Effective Verification of Interrupt-Driven Software



Lihao Liang
St Catherine's College
University of Oxford

A thesis submitted for the degree of
Doctor of Philosophy
Michaelmas 2018

Dedicated to my beloved parents Guosen and Jianfeng for their unconditional love and enduring inspiration.

Abstract

Interrupt-driven software is difficult to test and debug, especially when interrupts can be nested and subject to priorities. Interrupts can arrive at arbitrary times, leading to an exponential blow-up in the number of cases to consider.

The first part of this thesis presents two different approaches to verifying software with nested interrupts. The first approach uses source-to-source transformations that translate an interrupt-driven program into a sequential and a multi-threaded program. The second method is a new formal approach based on symbolic execution. This approach leverages recent advances in the encoding of the execution traces of interacting, concurrent threads. We assess the performance of these techniques on benchmarks drawn from real-world embedded systems code and Linux device drivers. To the best of our knowledge, our work is the first to demonstrate effective formal verification of low-level embedded software with nested interrupts.

The second part of this thesis studies the verification problem of Read-Copy Update (RCU), a scalable high-performance synchronization mechanism in the Linux operating system that allows extremely low-overhead readers to run concurrently with updaters. Production-quality RCU implementations are highly sophisticated and make heavy use of the scheduling-clock interrupt. Previous validation efforts on RCU either focus on simple implementations or use modelling languages that are difficult to translate from C. The resulting error-prone manual translation step must be repeated frequently due to the rapid change rate of the RCU implementation in Linux. We first explain in detail the RCU implementation in the Linux kernel; then we describe how to construct a model from the RCU source code, and use the CBMC model checker to verify its safety and liveness properties. This is the first attempt to verify the core components of the Tree RCU implementation in the Linux kernel. We hope it will be a step towards integrating formal verification into RCU's regression test suite in Linux.

Credit

Credit for technical content and presentation of this thesis is due to:

- Paul E. McKenney for providing detailed feedback on Section 6.4, all the bug-injection scenarios in Section 7.2.1, and Figures 6.1–6.4
- Peter Schrammel for providing the Logger and Brake benchmarks, test cases for all the benchmarks in Chapter 5, and the initial version of Figures 4.4 and 4.5
- Ilja Zakharov for providing the RcCore benchmark in Chapter 5
- Doina Bucur for providing the Blink benchmark in Chapter 5
- CBMC is developed and maintained by Daniel Kroening and his research group in Oxford
- Read-Copy Update (RCU) in the Linux kernel is developed and maintained by Paul E. McKenney

List of Publications

- Lihao Liang, Tom Melham, Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Effective Verification of Low-Level Software with Competing Interrupts. In *ACM Transactions of Embedded Computing Systems (TECS)*, Volume 17, Issue 2, April 2018.
- Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of Tree-Based Hierarchical Read-Copy Update in the Linux Kernel. In *Proceedings of the 2018 Design, Automation and Test in Europe Conference (DATE)*, pages 61–66, 2018.
- Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of the Tree-Based Hierarchical Read-Copy Update in the Linux Kernel. In *arXiv Computing Research Repository*, arXiv:1610.03052, 2016.
- Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective Verification of Low-Level Software with Nested Interrupts. In *Proceedings of the 2015 Design, Automation and Test in Europe Conference (DATE)*, pages 229–234, 2015.
- Alex Horn, Michael Tautschnig, Celina Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. Formal Co-Validation of Low-Level Hardware/Software Interfaces. In *Proceedings of the 13rd International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 121–128, 2013.

Acknowledgements

First of all, I would like to express my gratitude to my supervisors, Professors Daniel Kroening, Tom Melham, and Luke Ong. Daniel gave me the freedom to work on research topics that interested me and provided support whenever needed. His high standard and hands-on approach to writing code in a research lab gave me a rare opportunity to work on industrial-strength software in academia. I also thank him for introducing me to the fascinating topic of Read-Copy Update (RCU) verification. I appreciate the time and effort that Tom spent in our numerous discussions, which led to substantial improvement to clarity and the technical formulation in the first part of this thesis. Tom also provided valuable advice on how to write a good research paper, in particular how to express complex, technical ideas in a clear and precise way. Luke could not have given me more support to pursue my academic goal. He once told me that getting a D.Phil degree from Oxford is not easy, yet it depends on how badly you want it. Luke genuinely cares about my wellbeing and continuously supports my career development, for which I am extremely grateful.

My research was generously funded by a scholarship from Intel Corporation and the Department of Computer Science. I greatly appreciate their financial support, which makes my dream of studying at Oxford possible.

I thank my examiners Professors Bill Roscoe and Peter Sewell. Peter and Bill provided many constructive feedback and comments that helped improve this thesis significantly.

My special thanks go to Paul E. McKenney, with whom I was very fortunate to be able to work closely. Paul is always keen in sharing his knowledge and insights about Read-Copy Update (RCU) as well as the Linux kernel. I have learned a great deal from our countless discussions, for which I am truly grateful.

It was a great pleasure to work with Peter Schrammel. Peter is always supportive, responsible, and kind. I enjoy our discussions about how our field

is involved in the modern era. I am also thankful to Michael Tautschnig, who helped me better understand the non-trivial CBMC codebase, and provided thorough reviews of my code.

I appreciate the illuminating discussions in various meetings with our project partners Moshe Vardi, Sharad Malik, Alan Hu, Jim Grundy, Celina Val, Kuldeep Meel, Sonali Dutta, Sunha Ahn, Sam Bayless, and Divjyot Sethi.

I did an internship in the Operating Systems Kernel Lab at Huawei Technologies. I want to express my gratitude to the Lab Director Haibo Chen for his huge support that allowed me to continue working on RCU as my intern project. I also want to thank my colleagues at Huawei who taught me a lot during my internship.

Over the years, my friends and colleagues in the Department of Computer Science provided me with ideas, support, and advice. I am grateful to Hongseok Yang and Nando De Freitas for those fruitful discussions. I also want to thank James Worrell, Joel Ouaknine, Michael Benedikt, Stefan Kiefer, Dan Olteanu, and Boris Motik for their advice and encouragement at various stages of my doctoral studies.

I had many fond memories with my friends and colleagues in the Verification Group. Anthony Lin and I had lots discussions about career and life, many of which occurred in Oxford's best Fish and Chips place in Old Marston. I shared the same office with Alex Horn, with whom I often talked about research and programming. His conscientiousness inspired me to produce better work. Vincent Nimal, Ganesh Narayanaswamy, and I spent many late nights together before paper deadlines. G & D coffee and ice cream kept us productive. I thank Daniel Poetzl for his feedback on the partial-order encoding in Section 4.5, and Rajdeep Mukherjee for sharing his research lessons and experiences. I also thank Björn Wachter, Youcheng Sun, Cesar Rodriguez, Marcelo Sousa, Leopold Haller, Dario Cattaruzza, Georg Weissenbacher, Vijay D'silva, Ruben Martins, Cristina David, Liana Hadarean, Vojtech Forejt, Pascal Kesseli, Matt Lewis, Lucas Cordeiro, Martin Brain, and Ashutosh Natraj for their valuable advice and insightful discussions.

I had various interesting conversations with Ines Marusic, Pengyu Wang, Yishu Miao, Ziyu Wang, Lei Yu, Brendan Shillingford, Ventsislav Chonev, Joao Sousa-Pinto, Josh Ko, Rastislav Lehardt, and Daniel Bundala.

The Department of Computer Science was very supportive throughout my studies. I am thankful to Julie Sheppard and David Kay who provided the support whenever I needed and helped me focus on my research. I am also grateful to my college St Catherine's and its staff for providing a supportive and friendly environment to all of its students. I enjoyed studying and working in the library on numerous evenings and weekends.

My friends made my Oxford life much more fun and memorable. I truly enjoyed my time at my College formal dinners, MCR parties, and pizza nights where we talked about almost everything, life, culture, politics, international relations, etc. I was elected as our MCR bar officer and it was a fantastic experience. I thank all my friends at Catz for distracting me enough, but too much, when my research became challenging. Xi Geng and I studied mathematics at the same university in China. It was amazing that we both pursued a doctoral degree at Oxford. Discussing mathematics and doing research together in the library and office were some of the most memorable moments of my university life. I thank Junsong Lin, Sissy Liao, Kaiwei Wang, Jiena Chen, Kun Ma, Ting Chu, Haiwei Long, Wei Pan, Hao Ni, Yijing Song, and Wei Wei for those wonderful social events we enjoyed together. My thanks also go to Shanshan Peng for the advice in the early stage of my D.Phil and to Qianxi Jiang for the encouragement when things were getting difficult.

During my D.Phil years, I had the rare opportunities to meet and talk with many Turing Award recipients, including Donald Knuth, Michael O. Rabin, Dana S. Scott, Tony Hoare, Stephen A. Cook, Richard Karp, Robert Tarjan, John Hopcroft, William Kahan, Butler W. Lampson, Andrew Yao, Adi Shamir, Vinton G. Cerf, Edmund M. Clarke, Barbara Liskov, Leslie G. Valiant, Shafi Goldwasser, Leslie Lamport, Geoffrey Hinton, and Yann LeCun, whose achievements have influenced our field and society in profound ways.

I was incredibly excited to meet Demis Hassabis, Andrew Wiles, and Yitang Zhang, whose stories have inspired me and will continue to do so.

Last but not least, I am indebted to my parents, whose unconditional love and support are beyond words. Without them, none of these would have been possible. 读博期间我和父亲有很多交流和沟通，父亲大多时候在聆听，不时会给点建议，在最困难的时候是他的鼓励让我能够坚持下去。母亲在 15 年曾来牛津探望，我们游走了著名的景点和学院，拍了不少照片，还在我学院食堂一

起进餐，都成为我读博期间最美好的回忆。博士期间每年回家的机会和时间并不多，爷爷和我时常通过电话和视频分享生活，让我在科研的忙碌和压力下得到放松。博士后期却因繁忙和时差不时错过爷爷的来电，深感遗憾。我多么希望可以更快毕业，岁月如梭，爷爷，奶奶和外婆未能看到我穿上博士袍的那一天，不过无论在哪里我相信他们都会为我能成为牛津大学的博士感到高兴。我的感谢难以用寥寥数语表达，就此搁笔。最后由衷感激家人一直以来对我的爱和关怀，无以为报，只有感恩。谢谢你们!

Contents

1 Introduction **1**

1.1 Verification of Software with Nested Interrupts 2

1.2 Verification of Read-Copy Update in the Linux Kernel 4

2 Literature Review **6**

3 Program Transformations for Nested Interrupts **27**

3.1 Nested Interrupts 27

3.2 Interrupt Sequentialisation 30

3.2.1 Modelling Nested Interrupts using a Scheduler 31

3.2.2 Reducing Scheduler Calls 33

3.2.3 Example of Interrupt Sequentialisation 35

3.3 Modelling Interrupts with Threads 38

4 Partial-Order Encoding for Nested Interrupts **43**

4.1 Introduction 43

4.2 Static Single Assignment Form 45

4.3 Symbolic Events 47

4.4 Partial-Orders for Nested Interrupts 48

4.5 Algorithms for Computing Constraints for Symbolic Partial Orders . 52

4.6 Analysis of the Algorithms 61

5 Experimental Studies of Nested Interrupts Verification 65

5.1 Verification Tools 65

5.2 Benchmarks 67

5.3 Results and Discussion 68

6 Read-Copy Update in the Linux Kernel 78

6.1 Introduction 78

6.2 What is RCU? 80

6.3 Core RCU API Usage 81

6.4 Implementation of Tree RCU 83

6.4.1 Overview 84

6.4.2 Read/Write-Side Primitives 85

6.4.3 Data Structures of Tree RCU 85

6.4.4 Quiescent State Detection 89

6.4.5 Grace Period Detection 91

6.5 Verification Scenario 94

7 Modelling and Verification of Tree RCU 97

7.1 Modelling RCU for CBMC 97

7.2 Verification of Tree RCU 101

7.2.1 Bug-Injection Scenarios 101

7.2.2 Validating the RCU Model in User-Space 104

7.2.3 Getting CBMC to work on Tree RCU 105

7.2.4 Results and Discussion 106

8 Conclusion **112**

Bibliography **115**

List of Figures

3.1 A complex scenario of nested interrupts 29

3.2 Motivating example 29

3.3 Motivating example (modified) 30

4.1 Overview of encoding into partial-order constraints 44

4.2 An example of the static single assignment form for concurrent programs 47

4.3 Symbolic memory events for the program in Figure 4.2 48

4.4 Event graph for the example program in Figure 4.2 49

4.5 Invalid execution in an event graph 52

4.6 Counterexample 62

4.7 ISR A preempts the main program right before event m 63

5.1 i-CBMC v.s. CBMC with threads modelling 73

5.2 i-CBMC v.s. i-CBMC with precise encoding 74

5.3 Number of constraints in the encodings 75

5.4 Number of variables in the SAT formulas 75

5.5 Number of clauses in the SAT formulas 76

6.1 RCU concepts 81

6.2 Verifying RCU grace periods 82

6.3 Tree RCU hierarchy 86

6.4 Grace-period detection state diagram 91

7.1 Number of constraints in the SAT formulas 108

7.2 Number of variables in the SAT formulas 108

7.3 Number of clauses in the SAT formulas 109

7.4 Total runtime in seconds 109

7.5 Maximum memory consumption in gigabytes 109

Chapter 1

Introduction

Interrupts are a key design primitive for embedded and systems software that interacts closely with hardware. They are signals to the processor generated by hardware or software indicating an event needs immediate attention, which enables timely responses to outside stimuli in a power-efficient way. Interrupts are common in all styles of computing platforms, including safety-critical embedded software, low-power mobile platforms, and high-end information systems.

But interrupt-driven code is difficult to engineer. Device drivers, typical examples of software that use interrupts heavily, are known as the "fault hotspots" of the Linux kernel [CYC+01, PTS+11]. The crux of the problem is the nondeterminism inherent in systems that use interrupts: The hardware can divert control to the interrupt service routine (ISR) at any time, resulting in surprising interactions between the code that is interrupted and the ISR. The problem is exacerbated by *interrupt nesting*, where the ISR itself can be preempted by interrupts with higher priority.

Most existing approaches to validating interrupt-driven software rely on testing. But testing is particularly ineffective in the case of nested interrupts, as the number of possible interleavings—i.e. interspersions of interrupt arrivals into a run of the code—grows exponentially in the number of interrupts that occur. Bugs are therefore easily

missed, and any errors that are observed are hard to reproduce.

The concurrent nature of interrupt generation and handling as well as the sheer size of the search space suggest a formal approach to validation. Model checking is a particularly effective technique when applied to problems with many subtle corner cases. Several model-checking-based approaches to the problem have therefore been explored [BK11, HTV+13, LRJ13, SNBB09, VHHK15, WBKW07, WWC+13]. For example, we use the C Bounded Model Checker (CBMC) [CKL04] to co-validate an Ethernet MAC hardware/software interface model with interrupts [HTV+13]. But all these methods only consider programs with a single kind of interrupt—none of them handle programs with multiple interrupt kinds that can be nested.

The goal of this thesis is to design and implement effective techniques and tools to verify software with nested interrupts.

1.1 Verification of Software with Nested Interrupts

The first part of this thesis introduces the first approaches for verification of software with nested interrupts. Our experimental studies demonstrate that these methods are effective in verifying real-world interrupt-driven code.

Program Transformations. The first contribution of this thesis is two program transformation techniques that transform programs with nested interrupts into sequential and multi-threaded programs, respectively, which can then be passed to off-the-shelf sequential and concurrent software verifiers.

The first approach is to instrument the source code of the program by inserting conditional calls to the interrupt procedure, in effect mimicking the semantics of the hardware interrupt. The instrumented program is then passed to a conventional analyzer for sequential programs. The approach of program sequentialisation is straightforward to implement, and is in principle viable in the case of nested inter-

rupts as well. Sophisticated instances of this type of approach use over-approximating analyses, akin to partial-order reduction, to determine program locations where the call to the ISR can be omitted safely—for example, when only variables that are not shared with the ISR are accessed. We present this approach in detail in Section 3.2.

The second approach takes advantage of the close resemblance of the interleaving semantics of programs with nested interrupts and the semantics of programs with multiple threads. In particular, the behaviours under interrupt semantics are a subset of the behaviours under thread semantics. This suggests analysis using verification tools for concurrent programs. False alarms caused by over-approximation can be addressed by means of suitable instrumentation. We describe this method in Section 3.3.

Partial-Order Encoding. Our second contribution is a symbolic execution technique that leverages recent advances in concurrent software verification based on the boolean satisfiability problem (SAT) [AKT13]. The work on memory models reported by Alglave et al. [AKT13] explores a weakening of the thread semantics to capture the additional behaviours in the multi-core case. By contrast, our contribution is to employ a specially designed *strengthening* of the interleaving semantics of threads over the Sequential Consistency (SC) memory model, in the form of a symbolic partial order that captures more precisely the subset of interleavings required to model nested interrupts.

This technique is explained in detail in Chapter 4. The central idea is to collect the set of possible events (reads or writes) in the system and construct a formula that relates their relative ordering by means of a symbolic *happens-before* relation. The formula is then passed to a modern boolean satisfiability problem (SAT) or satisfiability modulo theories (SMT) solver together with a constraint that specifies the set of error states. If the solver determines that the formula is satisfiable, which

means an error state is reachable, an error trace giving the exact sequence of events can be extracted from the satisfying assignment.

Experimental Studies. The third contribution of this thesis is to provide the first experimental studies on techniques for verification of programs with nested interrupts (Chapter 5).

To evaluate the performance of our proposed methods, we implement our partial-order encoding in a prototype tool called *i-CBMC*, which is an extension of CBMC. Moreover, we apply a range of existing verifiers to sequentialised programs and programs that have been instrumented with threads. We assess the performance of these methods on real-world benchmarks drawn from embedded systems code in TinyOS and Simulink, as well as Linux device drivers.

This line of work provides the first method and tool implementation for effective verification of software with nested interrupts, which is published in the *Proceedings of the 2015 Design, Automation and Test in Europe Conference (DATE)* [KLM+15]. A full description of our program transformation techniques and partial-order encoding, as well as more extensive experimental studies, are reported in a journal publication in the *ACM Transactions on Embedded Computing Systems (TECS)* [LMK+18].

1.2 Verification of Read-Copy Update in the Linux Kernel

The second part of this thesis studies the verification problem of Read-Copy Update (RCU), a scalable high-performance synchronization mechanism in the Linux kernel that allows extremely low-overhead readers to run concurrently with updaters [MS98, MW08]. Production-quality RCU implementations are highly sophisticated and make heavy use of the scheduling-clock interrupt.

Previous validation efforts on RCU either focus on simple implementations [AKT13, McK15] or use modelling languages that are difficult to translate from its source code in C [DMD13, McK08a]. The resulting error-prone manual translation must be repeated frequently due to the rapid change rate of the RCU implementation in the Linux kernel.

