification algorithms

The verification algorithms' implementations need to be fast performance-wise, as they are one of the core components that is run often.

Temporal tagging

Since most operations of Algorithm 3.1 operate on temporal tags, which are bit strings, operations such as `MostSignificantBitSet` are implemented with the help of architecture-specific specialized instructions. An example is the use of the `LZCNT` (count leading zeroes) instruction for the x86-64 architecture.

The major concern for this algorithm's implementation is cache utilization. Since each event's temporal tag needs to be retrieved from each event's temporal store, which is a hash table, the memory access patterns are not sequential and are instead scattered. However, this is a problem only for non-static events, as retrieving the temporal tag of static events requires a single memory access.

Linearised temporal history

Algorithm 3.2's implementation is straightforward, as most operations are just memory comparisons. Cache utilization is improved with this algorithm, since temporal observations are stored sequentially and multiple observations easily fit in a single cache line.

However, a high number of observations can result in sub-optimal performance as all of them need to be checked for a possible match individually, even if they refer to the same event and the same runtime values. This is avoided with per-event temporal tagging as multiple identical observations are coalesced into the same temporal tag.

4.2.4 Late initialization optimization

ThinTESLA borrows the same technique used by TESLA to avoid the overhead of initializing automata instances which will not reach the assertion site. Instead of initializing instances at the first temporal bound, automata are instead activated whenever any *starting event* is reached for the first time. A *starting event* is an event which is a direct successor of the first temporal bound. This makes it possible to completely ignore events that would not cause any transition and would not be of any interest to the automaton in the first place.

4.2.5 Optimizing the second temporal bound

ThinTESLA employs a highly-effective optimization in the kernel which allows it to remove the second temporal bound if it's shared by all assertions. As automata are enabled, verified, and disabled, ThinTESLA keeps a per-thread counter of how many assertions have reached the assertion site but have not been verified to be correct yet. As the second temporal bound is reached, it is enough to check the counter to know whether any assertion has failed, instead of calling a transition event for each of them. Unfortunately, this optimization is not available for OR-ed automata, which still need to be re-grouped at the second temporal bound.

4.2.6 Sacrificing correctness for performance

ThinTESLA introduces a *non-precise* mode which can optionally be enabled by the developer. This mode allows to trade the correctness guarantees in exchange of improved runtime performance. Specifically, when non-static events' data is retrieved from the temporal store (§4.2.2), both the hash and the full data are compared in order to guarantee a perfect match.

In non-precise mode, only the hash is stored and compared. This can cause both false positive and false negatives, and therefore the result of automata

verification cannot be considered always correct. By using a good hash function, the risk of incurring in false positives and/or negatives is greatly reduced, especially when the number of unique sets of data (i.e. the number of unique invocations of an event's function in terms of arguments' values) is small.

Chapter 5

Evaluation

ThinTESLA aims to address the performance implications of TESLA so that it may be used both in testing and production environments without rendering them unnecessarily slow. For example, enabling TESLA with 80 assertions in the FreeBSD kernel can result in more than 40% overhead over the non-instrumented kernel baseline. The evaluation of ThinTESLA therefore focuses on its performance impact on a range of workloads and the performance improvements over TESLA, and reasons on the underlying algorithmic and micro-architectural aspects that contribute to the performance gain. Furthermore, this evaluation measures the usefulness of specific optimizations and the impact of slightly different semantics between the two frameworks for different workloads.

This evaluation addresses the following research questions:

1. For which workloads does one framework perform better than the other and what are the root causes?
2. Are the performance improvements mainly a result of a better algorithm or of better hardware utilization (e.g. cache behaviour)?
3. As the work of the underlying benchmark increases, does the overhead of one or both frameworks become statistically insignificant? If so, how do they compare?

4. How do the two frameworks scale with respect to the number of assertions?
5. What is the effort required to integrate the two frameworks in existing codebases?
6. What is the impact of specific optimizations on the observed overhead?
7. How do the two frameworks compare against commonly-used debugging frameworks such as the FreeBSD kernel's WITNESS and INVARIANTS utilities?

5.1 Methodology and statistical procedures

Run-time overhead is analysed as the comparison between the original non-instrumented version of a program and the version instrumented by TESLA or ThinTESLA, unless stated otherwise. Graphs always show the median of multiple runs, where the number thereof is specified for each benchmark and indicated in the graph, and error bars show the interquartile range (IQR) of the results data set. Additionally, at least two warm-up runs are discarded. When statistical comparisons are performed, non-parametric (i.e. distribution-free) statistical tests are used, instead of those that rely on the assumption of a normal distribution. This is because performance measurements in computer systems cannot be safely assumed to be normally distributed [9] and usually have a long tail.

5.2 Benchmark environment and hardware

All benchmarks have been performed on a machine equipped with an Intel® Xeon® E5-1620 (Sandy Bridge) CPU with a frequency of 3.60GHz, a 512GB SSD for storage and 64GB of DDR3 RAM. The CPU has 4 physical cores (8 logical), 32KB of instruction and 32KB of data L1 cache per physical core, 256KB of shared L2 cache per physical core, and 10MB of shared L3 cache

for the whole processor. This machine has the same configuration as the one used in the original TESLA evaluation. The software environment is a FreeBSD 11.1-RELEASE installation. When custom kernels are used in the benchmarks, they are all based on the FreeBSD 11.1-RELEASE kernel.

5.3 Run-time overhead

Each test program instrumented with ThinTESLA is compared against two baselines: the non-instrumented program, and the program instrumented using the original TESLA. Comparisons are performed on a series of micro and macrobenchmarks that assess the behaviour of the two frameworks in user and kernel space.

5.3.1 Userspace microbenchmarks

A comparison of TESLA and ThinTESLA on a series of userspace microbenchmarks that aim to gauge the overhead of both libraries shows the latter performing up to 20 times better than the former. Performance improvements mainly depend on the assertions under test and whether global or per-thread automata are used. ThinTESLA is configured to run in guideline mode and with the linearised temporal history verification algorithm unless stated otherwise.

Pure overhead

The benchmarks in this section aim to measure the *pure overhead* of TESLA and ThinTESLA when a single assertion is enabled in a series of microbenchmarks which do not perform any actual CPU work. Pure overhead is defined as the total time spent in the two frameworks, which by definition is zero when the non-instrumented program is considered. Therefore, benchmarks are constructed in such a way that the runtime of the baseline program is

very close to zero. Because of that, graphs show the execution time of the programs when instrumented by both frameworks as measured by a wall clock as opposed to the overhead in percentage.