Tree-RCU Verification. Our fourth contribution is to provide a detailed explanation of the Tree RCU implementation in the Linux kernel (Chapter 6), and use CBMC to verify Tree RCU's safety and liveness properties from its C code over the SC, TSO, and PSO memory models (Chapter 7). This is the first attempt to verify the core components of the Tree RCU implementation. Although we believe this verification effort sits on the current limit of CBMC and SAT solving in terms of scalability, it demonstrates that RCU is a rich example to drive research. We hope our work will be a step towards integrating formal verification into RCU's regression test suite in the Linux kernel. This research is published in the *Proceedings of the 2018 Design, Automation and Test in Europe Conference (DATE)* [LMKM18].

Chapter 2

Literature Review

Interrupts are widely used in embedded and systems software, particularly in safety critical applications and systems in which the consequences of a bug can be severe. As discussed in [Reg07], the following are typical problems in interrupt-driven systems:

1. Stack overflow: Embedded systems are limited in resources such as memory and it is very important that interrupt-driven programs running on such systems do not overflow the stack, corrupting other memory regions and crashing the system.

2. Real-time constraints: In hard real-time systems, it is part of the correctness specification that deadlines of interrupt handling are met, which is difficult to ensure by testing.

3. Functional correctness: Interrupt-driven software is often low-level system code, interacting with the operating system kernel and the hardware, which is difficult to write. Studies in [CYC+01] and [PTS+11] show that device drivers, which are typical examples of read-world interrupt-driven software, contain the most faults in the Linux operating system.

4. Subtle concurrency: Preemptive threads run with the presence of (nested) inter-

rupts. The differences of interleaving semantics between threads and interrupts are quite subtle (see Section 3.1).

In the rest of this chapter, we first survey techniques to address these problems, including static analysis of stack usage and execution time, and formal methods to detect bugs and prove the absence of them. Then we discuss research closely related to the methods in this thesis. Finally, we review various efforts to validate Read-Copy Update (RCU), a sophisticated interrupt-driven program in the Linux kernel [MS98, MW08].

Analysis of Interrupt-Driven Software

We review techniques that statically analyze usage of two important resources of interrupt-driven programs in real-time embedded systems, stack size and execution time, and ensure their safety in such systems.

Stack Size Analysis

Brylow et al. [BDP01] present a design and implementation of a static analyzer for Z86-based assembly code with interrupts. Their method involves building a control-flow graph (CFG) of the program that models only the program counters and the interrupt mask registers (IMR) to reduce the state space, and labelling its edges with information about stack size, operation and time. Then the maximum stack size and interrupt latency can be computed by finding a path in the CFG with the largest weights. Also, stack operations can be checked, e.g. whether pop and push operations are well matched. But their method can not handle programs with complicated loops. Their tool is able to find a tight upper bound of stack size and type-check stack operations in 6 of 7 benchmark programs, each of which has about 1000 lines of

code. Also it can find a latency bound for relatively few of the nodes in most of the programs.

Palsberg and Ma [PM02] propose a generic assembly language with interrupts, called the Interrupt Calculus, that contains essential constructs for programming interrupt-driven systems. Programs written in this calculus have access to the interrupt mask register, which can be updated during execution, and a memory of integer variables. As they allow programs to enable disabled interrupts, re-entrance of interrupt handlers is possible, causing the stack to overflow. To avoid this situation, the calculus also has a static type system that enables modular type checking to ensure stack boundedness. A soundness theorem asserts that the stack size for interrupt handling of an Interrupt Calculus program is guaranteed to be bounded if the program is well-typed. However, the type system is not complete. Also their type system requires programmers to provide type annotations, which are upper bounds of the stack size in a subset of all possible program statements, and are not always easy to obtain. Experimental studies show that their prototype implementation was able to prove stack boundedness of four out of five small programs.

Chatterjee et al. [CMM+04] propose a path-insensitive static analysis for computing an upper bound of Interrupt Calculus programs introduced in [PM02]. In contrast to the type checking method in [PM02], their method can be considered as a type inference algorithm, and is both sound and complete. They consider three categories of Interrupt Calculus programs with increasing sophistication. Computational complexity of these three kinds of programs are studied in [CMM+04] and [LO11], ranging from NL-complete to EXPTIME. However, the analysis requires building the complete state graph of a program, which may suffer from the state explosion problem, whereas the type-based method in [PM02] enables modular checking as only the type signature of each interrupt handler needs to be examined. No experimental studies are performed to examine the scalability of the approach in practice.

It is worth noting that [PM02], [CMM+04] and [LO11] consider situations where reentrant interrupts are allowed. Reentrant interrupts, however, are considered as a bad design of interrupt-driven software [RRW05]. More often they are unintentional, hence highly likely to be incorrect [Reg05]. Without reentrant interrupts, the stack size of interrupt handling is automatically bounded, which is the number of IRQ lines supported by the system. In specialised situations where reentrant interrupts are used, it is possible for the stack to overflow if interrupts are signalled too frequently. Interrupt overload can be prevented by either software or hardware though [RD05].

Instead of analysing assembly code [BDP01] or an intermediate language like the Interrupt Calculus we discussed above, [RRW05] introduces a two-phase static analysis to find an upper bound of stack space usage for object code of interrupt-driven embedded programs. The first phase of their analysis is a context-sensitive, path-insensitive, forward data flow analysis of object code based on abstract interpretation [CC77]. Empirically valid but (in general) theoretically unsound techniques are used to handle issues of indirect branches, recursion, spilling registers to the stack, and indirect stores in their data flow analysis. The second phase is to compute the interrupt-preemption graph that is a weighted, directed graph, where each edge represents a potential preemption by an interrupt handler and edge weights represent stack memory requirements. As in [BDP01], the maximum stack depth can be computed by finding the longest path through the interrupt-preemption graph. There are two assumptions in the second phase: 1) Reentrant interrupts are considered as a poor design of the program and are not allowed; 2) Interrupt handlers do not always return to the interrupted program location. They also investigate function inlining and simulation for eliminating interrupt preemption to reduce the stack usage during program execution. Empirical studies show that their prototype tool is able to compute tight stack usage bounds of embedded applications for Atmel AVR microcontrollers compiled from up to 30,000 lines of C in a reasonable amount of time.

Deadline Analysis

Brylow and Palsberg [BP04], based on previous work [BDP01] described above, propose an approach that combines testing and static analysis to compute the execution time of interrupt-driven programs. Their method uses testing oracles [RAO92] to specify the worst-case execution time (WCET) of certain program paths (e.g. cycles) in the control flow graph, which is verified by testing and simulation. Precision of the control flow graph can be improved by applying a lazy multi-resolution k-CFA analysis [Shi91] to reduce the number of oracles. Then the problem is reduced to a multi-source longest path problem, which can be computed in linear time once the control flow graph has been transformed into an acyclic digraph. The algorithms are implemented in a prototype tool called ZARBI, which is tested on six commercial micro-controller programs of 800 – 1600 lines of assembly code and three small examples with less than 100 lines of code. Their experiments show that the method is quite precise. At most 17 oracles were inserted, which were done by an expert in less than one hour in an interactive style during the analysis. Run-time ranged from less than 2 seconds up to an hour for the largest benchmark, with an average of 15 minutes.

Kotker et al. [KSS11] describe an approach that, assuming the semantics of prioritized, preemptive scheduling policy, reduces an interrupt-driven program with bounded context switches to a sequential program similar to [LR09]. It then predicts the execution time of *all* paths of the sequential program by analysing only a subset of its paths [SR12, SR08], whereas the method in [BP04] above uses testing oracles to approximate the worst-case execution time of certain program fragments. In their empirical studies, the method was able to accurately predict the execution time of interrupt-driven programs of a few hundred lines of C code in a couple of context switches on a 32-bit ARM Cortex M3 micro-controller, with the error margin between predicted and measured value less than 5%.

Formal Methods for Interrupt-Driven Software

Interrupt-driven software is often low-level system code, interacting with the operating system kernel and the hardware. It is difficult to write and tends to be buggy. As interrupts can happen at arbitrary times, bugs are extremely hard to detect and reproduce by testing. In this section, we survey various techniques for verification of interrupt-driven software. These include model checking, static analysis, inference-based methods, partial order reduction, and program transformation.

Model Checking

Ball et al. [BBC+06] design and implement a tool called Static Driver Verifier tool (SDV) that verifies temporal safety properties [Lam77] of Windows Device Drivers. There are three key components in SDV. The first one is a precise model of the environment in the form of a C program that consists of two parts: 1) the stub code that provides the semantics for the kernel API that the driver may call. 2) the test harness simulating the operating system invoking the device driver. Non-determinism is introduced to the environment model that simulates the interaction between the kernel and the driver in various ways to achieve a high coverage of real-world scenarios. The second key component is a set of over 60 API usage rules written in a C-like specification language called SLIC [BR02b], which specify the kinds of behaviours that device drivers are expected to have, ranging from "simple locking properties to complex properties dealing with completion routines, plug-and-play, and power management". These rules are checked against the device driver and the environment model for compliance by the verification engine called SLAM [BR02a], which is the third key component of the tool. SLAM uses Boolean programs [BR00] as an abstraction for sequential C programs and the counterexample-guided abstraction refinement loop (CEGAR) [CGJ+00] to *automatically* refine the abstraction until it

is precise enough to prove the program correct with respect to the API usage rules or find a real bug. Experimental studies show that SDV is able to find multiple errors in almost every device driver (in total 146 windows device drivers were checked) with about 25% false alarms. Despite the effectiveness of device driver verification, as other static analysis and verification tools, there are limitations in SDV. For example, all of the false alarms are due to inaccuracy of either the API usage rules or the environment model. It can only scale up to tens of thousands lines of code. It does not check memory safety, but assumes it. And it does not support concurrency.

In [BBKL10] Ball et al. report on improvements in the SLAM verification engine [BBC+06] to reduce the false alarms: 1. a more precise predicate transformer over statement sequences; 2. a combination of backward and forward symbolic execution to detect inconsistencies in various places of a counterexample trace, which is, comparing with SLAM, more flexible and reduces the chance for the refinement loop to get stuck. Besides these two major improvements, the new verification engine SLAM2 has a more efficient pointer analysis and encoding of predicates, as well as a new set of axioms for the memory model [BBLdM10], and new state-of-the-art SMT solver Z3 [dMB08]. Empirical results show that SLAM2 is able to reduce the percentage of false alarms from 25% in SLAM to under 4% for the Windows Driver Model (WDM) class of device drivers.

Witkowski et al. [WBKW07] present a tool DDVerify that integrates the model checkers Cadence SMV [McM92] and BOPPO [CKS05] into the predicate abstraction-based verification tool SatAbs [CKSY05]. Their tool is also able to generate an accurate sequential and concurrent model of the relevant parts of the Linux kernel, enabling automated verification of Linux device drivers and shared-memory concurrent programs. Experimental studies report two previously unknown bugs in the Linux device drivers, confirming the results of the SLAM project [BBC+06, BBKL10] in the Linux world. However, when introducing additional threads, the DDVerify

framework, as most model-checking based approaches, suffers from the state-space explosion problem. A concurrent model with two threads resulted in a significant increase in the number of predicates required, yielding time-outs for most of the claims.

Li et al. [LRJ13] propose the use of formal verification techniques in medical device software. Similar to Bucur and Kwiatkowska [BK11], they translate embedded code for the MSP430 microcontroller into standard C, which can be verified by CBMC [CKL04]. Then interrupt handlers are called after each basic block of the translated program. Nested interrupts are not allowed.

Vörtler et al. [VHHK15] propose a verification framework for applications in the embedded operating system Contiki, and present a new modelling approach for periodically occurring interrupts that cannot reply on interrupt reduction such as partial-order reduction. In the experiments, they compare different interrupt modelling approaches on three Contiki applications using CBMC.

Static Analysis

Beside model checking, recently there has been advance on verification of device drivers using static analysis. [NRTT09] presents a tool called Yogi that checks temporal safety properties of C programs by combing static analysis and testing. Built on the Synergy algorithm [GHK+06], the Dash algorithm [BNRS08] that Yogi implements performs testing and program abstractions simultaneously. Tests are used to detect bugs and refine program abstractions, which are then used to prove program correctness. Experimental results on Windows Device Drivers show that Yogi scales much better than SLAM [BBC+06] and is comparable to SLAM2 [BBKL10] in terms of precision.

Improvements have been made since. The Smash algorithm described in [GNRT10] uses a compositional analysis in which summaries of a procedure call or test genera-

tion and abstraction construction are computed, stored and can be re-used in other calling contexts. On top of this, [NR10] describes several optimizations such as initial abstraction using predicates from the safety properties, heuristics to generate predicates for refinement, and interprocedural analysis using global modification analysis and summaries. In [AKNR12], Map-Reduce style parallelism is applied to the top-down interprocedural analysis of a test suite consisting of 45 Microsoft Windows device drivers and 150 safety properties, gaining an average speedup of 3.71x and a maximum speedup of 7.4x over a sequential Dash algorithm.

All techniques and tools discussed so far have focused on verification of temporal safety properties in device drivers. For example, SLAM cannot prove properties that depend on establishing invariants of heap data structures [BLR11]. Hence SLAM does not check memory safety, but assumes it [BBC+06]. Yang et al. [YLB+08] develop an efficient yet precise join operator (in the context of abstract interpretation [CC77]). Together with a new shape analysis that uses and synthesises a restricted subset of higher-order Separation Logic [BBT05] to describe composite data structures (i.e. as cyclic doubly-linked lists) that are commonly found in system code [BCC+07], and a version of the Reps-Horowitz-Sagiv algorithm with localization [GBC06] to perform a whole-program interprocedural analysis, they were able to, for the first time, perform automatic shape analysis of whole industrial programs such as Windows and Linux device drivers (10K lines of code). Berdine et al. [BCI11] describe SLAyer, a *fully automatic* static analysis tool designed to prove the absence of memory safety errors such as double frees, dereference of dangling pointers, and memory leaks. It is able to verify memory safety of system code written in C of moderate size (10K – 30K lines of code).

Ouadjaout et al. [OMLB16] present a static analysis based on abstract interpretation to verify device drivers of the TinyOS operating system. The proposed method can prove absence of interrupt-related and hardware-software interaction bugs, with

several partitioning techniques to help improve precision of their analysis. They compare their tool SADA with i-CBMC [KLM+15] that implements a partial-order encoding (Chapter 4) on their TinyOS 2.x benchmarks. The runtime and memory consumptions of SADA were lower due to their abstraction techniques, whereas i-CBMC is based on CBMC that implements bit-precise bounded model checking. The major drawback of SADA is false alarms. i-CBMC also produced false alarms because of the imprecise modelling of the hardware in some of the benchmarks. In several cases, i-CBMC missed bugs due to the fixed loop unwinding depth.

Miné [Min17] describes how to apply abstract interpretation, based on a denotational semantics in thread-modular form, to obtain an effective static analyzer to detect run-time errors, data-races, as well as deadlocks in programs with dynamic priority and priority ceiling protocol. They implement their technique inside the Astrée analyzer and perform preliminary experiments on small programs.

Sung et al. [SKW] propose an abstract interpretation framework for static verification of interrupt-driven programs. Their technique first analyzes each interrupt handler in isolation and then propagate the results to other handlers until a fixed point is reached. They implement their method in a tool called IntAbs and compare it with i-CBMC [KLM+15] on 35 interrupt-driven applications with a total of 22,541 lines of code. Their tool is able to prove more properties but produces more false alarms with longer runtime.

Schwarz et al. [SSVA14] derive a static analysis technique to detect data races in programs with multiple interrupts of static priorities. Their analysis focuses on programs that use a restricted set of synchronization patterns with global flag variables. The experimental results show that the technique is effective in reducing the number of race warnings.

Formal verification has started to make its way into real-world practice of verifying large non-trivial code bases. Calcagno et al. [CDD+15] describe their experience of

integrating a formal verification tool based on static analysis into Facebook's software development cycle. Malacaria et al. [MTD16] apply CBMC to analyze information leakage of the OpenSSL implementation.

Inference-Based Methods

Suenaga and Kobayashi [SK07] introduce an ML-style calculus equipped with primitives for threads and interrupt handling as "the first step towards the goal of establishing a type-based method for verification of certain critical properties (such as deadlock and race-freedom) of operating system kernels". A type system is proposed to guarantee the absence of deadlocks of programs written in the language. Nevertheless, it does not support records, arrays and pointer arithmetic that are regularly found in operating system (OS) kernels. Although the authors claimed that their type system was "the first type system for deadlock-freedom that can deal with both threads and interrupt primitives", it is not clear how to adopt their method to verify real-world programs.

Feng et al. [FSDG08] present a Hoare Logic–like framework to verify low-level system code that has both preempted threads and hardware interrupts. By using memory ownership-transfer semantics to formalise the interaction between interrupt handlers and threads, they were able to—for the first time—certify a preemptive thread implementation and a large number of common synchronization primitives. In their implementation, the program logic is adopted for the 16-bit real-mode x86 architecture, and a subset of the x86 assembly language is formalised in the Coq proof assistant [Coq]. However, the instruction set for this architecture is rarely used in real-world systems these days. Also, although the size of the assembly source code is about 300 lines, the whole proofs contain more than 80,000 lines of code, and are done manually. As different implementations would need different Coq proofs, to make their method scalable, an automatic technique is needed.

Klein et al. [KEH+09] describe their experience in performing machine-checkable proof of the seL4 microkernel [EDE08] for functional correctness. The proof is done in Isabelle/HOL [NPW02] by using refinement [RE08]: checking correspondence between an abstract specification and an executable specification written in Haskell, and the Haskell executable specification and the final implementation in C. seL4 is the first complete, general-purpose operating system kernel that has been fully verified against a high-level specification of kernel behaviour. To reduce verification effort, effects of interrupts are minimised by disabling them during kernel execution, except for a few carefully-placed program points modelled by polling. The kernel only contains a single device driver so nested interrupt is not supported. Concurrent execution of the kernel is completely avoided. A more fundamental issue is how to control the cost of proof maintenance. While seL4 contains 8700 lines of C code and 600 lines of code in assembly and the execution specification has 5700 lines of Haskell code, the Isabelle code is more than twice larger (32900 lines). It is reported that a fundamental change to existing features took about 1 man-year or 17% of the original proof effort to re-verify, although the code size of this change was small (less than 5% of the total code base). It is an open problem how to synthesise proofs *automatically* from code and specifications of large-scale real-world systems.

Closely Related Work

Program Transformation

There has been much research on tools to verify assertion-based properties in sequential (e.g. [CKL04], [CKSY05], and [CDE08]) and multithreaded code (e.g. [AKT13], [SW10], and [SW11]). These tools can be used to verify interrupt-driven programs if a semantics-preserving source-to-source transformation is possible. In this section, we briefly survey program translation techniques that could be useful for interrupt-driven

software verification.

Kidd et al. [KJV10] introduce a program transformation that translates a multi-threaded program into a sequential program by encoding a thread scheduler, a call to which is inserted between each instruction of the original program. A correctness proof based on a reduction from concurrent pushdown system [QR05] to a single pushdown system is given. Their method applies to only threads with fixed priorities, hence is also applicable to interrupt-driven programs with fixed interrupt priorities. The key idea is to use non-determinism in the scheduler to decide in each program location whether to perform a context-switch as well as which thread to switch to taking into account the priority of the running thread. The scheduler runs for a "hyperperiod" in which every interrupt must happen exactly once, which is an under-approximation of all possible situations, e.g. the same thread can not be executed consecutively. Moreover, calling the scheduler between every single instruction is likely to cause the state space to explode at the verification stage (although no experimental studies are reported). In Section 3.2, we describe a program transformation that translates a program with interrupts into a sequential program, similar to the approach of Kidd et al. [KJV10]. Our transformation combines generalisation through a scheduling function that models potentially nested arrivals of diverse interrupts with an optimization that strongly resembles partial-order reduction to verify low-level software with *nested* interrupts. We also provide experimental results in Chapter 5.

Regehr and Cooprider [RC07] suggest a source-to-source transformation that turns interrupt-driven code into multi-threaded code by assigning a fixed priority to a thread when it is created. Locks are used in each thread to prevent interrupt re-entrance. During program execution, threads are scheduled according to their predefined priorities. Hence the possible interleavings of the resulting code are the same as the original interrupt-driven code. This approach, however, requires thread-based verification tools to be able to model the scheduler of operating system kernels. And

no experimental studies are performed. In Section 3.3, we describe in detail a variant of their approach in which the thread priorities are enforced by adding further instrumentation. Experimental results of this approach are reported in Chapter 5.

Wu et al. [WWC+13] describe a translation from programs with nested interrupts into sequential code, which makes the conservative assumption that interrupts may arrive after every instruction. This approach suffers from state explosion, which in our translation is addressed by an optimization that greatly reduces the number of calls to interrupt handlers.

Wu et al. [WCM+16] present a numerical static analysis technique based on sequentialization for embedded software with interrupts. They also design specific abstract domains to analyze numerical properties of the sequentialized interrupt-driven programs to improve precision and scalability.

Liu et al. [LJZ+16] describe a sequentialisation framework for verifying industrial Multifunction Vehicle Bus controller (MVBC) systems. It uses a happen-before interrupt graph to model interrupt dependency and reduce false alarms. The resulting sequential program is verified by CBMC. They have applied their framework to an industrial MVBC product and found three previously unknown bugs.

Partial-Order Reduction

Interrupts can arrive at arbitrary time. Interrupts are usually modelled by calling handler functions at program points where the corresponding interrupts may happen. Partial order reduction is an effective technique to reduce calls to interrupt handlers while preserving the verification results.

Schlich et al. [SNBB09] introduce an abstraction technique for microcontroller assembly code called interrupt handler execution reduction (IHER) based on partial order reduction [God94, Pel93] that reduces the number of program locations where an interrupt handler needs to be executed. They are able to prove the abstraction

preserves the validity of CTL*-X formulas [Eme91]. Their empirical studies use a benchmark that consists of five programs with 200 – 400 lines of code written by students during lab sessions or theses, and shows that the IHER technique greatly reduces state spaces for a number of different programs, from 64% to 98% depending on "the number of interrupts used, dependencies between interrupt handlers and instructions, dependencies between interrupt handlers, the overall structure of the program, and the property to be verified" [SNBB09].

Bucur and Kwiatkowska [BK11] present a platform-dependent, OS-independent software verification tool for OS-wide programs written in embedded C with asynchronous hardware interrupts. Their tool automatically translates MSP430 embedded C programs into standard C by modelling the MCU memory map and direct memory access, and adds calls to interrupt handlers to emulate arrivals of hardware interrupts. As [SNBB09], it also uses partial order reduction to reduce the number of calls to an interrupt handler. But their method does not consider the invisibility and cycle conditions [CGP01], hence only considers an under-approximation of the program state space. Also, the translated C programs are verified by CBMC [CKL04] to which bounds of loops are manually given, which can be time consuming. Moreover, although it is natural to use non-determinism to model the hardware, the range of non-deterministic values is platform-dependent, and is "strictly an issue of knowledge of the platform workings" [BK11]. Hence, it would require some work to port the tools to different platforms.

Symbolic Execution

In Chapter 1, we mention that interleavings between threads and interrupts are quite subtle. In this section, we review recent advances on symbolic encoding for concurrent software verification. We then discuss, in Chapter 4, how to extend symbolic encoding of concurrent software to verify programs with nested interrupts.

The C Bounded Model Checker (CBMC)[1] is a program analyzer that implements bit-precise bounded model checking for C programs [CKL04]. It can show violation of assertions in C programs, or prove their safety under a given loop unwinding bound. CBMC first unwinds loops of an input C program into a given depth. It then translates the unwound program into a formula, which is passed to a modern SAT or SMT solver together with a constraint that specifies a set of error states. If the solver determines the formula to be satisfiable, an error trace is extracted from the satisfying assignment. More information about CBMC's encoding can be found in Section 4.2. CBMC supports verification of concurrent programs over a range of memory models, including SC, TSO, and PSO [AKT13].

Alglave et al. [AKT13], based on the framework of [AMSS12], introduces a symbolic partial-order encoding of concurrent programs that supports SC as well as weaker memory models such as Intel's x86 and IBM's Power, hence generalising [SW10] and [SW11] at large. Their encoding consists of two parts: the conjunct `ssa` encodes the data and control structure in a static single assignment (SSA) [CKY03] form with fresh indices introduced for each occurrence of shared variable (similar to [WCGY09] and [SW11]); the second conjunct `pord` encodes the communications between threads, e.g. linking shared reads to appropriate writes in different memory models including SC. The experimental results show that their prototype implementation based on CBMC [CKL04] scales to programs of sufficient size (up to 28000 lines of C code) to achieve first-time formal verification of non-trivial concurrent systems code, whereas most tools fail to parse the examples or abort with other errors. By contrast, the encoding described in Chapter 4 is a *strengthening* of SC. We implemented our encoding in a tool called i-CBMC, which is also based on CBMC.

Cordeiro and Fischer [CF10] describe three approaches based on bounded model checking and Satisfiability Modulo Theories (SMT) to verify multi-threaded software.

[1] `http://www.cprover.org/cbmc/`

The lazy approach generates all possible interleavings of a multi-threaded program and explores each one of them individually. The schedule recording approach encodes all possible interleavings into a single formula and leverage the performance of modern SMT solvers. The underapproximation and widening approach abstracts the number of interleavings from the unsatisfiable proofs generated by SMT solvers. They also combine these techniques with context-bounded model checking to verify non-trivial multi-threaded programs in C [CF11]. They implement these techniques into model checker ESBMC [Cor10] and their experiments show that ESBMC is more scalable and efficient than state-of-the-art techniques on benchmarks with up to 6,300 lines of code.

Sinha and Wang [SW10] propose a two-stage analysis that separates the intra- and inter-thread reasoning for checking assertion violations and other concurrency errors in concurrent programs. The first stage unrolls loops and recursions to a fixed depth and then codes the data and control structure of each thread, which introduces a fresh symbolic variable for read access to a shared memory location. The second stage links global read accesses with appropriate writes by using the sequential consistency (SC) axioms [AG96] specified in first-order logic, followed by checking violations using an SMT solver. Optimizations are also applied to prune redundant constraints in their symbolic encoding. Experimental results show that implementation of the method in the FUSION verification system [WCGY09] is relatively scalable.

Sinha and Wang [SW11] introduce an automatic refinement procedure to systematically under-approximate the encoding of concurrent programs in their earlier work [SW10] for finding a violating witness and over-approximate it for proving (safety) properties. As [SW10], the algorithm is implemented in the FUSION system [WCGY09]. Experiments show that small encodings for over-approximation are often sufficient to check properties in their benchmarks, and significantly improve the verification time.

The encoding in [AKT13] only relates memory events that access the same memory locations, thus is cubic in the maximum number M of access to a *single* memory as opposed to *all* memory access N, which is the case in [SW10] and [SW11]. In their experimental studies, "N on average was 5.7 times larger than M, which, on average, extrapolates to an overall difference of more than two orders of magnitude in the size of the formula" [AKT13]. In addition, the partial-order encoding does not build transitive closure, whereas [SW10] and [SW11] do, although [SW10] has an optimized step to prune redundant constraints. A novelty of [SW11] is a refinement procedure to systematically over- and under-approximate the symbolic encoding to reduce verification time. Finally, it is worth mentioning that the partial-order encoding in [AKT13] is significantly different from partial order reduction described in the previous section, which considers total orders as primary, and uses partial orders to reduce the size of the state space to explore.

Static Single Assignment Form

Translation from control-flow graph to static single assignment form (SSA) in verification tools [CKY03, AKT13] is not new. It has also been commonly found in compilers [AWZ88, RWZ88, CFR+89, CFR+91, BM94, GSW95, BCHS98, SS07, LLV].

Alpern et al. [AWZ88] describe an efficient algorithm to statically detect equality of variables in a program. Their algorithm translates a program's control-flow graph into SSA for later construction and analysis.

Rosen et al. [RWZ88] propose an algorithm to identify expressions that are lexically different but certain to compute the same value. Their algorithm takes advantage of the SSA form which provides further opportunities to identify equivalence among expressions that would not be lexically identical in the original program.

Gerlek et al. [GSW95] introduce a method based on a demand-driven representation of an SSA form for detecting a board class of linear induction variables and

several other sequence forms, which is useful for strength reduction in compiler optimization.

Much research has been focused on how to compute SSA form efficiently. For example, Cytron et al. [CFR+89, CFR+91] propose an algorithm to efficiently compute SSA form for arbitrary control-flow graphs. They also provide analytical and experimental evidence that the resulting SSA is usually linear in the size of the original program, which can be of practical use in compiler optimization.

Brandis and Mössenböck [BM94] present a single-pass algorithm to generate SSA form of a program without goto statements. They further conclude that there is no reason for compilers of such languages to build other internal representations such as abstract syntax tree.

Briggs el al. [BCHS98] discuss different forms of SSA, and present two improvements to the SSA construction algorithm by Cytron et al. [CFR+91]. They also consider the problem of converting SSA form back into executable code.

The modern compiler infrastructure LLVM [LLV] also uses a SSA form as its internal representation (LLVM IR) to facilitate program analysis and optimization.

Validation of Read-Copy Update

Read-Copy Update (RCU) is a high-performance scalable synchronization mechanism that allows extremely low-overhead readers to run concurrently with updaters [MS98]. Production-quality RCU implementations in the Linux kernel make heavy use of interrupts and are highly sophisticated to deal with Linux kernel's diverse workloads and platforms [MW08]. Formal verification has been applied to various aspects of RCU design and implementation.

McKenney has applied the SPIN model checker to verify RCU's `NO_HZ_FULL_SYSIDLE` functionality [McK15], and interactions between dyntick-idle and non-maskable interrupts [McK08a]. Desnoyers et al. [DMD13] propose a virtual architecture to

model out-of-order memory accesses and instruction scheduling. User-level RCU [DMS+12] is modeled and verified in the proposed architecture using the SPIN model checker. These efforts require an error-prone translation from C to SPIN's modelling language, and therefore are not appropriate for regression testing.

Alglave et al. [AKT13] introduce a symbolic encoding for verifying concurrent software over a range of memory models including SC, TSO and PSO. They implement the encoding in the CBMC bounded model checker [CKL04] and use the tool to verify `rcu_assign_pointer()` and `rcu_dereference()`. McKenney has used CBMC to verify Tiny RCU [McK15], a trivial Linux-kernel RCU implementation for uni-core systems. Roy has applied the same tool to verify a significant portion of Sleepable RCU (SRCU). CBMC is now part of the regression test suite of Classic SRCU in the Linux kernel [Roy17].

Kokologiannakis et al. apply Nidhugg to verify Tree RCU [KS17]. Since Nidhugg has better list support and does not model data non-determinism, they are able to verify more scenarios with less CPU and memory consumption. But some portions of RCU use atomic read-modify-write operations that can give nondeterministic results. We therefore hypothesise that data non-determinism will be required to verify RCU' s dyntick-idle and `rcu_barrier()` components.

Groce et al. [GAJM15] introduce a falsification-driven verification methodology based on mutation testing. Using CBMC, they are able to find two holes in `rcutorture`, RCU's stress testing suite, one of which was hiding a real bug in Tiny RCU. Further work on real hardware has identified two more `rcutorture` holes: one was hiding a real bug in Tasks RCU [Cor14] and the other was hiding a minor performance bug in Tree RCU.

Gotsman et al. [GRY13] use concurrent separation logic extended with temporal operators to formalize grace periods and prove an abstract implementation of RCU over SC. Tassarotti et al. [TDV15] use the GPS program logic to verify a simple

implementation of user-level RCU for a singly-linked list. They assume "release-acquire" semantics, which is weaker than SC but stronger than memory models used by real-world RCU implementations. These proofs are done manually on simple RCU implementations.

In Chapter 6, we describe our experience of constructing a model from the source code of Tree RCU in the Linux kernel, and apply CBMC to verify its grace-period property over SC, TSO, and PSO.

Chapter 3

Program Transformations for Nested Interrupts

Interrupt-driven software is difficult to engineer and debug, since interrupts can arrive at arbitrary times, resulting in an exponential number of possible cases to consider. The situation is even worse when there are multiple kinds of interrupts whose executions can be nested.

This chapter describes our first approach based on program transformations for verifying software with nested interrupts. First, Section 3.1 introduces the interrupt semantics on which the program transformations in this chapter and the partial-order encoding in the next chapter are based. We then explain the subtle interleaving semantics of nested execution of interrupts. Finally, we describe in detail two program transformation techniques that translate a program with nested interrupts into a sequential and a multi-threaded program in Sections 3.2 and 3.3, respectively.

3.1 Nested Interrupts

Interrupt executions are platform-dependent. On some microcontrollers (e.g. ARM Cortex-M, AT89CXX), priorities or priority groups can be configured, whereas oth-

ers use fixed priorities (e.g. AVR1305). There are architectures where lower-priority interrupts, when enabled, can preempt higher-priority ones (e.g. ATMEL256). The primary way to implement priorities is by enabling and disabling interrupts appropriately, e.g. by means of API functions such as AVR's `sei` and `cli` instructions.

In this thesis, we use the following setting on which the techniques described in Chapter 3 and Chapter 4 are based. When entering an interrupt service routine (ISR), all interrupts are masked by the hardware and thus ignored. The ISR can then change which specific interrupts are globally enabled or disabled by issuing a command, e.g. by setting particular bits in a global control register. The interrupt mask can then be lifted, allowing whatever interrupts are enabled at that point to subsequently preempt the execution of the ISR. We assume the number of interrupt arrivals during a program execution is fixed and an interrupt can arrive at any program point at which it is enabled. If multiple interrupts arrive at the same time, the order of handling them is non-deterministic.

As an example consider Figure 3.1, where interrupt a is preempted by interrupts b and c, which is in turn preempted by interrupt d. The state recording which interrupts are enabled and which are disabled is global, and so any changes made by an ISR to this state remain in force when it completes execution. Interrupt re-entrance is not allowed.

Note that interrupts with priorities are a special case of this setting, where the interrupts enabled during execution of an ISR are only those with higher priority. The handling of lower-priority interrupts is thus suspended until the handling of those with higher priority is completed. So, in principle, our setting covers the vast majority of microcontrollers. For the modelling approaches we discuss later in this thesis, however, we propose more direct ways in which priorities can be handled.

We illustrate the execution of programs with nested interrupts through the simple code fragment in Figure 3.2. In this example, the interrupt handler `ISR_A` enables the

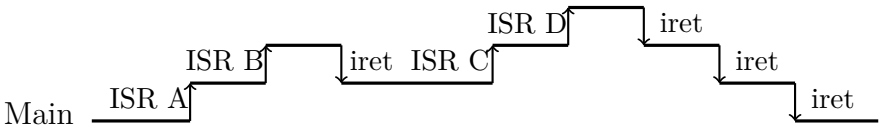


Figure 3.1: A complex scenario of nested interrupts

interrupt handled by `ISR_B`. This permits the execution of `ISR_A` to be preempted by an interrupt that is then serviced by `ISR_B`.

```
irqreturn_t ISR_A(...)
{
  // start execution
  // enable interrupt B
  ...
  y = 1;
  ...
  ...
  return IRQ_HANDLED;
}
```

```
irqreturn_t ISR_B(...)
{
  ...
  y = 2;
  x = 3 * y;
  assert(x == 6); ✓
  ...
  return IRQ_HANDLED;
}
```

Figure 3.2: Motivating example

In this work, we focus on verification of safety properties, such as user-defined assertions. An example occurs in the code of `ISR_B` in Figure 3.2. Suppose interrupt `ISR_A` enables interrupt 2, as shown, and interrupt `ISR_B` does not enable any interrupts. Then the property `(x == 6)` in `ISR_B` always holds because the execution of `ISR_B` cannot be interrupted. But if interrupts are enabled incorrectly—as depicted in Figure 3.3—the assertion `(x == 6)` will be violated. The ISR `ISR_B` can be interrupted and the invoked ISR can change the value of `y` just before the assignment to `x` in `ISR_B` is executed.

We note that an interleaving semantics in which ISRs are treated as threads with arbitrary interleaving semantics would admit an assertion-violating execution in the example of Fig. 3.2. This is because in a multi-threaded program, between statements `y = 2` and `x = 3 * y` of `ISR_B`, the control can return to `ISR_A` which then updates the value of `y`. Consequently, tools for multi-threaded software verification, naïvely

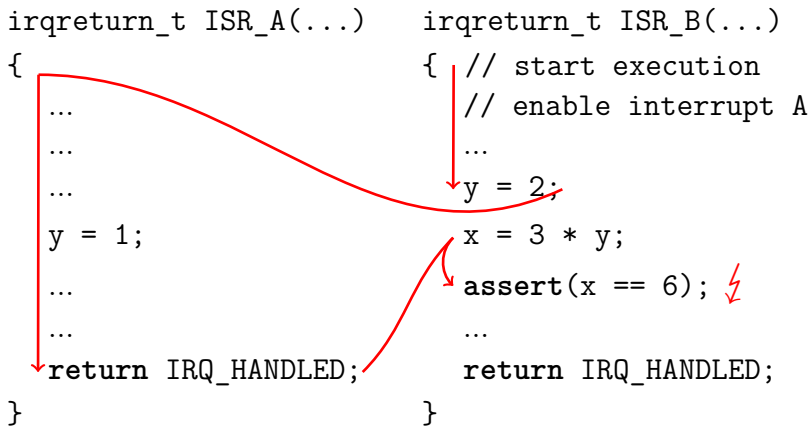


Figure 3.3: Motivating example (modified)

applied, can report an assertion violation that is not possible in any actual execution of the program. This motivates the need for techniques that model the execution of interrupts more precisely than threads do. The program analysis principles discussed in this paper for interrupt handling also apply to the analysis of prioritised, preemptive task scheduling policies.