Benchmark	events	static events	non-static events	optional events	OR-ed events	OR-ed automata	events called before	events called after
A	4	3	1	0	0	0	1	2
B	4	3	1	1	0	0	1	3
C	6	5	1	1	4	0	1	3
D	6	5	1	1	4	0	3	3
E	6	5	1	1	4	0	2	2
F	6	5	1	1	4	0	3	2
G	4	2	2	0	2	0	4	1
H	5	2	3	0	4	0	4	2
I	6	6	0	0	0	2	1	2
J	6	5	1	0	0	2	1	1
K	9	8	1	0	0	3	1	1
L	9	9	0	0	2	3	2	1
M	9	9	0	1	2	3	2	1

Table 5.1: Microbenchmarks and assertions' characteristics

```
void examplecall(int i) {
    callOne(1);             // Event called before.

    TESLA_WITHIN(examplecall, TSEQUENCE(
          call(callOne),
          TESLA_ASSERTION_SITE,
          call(callArgs(i, ANY(ptr), 3)),
          call(callTwo)));

    callArgs(i, NULL, 3);   // Event called after.
    callTwo(2);             // Event called after.
}
```

Assertion 5.1: Benchmark A's temporal assertion, with an indication of the events called before and after by the benchmark framework

A skeleton program is responsible of running a function 10 million times, where that function represents the temporal bounds of the assertion being benchmarked. Inside the function there are calls to empty functions that constitute the events checked by the assertion. Table 5.1 shows what the characteristics of each benchmark's assertion are, namely:

- The total number of events (excluding temporal bounds).
- The number of static and non-static events.
- The number of optional events and events in OR blocks.
- The number of top-level OR-ed automata, e.g. an expression such as **TSEQUENCE**`(...) ||` **TSEQUENCE**`(...)`.
- The number of events that are called by the benchmark before and after the assertion site is reached.

Assertion 5.1 shows the assertion used for benchmark A as an example. Many combinations that include OR-ed automata (e.g. OR-ed automata with many non-static events) could not be compared, as TESLA behaves incorrectly by determining the assertion has failed when in reality has succeeded (see §2.4).

The top graph of Figure 5.1 shows the pure overhead of the two frameworks when global semantics are used. TESLA's overhead varies widely depending on the underlying assertion semantics, while ThinTESLA's behaviour shows little variance across the entire set of benchmarks. Specifically, TESLA seems very sensitive to benchmarks that include optional and/or events in OR blocks, and even more so when non-static events are referenced. Per-benchmark variance is extremely small due to a carefully controlled testing environment.

The bottom graph of Figure 5.1 shows the same set of benchmarks run with per-thread semantics and four threads executing the function representing the temporal bounds in a loop. ThinTESLA still performs considerably better than TESLA but employing per-thread semantics does incur a performance penalty as opposed to the non-synchronized, global semantics. TESLA's per-thread performance is very similar to the global case, as the per-thread

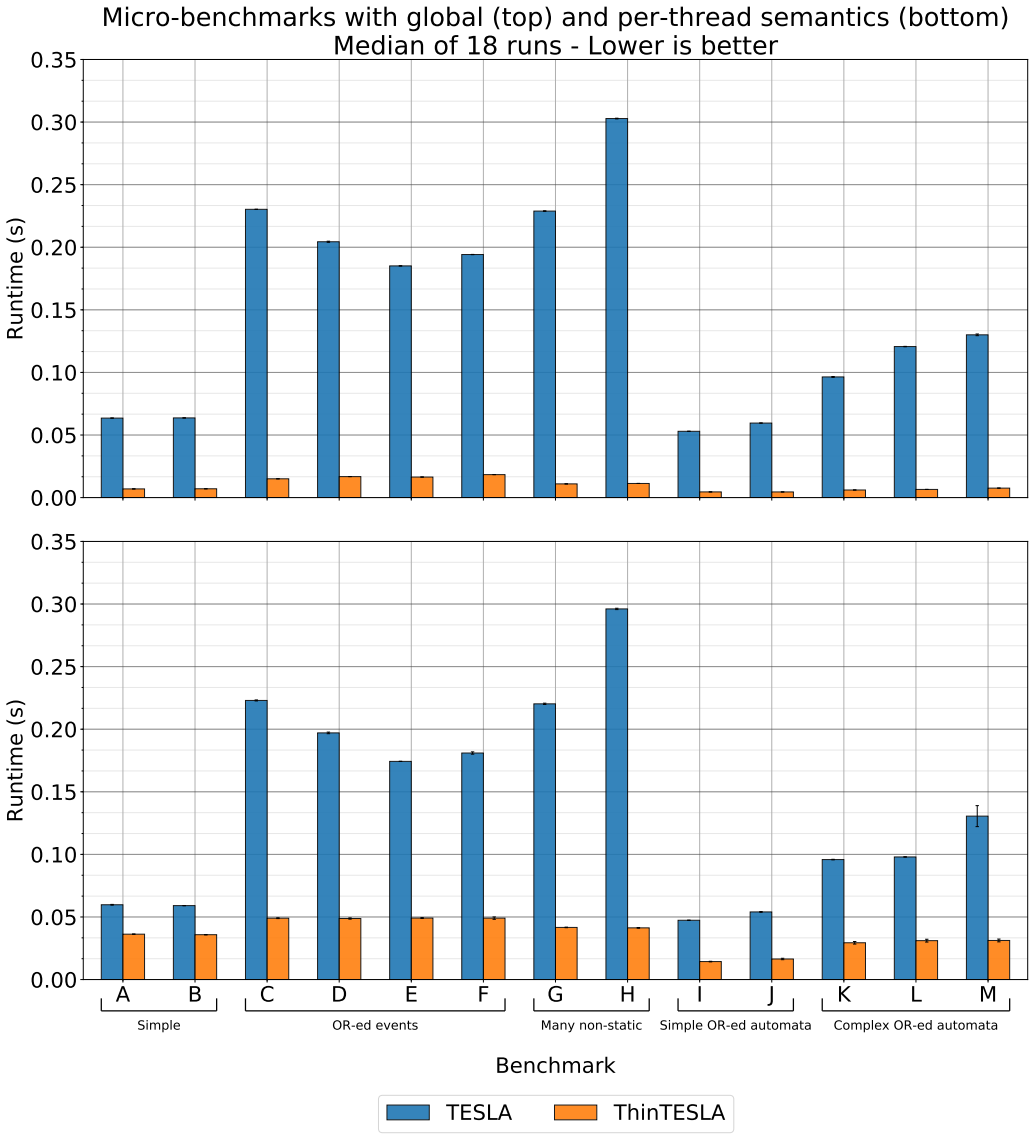


Figure 5.1: TESLA's and ThinTESLA's pure overhead as observed in a series of microbenchmarks with a single global or per-thread assertion

`pthreads` storage implementation is used to provide thread-local semantics. ThinTESLA's implementation aims to support arbitrary thread libraries and even lightweight thread (fiber) libraries, at some cost to performance.

Table 5.2 shows the median performance and the variance (IQR) observed for the two frameworks across the entire set of benchmarks. On average, Thin-

TESLA performs more than 16 times better when global semantics are used and more than 2.5 times better when per-thread semantics and four threads are employed. Similarly, variance is also drastically reduced by ThinTESLA, as TESLA's variance ranges from being more than 15.5 and 6.5 times worse in the two cases.

Implementation	**Global benchmarks**		**Per-thread benchmarks**	
	Median	**Variance (IQR)**	**Median**	**Variance (IQR)**
TESLA	0.130	0.141	0.131	0.137
ThinTESLA	0.008	0.008	0.036	0.018
ThinTESLA improvement	**16.160x**	**15.568x**	**2.600x**	**6.764x**

Table 5.2: TESLA and ThinTESLA's median performance and variance across all microbenchmarks

Change in overhead as the underlying work is increased

Figure 5.1 shows the pure overhead of TESLA and ThinTESLA, since the underlying program is not performing any actual work. But it is interesting to determine at which point the overhead of the two frameworks becomes statistically insignificant as the instrumented program increases the amount of CPU work it is performing. We consider two workloads: the first is ALU intensive, performing integer additions, while the second is memory intensive, writing large amounts of data to memory.