3.2 Interrupt Sequentialisation

The first conventional approach we compare our new method with is a source-to-source transformation that translates a program with interrupts into a sequential program, similar to the approach proposed by Kidd et al. [KJV10]. This allows us to analyze interrupt-driven code by leveraging existing verification tools for sequential programs.

Interrupts can happen at any program point where they are enabled. An intuitively simple way to model this as a sequential program is to insert a *non-deterministic call* to the handler of an interrupt wherever it is enabled. Semantically, a non-deterministic call is a function call that—independently—either is or is not invoked each time program control passes through the point at which it occurs. This models the contingent arrival, at that point of program execution, of interrupts that are

generated externally.

We extend this idea to encompass the arrival and handling of nested interrupts. In this case, the transformation is realized by the insertion of (unconditional) calls to a certain *scheduling function*, instead of to a fixed interrupt handler. The scheduling function non-deterministically invokes handlers for the interrupts that are enabled. An optimization resembling partial-order reduction is then used to reduce the number of calls to the scheduling function. This retains an invocation of the scheduler only in certain program locations, namely just before any program statement that has potential run-time interactions with interrupts through shared variables.

The basic idea is well-known [BK11]. To the best of our knowledge, however, our transformation is the first to combine generalisation through a scheduling function that models potentially nested arrivals of diverse interrupts with an optimization similar to partial-order reduction to verify low-level software with *nested* interrupts. We also provide experimental results (Chapter 5).

3.2.1 Modelling Nested Interrupts using a Scheduler

We define a scheduling function together with two global variables: `count` is a bound on the number of interrupt arrivals that can occur during program execution, and the Boolean `irq_enabled[i]` indicates whether interrupt i is currently enabled. At the beginning of the interrupt-driven code, `count` is set to a positive integer and the array `irq_enabled` is initialized to zeros. Then `irq_enabled[i]` is set to 1 right after any statement in the original code that explicitly enables interrupt i.

The scheduling function is implemented as follows:

```
void schedule_irq(void)
{
  int j = count;
  int irq;

  for (int i = 0; i < j; i++)
```

```
{
  // non−deterministically generate candidate irq
  irq = nondet_int() ;
  assume(irq >= 0 && irq < num_irqs);

  // if irq is enabled, contingently generate the interrupt
  if (count != 0 && irq_enabled[irq] && nondet_bool())
  {
    count−−;

    // invoke the ISR, preempting the calling program
    switch (irq)
    {
      case 0: ISR_0(); break;
      case 1: ISR_1(); break;
      ...
      default: ;
    }
  }
 }
}
```

The scheduling function works as follows. First, if `count` is already zero the scheduler does nothing and exits. Otherwise, up to `count` interrupts might still occur. To decide which ones do occur, at this point of program execution, the scheduler loops `count` times and, in each iteration, makes a non-deterministic choice among all possible interrupts. If the chosen interrupt is enabled, according to `irq_enabled`, and `count` has not reached zero, the scheduler makes a non-deterministic choice between doing nothing (to simulate that this interrupt does not occur) and calling the corresponding interrupt handler, decrementing `count`. The variable `count` is checked in the conditional inside the loop because the ISRs may themselves be interrupted and so have invocations of the scheduler within them that also decrement `count`.

In the special case where interrupt priorities are explicit, a global array `irq_pty` is used to store the priority of each interrupt. When the scheduler generates a candidate interrupt, it is considered further only if its priority is strictly higher than the currently-executing interrupt (if any) stored in a global variable `curr_pty`. Then if the scheduling function decides to invoke an interrupt handler, it stores the cur-

rent interrupt priority in a variable local to the scheduler, and updates `curr_pty` to the priority of the interrupt being handled, in the same way as Kidd et al. [KJV10]. Then it can restore `curr_pty` to the previous priority upon return from the invoked handler.

Although in our experiments we bound the number of interrupt arrivals, the sequential translation can be extended to handle an unlimited number of interrupt arrivals. To do so, we eliminate the `count` variable and replace the main loop with an infinite one. Each time through the loop, the scheduler makes a non-deterministic choice between exiting and generating another candidate interrupt. The resulting code might then be analyzed by an unbounded model checker.

3.2.2 Reducing Scheduler Calls

In the naïve approach, interrupt arrivals are modelled by inserting a call to the scheduler at every boundary between instructions in the program to be translated, including the code of each of its interrupt handlers. But this will almost certainly make formal analysis computationally intractable. We therefore use an optimization that strongly resembles *partial-order reduction* [God94, GW91, KP88, Pel94, CGP01] to lessen the number of calls to our scheduling function, while preserving the semantics given by the naively transformed program.

Background on Partial-Order Reduction

The idea of partial-order reduction is to reduce the number of permutations of a sequence of individual state transitions that need to be considered, when the order of the transitions does not affect the result of formal analysis. It proceeds by eliminating some of these transitions by observing that some pairs of transitions are independent, in the sense that the order in which they are executed does not matter—and so one order may arbitrarily be chosen over all others, eliminating these other orderings from

the analysis.

We denote the set of states and transitions of a concurrent system by S and Γ, respectively. For transition $t \in \Gamma$, t is a partial function from S to S. We use $s \xrightarrow{t_0 \ldots t_n} s'$ to denote a sequence of transitions $t_0 \ldots t_n$ from state s to s'. For state $s \in S$, we define $enabled(s) \subseteq \Gamma$ to be the set of enabled transitions in state s, i.e. $\{t \in \Gamma \mid t(s) \text{ is defined}\}$.

Definition. Two transitions t_1 and t_2 are *independent* if for all $s \in S$, the following hold:

1. if $t_1 \in enabled(s)$ and $s \xrightarrow{t_1} s'$ (i.e. $t_1(s) = s'$), then $t_2 \in enabled(s)$ if and only if $t_2 \in enabled(s')$;

2. if $t_1, t_2 \in enabled(s)$, then there is a unique state s' such that $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$

The first condition implies that independent transitions can neither enable nor disable each other. The second condition indicates that enabled independent transitions commute.

Definition. For state $s \in S$, a set of transitions $T \subseteq enabled(s)$ is *persistent* in s if for all nonempty sequences of transitions $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots s_{n-1} \xrightarrow{t_n} s_n$ from s using only transitions $t_i \notin T$, $1 \leq i \leq n$, t_i and t are independent for all $t \in T$.

Applying Partial-Order Reduction

In a state of an interrupt-driven program, there are two kinds of deterministic transitions: the next program statement to execute, and a call to the ISR for an enabled interrupt. We denote the transition for the next program statement $stmt_s$ in state s by t_{stmt_s}. For programs with nested interrupts, there are multiple ISRs that can possibly be invoked in a program state. Let $ISR(s)$ be the set of all possible nested executions of ISRs in state s, each of which is represented as a transition. The reason we can model a nested execution of ISRs as a single transition is because the next

program statement will not be executed until after the execution of nested ISRs returns. Thus, we have *enabled*(s) is equal to $\{t_{stmt_s}\} \bigcup ISR(s)$. For $t \in ISR(s)$, we define *stmts*(t) to be the set of program statements in the execution represented by t. Then t and t_{stmt_s} are independent if the following hold:

1. $stmt_s$ neither enables nor disables any interrupts; and

2. there does not exist a statement *stmt* in *stmts*(t) s.t. $stmt_s$ and *stmt* share a program variable that is written by one of them.

In our transformation, the set of transitions $ISR(s)$ is represented by the scheduling function that uses nondeterminism to model all possible executions of ISRs. Then $\{t_{stmt_s}\}$ is a persistent set in state s if the following holds:

1. $stmt_s$ neither enables nor disables any interrupts; and

2. none of the enabled ISRs has a program statement *stmt* that contains a shared variable written by $stmt_s$ or *stmt*. That is, for all $t \in ISR(s)$, t and t_{stmt_s} are independent.

If a program statement violates the above condition, we explore the whole enabled set by inserting a call to the scheduling function before and after the program statement $stmt_s$.

3.2.3 Example of Interrupt Sequentialisation

We demonstrate this transformation process with an artificial example that has a user application and two ISRs, all of which are loop-free. We assume that `ISR_1` and `ISR_2` have only one shared variable, `y`, and that `ISR_2` enables interrupt 1 throughout its execution.

```
irqreturn_t ISR_1(...)
{
  // all interrupts are masked upon entry of an ISR
   ...

  y = 1;
```

```
  ...

  return IRQ_HANDLED;
}


irqreturn_t ISR_2(...)
{
  // all interrupts are masked upon entry of an ISR
  ...

  // ISR_2 enables interrupt 1
  enable_irq(1);

  ...

  x = 3 * y;
  assert(x == 6);

  ...

  return IRQ_HANDLED;
}
```

First, we define global variables to record the basic parameters of the execution environment of the code to be analyzed:

```
int num_irqs = 2; // no. of ISRs
int count;        // upper bound on no. of interrupt arrivals

// Boolean array to record which interrupts are enabled
bool irq_enabled[] = {1, 1}
```

It remains to decide where to call the scheduling function. As discussed earlier, it suffices to check for each program statement P of the user application and ISRs, whether there exists a statement Q from an enabled ISR such that either P or Q writes a shared variable. In our example, the statement `x = 3 * y` in `ISR_2` reads the shared variable `y`, which is written by `ISR_1`. As we assume `ISR_2` enables interrupt 1, a function call to `schedule_irq()` is inserted before statement `x = 3 * y` in `ISR_2`.

The main function for the translated code has the following structure:

```
int main(void)
{
  // allow up to ten interrupt arrivals
  count = 10;

  // admit all enabled interrupts
  schedule_irq();

  // user application
  ...

  return 0;
}
```

The ISRs are then instrumented with calls to the scheduling function. Owing to our optimization, only two calls are inserted in `ISR_2`. In fact, since the shared variable `y` is not read again after `x = 3 * y`, a call to `schedule_irq()` before `x = 3 * y` is sufficient.

```
irqreturn_t ISR_1(...)
{
  // all interrupts are masked upon entry of an ISR
  ...

  y = 1;

  ...

  return IRQ_HANDLED;
}

irqreturn_t ISR_2(...)
{
  // all interrupts are masked upon entry of an ISR
  ...

  // ISR_2 enables interrupt 1
  enable_irq(1);

  ...

  schedule_irq();
  x = 3 * y;
  schedule_irq(); // this call can be omitted
  assert(x == 6);
```

```
    ...

    return IRQ_HANDLED;
}
```

3.3 Modelling Interrupts with Threads

The second source-to-source transformation we use for comparison generates multi-threaded code and was suggested by Regehr and Cooprider [RC07]. This method verifies program properties for specific, user-defined interrupt arrival scenarios, rather than modelling all possible patterns of interrupt arrival. The method requires us to know which interrupts are enabled at every point of program and ISR execution. We then choose a collection of locations, at each of which we insert some instrumentation that models the potential for a specific enabled interrupt to arrive at that or any subsequent point of program execution. These locations can be in the ISRs as well as in the main program. Interrupt re-entrance is not allowed, but a particular interrupt can arrive and be handled multiple times in succession. Chapter 4 also uses this modelling approach.

This transformation uses threads to model interrupts. The basic setup is as follows:

1. We start the main program as a thread.

2. At any point in the original code where an interrupt is enabled, we may insert code that explicitly spawns a further thread that executes that interrupt's ISR. We endow each such thread with a unique and distinct copy of its ISR function body, so that we can distinguish multiple arrivals of the same interrupt.

3. We instrument the code so that a thread modelling an interrupt arriving and being handled starts executing only if its corresponding interrupt is enabled.

4. Our instrumentation also ensures that execution of an enabled ISR preempts execution of the main program or the ISR that enabled it, but not vice versa.

Using threads this way means that an ISR can start running any time after the thread that models it is spawned, by the interleaving semantics of multi-threaded programs. Note that when we model multiple invocations of the same ISR, we create separate function bodies for each one. This allows them to have different instrumentations.

To ensure that only the threads of enabled ISRs can preempt those of enabling ones or the main program, we assign a *depth* to each thread according to the following rule. If thread j is spawned by thread i, then the depth of thread j is greater than that of thread i. If two threads are spawned by the same ISR thread (or the main program), then their depth is the same. We then instrument the code so that execution of all threads with smaller depth is blocked by any execution of a thread with greater depth.

Just as in the sequential transformation of Section 3.2, since we instrument at the source code level, we treat each program statement as the smallest atomic block in any interleaving of the execution of program threads. So it is an under-approximation of all possible executions. In addition, function calls in the ISRs are inlined, because two ISRs may call the same function and may therefore need different instrumentations. We further assume the existence of a global Boolean array `irq_enabled[i]` that records whether each interrupt i is currently enabled or not. We suppose the state of this array is maintained either by the programming language or explicitly by the programmer. At the beginning of ISR i's function body, we insert the atomic statement below to make sure ISR i starts executing only if interrupt i is enabled.[1]

```
atomic { assume(irq_enabled[i]); }
```

[1]In many verification tools, atomic sections are supported by a pair of constructs `__VERIFIER_atomic_begin()` and `__VERIFIER_atomic_end()` that indicate the beginning and the end of an atomic section, respectively. The atomic assume statement can be realized by using an atomic section and the `__VERIFIER_assume` primitives.

This ensures that a thread modelling an interrupt, spawned at a point in the program where the interrupt is enabled, will not begin execution at some later time if the interrupt has in the meantime been explicitly disabled.

Another global array `thread_running` records the currently executing thread, along with all the threads the executions of which have begun but have been suspended. To do this, the code for each thread i is instrumented with assignment statements that set `thread_running[`i`]` to `1` initially and to `0` when the thread completes. We then insert at every boundary between statements throughout the code for thread i the following atomic statement

`atomic { assume(thread_running[`j_0`]==0 && ... && thread_running[`j_n`]==0); }`

where threads j_0 to j_n are all those threads, excluding thread i itself, the depths of which are greater than or equal to that of thread i. This assume statement ensures that the program statement it precedes is executed only if no thread of greater or equal depth is running—i.e. execution of this ISR has not been interrupted or no other ISRs that cannot be preempted by this ISR are currently running.

With this approach, it is important to note that to model multiple invocations of the same ISR we have to create separate ISR function bodies for each invocation, each with different instrumentations.

To see when we must do this, consider the following example:

```
void* ISR_1(void* arg)
{
  // all interrupts are masked upon entry of an ISR
   ...

  // enable interrupt 2
  enable_irq(2);

   ...
```

```
    // spawn a thread of ISR_2()
    async: ISR_2();

    ...
}

void main()
{
    ...

    // enable interrupts 1 and 2
    enable_irq(1);
    enable_irq(2);

    ...

    // spawn threads of ISRs 1 and 2
    async: ISR_1();
    async: ISR_2();

    ...

}
```

In this example, the execution depth of `ISR_1` is 1. But there are two execution depths for `ISR_2`, since it is called once in the main program with depth 1 and again in `ISR_1` with depth 2. So we need two copies of the function body for `ISR_2` that are instrumented with different depths.

In the special case where interrupts have explicit priorities, we take an approach similar to that explained in Section 3.2. A global array `irq_pty` is used to store the priority of each interrupt. All threads that model an arrival of the same interrupt will have the same priority. We then use a slightly modified form of instrumentation, in which the atomic assume statement is as shown above but threads j_0 to j_n are all the threads, excluding thread i, the *priorities* of which are greater than or equal to that of thread i. That is, we simply replace 'depth' by 'priority'.

With all this instrumentation, the resulting translated program can be checked by verification tools for concurrent programs, such as CBMC [AKT13], ESBMC [Cor10],

or CSeq [ITF+14].

Example of an instrumented ISR is depicted below. The full source code of the examples can be found online at http://www.cprover.org/interrupts/.

```
irqreturn_t ISR_i(param_list)}
{
  atomic { assume(irq_enabled[i]) };
  atomic { assume(thread_running[i_0]==0 && ...
                  thread_running[i_k]==0) };
  atomic { thread_running[i] = 1 };

  /* original code */
  atomic { assume(thread_running[i_0]==0 && ...
                  thread_running[i_k]==0) };
  atomic { statement 1 };

  atomic { assume(thread_running[i_0]==0 && ...
                  thread_running[i_k]==0) };
  atomic { statement 2 };

  ...

  atomic { assume(thread_running[i_0]==0 && ...
                  thread_running[i_k]==0) };
  atomic { thread_running[i] = 0 };
  return IRQ_HANDLED;
}
```

Summary

In this chapter, we describe two source-to-source transformations that translate programs with nested interrupts into sequential and multi-threaded programs, respectively. The main advantage of this approach is that we can apply a wide range of existing model checkers for sequential and concurrent software to verify interrupt-driven programs.

In the following chapter, we will introduce a different technique for verifying software with nested interrupts that is based on symbolic execution. We compare these two approaches experimentally in Chapter 5.

Chapter 4

Partial-Order Encoding for Nested Interrupts

In Chapter 3, we described two program transformations to verifying programs with nested interrupts. In this chapter, we introduce a symbolic execution technique to tackle the same verification problem. This technique leverages recent advances in concurrent software verification based on partial-order semantics [AKT13]. We provide extensive experimental studies to compare all these approaches in Chapter 5.

4.1 Introduction

We propose a new technique based on symbolic execution for verifying programs with nested interrupts. This method assumes the same interrupt semantics described in Section 3.1. It also uses the same modelling approach presented in Section 3.3, that is the user verifies properties for specific interrupt arrival scenarios, which are modelled by spawning interrupt service routine (ISR) threads at selected points in the program. As with the approach in Section 3.3, we can model multiple arrivals of the same interrupt by creating separate copies of the ISR function body for each interrupt. Instead of instrumenting the source code to constrain the resulting concurrency interleavings

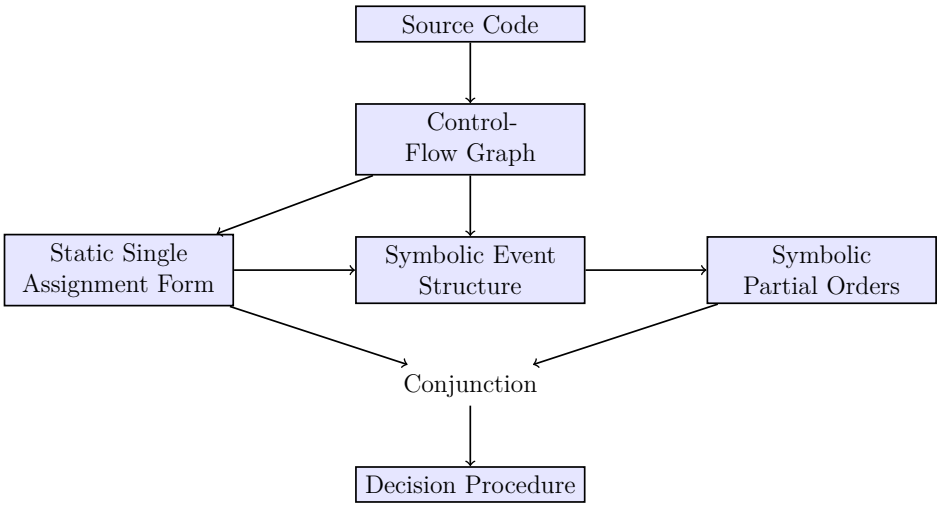


Figure 4.1: Overview of encoding into partial-order constraints

(Section 3.3), we encode the program execution directly as a SAT formula.

The idea is to translate a concurrent program into atomic memory read/write events (Section 4.3), and then describe the interleavings of these events that can occur as a symbolic partial order expressed by a formula (Section 4.4). The overall encoding is a formula of the form $SSA \wedge PORD$, where the conjunct SSA encodes the data and control structure of the program, in the style expounded by Alglave et al. [AKT13], and the conjunct $PORD$ is a predicate that over-approximates the set of all possible interleavings of ISRs that can occur during execution of the program. (We discuss the over-approximation in Section 4.6, after presenting the algorithms of computing the symbolic encoding in Section 4.5.) The program may contain assertions for verification, which will be included in SSA. A satisfying assignment to the variables in the formula corresponds to a potential error trace in the interrupt-driven program, identifying an assertion violation.

We emphasize that our partial-order encoding for nested interrupts is not to be confused with partial-order *reduction*, which was discussed in Section 3.2.2. An overview of our method is shown in Figure 4.1. We describe each key component of this figure in subsequent sections.

4.2 Static Single Assignment Form

The first conjunct *SSA* in our encoding is built in the same way as is employed by Alglave et al. [AKT13], which is based on a variant of the logical encoding of the *static single assignment* (SSA) form of Clarke et al. [CKY03]. In SSA, each occurrence of a program variable is annotated with an index; assignments can then be turned into equalities, with the indices yielding distinct variables in the resulting equation. For example, the assignment `x := x + 1` results in the equality $x_1 = x_0 + 1$. Unique indices are also used in loop unwinding. Consider the loop below:

```
int x = 0;
for (int i=0; i<3; i++) {
  x = x + 1;
}
```

The SSA encoding for the loop above is

$$x_0 = 1 \wedge x_1 = x_0 + 1 \wedge x_2 = x_1 + 1 \wedge x_3 = x_2 + 1$$

Each equation has a *guard*, which is the disjunction over the conjunctions of the branching conditions that occur along all the program paths leading to the assignment statement. So guards encode the control flow of the program. Consider the following example:

```
if (a > 0) {
  x = 1;
} else {
  x = 0;
}
y = x;
```

The SSA encoding of the above code snippet is

$$x_0 = 1 \wedge x_1 = 0 \wedge (x_2 = a_0 > 0 \; ? \; x_0 : x_1) \wedge y_0 = x_2$$

Recall that we annotate each occurrence of a program variable with an index. The `x` variable in the `if` and the `else` block is annotated with index 0 and 1, respectively. At the control-flow joint point a new variable x_2 (with a fresh index) is introduced to ensure the variable `y` gets the correct value of `x` from both conditional branches.

Pointer dereferences are removed recursively. First, expression `&*p` is simplified to `p`, and `*p` is translated array expression `p[0]`. Then the remaining dereferencing operators in all sub-expressions are removed. Consider the following example:

$$
\begin{array}{l}
*\mathrm{p} = \&\mathrm{a}; \\
*\mathrm{q} = \&\mathrm{b}; \\
\mathrm{x} = *\mathrm{p} + *\mathrm{q};
\end{array}
$$

The corresponding SSA encoding is

$$p_0[0] = a \wedge q_0[0] = b \wedge x_0 = p_0[0] + q_0[0]$$

For a concurrent program, a distinct index is used for each occurrence of a shared variable. Consequently, the connection between reads and writes across concurrent components of the program is broken. Reading a shared variable is then non-deterministic, over-approximating the behaviours of the program. For example, in the program given in Figure 4.2, `ISR_B` reads the value of global variable `x` and uses it to compute `b`. Depending on whether `ISR_B` starts its execution before `ISR_A`, `x` in `ISR_B` can get value x_1 from `ISR_A` or x_0 from the initialization step. As both scenarios are possible, we use a new index x_2 to capture both cases in the encoding. The construction of *PORD* then includes constraints to enforce consistency of corresponding reads and writes [AKT13]. The following two sections elaborate on how to construct *PORD*.

```
volatile unsigned x = 0, y = 0;                    x0 = 0  ∧  y0 = 0
volatile unsigned a = 0, b = 0;                    a0 = 0  ∧  b0 = 0

void* ISR_A(void* arg)
{
  x = 1;                                           x1 = 1
  a = y + 1;                                       a1 = y1 + 1
}

void* ISR_B(void* arg)
{
  async: ISR_A(); // spawn ISR_A thread
  y = 1;                                           y2 = 1
  b = x + 1;                                       b1 = x2 + 1
}

void main()
{
  async: ISR_B(); // spawn ISR_B thread
  assert(!(a == 1 && b == 1));                     a2 = 1  ∧  b2 = 1
}
```

Figure 4.2: An example of the static single assignment form for concurrent programs

4.3 Symbolic Events

Once the SSA encoding *SSA* is obtained, we construct a collection of *symbolic memory events* based on the occurrences of variables being read or written. A symbolic memory event has five parts: a unique identifier, a direction (read or write), the memory location that is read or written, a value, which may be given by a symbolic expression containing variables, and a guard, which is defined the same way in Section 4.2.

Consider the example in Figure 4.3. The SSA equation $a_1 = y_1 + 1$ arises from the statement `a = y + 1` in `ISR_A`. This generates two symbolic memory events: (A2) R `y` y_1 and (A3) W `a` a_1. Event A2 reads memory location `y` (i.e. the variable `y` in the original program) with symbolic value y_1. Event A3 writes to the memory location `a` with symbolic value a_1, which is equal to $y_1 + 1$ as given in the SSA equation.

$x_0 = 0 \quad \wedge \quad y_0 = 0$	(m1) W x 0 (m2) W y 0
$a_0 = 0 \quad \wedge \quad b_0 = 0$	(m3) W a 0 (m4) W b 0
$x_1 = 1$	(A1) W x 1
$a_1 = y_1 + 1$	(A2) R y y_1 (A3) W a a_1
$y_2 = 1$	(B1) W y 1
$b_1 = x_2 + 1$	(B2) R x x_2 (B3) W b b_1
$a_2 = 1 \quad \wedge \quad b_2 = 1$	(m5) R a a_2 (m6) R b b_2

Figure 4.3: Symbolic memory events for the program in Figure 4.2

4.4 Partial-Orders for Nested Interrupts

Using the symbolic memory events just described, we construct a symbolic partial order as a formula *PORD* that over-approximates the set of all possible interleavings of reads and writes that can occur in any execution of the original program. The motivation for making an over-approximation is to reduce the size of *PORD*. As will be seen in Section 4.6, this is still precise for non-nested interrupts—but it may signal false alarms in some more complex scenarios. Our experimental studies in Chapter 5 show that the encoding is still precise enough for real-world programs with nested interrupts.

Our encoding is an extension of the framework explained by Alglave et al. [AMSS12]. It is based on the idea of an *event graph* of reads and writes of shared memory locations. An event graph represents a set of candidate executions of the program with a particular set of control-flow choices for each thread, in which memory reads and writes happen in a certain partially-constrained order between different threads. There may be many different event graphs consistent with a given program, each describing different interrupt-arrival scenarios. The nodes of an event graph are the symbolic memory events of the program, the guards of which are evaluated to be true under a given set of control-flow choices for each thread. For clarity of presentation, we sometimes just write the unique identifier component of a symbolic memory event

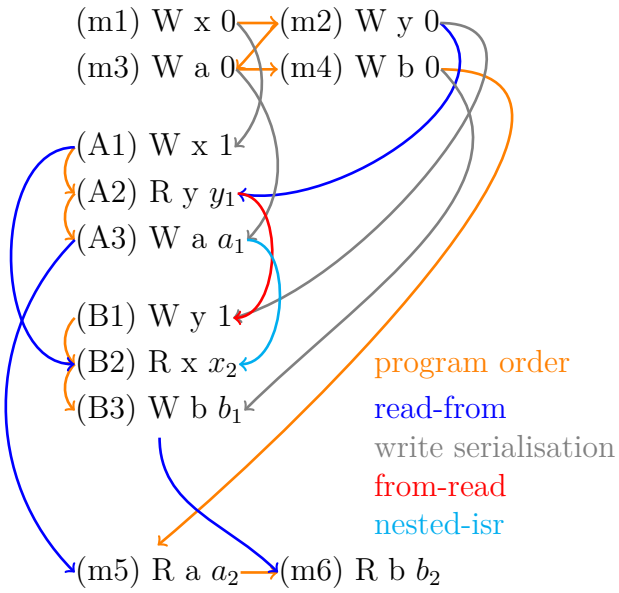


Figure 4.4: Event graph for the example program in Figure 4.2

as a node, it being understood that the other three components are obtainable from the identifier they are associated with. The edges of an event graph model a particular *global happens-before* order of the memory events that occur across the main program and the ISR threads.

There are five kinds of edges in an event graph that indicate specific types of ordering relationship between events: *program order* (po), *write serialisation* (ws), *read-from* (rf), *from-read* (fr), and *nested-isr* (ni). An event graph is said to be *valid* if po $\cup$ ws $\cup$ rf $\cup$ fr $\cup$ ni is acyclic—i.e., there are no cyclic dependencies among the read and write events in the event graph. One valid event graph for the example program introduced in Section 4.3 is given in Figure 4.4. This event graph corresponds to the arrival and handling of interrupt A during the handling of interrupt B, just after execution of the first assignment statement in `ISR_B`.

We now explain each relation in detail. We write thread(e) to denote the thread that generates memory event e, whether of an ISR or the main program. Note that memory events e and e' may belong to the same ISR, but this does not necessarily

mean that thread(e) = thread(e'), as they may belong to two different copies of the ISR function body—i.e. two different threads. We define depth(e) to be the execution depth of the thread that contains e, the same way we define depth in Section 3.3.

- The relation *program order* (po) is defined by the textual order of instructions within the code for each thread, or the main program. It relates only memory events that belong to the same thread. In the example event graph shown in Figure 4.4, the events within each handler and the main program are ordered as indicated by the numeric suffix of their identifiers.

- The relation *write serialisation* (ws) is the union of a set of total orders, each of which orders all the write events that access a specific memory location. For example, in the event graph in Figure 4.4, ws = {(m1, A1), (m2, B1), (m3, A3), (m4, B3)}. The *write serialisation* relation represents an order in which the write events to each memory location occur during an execution of the program and the modelled interrupts.

- The relation *read-from* (rf) relates each read event to a unique write event from which it is assumed that the read gets its value. More precisely, for all read events (r) R x a there is unique write event (w) W x b which accesses that same memory location x such that $(w, r) \in$ rf. As will be seen later, each such edge in the graph adds a constraint $a = b$ to the overall partial-order encoding—because, of course, r must read the same value that was written by w. In our example event graph, read event A2 reads the value of memory location y that is written by write event m2.

- The relation *from-read* (fr) contains certain read-write event pairs (r, w) that access the same memory location and thereby constrain the read r to happen before the write w. It is a derived relation and constructed as follows. Suppose

there is a pair (w, r) in the *read-from* relation rf. That is, it is assumed that r reads the value written by w. Now consider any write w' that occurs *after* w, according to the write serialisation relation ws. The read event r must happen *before* the write event w', for otherwise r would have gotten its value from w' instead of w. We express this constraint by having $(r, w') \in$ fr. In our example, (m2, A2) $\in$ rf and (m2, B1) $\in$ ws, so (A2, B1) $\in$ fr.

Finally, we define the *nested-isr* (ni) relation. It is a derived relation and represents constraints on the order of memory events that arise due to interrupt handling. Recall from Section 3.1 that all interrupts are masked by the hardware when an ISR is entered. The ISR can then selectively enable or disable individual interrupts, before lifting the mask. If ISR i enables interrupt j, which subsequently arrives during the execution of ISR i, the execution of ISR j must finish before the execution of ISR i can resume. The *nested-isr* relation captures this.

Formally, the relation *nested-isr* (ni) is defined as follows. For all memory events e, e' such that thread$(e) \neq$ thread(e') and depth$(e) \leq$ depth(e'), if $(e', e) \in$ ws $\cup$ rf $\cup$ fr, then for all events e'' such that $(e', e'') \in$ po, we put $(e'', e) \in$ ni. Intuitively, suppose e' is in a thread executing ISR j and e' happens before e, which is in a thread with equal or lower depth that is executing ISR i. This means that ISR i has been interrupted by ISR j, which generates memory event e' before execution of ISR i has reached memory event e. Since ISR j must exit before control is returned to ISR i, any memory events e'' in the execution of ISR j that occur after e' in program order must also happen before e. So we have $(e'', e) \in$ ni. In our example, interrupt A has arrived and is handled during the execution of `ISR_B`, indicated by (A1, B2) $\in$ rf. Then `ISR_A` must complete its execution before the control goes back to `ISR_B`, so we have (A3, B2) $\in$ ni. This is depicted as a solid, bold edge in Figure 4.4.

The event graph of our previous example in Figure 4.4 is acyclic, hence represents a valid execution of the program from which it is constructed. However, suppose that

event (B2) were to receive its value from (m1) instead of (A1), i.e., (m1, B2) $\in$ rf. Then (B2, A1) would be in fr as (m1, A1) is in ws. The event graph would have a cycle B1 $\rightarrow$ B2 $\rightarrow$ A1 $\rightarrow$ A2 $\rightarrow$ B1, thus does not correspond to a valid execution of the program. This scenario is depicted below in Figure 4.5.

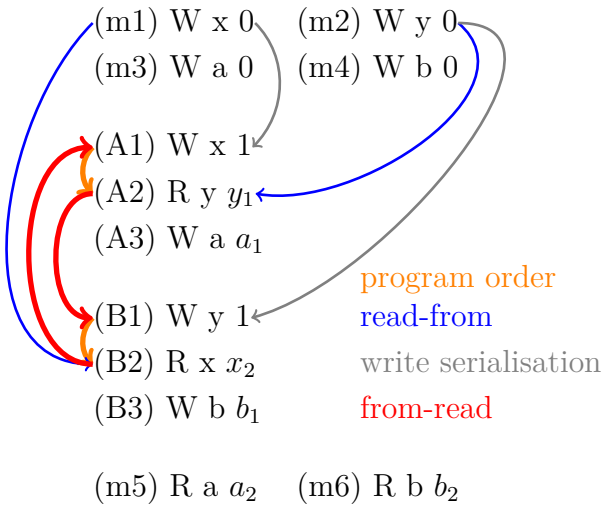


Figure 4.5: Invalid execution in an event graph

Now we are ready to give a semantics of program execution based on the event graph model described above. Given a valid event graph without cycles, the executions of an interrupt-driven program are given by the set of total orders induced by the partial order of the event graph.

4.5 Algorithms for Computing Constraints for Symbolic Partial Orders

We now elaborate on the algorithms for computing the formula *PORD*, which characterizes a set of possible interleavings among the ISRs and the main program under analysis. We can view *PORD* as the set of all valid event graphs that are consistent with the structure of the program and with the semantics of the interrupt scenarios

with which we have instrumented the program. As we shall see later, *PORD* is a slight over-approximation of this set; that is, it admits some event orderings that cannot occur in any real program execution. This will be explained in detail in Section 4.6.

Before describing our partial-order encoding, it is worth discussing a precise encoding that represents exactly the set of all possible interleavings of ISRs that can occur during program execution. It is a straight-forward modification of the partial-order encoding, where the constraints for the *nested-isr* (ni) relation of the precise encoding consider events that can access *different* memory locations. By contrast, the ni relation of the partial-order encoding only considers events that access the same memory locations in the ws, rf, and the fr relations. More precisely, for each pair of ISRs i and j such that the execution depth of ISR i is smaller than or equal to that of ISR j, the *nested-isr* (ni) constraints of the precise encoding are as follows: for each memory event e of ISR i and the first memory events e' of ISR j, which can either access the same or different memory locations, if e' happens before e, then for all memory events e'' after e' in ISR j, e'' must happen before e.

Algorithms 1–3 described below compute a set of constraints for our partial-order encoding. They take as input the set of all the symbolic events of a program with nested interrupts and its program order po, and produce as output sets of constraints that characterize the four relations between symbolic events in a valid event graph for the program. The formula *PORD* is the conjunction of all these constraints. Each algorithm computes a set of constraints for one of the relations *write serialisation* (ws), *read-from* (rf), or *from-read* (fr). In addition, each algorithm computes one part of the constraints for the *nested-isr* (ni) relation—namely those constraints on ni that arise from the constraints of other relations each algorithm computes. Our algorithms generate the same constraints for relations ws, rf, and rf as in Alglave et al. [AKT13]. Then we use these constraints to generate the constraints for the ni relation.

In what follows, we write mloc(e) for the memory location of event e, val(e) for

Algorithm 1: Constraints for *write serialisation*

Input : A set of symbolic events and program order po

Output: Sets of constraints C_{ws} and C_{ni_ws}

1 $C_{ws} \leftarrow \emptyset$; $C_{ni_ws} \leftarrow \emptyset$
2 $writes \leftarrow \{(\alpha, \{w_1 \ldots w_n\}) \mid w_i \text{ is a write event} \wedge \text{mloc}(w_i) = \alpha\}$
3 **for** α, W *s.t.* $(\alpha, W) \in writes$ **do**
4 **for** $w, w' \in W$ *s.t.* $\text{thread}(w) \neq \text{thread}(w')$ **do**
5 $C_{ws} \leftarrow C_{ws} \cup \{w \rightarrow w' \vee w' \rightarrow w\}$
6 **if** $\text{depth}(w) \geq \text{depth}(w')$ **then**
7 $C_{ni_ws} \leftarrow C_{ni_ws} \cup \{(w \rightarrow w' \wedge \text{grd}(w) \wedge \text{grd}(w')) \Rightarrow \bigwedge_{e \in \text{last}(w)} e \rightarrow w'\}$
8 **end**
9 **if** $\text{depth}(w) \leq \text{depth}(w')$ **then**
10 $C_{ni_ws} \leftarrow C_{ni_ws} \cup \{(w' \rightarrow w \wedge \text{grd}(w') \wedge \text{grd}(w)) \Rightarrow \bigwedge_{e \in \text{last}(w')} e \rightarrow w\}$
11 **end**
12 **end**
13 **end**

the symbolic value of event e, and $\text{grd}(e)$ for the guard of the SSA equation from which e is derived. For every event e, we introduce an integer variable ck_e, which is the *clock* of e. Given two memory events e and e', our encoding will then express that event e happens before event e' with the constraint $ck_e < ck_{e'}$. We write $e \rightarrow e'$ as a shorthand for the formula $\text{grd}(e) \wedge \text{grd}(e') \Rightarrow ck_e < ck_{e'}$. Note that in the precise encoding, although an edge (e', e) may not belong to any of the partial-order relations, the verification condition $e' \rightarrow e \wedge \text{grd}(e') \wedge \text{grd}(e)$ is checked by the underlying solver used in an algorithm that computes the constraints.