The benchmark framework has been modified so that, within the temporal bounds, the CPU is exercised by computing the sum $2+4+6+...+N$ for the first workload, and by writing N times to a large array for the second. Figures 5.2 and 5.3 show the overhead as the ratio between the execution time of the instrumented program and the execution time of the non-instrumented one. The program chosen for this test is benchmark F with global semantics. The value N is increased until the overhead of one of the two frameworks becomes statistically insignificant as determined by the two-sample

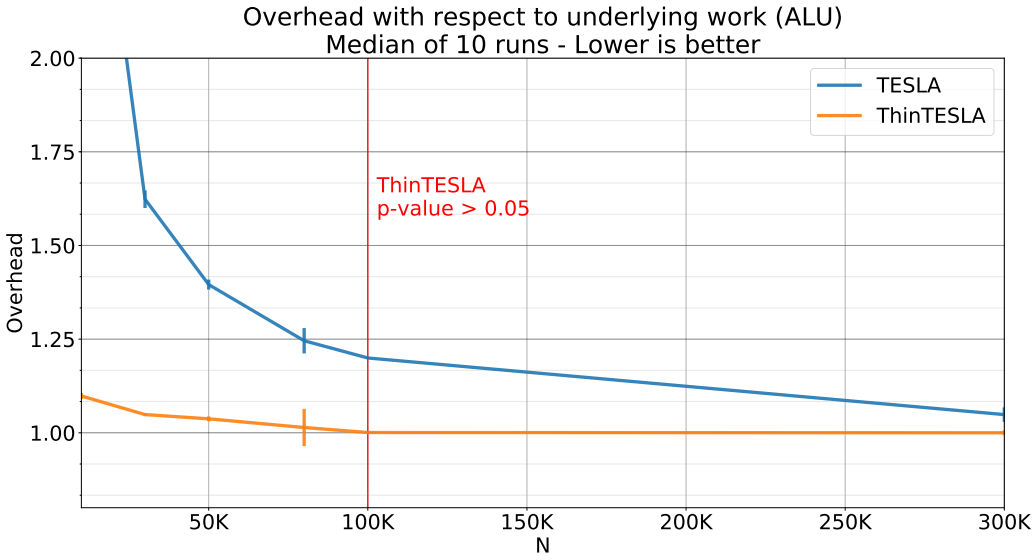


Figure 5.2: TESLA's and ThinTESLA's overhead as the underlying program (benchmark F) performs more and more ALU work

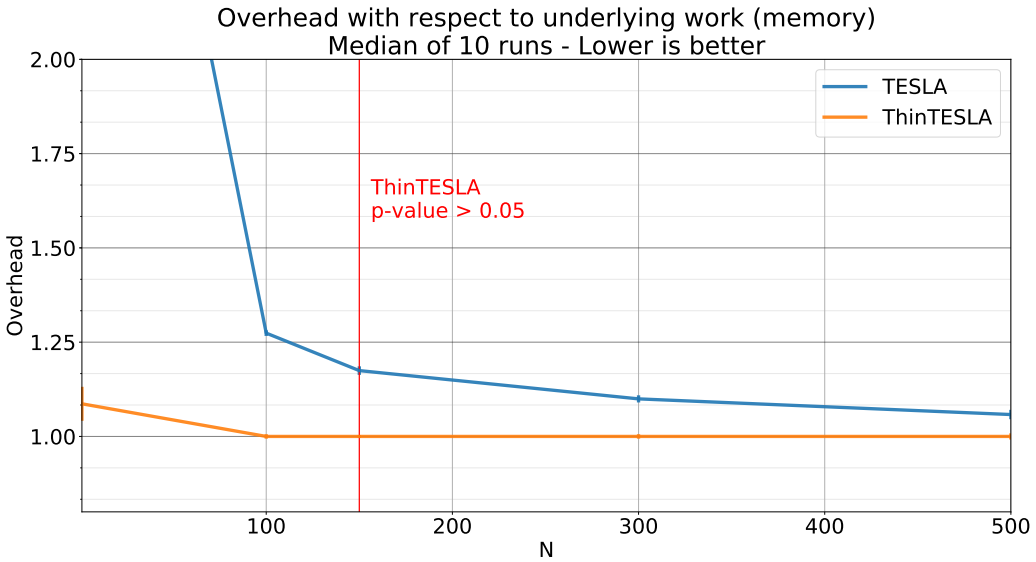


Figure 5.3: TESLA's and ThinTESLA's overhead as the underlying program (benchmark F) performs more and more memory stores

Kolmogorov-Smirnov equivalence test [10].

As the graphs show, when the overhead of ThinTESLA becomes statistically non-significant (i.e. $p\text{-}value > 0.05$), the observed overhead of TESLA is still

more than 19% (for the ALU-intensive workload) and 17% (for the memory-intensive one) over the non-instrumented baseline.

Scalability with respect to the number of assertions

ThinTESLA has been designed with scalability in mind, so that the runtime required to process N assertions should be roughly linear with respect to N. Figure 5.4 shows the running time of benchmark F instrumented by the two frameworks as the same assertion is duplicated N times, with N starting at 1 and becoming increasingly larger. Both axes are on a logarithmic scale. Even though the underlying assertion specification is the same, N individual instances are created as they are treated as different assertions.

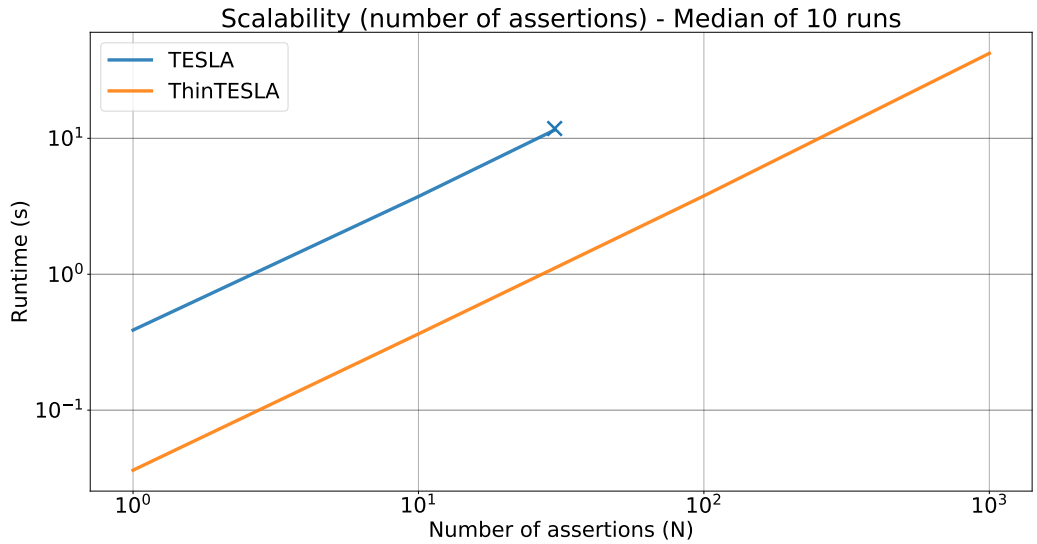


Figure 5.4: TESLA's and ThinTESLA's runtime (benchmark F) as the number of assertions increases. TESLA crashes when 50 or more assertions are enabled

TESLA scales linearly with small Ns, but trying to run TESLA with 50 or more assertions results in a segmentation fault. This seems to be a problem specific to userspace TESLA as the version for the kernel can sustain up to 100 assertions without crashing. ThinTESLA instead scales linearly with respect to N even with 1,000 assertions enabled at the same time.