Algorithm 1 computes the constraints C_{ws} on the relation *write serialisation* (ws) and constraints C_{ni_ws} on the part of the *nested-isr* relation that derives from event orderings in ws. Specifically, it adds, for each pair of write events w and w' that access the same memory location and do not belong to the same thread (lines 2–4), the constraint that either $(w, w') \in \text{ws}$ or $(w', w) \in \text{ws}$ (line 5). Because of the definition of $e \rightarrow e'$, the disjunction is exclusive. So a satisfying assignment of values to the integer clock variables in this constraint constitutes a set of (guarded) total

orders on all write events that access each location α.

If a write w happens before another write w' and $\text{depth}(w') \leq \text{depth}(w)$, then according to our definition of the *nested-isr* relation, all memory events e after w in the program order must also happen before w'. This is expressed by $\bigwedge_{e \in \text{last}(w)} e \rightarrow w'$ in line 7, where

$$\text{last}(e) = \{e' \mid (e, e') \in \text{po} \wedge \neg \exists e''.\text{thread}(e') = \text{thread}(e'') \wedge (e,' e'') \in \text{po}\}$$

Note that we state only the last events e in the program order of the ISR that contains w. This is an optimization, as all memory events between w and any e in program order will happen before w' by transitivity. Symmetrically, we also impose a similar constraint in cases where $\text{depth}(w) \leq \text{depth}(w')$ (lines 9–11).

More formally, for the *write-serialisation* (ws) relation we have the following proposition:

Proposition 4.5.1. Given a valid event graph G with respect to a set of control-flow choices for each thread, there is a satisfying assignment to the constraint C_{ws} generated by Algorithm 1.

Proof. Since G is a valid event graph, there is a total order on the set of its nodes, in particular the set of write events that access the same memory location, i.e. the *write-serialisation* (ws) relation. We can label each node of G with an integer based on this total order. Then for each pair of write events w, w' that access the same memory location α and are from different threads, there are two cases to consider:

- Case 1: if either w or w' does not belong to G, then there exists variable assignments derived from the set of control-flow choices associated with G that make their guards $\text{grd}(w)$ or $\text{grd}(w')$ false. Since $e \rightarrow e'$ is defined to be $\text{grd}(e) \wedge \text{grd}(e') \Rightarrow ck_e < ck_{e'}$, the conjunct $w \rightarrow w' \vee w' \rightarrow w$ in Algorithm 1 (line 5) is vacuously true;

Algorithm 2: Constraints for *read-from*

Input : A set of symbolic events and program order po

Output: Set of constraints C_{rf}, C_{ni_rf}

$C_{rf} \leftarrow \emptyset$; $C_{ni_rf} \leftarrow \emptyset$
reads $\leftarrow \{(\alpha, \{r_1 \ldots r_n\}) \mid r_i \text{ is a read event} \wedge \text{mloc}(r_i) = \alpha\}$
writes $\leftarrow \{(\alpha, \{w_1 \ldots w_n\}) \mid w_i \text{ is a write event} \wedge \text{mloc}(w_i) = \alpha\}$
for α, R, W *s.t.* $(\alpha, R) \in$ *reads* $\wedge$ $(\alpha, W) \in$ *writes* **do**
  for $r \in R$ **do**
    $tmp \leftarrow \emptyset$
    for $w \in W$ **do**
      if $(r, w) \notin po$ **then**
        $tmp \leftarrow tmp \cup \{s_{wr}\}$
        $C_{rf} \leftarrow C_{rf} \cup \{s_{wr} \Rightarrow (\text{grd}(w) \wedge \text{grd}(r) \wedge \text{val}(w) = \text{val}(r))\}$
        $C_{rf} \leftarrow C_{rf} \cup \{s_{wr} \Rightarrow w \rightarrow r\}$
        if $\text{thread}(w) \neq \text{thread}(r) \wedge \text{depth}(w) \geq \text{depth}(r)$ **then**
          $C_{ni_rf} \leftarrow C_{ni_rf} \cup \{(w \rightarrow r \wedge \text{grd}(w) \wedge \text{grd}(r)) \Rightarrow \bigwedge_{e \in \text{last}(w)} e \rightarrow r\}$
        end
      end
    end
    $C_{rf} \leftarrow C_{rf} \cup \{\text{grd}(r) \Rightarrow \bigvee_{s \in tmp} s\}$
  end
end

- Case 2: if both w and w' belong to G, then the same assignments in Case 1 would satisfy their guards $\text{grd}(w)$ and $\text{grd}(w')$. In addition, we can assign ck_w and $ck_{w'}$ to the integer values based on its total order induced from G defined above, then the conjunct $w \rightarrow w' \vee w' \rightarrow w$ is satisfied.

It follows that the conjunction C_{ws} is satisfied. □

We defer our proof for the *nested-isr* relation in Theorem 4.5.1.

Algorithm 2 computes constraints C_{rf} on the relation *read-from* (rf) and constraints C_{ni_rf} on the part of the *nested-isr* relation deriving from event orderings in rf. In addition to specifying orders between write and read events, it connects the symbolic *value* of each read event to the value written by one of the possible

writes. For each pair of a write w and a read r that access the same memory location (lines 2–7), we introduce a fresh Boolean variable s_{wr} (line 9) to indicate that r in fact does read the value written by this w (line 10) as well as say that w must happen before r (line 11). Technically, this is achieved by enforcing $\text{val}(r) = \text{val}(w)$ in line 10. The vector of all the s_{wr} variables introduced in this part of the algorithm constitutes a symbolic index into all the possible ways in which reads get their values from writes. Note that the disjunction in line 17 technically allows a read event to get its value from multiple writes. However, with constraints of relation *from-read* described in Algorithm 3, this will be prevented. To see this, suppose $(w, r), (w', r) \in$ rf and $(w, w') \in$ ws. Then we have $(r, w') \in$ fr, forming a cycle between r and w'. This is not a valid execution.

The constraints related to nested interrupts are generated in lines 12–13 in the same way as in Algorithm 1.

We have the following proposition for the *read-from* (rf) relation:

Proposition 4.5.2. Given a valid event graph G with respect to a set of control-flow choices for each thread, there is a satisfying assignment to the constraint C_{rf} generated by Algorithm 2.

Proof. For each write-read pair w, r that access the same memory location α, there are two cases to consider:

- Case 1: if r does not belong to G, then the variable assignments derived from the set of control-flow choices associated with G in Case 1 of Proposition 4.5.1 would make its guard $\text{grd}(r)$ false. In addition, we assign false to the auxiliary variable s_{wr}. So the conjuncts in lines 10, 11 and 17 of Algorithm 2 are vacuously true;

- Case 2: if r belongs to G, since G is a valid event graph, there exists a write event in G from which r gets its value. If r does not get its value from w, we

Algorithm 3: Constraints for *from-read*

Input : A set of symbolic events and program order po

Output: Set of constraints C_{fr}, C_{ni_fr}

1 $C_{fr} \leftarrow \emptyset$; $C_{ni_fr} \leftarrow \emptyset$
2 $reads \leftarrow \{(\alpha, \{r_1 \ldots r_n\}) \mid r_i \text{ is a read event} \wedge \text{mloc}(r_i) = \alpha\}$
3 $writes \leftarrow \{(\alpha, \{w_1 \ldots w_n\}) \mid w_i \text{ is a write event} \wedge \text{mloc}(w_i) = \alpha\}$
4 **for** α, R, W *s.t.* $(\alpha, R) \in reads \wedge (\alpha, W) \in writes$ **do**
5 **for** $(w, w') \in W \times W$, $r \in R$ *s.t.* $w \neq w' \wedge \text{thread}(r) \neq \text{thread}(w')$ **do**
6 $C_{fr} \leftarrow C_{fr} \cup \{(s_{wr} \wedge w \rightarrow w' \wedge \text{grd}(w')) \Rightarrow r \rightarrow w'\}$
7 **if** $\text{depth}(r) \geq \text{depth}(w')$ **then**
8 $C_{ni_fr} \leftarrow C_{ni_fr} \cup \{(s_{wr} \wedge w \rightarrow w' \wedge \text{grd}(w')) \Rightarrow \bigwedge_{e \in \text{last}(r)} e \rightarrow w'\}$
9 **end**
10 **end**
11 **end**

assign false to s_{wr}, then the conjuncts in lines 10 and 11 are vacuously true. If r gets its value from w in G, then (w, r) belongs to the *read-from* (rf) relation. The control-flow assignments in Case 1 would satisfy $\text{grd}(w)$ and $\text{grd}(r)$. In addition, we assign true to s_{wr} and $\text{val}(w) = \text{val}(r)$. Moreover, since w happens before r in G, based on the total order induced from G in the proof of Proposition 4.5.1, the assignment would satisfy $w \rightarrow r$. Thus, the conjuncts in lines 10, 11 and 17 of Algorithm 2 are all satisfied.

It follows that the conjunction C_{rf} is satisfied. □

Finally, **Algorithm 3** constructs constraints C_{fr} for the relation *from-read* (fr) and constraints C_{ni_fr} for the part of the *nested-isr* relation derived from event orderings in fr. For every read-write pair (r, w') that accesses the same memory location α (lines 2–4), we consider all other writes w that accesses this memory location too and happen after w' (expressed by $w' \rightarrow w$ line 6). Recall that in Algorithm 2 we introduce the Boolean variable $s_{w'r}$ to say that $(w', r) \in \text{rf}$. So if r does indeed get its value from w', then we require $r \rightarrow w$ to hold. That is r must happen before w, for otherwise it would get its value from w. The constraints related to nested interrupts

are generated in lines 7–8 in a way similar to those in Algorithms 1 and 2.

For the *from-read* (fr) relation, we prove the following proposition:

Proposition 4.5.3. Given a valid event graph G with respect to a set of control-flow choices for each thread, there is a satisfying assignment to the constraint C_{fr} generated by Algorithm 3.

Proof. For each write-write pair w, w' and read event r that access the same memory location α and w' and r belong to different threads, there are two cases to consider:

- Case 1: if (w, r) does not belong to the *read-from* (rf) relation or (w, w') does not belong to the *write-serialisation* (ws) relation, based on the assignments from Proposition 4.5.1 and 4.5.2, the auxiliary variable s_{wr} and constraint $w \rightarrow w'$ are false. Thus, the conjunct in line 6 of Algorithm 3 is vacuously true;
- Case 2: otherwise, the constraint $s_{wr} \wedge w \rightarrow w' \wedge \text{grd}(w')$ is satisfied by the assignment in Case 1 above. Since G is a valid event graph, (r, w') belongs to the *from-read* (fr) relation, we have r happens before w' in G. The assignment based on the total order induced from G in Proposition 4.5.1 would satisfy $r \rightarrow w'$. As a result, the conjunct in line 6 is satisfied.

It follows that the conjunction C_{fr} is satisfied. □

Now we show our algorithm correctly encodes a valid event graph.

Theorem 4.5.1. Given a valid event graph G with respect to a set of control-flow choices for each thread, there is a satisfying assignment to the constraint $C_{ws} \cup C_{rf} \cup C_{fr} \cup C_{ni_ws} \cup C_{ni_rf} \cup C_{ni_fr}$ generated by Algorithms 1–3.

Proof. Given a valid event graph G with respect to a set of control-flow choices for each thread, it follows from the proofs of Propositions 4.5.1, 4.5.2, and 4.5.3 that there exists a satisfying assignment to the constraint $C_{ws} \cup C_{rf} \cup C_{fr}$. It remains to show that this assignment also satisfies $C_{ni_ws} \cup C_{ni_rf} \cup C_{ni_fr}$.

We show the assignment satisfies the constraint C_{ni_ws} in Algorithm 1. The cases for constraints C_{ni_rf} and C_{ni_fr} in Algorithms 2 and 3 can be proved similarly.

For write-write pair w, w' that access the same memory location and belong to different threads, if the invocation depth of w is higher than or equal to that of w', there are three cases to consider:

- Case 1: if either w or w' does not belong to G, then their guards $\text{grd}(w)$ or $\text{grd}(w')$ are evaluated to be false. So the conjunct in line 7 of Algorithm 1 is vacuously true;
- Case 2: if both w and w' belong to G but w' happens before w, then the assignments based on the total order induced from G in Proposition 4.5.1 does not satisfy $w \rightarrow w'$. So the conjunct in line 7 is also vacuously true;
- Case 3: if both w and w' belong to G and w happens before w', since G is a valid event graph, by the definition of the relation *nested-isr* (ni), for all memory events e that happens after w in the same thread of w, we have e happens before w' in G. Thus, the assignment based on the total order induced from G in Proposition 4.5.1 would satisfy $\bigwedge_{e \in \text{last}(w)} e \rightarrow w'$.

Similarly, the conjunct in line 10 of Algorithm 1 is also satisfied. The cases for constraints C_{ni_rf} and C_{ni_fr} in Algorithms 2 and 3 can be proved in the same way as the constraint C_{ni_ws}.

□

In the special case where interrupts have explicit priorities, we take an approach similar to those used in the program transformation methods discussed in the previous sections. A global array `irq_pty` is used to record the priority of each interrupt, and we define the priority of a memory event $\text{pty}(e)$, to be the priority of the ISR function body it belongs to. Then to handle explicit priorities in our encoding, we only need to replace any occurrence of $\text{depth}(e)$ with $\text{pty}(e)$ in our algorithms.

In the next section, we analyze our algorithms based on the event graph model. In particular, we show that the partial-order encoding is an over-approximation and the precise encoding satisfies a nested execution property.

4.6 Analysis of the Algorithms

Our encoding does not relate events in the same ISR call as they are already related by program order, except for *read-from* as a read event can get its value from a write event in the same ISR call. The (asymptotic) size of our encoding is, in the worst case, quadratic for *write serialisation* and *read-from*, and cubic for *from-read*, in the maximum number of memory events for a *single* address. By contrast, the precise encoding relates events that access the same and different memory locations, it produces constraints that are potentially larger than those of the partial-order encoding. In Chapter 5, we experimentally compare these two encodings.

Note that the encoding produced by our algorithm is by no means minimal in terms of size. For example, line 9 of Algorithm 2 can, in principle, exclude cases where all of the following hold: w and r do not belong to the same ISR call, $\text{depth}(r) \leq \text{depth}(w)$ is true, and w is the last write event in the program order s.t. the conjunction of guards $\text{grd}(w)$ and $\text{grd}(r)$ is satisfiable. However, in each iteration, we would need to check whether $\text{grd}(w) \wedge \text{grd}(r)$ is satisfiable, which can be quite expensive. We need to balance the size of the encoding and the computational cost of producing it.

As discussed in Section 3.1, executions of an interrupt-driven program with n interrupts should satisfy the following: for any pair of ISRs i and j, if the execution depth of ISR j is greater than or equal to the execution depth of ISR i, then ISR j must finish its execution before the control can return to ISR j. In terms of our partial-order relation framework, we have the following defintion:

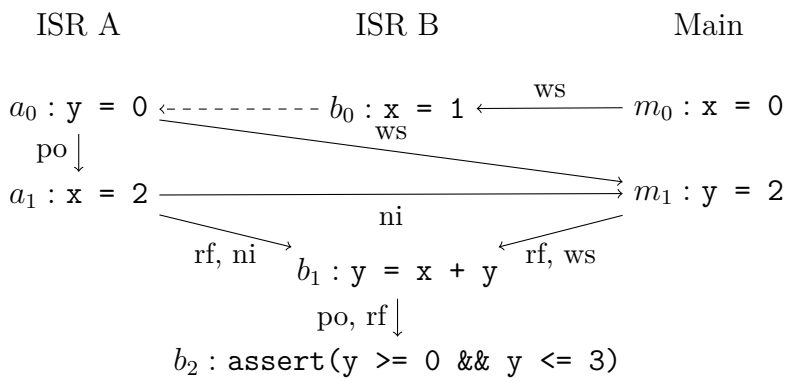


Figure 4.6: Counterexample

Definition 4.6.1. Suppose an event graph of a program satisfies the following: if 1) the depth of ISR j is greater than or equal to the depth of ISR i, and 2) event e of ISR j happens before event e' in ISR i, then for all events e'' in ISR j, we have e'' happens before e'. We say this event graph satisfies the *nested-execution property*.

As we have shown in Theorem 4.5.1 in Section 4.5, the partial-order encoding encodes all edges of a valid event graph. Thus, if a safety property holds in the partial-order encoding, it also holds in any valid event graph of the original program. On the other hand, if a safety property is violated in our partial-order encoding, it may not be violated in the original program. We exhibit one such event graph example that violates the nested-execution property in Figure 4.6.

In this example, the execution of the main program is preempted by ISR B, which is then preempted by ISR A. Since a_0 happens before m_1 represented by a ws edge, according to the definition of ni, we have an ni edge from a_1 to m_1. Similarly, since b_1 gets its x value from a_1, there is an ni edge from a_1 to b_1. However, there is no ws, rf, or fr edge from ISR B to the main program, so there is no ni edge to ensure the execution of ISR B finishes before the control can return to the main program.

For the non-nested case where the main program has only one ISR, if a safety property is violated in the partial-order encoding, then there exists a valid event graph that violates the property. We show this in the following theorem:

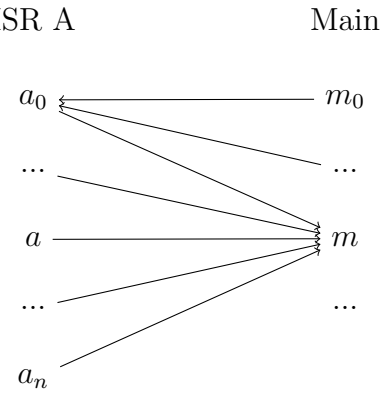


Figure 4.7: ISR A preempts the main program right before event m

Theorem 4.6.1. For an event graph of a program with one ISR that satisfies the partial-order constraints *program order* (po), *write-serialisation* (ws), *read-from* (rf), *from-read* (fr), and *nested-isr* (ni), there exists a valid event graph that satisfies the nested-execution property.