Scalability is important when large systems needs to be instrumented, and especially so when many assertions share the same temporal bounds (as in the benchmark). For example, the FreeBSD kernel's assertions all have the `syscall` function as the temporal bounds, which means automata are potentially activated for every system call. As it will be shown, optimizations such as late initialization help reduce the impact and generally improve scalability.

Verification algorithms comparison and correctness trade-offs

Figure 5.5 shows the benchmarks of Table 5.1 run with the two verification algorithms 3.1 and 3.2. The former is set to guarantee correctness while the second employs the correctness/performance trade-offs detailed in §4.2.6.

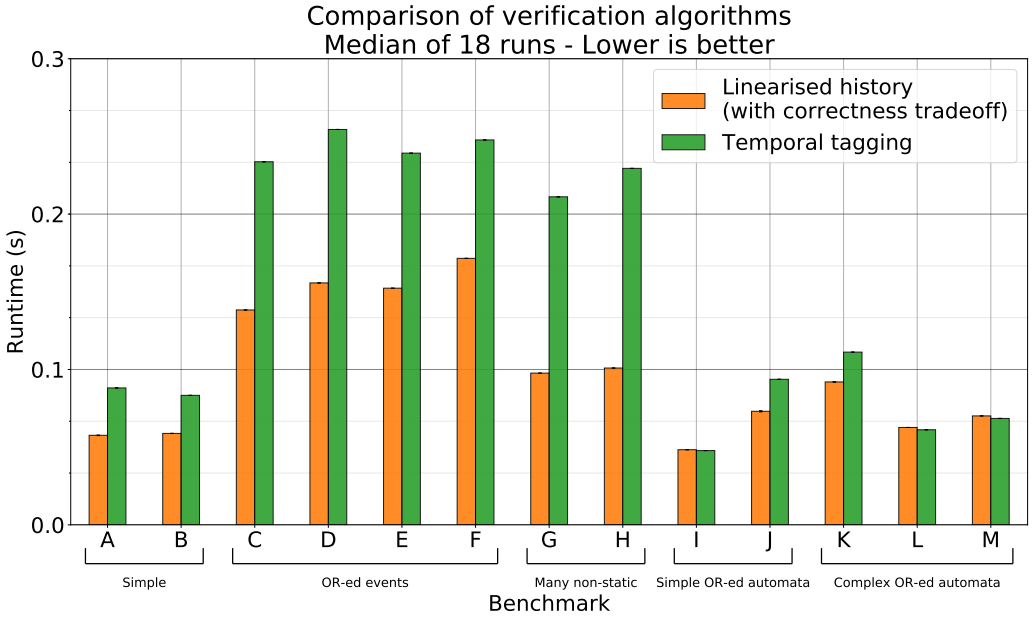


Figure 5.5: ThinTESLA's pure overhead with the two verification algorithms (one with correctness trade-offs)

The verification algorithm based on a linearised temporal history generally performs better, also thanks to the relaxed correctness semantics that allow the temporal history vector to be more compact as only 64-bit hashes need to be stored. Furthermore, as explained in §3.2.4, recording the temporal

history in a linearised fashion does not require static events' observations to be stored, while temporal tagging verification needs to process every event. The benchmarks in which there is no difference between the two algorithms are those with all-static events, where the verification algorithm is not necessary. Surprisingly, as the next section on cache behaviour shows, there are no drastic improvements on data cache utilization when linearised history is enabled, but instruction cache footprint greatly benefits from a more compact algorithm implementation.

Micro-architectural behaviour

Figure 5.6 shows TESLA's and ThinTESLA's cache behaviour for both L1 data and instruction cache as measured by the `pmcstat` tool during the execution of the benchmarks, which have been modified to run indefinitely in a loop. For ThinTESLA, the cache footprint of both verification algorithms is measured. The graph shows the median number of data and instruction cache misses incurred every second as exposed by hardware counters, after the program has entered a steady state in terms of cache utilization. This method of measurement has been adopted to minimize the probe effect, which has been determined to be already small in hardware event counting mode. As with every observation of micro-architectural aspects, however, the variance is expected to be large, as any unrelated event in the system can indirectly interfere with the processor's state.

TESLA's footprint on the data cache is much more significant than ThinTESLA, and especially so in the case of the three benchmarks that have 3 OR-ed automata. In case of multiple automata in the same assertion, TESLA maintains a full table of possible transitions which quickly becomes large as the number of automata increases. Walking the table at runtime is the main reason for poor cache utilization: in fact, the number of data cache misses reaches 44 million per second in benchmark M. ThinTESLA's automaton representation does not explicitly store every transition, and instead works them out at runtime, keeping the number of data cache misses per second

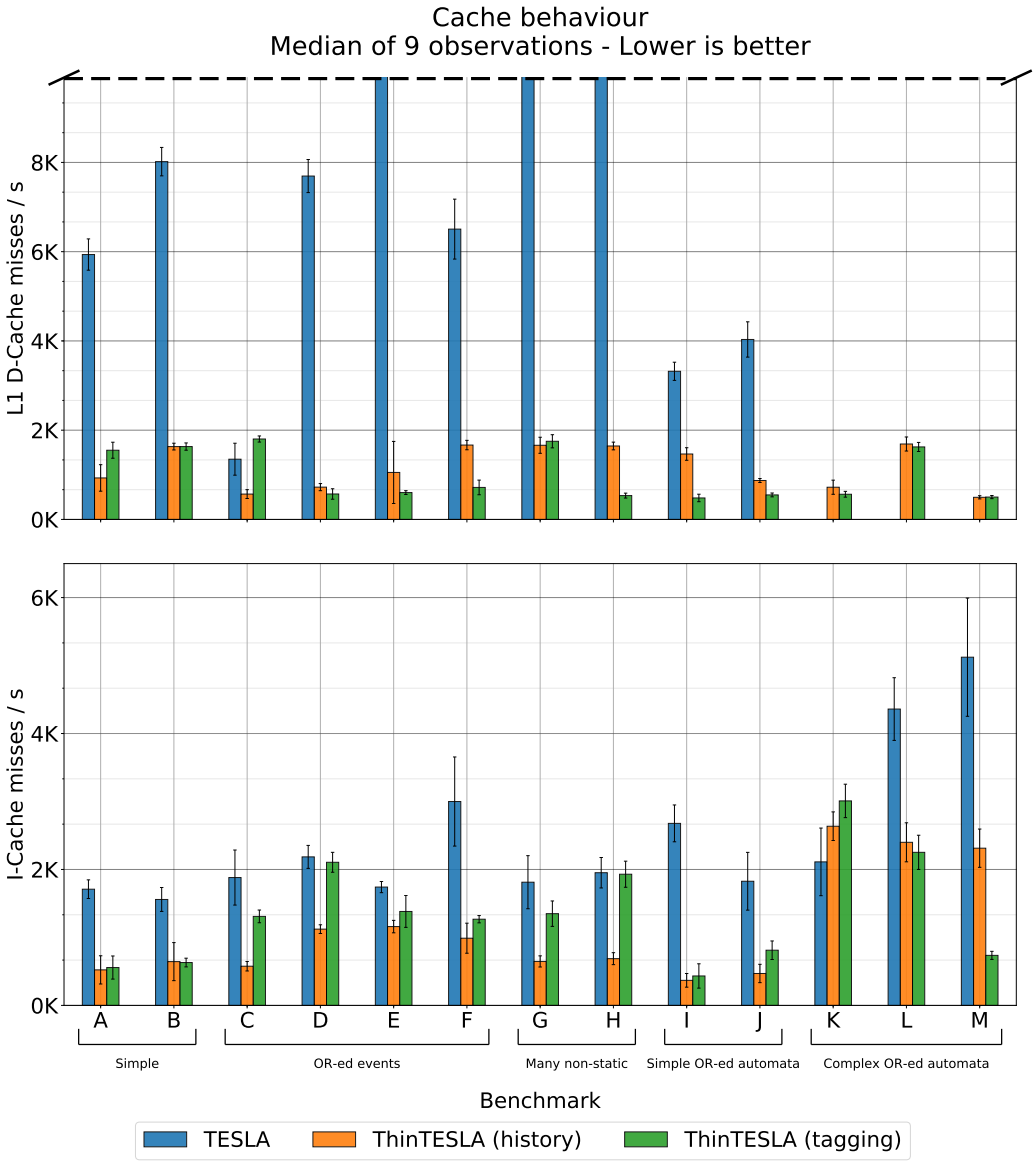


Figure 5.6: TESLA's and ThinTESLA's L1 data and instruction cache behaviour. L1 D-Cache miss rate for outliers is omitted from the graph

under 2'5000 across all set of benchmarks.

In terms of instruction cache misses, the behaviour roughly reflects the one observed for the data cache. However, ThinTESLA's footprint on the instruction cache is not always ideal. TESLA's code, and especially its hot paths, is mainly contained in a couple of files, making it possible for the compiler to

better inline small functions and perform optimizations that are usually only possible with link-time optimization (LTO) enabled. ThinTESLA's implementation spans more than 15 C translation units, and all of them contain code that is called in hot paths during runtime. Some of these functions are one or two lines long (e.g. calls to assembly intrinsics), and would greatly benefit from LTO. In order to perform a fairer comparison, however, LTO has not been enabled as it would be unavailable anyway in the kernel. Temporal tagging's impact on the instruction cache is especially significant, as the algorithm's implementation is not trivial and spans multiple functions.

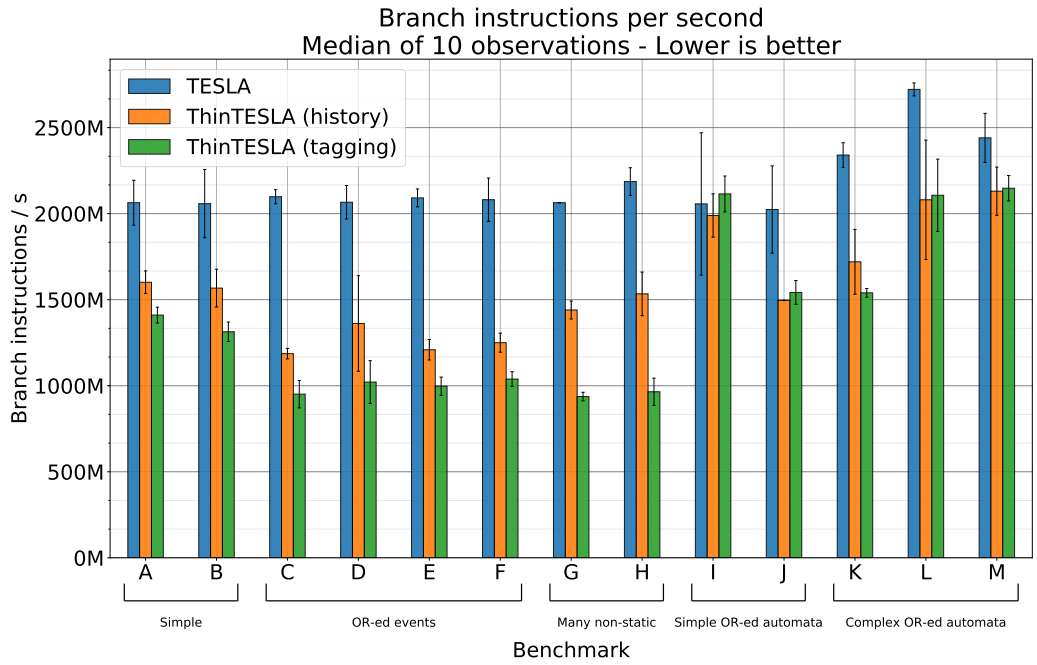


Figure 5.7: A comparison of the number of branch instructions per second between TESLA and ThinTESLA

While ThinTESLA's cache behaviour is better than TESLA's, the latter's impact on the cache is not enough to conclude that the overhead comes primarily from this micro-architectural aspect. This is notably true in non-pathological scenarios such as benchmark A, B, C, I and J, where ThinTESLA outperforms TESLA even though the number of cache misses for both frameworks is not especially high. In fact, an analysis of the number of retired instructions show that ThinTESLA's algorithms are much less com-

putationally expensive than the general approach used by TESLA, which for the same task requires 10x more instructions. When looking at a specific kind of instructions, namely branch instructions, ThinTESLA generally puts less pressure on the pipeline (especially when using temporal tagging), as Figure 5.7 shows. Therefore, we conclude that ThinTESLA trumps its predecessor both in asymptotic and micro-architectural behaviour.

5.3.2 FreeBSD kernel assertions

The FreeBSD kernel has been instrumented with ThinTESLA and TESLA with almost the same assertions used in the original paper. Some have been removed as they use modifiers not yet implemented in ThinTESLA such as `FLAGS`. Table 5.3 shows the number of assertions per type. As mentioned in 2.4, some assertions incorrectly fail under TESLA and some assertions are simply not up-to-date with FreeBSD 11.1-RELEASE. Therefore, both frameworks have been set to ignore assertion failures and continue execution. This is acceptable for comparing performance since both TESLA and ThinTESLA perform the work to verify the assertions regardless of the outcome.

Symbol	Description	Assertions	of which OR-ed assertions
MAC_SOCKET	MAC framework (sockets)	11	0
MAC_PROC	MAC framework (processes)	10	6
MAC_FS	MAC framework (filesystem)	24	6
MAC_ALL	MAC framework (all of the above)	45	12
PROC	Process lifetimes	35	12
ALL	**All of the above**	**80**	**24**

Table 5.3: Assertions inserted in the FreeBSD kernel by type

Effort required to integrate ThinTESLA

The original TESLA implementation required developers to modify part of the FreeBSD build system to output LLVM IR as an intermediate step in

order for it to be instrumented. The new build system integration mechanism of ThinTESLA allows the framework to be integrated into the kernel with no change to the build configuration files at all, and supports incremental rebuilds. This enables developers unfamiliar with the subtleties of a complex build system to integrate ThinTESLA without any special effort.

Microbenchmarks

Figure 5.8 shows the impact of TESLA and ThinTESLA on a micro-benchmark that tests the latency of system calls [1]. The results for the `read` system call are shown, but apply in general to any system call.

TESLA's impact is a consistent $\sim$ 2.5x degradation in latency, while ThinTESLA's increases as more assertions are enabled. However, ThinTESLA's impact on system call latency is less than 4% with socket assertions. This is because socket assertions do not have OR-ed automata and ThinTESLA can easily optimize away the temporal bounds. TESLA performs better with a large number of assertions, but that is partly because of its faulty and non-correct behaviour with OR-ed automata.