Proof. Let a and m be the first event in ISR A and the main program, respectively, such that (a, m) belongs to one of the ws, rf, and fr relations. By the definition of ni, all events that happen after a in ISR A will happen before m. For events a' and m' that happen before a and m in ISR A and the main program, respectively, they either access different memory locations or read the same memory location. Thus, their ordering does not matter, we can have m' happens before a_0 and a' happens before m as shown in Figure 4.7. Then the event graph satisfies the nested-execution property. □

In the counterexample given in Figure 4.6, the precise encoding would have an ni constraint indicating b_2 happens before m_1, since b_0 happens before m_1, even though they access different memory locations. Then there is a cycle $m_1 \to b_1 \to b_2 \to m_1$, excluding this counterexample.

More generally, we have the following theorem:

Theorem 4.6.2. If an event graph G satisfies the precise encoding, it also satisfies

the nested-execution property.

Proof. Suppose the depth of ISR j is greater than or equal to the depth of ISR i, and in the event graph G event e of ISR j happens before event e' in ISR i. We have the first event e_0 of ISR j also happens before e' (by program order). Since G satisfies the precise encoding, in particular the constraints for the *nested-isr* (ni) relation, for all events e'' that happen after e_0 in ISR j in program order, we have e'' happens before e' in G, satisfying the nested-execution property. □

Our experimental results in Chapter 5 show that the partial-order encoding obtains the same verification results as the precise encoding for all our benchmarks, while the runtime is faster on all but one of the benchmarks.

Summary

This chapter presents a symbolic execution technique that encodes executions of programs with nested interrupts into a symbolic partial order expressed by a SAT formula. By relating only the memory events that access the same memory location in the encoding, we are able to obtain a potentially smaller formula than that of threads with arbitrary interleaving semantics. At the same time, the formula can more precisely model the interleaving semantics of programs with nested interrupts. This is confirmed by our experimental studies in Chapter 5.

More specifically, we perform comprehensive empirical studies that compare the program transformation techniques introduced in Chapter 3 and the partial-order encoding developed in this chapter, using a variety of software verification tools and real-world benchmarks.

Chapter 5

Experimental Studies of Nested Interrupts Verification

In this chapter, we provide extensive experimental studies on techniques for verifying programs with nested interrupts. We compare the partial-order encoding technique introduced in Chapter 4 and the two program transformations described in Chapter 3 that translate interrupt-driven code into sequential and into multi-threaded code.

We implement our partial-order encoding in a prototype tool called *i-CBMC*, and apply a range of existing verifiers to sequentialised programs and programs that have been instrumented with threads. We evaluate the performance of these methods on real-world benchmarks drawn from embedded systems code in TinyOS and Simulink, as well as Linux device drivers. The experimental results show that our research provides the first method and tool implementation to effectively verify software with nested interrupts.

5.1 Verification Tools

Our prototype tool *i-CBMC*, which implements the partial-order encoding in Chapter 4, is an extension of the C Bounded Model Checker (CBMC) [CKL04]. Be-

sides using a collection of real-world benchmarks in our experimental studies reported in this chapter, CBMC and i-CBMC are also tested using a set of regression tests which can be found at https://github.com/lihaol/cbmc/tree/i-cbmc/regression/cbmc-concurrency.

To compare our partial-order encoding implemented in i-CBMC, we apply a range of existing verifiers to sequentialised programs and programs that have been instrumented with threads.

For *interrupt sequentialisation* as described in Section 3.2, we used the following tools:

- The sequential version of CBMC [CKL04], a bounded model checker for the C programming language;
- SatAbs [CKSY05], BLAST [SMM12], and CPAchecker [DLW15], which implement counterexample-guided abstraction refinement (CEGAR) [CGJ+00];
- UFO [ALGC12], which implements interpolation-based verification in an abstraction refinement loop [AGC12a, AGC12b, AGC12c].

For *modelling interrupts with threads* as discussed in Section 3.3, we used:

- The concurrent version of CBMC [AKT13];
- Impara [WKO13], which combines a new, symbolic form of partial-order reduction with the Impact algorithm [McM06];
- ESBMC [MRC+14], a context-bounded symbolic model checker for multi-threaded C programs;
- Lazy-CSeq [INF+15], a program transformation tool that translates a multi-threaded C program into a sequential C program with a given bound on the schedule rounds;

- The concurrent version of SatAbs [CKSY05];
- Threader [PR13], which implements thread-modular reasoning [GPR11].

The tools were run using the SV-COMP17 configuration (http://sv-comp.sosy-lab.org/2017/rules.php). We make our benchmarks and detailed experimental data available for other researchers at http://www.cprover.org/interrupts/. The source code of i-CBMC can be found at https://github.com/lihaol/cbmc/tree/i-cbmc.

5.2 Benchmarks

We assess the effectiveness of each method on benchmarks derived from embedded software and Linux device drivers.

For each benchmark, we have a version where interrupts are correctly disabled and enabled, and another version where they are *incorrectly* managed, and hence, safety properties are violated.

Logger models parts of the firmware of a temperature logging device from a major industrial enterprise (112 LOC, and 172 LOC for the extended version). It has two interrupt-triggered tasks: the measurement task should not be preempted by the communication task. Otherwise, a measurement is written to a wrong position in the log.

Blink is an application from TinyOS[1] distribution (2652 LOC). It displays a pattern on three LEDs with the help of timers. TinyOS provides a virtualization API to handle several alarms with one hardware timer. There are two interrupt-triggered tasks, for firing the alarm and updating the timer when it overflows. If the alarm firing can preempt the updating of the timer, the timer keeps running even though the alarm has already fired.

[1] http://www.tinyos.net

Brake is generated from the Simulink model of a brake-by-wire system provided by Volvo Technology AB (3938 LOC). The system consists of four threads communicating with four wheel brake controllers, together with one main thread, responsible for computing the braking torque, which should not be preempted. Otherwise, the brake torque can be miscalculated. We parameterized this benchmark by the number of wheels.

RcCore is a Linux device driver for a remote control together with a model of the Linux kernel (7035 LOC). The original driver uses locks to enforce priorities—namely that querying the availability of transmission protocols may not be interrupted by a modification to the protocol settings. We introduced a bug such that the driver may crash due to a NULL pointer dereference if priorities are disregarded.

We performed all experiments on a 64-bit machine running Linux with eight Intel Xeon 3.07 GHz cores and 48 GB of main memory.

5.3 Results and Discussion

Our results indicate that the transformation to sequential code performs worst, whereas our partial-order encoding is clearly more effective than the two program transformation techniques. We discuss our experimental results of each approach in more detail below.

Interrupt Sequentialisation

Table 5.1 presents the results of our interrupt sequentialisation. Column 1 lists our five benchmarks: Logger, extended Logger, Blink, RcCore and Brake (parameterised by the number of wheels). Columns 2–4 give the number of lines of code, the number of interrupt arrivals we modelled, and the number of scheduler calls after our optimization (Section 3.2) in each benchmark. Column 5 gives results for the interrupt

	LOC	Interrupts	Scheduler Calls	Interrupt Sequentialisation			
				BLAST 2.7.3	CPAChecker 1.6.12	UFO SV-COMP14	CBMC ea3a21f
Logger	112	2	8	✓ 17.7 s	✓ 2.7 s	✓ **1.4 s**	TO
+ incorrect	112	2	9	! **0.5 s**	! 7.0 s	! 36.2 s	TO
Logger (extended)	172	3	22	* 0.4 s	unknown	TO	TO
+ incorrect	172	3	18	! 84.1 s	! **9.1 s**	TO	TO
Blink	2,652	2	193	✗	unknown	✓**1425.1 s**	TO
+ incorrect	2,652	2	198	✗	unknown	? 1420.5 s	TO
RcCore	7,035	3	347	* 2.7 s	unknown	TO	TO
+ incorrect	7,035	3	347	! **2.7 s**	unknown	TO	✗
Brake (1 Wheel)	3,938	2	6	* 0.1 s	unknown	✓ **0.8 s**	✓ 80.8 s
+ incorrect	3,938	2	7	TO	unknown	? 1.1 s	✗
Brake (2 Wheels)	3,938	3	10	TO	unknown	✓ **7.8 s**	TO
+ incorrect	3,938	3	7	TO	unknown	? 1.1 s	TO
Brake (3 Wheels)	3,938	4	14	TO	unknown	✓ **854.1 s**	TO
+ incorrect	3,938	4	9	TO	unknown	? 3.8 s	TO

✓= proved correct, ! = bug exposed, **?** = bug missed, * = false alarm
✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

Table 5.1: Experimental results of interrupt sequentialisation

sequentialisation. SatAbs crashed on all our benchmarks so we did not include them in the table. The best correct entry for each benchmark is marked in bold.

In our experiments with interrupt sequentialisation, the number of interrupt arrivals is bounded. All sequential tools we tried except CBMC could handle both versions of the simplest benchmark Logger. CBMC timed out on this benchmark. For other benchmarks, the tools we tried could obtain only partial results. For the extended Logger benchmark, both UFO and CBMC timed out on the correct and faulty versions. BLAST reported a safety violation on both versions of this benchmark, whereas CPAchecker reported unknown and safety violation on the correct and faulty versions, respectively. For the Blink benchmark, only UFO was able to prove safety on the non-faulty version in about 24 minutes; but it failed to report a safety

	LOC	Interrupts	Modelling Interrupts with Threads			
			IMPARA 0.5.0	ESBMC 4.4.0-incr	Lazy-CSeq SV-COMP16	CBMC ea3a21f
Logger	112	2	✓ 0.5 s	✓ **0.1 s**	✓ 12.2 s	✓ 0.4 s
+ incorrect	112	2	! **0.6 s**	? 0.2 s	unknown	! 1.5 s
Logger (extended)	172	3	TO	✓ **9.5 s**	* 17.6 s	✓ 399 s
+ incorrect	172	3	TO	? 9.5 s	! **17.6 s**	! 383.2 s
Blink	2,652	2	TO	✓ **1.6 s**	unknown	✓ 4.5 s
+ incorrect	2,652	2	! **212.7 s**	? 2.4 s	unknown	? 19.4 s
RcCore	7,035	3	✗	PE	unknown	✓ **74.9 s**
+ incorrect	7,035	3	✗	PE	unknown	! **73.2 s**
Brake (1 Wheel)	3,938	2	TO	* 0.5 s	unknown	TO
+ incorrect	3,938	2	TO	! **0.6 s**	unknown	! 6.8 s
Brake (2 Wheels)	3,938	3	TO	* 0.8 s	unknown	✓ **669.3 s**
+ incorrect	3,938	3	TO	! **0.8 s**	unknown	! 16.9 s
Brake (3 Wheels)	3,938	4	TO	* 1.0 s	unknown	TO
+ incorrect	3,938	4	TO	! **1.0 s**	unknown	! 39.3 s

✓= proved correct, ! = bug exposed, **?** = bug missed, * = false alarm
✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

Table 5.2: Experimental results of modelling interrupts with threads

violation on the faulty one. BLAST crashed, CPAchecker returned unknown, and CBMC timed out on both versions of this benchmark. For the RcCore benchmark, only BLAST managed to report a safety violation in about 3 seconds; but it failed to prove safety on the non-faulty version. CPAchecker returned unknown and UFO timed out on both versions. Finally, UFO and CBMC were able to prove safety of the correct versions of the Brake benchmark in 1 s and 81 s, respectively, whereas BLAST returned a counterexample. None of the tools we tried was able to report a safety violation on the faulty version. It is worth noting that for most benchmarks, CBMC was not able to finish unwinding the recursion after 30 minutes. The reason is that non-determinism is encoded directly into a SAT formula.

Modelling Interrupts with Threads

Table 5.2 presents the results of modelling interrupts with threads. SatAbs crashed and Threader reported parse errors on all our benchmarks so we did not include them in the table. The results are better compared to sequentialisation: CBMC performed the best among the tools we have tried. It managed to report the expected results for most versions of our benchmarks. It timed out on the correct version of the Brake benchmark with one and three wheels, In addition, it failed to return a safety violation on the faulty version of the Blink benchmark. We suspect this may be caused by a potential bug in the current CBMC implementation, since Impara managed to report a safety violation on the faulty Blink. Impara was also able to return the expected results of the simplest Logger benchmark. For other benchmarks, it either timed out or crashed. ESBMC was able to prove safety on the correct Logger, extended Logger, and the Blink benchmarks; but it missed the bug on their incorrect versions. It could not parse the Linux device driver RcCore. For the Brake benchmark, it returned a safety violation on all its incorrect versions, and a false alarm on the correct ones. Finally, Lazy-CSeq managed to prove safety on the correct Logger, and report a safety violation on the incorrect extended Logger. But it reported a false alarm on the correct extended Logger, and unknown for other benchmarks.

In the thread instrumentation approach, each thread requires a separate copy of its ISR function, so modelling with threads can verify only a bounded number of interrupts, whereas interrupt sequentialisation can model an unbounded number of interrupts.

Partial-Order Encoding

Overall, *i-CBMC with partial-order encoding* is the most effective and time-efficient, outperforming other approaches on most benchmarks. It timed out in the non-faulty Brake benchmark with one and three wheels, and yields the expected results in all

	LOC	Interrupts	Naïve	Threads	Precise	PO
			CBMC ea3a21f	CBMC ea3a21f	i-CBMC -PRE	i-CBMC
Logger	112	2	* 0.2 s	✓ 0.4 s	✓ 0.3 s	✓ **0.2 s**
+ incorrect	112	2	! 0.2 s	! 1.5 s	! 0.3 s	! **0.2 s**
Logger (extended)	172	3	* 13.0 s	✓ 399 s	✓ 41.8 s	✓ **28.1 s**
+ incorrect	172	3	! **13.0 s**	! 383.2 s	! 35.3 s	! 25.9 s
Blink	2,652	2	* 3.4 s	✓ 4.5 s	✓ 26.5 s	✓ **2.0 s**
+ incorrect	2,652	2	! **3.0 s**	? 19.4 s	! 30.5 s	! 3.4 s
RcCore	7,035	3	* 73.4 s	✓ **74.9 s**	✓ 76.7 s	✓ 76.6 s
+ incorrect	7,035	3	! 75.1 s	! 73.2 s	! **56.1 s**	! 79.5 s
Brake (1 Wheel)	3,938	2	* 3.9 s	TO	TO	TO
+ incorrect	3,938	2	! 3.6 s	! 6.8 s	! 4.5 s	! **3.3 s**
Brake (2 Wheels)	3,938	3	* 7.1 s	✓ **669.3 s**	TO	✓883.9 s
+ incorrect	3,938	3	! **6.1 s**	! 16.9 s	! 9.1 s	! 7.2 s
Brake (3 Wheels)	3,938	4	* 7.2 s	TO	TO	TO
+ incorrect	3,938	4	! **6.8 s**	! 39.3 s	! 15.5 s	! 8.1 s

✓= proved correct, ! = bug exposed, **?** = bug missed, * = false alarm
✗ = tool crashes, PE = parse errors, TO = timeout 1800 s

Table 5.3: Experimental results of partial-order encoding

other experiments.

Table 5.3 lists the results of running CBMC on the benchmarks without thread instrumentation. CBMC reported false alarms on all correct versions of the benchmarks. This is expectable because without the additional constraints generated by i-CBMC, CBMC would allow interleavings that are not possible in any execution of the program, as described in Figure 3.3 of Section 3.1. Table 5.3 also compares the results of i-CBMC and CBMC with threads modelling (see the scatter plot in Figure 5.1). Both i-CBMC and CBMC timed out on the correct version of the Brake benchmark with one and three wheels. However, CBMC missed the bug in the faulty version of the Blink benchmark, whereas i-CBMC was able to report a safety violation on it. In addition, the runtime of i-CBMC was a lot faster than CBMC on all but two benchmarks. For example, i-CBMC was more than ten times faster than CBMC

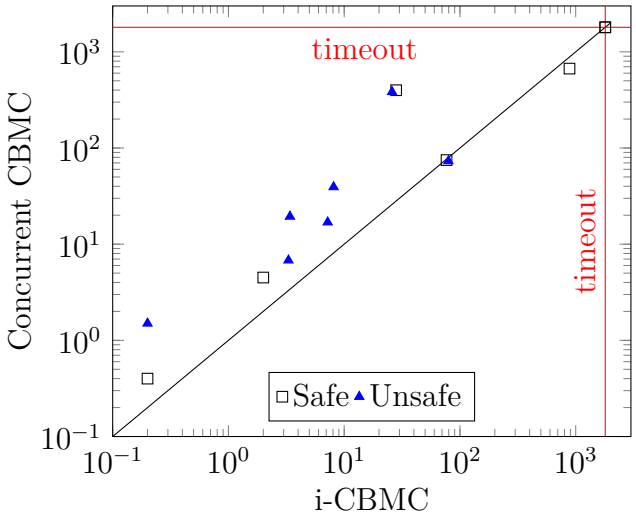


Figure 5.1: i-CBMC v.s. CBMC with threads modelling

on the Blink benchmark.

We also implement the precise encoding described at the end of Section 4.5 in a variant of i-CBMC (i-CBMC-PRE). As we can see in Table 5.3, i-CBMC-PRE was able to return the expected results for most versions of our benchmarks except the correct version of the three Brake benchmarks on which it timed out. In general, its runtime is between that of i-CBMC and concurrent CBMC except that it is the slowest on Blink and the fastest on the faulty version of RcCore.

In our modelling of interrupts with threads, we made each line of code atomic, which is an under-approximation of the behaviours of the original code. Our partial-order encoding models reads and writes as different atomic events and hence captures more behaviours. Yet, the runtime of i-CBMC is significantly better than that of CBMC with threads modelling for most variants of our benchmarks. The reason is that the assume statements in the modelling with threads approach increases the size of the SAT formula that CBMC generates.

Similarly, since i-CBMC-PRE implements the precise encoding that relates memory events that access different memory addresses, it generates more constraints than the partial-order encoding in i-CBMC that only relates events with the same memory

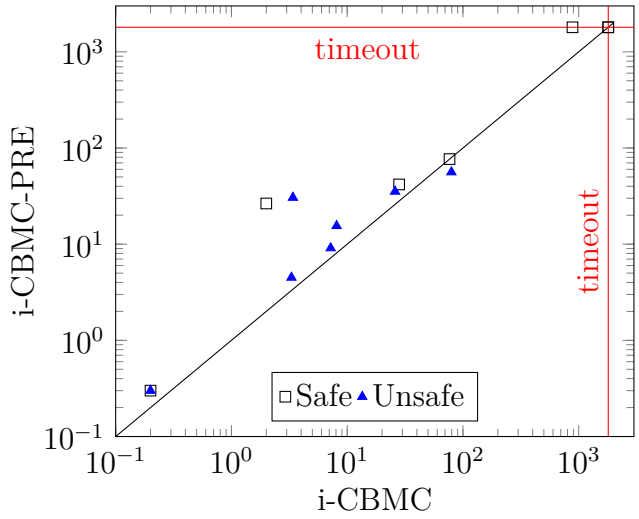


Figure 5.2: i-CBMC v.s. i-CBMC with precise encoding

address, resulting in a slower runtime in general (see the scatter plot in Figure 5.2).