Performance improvements from temporal bounds optimizations

Figure 5.9 shows the performance of ThinTESLA when temporal bounds optimizations are disabled. A system call becomes extremely expensive, even when only a few assertions are enabled. Temporal bounds optimizations play a crucial role in lowering the general overhead, and can result in almost 10x improvement in microbenchmarks when all kernel assertions are enabled.

[1] `https://github.com/cracauer/ulmbenchmarks`

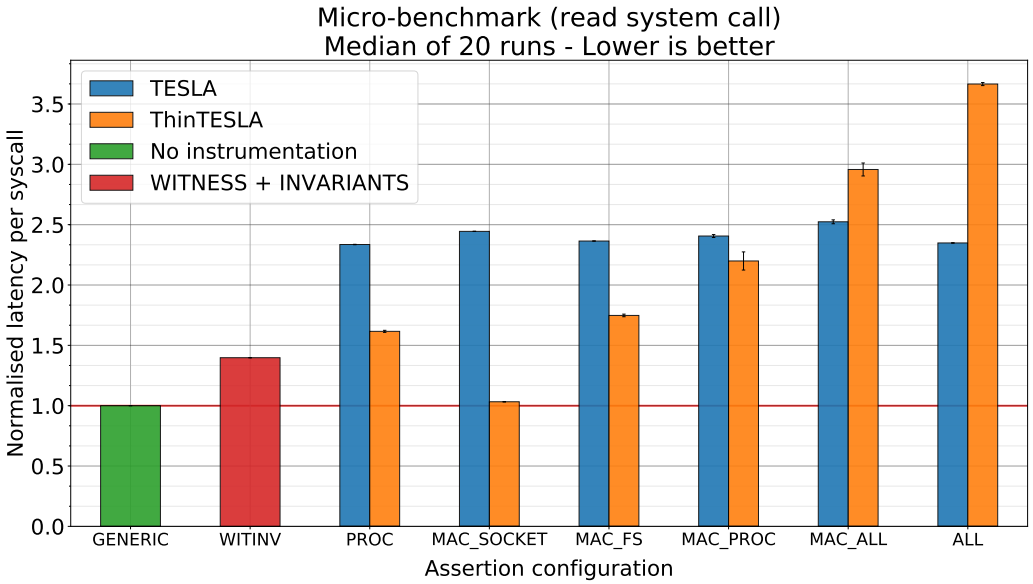


Figure 5.8: TESLA's and ThinTESLA's impact on the `read` system call in a micro-benchmark

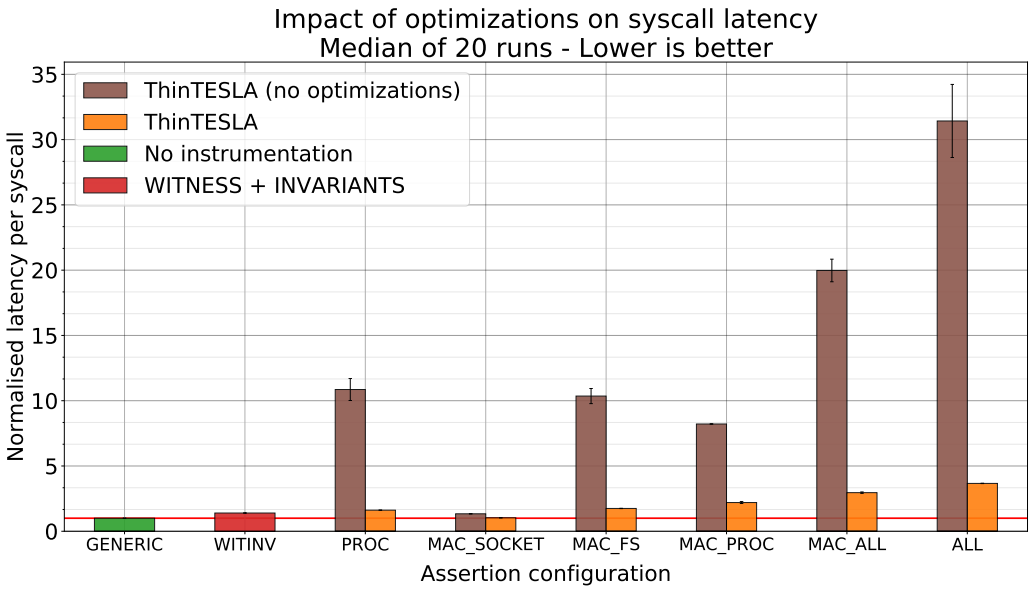


Figure 5.9: The impact on syscall latency when temporal bounds optimizations are disabled in ThinTESLA

Macrobenchmarks

TESLA and ThinTESLA's performance has been compared with respect to two macrobenchmarks that aim to stress-test the network and the filesystem components of the system. The first, Siege [2], sends a large number of requests to a local Apache HTTPD server. The second, Postmark [11], exercises the file system by creating, modifying, and deleting a large number of small files on a RAM disk. I was unable to reproduce the results of the original paper using the sysbench [3] OLTP benchmark, since the specific version of the benchmark is not available anymore.

Figure 5.10 shows the results obtained after running Siege (shown at the top) and Postmark (bottom). As expected, performance when running Siege with TESLA is most degraded (18%) when socket-related assertions are enabled. ThinTESLA's impact when socket assertions are enabled, on the other hand, is much lower (4%). The difference in performance between the two frameworks is even more evident with the Postmark benchmark: when filesystem assertions are enabled, TESLA and ThinTESLA have an impact of 44% and 14% respectively.

Both frameworks stay well below the overhead of FreeBSD's debug utilities WITNESS and INVARIANTS when considering the Apache workload, but the Postmark benchmark sees TESLA's overhead being close to that of WITNESS and INVARIANTS (44% and 53% respectively).

In kernel-space, the two verification algorithms compete on the same level of performance. The biggest benefit of using the linearised temporal history algorithm comes from when the number of events that are called is low (i.e. a low number of temporal observations). This is not the case in the kernel: many events lie in hot paths and are likely to be called a large number of times, even if they do not belong to the assertion being verified.

[2] `https://github.com/JoeDog/siege`
[3] `https://github.com/akopytov/sysbench`

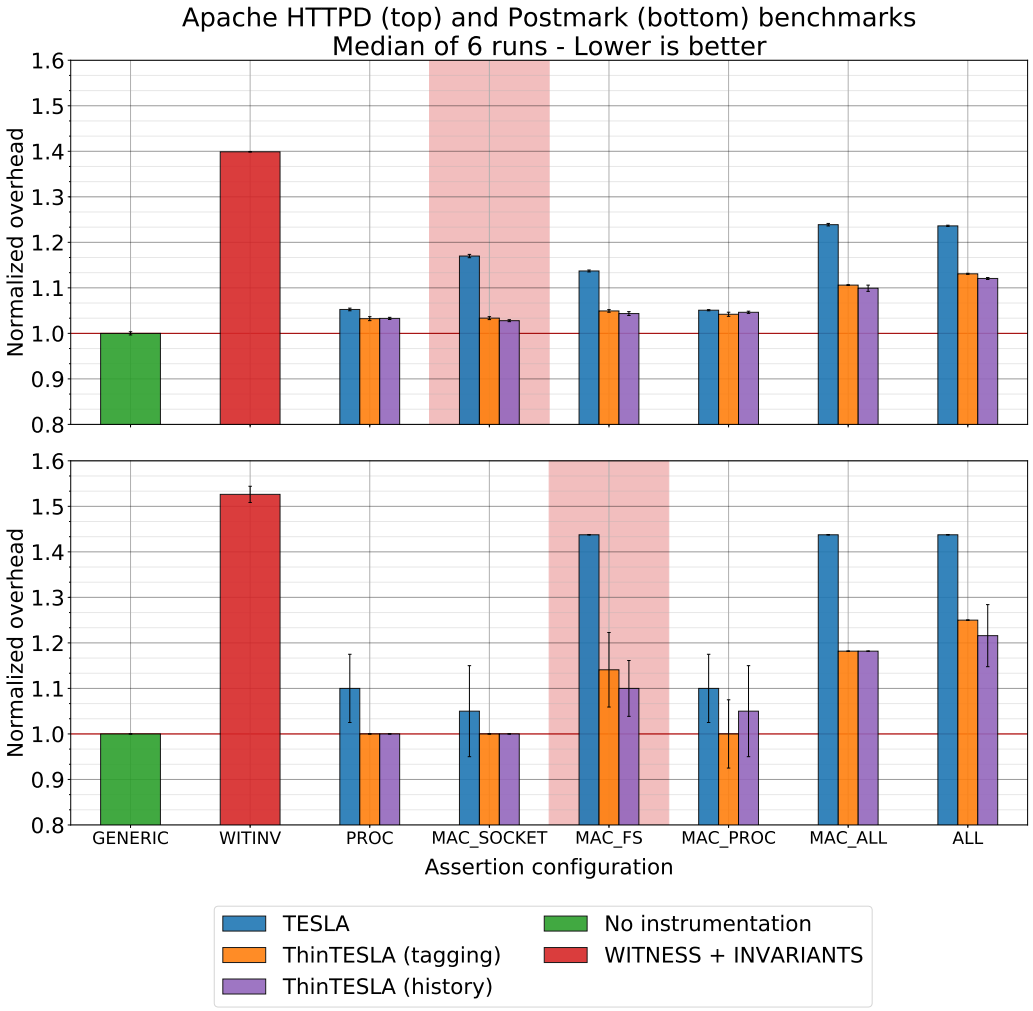


Figure 5.10: TESLA's and ThinTESLA's performance on socket-intensive and filesystem-intensive benchmarks. The column shaded in red represents the set of assertions being most frequently exercised

Chapter 6

Related Work

There is a fairly large amount of related work in the field of temporal assertions, program checking by formal models and instrumentation-based debugging, but TESLA and ThinTESLA can be considered unique in that it addresses a problem which is not taken into consideration by others. The majority of these approaches are mentioned in the original TESLA paper [1], and are here summarised along with the presentation of other relevant research that has been conducted after TESLA was published.

6.1 Model checking and static analysis

An automaton-based representation of security properties statically verified by model checking techniques was put forward by Chen et al. [12]. This approach cannot deal with information only available at runtime (e.g. memory or pointer values) and suffers from being sound but incomplete as most static analysis approaches.

Other approaches focus on specific classes of bugs and therefore lack the expressiveness of a general-purpose framework. Concurrent Predicates [13] by Gottschlich et al. aim to reproduce concurrency-related bugs by inserting in-code predicates that identify the sequence triggering the bug. The MAC

security framework of FreeBSD has been statically analysed to ensure the label management component doesn't cause memory leaks [14]. Muthukumaran et al. propose a static analysis approach for the automation and minimization of authorization and access control hooks by exploiting the developers' domain knowledge [15].

A large number of techniques and tools rely on proving or finding valid counterexamples to a formal model, leading to non-deterministic output and frequent time-outs, especially when the codebase under analysis is large. As an example, MUVI [16] detects concurrency bugs characterized by multi-variable access correlation, taking up to three hours when analysing large projects.

Tellez and Brotherston present a proof system able to statically verify temporal assertions that refer to memory state in pointer programs [17]. The proof system, implemented with the Cyclist theorem prover [18], has been tested on hand-crafted example programs and shows it is a valid alternative to traditional model checking. However, temporal assertions of complex and/or large programs are still impractical to prove.

Cook et al. propose an encoding to bring temporal logic semantics to off-the-shelf analysis tools [19], by introducing a proof system and a transformation that can be implemented in most existing analysis tools. Evaluation on non-trivial software such as the Windows OS kernel (after being adapted by hand) shows the potential viability of the approach, that is however still characterized by the limitations of static analysis.

TESLA, on the other hand, relies on runtime verification of temporal assertions, and only the assertions parser and the instrumentation library operate at compile time. On top of that, ThinTESLA introduces a static analysis pass that aims to reduce the complexity of assertions where possible, therefore reducing the performance overhead induced by the runtime library.

6.2 Dynamic verification of temporal assertions

Stolz and Bodden [20] propose a temporal assertion framework for Java based on AspectJ, an aspect-based extension for the Java language. AspectJ is used for instrumentation and is therefore a requirement. While the temporal properties that can be expressed are similar to TESLA's, they cannot contain references to runtime values. The framework is claimed to likely have a performance impact of no more than two percent, but no evaluation is present.

Stolz later introduced the theoretical foundation for temporal assertions with parametrised prepositions [21]. This enables support for assertions relying on runtime values and free variables, but no implementation has been made available.

Dynamic Temporal Assertions (DTA) [22] by Al-Sharif et al. for the Unicon programming language make it possible to express simple temporal properties such as `previously` or `eventually` with a limited notion of temporal intervals. DTA are injected at debugging time by a modified version of the Unicon debugger (UDB). Evaluation, although performed on a single assertion, shows the induced performance overhead is close to an order of magnitude.

Recent approaches suggest the combined use of static and dynamic verification [23, 24] in order to take advantage of the former where possible and resort to the latter only when needed. However, they focus far more on the static analysis component, do not target commonly-used programming languages, and program-wide (interprocedural) temporal properties are not supported. ThinTESLA does provide a static analysis component that aims to reduce the effort required by the runtime library.

A hybrid approach making use of static verification of a trace collected during the execution of a program is put forward by Ferlin et al. [25]. The static verifier relies on Linear Temporal Logic (LTL) and Büchi automata

to determine whether a trace respects a certain temporal property. The implementation is inaccessible due to industry confidentiality reasons, but is reported to be around 30,000 lines of C++ code. ThinTESLA's runtime library, which does not require traces to be collected but instead runs alongside the target program, is implemented in around 2,500 lines of C.

In contrast to the above-mentioned approaches, TESLA provides a general-purpose framework for C and Objective-C programs that supports dynamic verification of temporal assertions which expressiveness includes references to runtime values and concurrency awareness. ThinTESLA limits the scope of the framework to C programs and slightly changes the semantics and the temporal properties supported, but is able to guarantee better performance in all scenarios.

6.3 Program instrumentation

Program instrumentation is a flexible technique that allows for the modification of a program at compile-time or even after the binary has been assembled.

DTrace [26], originally for the Solaris operating system, is one of the most popular instrumentation tools that have been ported to FreeBSD. DTrace is able to instrument a large number of events in the kernel and run user-defined code at those points to collect data, process it and report any result to the user. DTrace often introduces a significant probe effect, and is not able to flexibly express temporal properties.

Tools such as DynamoRIO [27] perform binary instrumentation to insert arbitrary code into an existing program. The performance overhead is usually prohibitive as the code that is added to the binary needs to deal with registers and instructions allocation decisions that the compiler took and which are not reversible. Furthermore, binary instrumentation can interfere with some properties about the final binary layout that are assumed to be always true by the program.

Tools that aim to make compile-time instrumentation more accessible and intuitive are becoming more and more common. CSI [28] by Schardl et al., for example, provides a framework for easily instrumenting programs with the the help of the LLVM compiler infrastructure. However, it requires link-time optimization (LTO) in order to provide acceptable performance. Some codebases such as the FreeBSD kernel do not support LTO yet.

TESLA and ThinTESLA rely on custom compile-time LLVM [2] instrumentation passes. Binary instrumentation would not be viable due to the performance overhead, and a general solution such as CSI would impose unnecessary limitations such as the impossibility to instrument the FreeBSD kernel. ThinTESLA refines the instrumentation pass to take advantage of the fact that only function calls are instrumented, and provides an easy way to integrate the framework into existing build systems.

Chapter 7

Conclusions and future work

In this dissertation I presented ThinTESLA, a low-overhead framework for temporal assertions. ThinTESLA, which addresses the bugs and performance concerns of TESLA [1], performs considerably better both in userspace and in the FreeBSD kernel. The overhead of temporal assertions is decreased by up to 20 times in the former case and 3 times in the latter. Performance gains come mainly from a better algorithmic solution for runtime automata verification that exploits a simpler, light-weight automaton representation, and from an implementation that takes into account micro-architectural aspects such as cache footprint.

Future work could be directed at further improving the representation of automata, especially in the case of automata derived from assertions with boolean OR expressions at the top level. Those interfere with useful optimizations, and are one of the causes of ThinTESLA's overhead. The static analysis algorithm could be made more mature and expanded to include both static and non-static events' analysis, therefore enlarging the space of assertions that can be optimized statically. It would be interesting to determine whether ThinTESLA's design can be applied to arbitrary programming languages with minimal effort. If so, it could become a standard low-overhead debugging framework that works across languages, operating systems and environments.

Bibliography

[1] Jonathan Anderson, Robert N. M. Watson, David Chisnall, Khilan Gudka, Ilias Marinos, and Brooks Davis. TESLA: Temporally Enhanced System Logic Assertions. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 19:1–19:14, New York, NY, USA, 2014. ACM.

[2] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[3] Kenton Varda. Protocol buffers: Google's data interchange format. Technical report, Google, June 2008.

[4] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP (extended version). Technical Report UCAM-CL-TR-873, University of Cambridge, Computer Laboratory, August 2015.

[5] Sung-Eun Choi and E. Christopher Lewis. A study of common pitfalls in simple multi-threaded programs. In *Proceedings of the Thirty-first SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '00, pages 325–329, New York, NY, USA, 2000. ACM.

[6] D. Cederman, B. Chatterjee, N. Nguyen, Y. Nikolakopoulos, M. Papatriantafilou, and P. Tsigas. A study of the behavior of synchronization methods in commonly used languages and systems. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 1309–1320, May 2013.

[7] T. Johnson, M. Amini, and X. David Li. Thinlto: Scalable and incremental lto. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 111–121, Feb 2017.

[8] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *Ann. Math. Statist.*, 18(1):50–60, 03 1947.

[9] T. Chen, Q. Guo, O. Temam, Y. Wu, Y. Bao, Z. Xu, and Y. Chen. Statistical performance comparisons of computers. *IEEE Transactions on Computers*, 64(5):1442–1455, May 2015.

[10] A. Kolmogorov. Sulla determinazione empirica di una legge di distribuzione. *Giornale dell' Istituto Italiano degli Attuari*, 4:83–91, 1933.

[11] Jeffrey Katcher. Postmark: a new file system benchmark. Network Appliance Tech Report TR3022, October 1997.

[12] Hao Chen and David A. Wagner. MOPS: An Infrastructure for Examining Security Properties of Software. Technical report, University of California at Berkeley, Berkeley, CA, USA, 2002.

[13] Justin E. Gottschlich, Gilles A. Pokam, Cristiano L. Pereira, and Youfeng Wu. Concurrent predicates: A debugging technique for every parallel programmer. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 331–340, Piscataway, NJ, USA, 2013. IEEE Press.

[14] Xinsong Wu, Zhouyi Zhou, Yeping He, and Hongliang Liang. Static Analysis of a Class of Memory Leaks in TrustedBSD MAC Framework. In Feng Bao, Hui Li, and Guilin Wang, editors, *Information Security Practice and Experience*, pages 83–92, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[15] Divya Muthukumaran, Nirupama Talele, Trent Jaeger, and Gang Tan. Producing hook placements to enforce expected access control policies. In Frank Piessens, Juan Caballero, and Nataliia Bielova, editors, *Engineering Secure Software and Systems*, pages 178–195, Cham, 2015. Springer International Publishing.

[16] Shan Lu, Soyeon Park, Chongfeng Hu, Xiao Ma, Weihang Jiang, Zhenmin Li, Raluca A. Popa, and Yuanyuan Zhou. MUVI: Automatically inferring multi-variable access correlations and detecting related semantic and concurrency bugs. In *In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP07)*, 2007.

[17] Gadi Tellez and James Brotherston. Automatically verifying temporal properties of pointer programs with cyclic proof. In Leonardo de Moura, editor, *Automated Deduction – CADE 26*, pages 491–508, Cham, 2017. Springer International Publishing.

[18] James Brotherston, Nikos Gorogiannis, and Rasmus L. Petersen. A generic cyclic theorem prover. In Ranjit Jhala and Atsushi Igarashi, editors, *Programming Languages and Systems*, pages 350–367, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[19] Byron Cook, Eric Koskinen, and Moshe Vardi. Temporal property verification as a program analysis task. *Formal Methods in System Design*, 41(1):66–82, Aug 2012.

[20] Volker Stolz and Eric Bodden. Temporal Assertions using AspectJ. *Electronic Notes in Theoretical Computer Science*, 144(4):109 – 124, 2006. Proceedings of the Fifth Workshop on Runtime Verification (RV 2005).

[21] Volker Stolz. Temporal assertions with parametrised propositions. In Oleg Sokolsky and Serdar Taşıran, editors, *Runtime Verification*, pages 176–187, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[22] Z. A. Al-Sharif, C. L. Jeffery, and M. H. Said. Debugging with dynamic temporal assertions. In *2014 IEEE International Symposium on Software Reliability Engineering Workshops*, pages 257–262, Nov 2014.

[23] Johannes Bader, Jonathan Aldrich, and Éric Tanter. Gradual program verification. In Isil Dillig and Jens Palsberg, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 25–46, Cham, 2018. Springer International Publishing.

[24] Julian Tschannen, Carlo A. Furia, Martin Nordio, and Bertrand Meyer. Usable verification of object-oriented programs by combining static and dynamic techniques. In Gilles Barthe, Alberto Pardo, and Gerardo Schneider, editors, *Software Engineering and Formal Methods*, pages 382–398, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[25] A. Ferlin, P. Bon, S. Collart-Dutilleul, and V. Wiels. Parallel verification of temporal properties using dynamic analysis. In *2015 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 29–38, Oct 2015.

[26] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the*

Annual Conference on USENIX Annual Technical Conference, ATEC '04, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[27] Derek L. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004. AAI0807735.

[28] Tao B. Schardl, Tyler Denniston, Damon Doucet, Bradley C. Kuszmaul, I-Ting Angelina Lee, and Charles E. Leiserson. The CSI framework for compiler-inserted program instrumentation. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(2):43:1–43:25, December 2017.