We present the number of constraints (symbolic program expressions and partial orders), variables, and clauses of the SAT formulas that CBMC without thread instrumentation (naïve CBMC), i-CBMC, i-CBMC-PRE, and CBMC with thread instrumentation (concurrent CBMC) produce in Figures 5.3, 5.4, and 5.5, respectively. As we can see, the additional constraints generated by i-CBMC result in a slightly larger formula than those of CBMC *without* thread instrumentation. By contrast, the formula of CBMC *with* thread instrumentation are much larger than those of i-CBMC on most benchmarks. For example, on the extended Logger benchmark, the numbers of i-CBMC are substantially smaller than those of concurrent CBMC, which is reflected by the fact that the runtime of i-CBMC is an order of magnitude faster than concurrent CBMC. Similarly, i-CBMC produces smaller SAT formulas for the Logger, Blink, and the faulty version of the Brake benchmarks, so its runtime is also much lower than that of concurrent CBMC. Both tools performed similarly on the RcCore benchmark, and timed out on the correct version of the Brake benchmark with one and three wheels. The interesting case is the safe version of the Brake benchmark with two wheels. Although i-CBMC yields a similar or smaller encoding, it turns out they

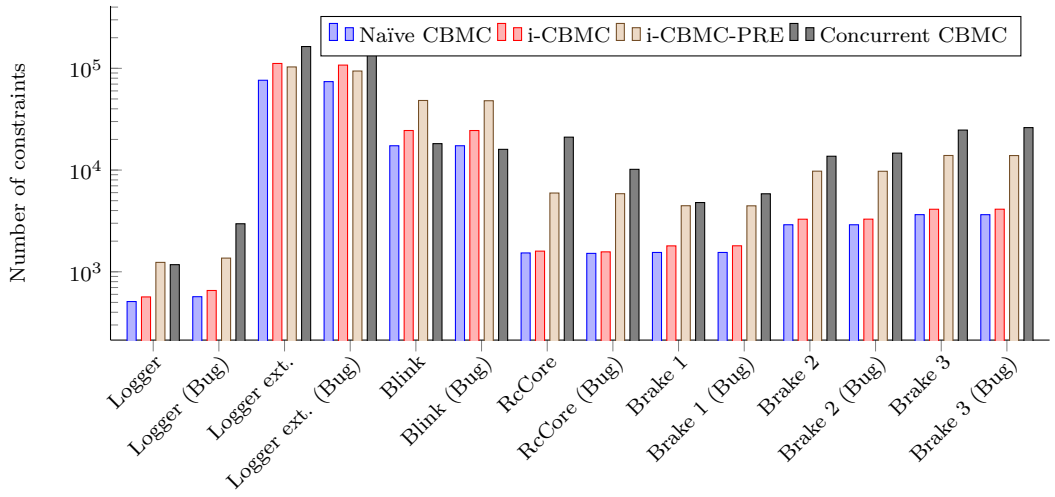


Figure 5.3: Number of constraints in the encodings

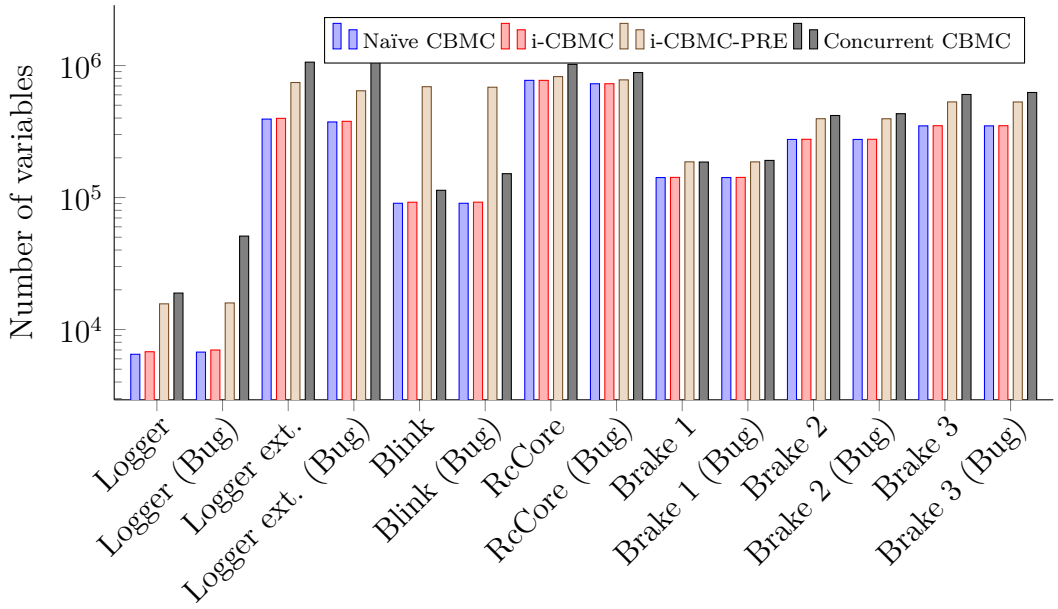


Figure 5.4: Number of variables in the SAT formulas

are more difficult to solve than those generated by concurrent CBMC (Table 5.3).

Following the same trend in runtime, the measurements for the formulas that i-CBMC-PRE generated on our benchmarks are in general between i-CBMC and concurrent CBMC. The exceptions are Blink on which i-CBMC-PRE generated the largest formula and had the slowest runtime. And it obtained the fastest runtime on the faulty version of the RcCore.

A limitation of the partial-order encoding is that it is based on bounded model

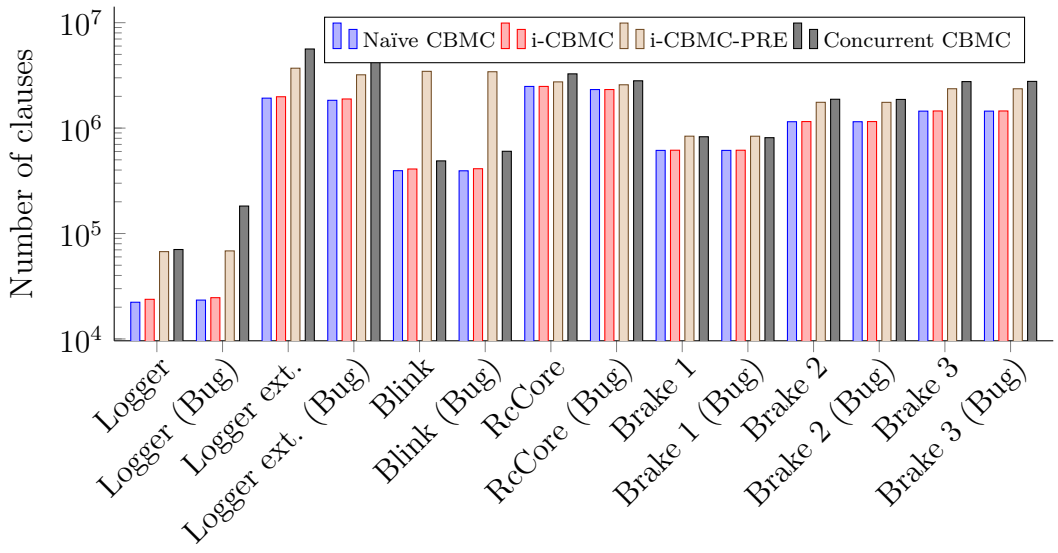


Figure 5.5: Number of clauses in the SAT formulas

checking, and thus is able to verify only a bounded number of interrupts. By contrast, the sequentialised programs can be passed to unbounded model checkers. In addition, since our partial-order encoding is an over-approximation, in real-world software engineering process, user inspection is required to decide whether a counterexample reported by i-CBMC is a false alarm.

Summary

In this chapter we perform comprehensive experimental studies on verification techniques for programs with nested interrupts. Our results show that the partial-order encoding developed in Chapter 4 performs the best, followed by the thread instrumentation introduced in Section 3.3. The interrupt sequentialisation described in Section 3.2 performs the worst, which is due to the recursion in its transformation.

Although the partial-order encoding is shown to be effective, the benchmarks in our experiments are mainly embedded programs. There is real-world systems code, such as the Linux kernel, that is significantly more complex and makes extensive use of interrupts. In the second part of this thesis, we describe our effort and experience

in modelling and verifying a highly sophisticated piece of interrupt-driven code in the Linux kernel.

Chapter 6

Read-Copy Update in the Linux Kernel

The second part of this thesis studies the verification problem of Read-Copy Update (RCU), a scalable high-performance synchronization mechanism in the Linux kernel [MS98, MW08]. Production-quality RCU implementations are non-trivial and sophisticated, making heavy use of the scheduling-clock interrupt. We start this chapter by introducing the concept of RCU and its core API usage. Next, we provide a detailed explanation of the tree-based RCU implementation in the Linux kernel. At the end, we illustrate how the implementation of Tree RCU works using a running example, which will also serve as a verification scenario in the next chapter.

6.1 Introduction

The Linux operating system kernel [Lin] is widely used, for example in servers, safety-critical embedded systems, household appliances, and mobile devices. Over the past 25 years, many technologies have been added to the Linux kernel, one example being Read-Copy Update (RCU) [MS98].

RCU is a synchronization mechanism used to replace reader-writer locks in read-

mostly scenarios, allowing low-overhead readers to run concurrently with updaters. Production-quality implementations for multi-core systems must provide excellent scalability, high throughput, low latency, modest memory footprint, excellent energy efficiency, and reliable response to CPU hotplug operations. They must therefore avoid cache misses, lock contention, frequent updates to shared variables, and excessive use of atomic read-modify-write and memory-barrier instructions. Finally, implementations must cope with the extremely diverse workloads and platforms of Linux [MW08].

RCU is now widely used in the Linux-kernel networking, device-driver, and file-storage subsystems [MW08, MBWW13]. To date, there are at least 75 million Linux servers [Lin13] and 1.4 billion Android devices [Cal15], so a "million-year" bug can occur several times per day across the installed base. Stringent validation of RCU's complex implementation is thus critically important.

Most validation efforts for concurrent software rely on testing, but unfortunately there is no cost-effective test strategy that can cover all corner cases. Worse still, some of errors that testing does detect might be difficult to reproduce, diagnose, and repair. The concurrent nature of RCU and the sheer size of the search space suggest use of formal verification, particularly model checking [BCM+92].

Formal verification has already been applied to some aspects of RCU design, including Tiny RCU [McK15], userspace RCU [DMD13], sysidle [McK15], and interactions between dyntick-idle and non-maskable interrupts (NMIs) [McK08a]. But these efforts either validate trivial single-CPU RCU implementations in C, or use special-purpose languages such as Promela [Hol97]. A major disadvantage of special-purpose modeling languages is difficult and error-prone translation from source code. Other research has done manual proof of simple RCU implementations using formal logic [GRY13, TDV15], but this requires significant effort beyond translation. Linux kernel releases are only about 60 days apart, and RCU changes with each release. So

any manual work must be replicated about six times a year.

If formal verification is to be part of Linux-kernel RCU's regression suite, the verification methods must be scalable and automated. To this end, this chapter describes how to build a model of Tree RCU from the Linux kernel source code, and use the C Bounded Model Checker (CBMC) [CKL04] to verify its safety and liveness properties. To the best of our knowledge, this is the first attempt to verify the core components of the Tree RCU implementation.

6.2 What is RCU?

Read-copy update (RCU) is a scalable, high-performance synchronization mechanism that is often used in read-mostly situations. RCU readers run concurrently with updaters, so RCU avoids read-side contention by maintaining multiple versions of objects and ensuring they are not freed until all pre-existing readers complete, that is, after a *grace period* elapses.

The basic idea is to split updates into removal and reclamation phases by a grace period [MS98]. The removal phase makes objects inaccessible to readers, waits during the grace period, and then reclaims them. As illustrated in Figure 6.1, grace periods need wait only for readers whose runtime overlaps the removal phase (represented by the red box). Readers starting after the removal phase ends cannot hold references to any removed objects and thus cannot be disrupted by objects being freed during the reclamation phase.

Modern CPUs guarantee that writes to single aligned pointers are atomic, so readers see either the old or new version of a data structure. This enables atomic insertions, deletions, and replacements in a linked structure. Readers can then avoid expensive atomic operations, memory barriers, and cache misses. In the most aggressive configurations of Linux-kernel RCU, readers use the same sequence of instructions

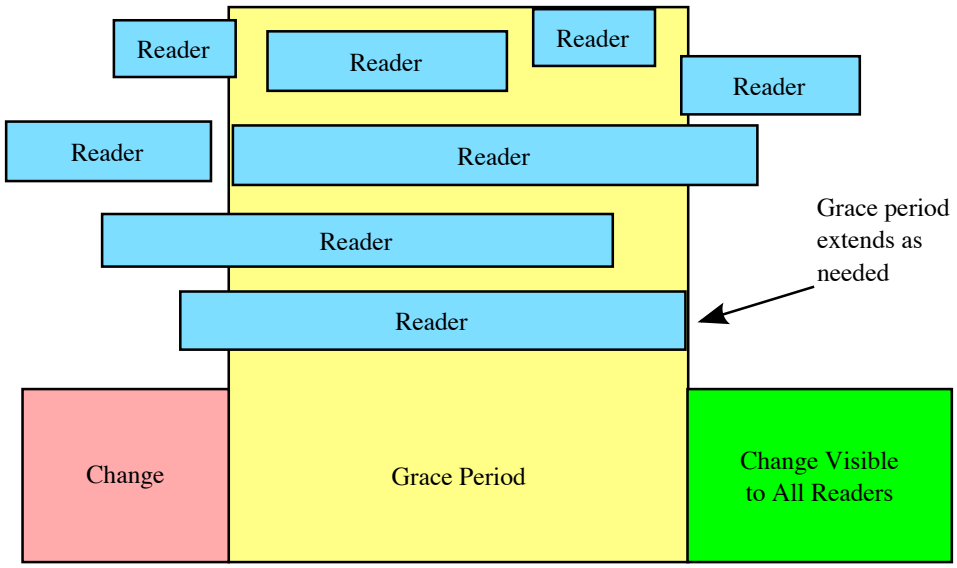


Figure 6.1: RCU concepts

that would be used in a single-threaded implementation, providing RCU readers with excellent performance and scalability.

6.3 Core RCU API Usage

The core RCU API is quite small and consists of only five primitives [MW08]: `rcu_read_lock()`, `rcu_read_unlock()`, `synchronize_rcu()`, `rcu_assign_pointer()`, and `rcu_dereference()`.

An RCU read-side critical section begins with `rcu_read_lock()` and ends with a corresponding `rcu_read_unlock()`. When nested, they are flattened into one large critical section. Within a critical section, it is illegal to block, but preemption is legal (in a preemptible kernel). RCU-protected data accessed by a read-side critical section will not be reclaimed until after that critical section completes.

The function `synchronize_rcu()` marks the end of the updater code and the beginning of the reclaimer code, so must block until all pre-existing RCU read-side critical sections have completed. But `synchronize_rcu()` need not wait for critical sections that begin after it does.

```
int x = 0;
int y = 0;
int r1, r2;

void rcu_reader(void) {
  rcu_read_lock();
  r1 = x;
  r2 = y;
  rcu_read_unlock();
}

void rcu_updater(void) {
  x = 1;
  synchronize_rcu();
  y = 1;
}

...

// after both rcu_reader()
// and rcu_updater() return
assert(r2 == 0 || r1 == 1);
```

Figure 6.2: Verifying RCU grace periods

Consider the example given in Figure 6.2. If the RCU read-side critical section in function `rcu_reader()` begins before `synchronize_rcu()` in `rcu_updater()` is called, then it must finish before `synchronize_rcu()` can return, so the value of `r2` must be 0. If it ends after `synchronize_rcu()` returns, then the value of `r1` must be 1.

Updaters use `rcu_assign_pointer()` to assign a new value to an RCU-protected pointer; readers use `rcu_dereference()` to fetch an RCU-protected pointer, which can then be safely dereferenced, but only within the enclosing read-side critical section. The `rcu_assign_pointer()` and `rcu_dereference()` functions work together to ensure that if a given reader dereferences an RCU-protected pointer to a just-inserted object, the dereference operation will return valid data rather than pre-initialization garbage.

In this thesis, we focus on verifying the implementation of `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()`.

6.4 Implementation of Tree RCU

The primary advantage of RCU is that it is able to wait for a very large number of readers to finish without tracking them individually. The number of readers can be up to the number of CPUs in non-preemptible implementations and up to the number of tasks in preemptible implementations.

Although RCU's read-side primitives enjoy excellent performance and scalability, update-side primitives must defer the reclamation phase till all pre-existing readers have completed, either by blocking or by registering a callback that is invoked after a grace period. The performance and scalability of RCU relies on efficient mechanisms to detect when a grace period has completed. A simplistic RCU implementation might require each CPU to acquire a global lock during each grace period, but this would not scale beyond a few hundred CPUs. The fact that Linux runs on systems with thousands of CPUs motivated the creation of Tree RCU.

6.4.1 Overview

Classic RCU has two implementations: Tiny RCU and Tree RCU. Tiny RCU is only for uni-processor systems, whereas Tree RCU is designed for muti-core systems. Tree RCU further has non-preemptible and preemptible variants, configured by the kernel Kconfig option `CONFIG_PREEMPT`. There is no preemptible implementation of Tiny RCU: Instead, preemptible Tree RCU is used in single-CPU preemptible kernel builds.

There are several flavours of RCU, including RCU-sched, RCU-preempt, RCU Bottom-Half (RCU-bh), and Sleepable RCU (SRCU) [McK08b]. Tree RCU implements RCU-sched, RCU-bh, and RCU-preempt when `CONFIG_PREEMPT=y`. If `CONFIG_PREEMPT=n`, then RCU-preempt is mapped into RCU-sched. Tiny RCU requires `CONFIG_PREEMPT=n`, so it also maps RCU-preempt into RCU-sched.

We focus on the "vanilla" API of RCU-sched in a non-preemptible build of the Linux kernel (`CONFIG_PREEMPT=n`), specifically on the `rcu_read_lock()`, `rcu_read_unlock()`, and `synchronize_rcu()` primitives. The key idea is that RCU read-side primitives are confined to kernel code and do not voluntarily block. So when a CPU passes through a *quiescent state* (context switch, is idle/offline, or runs in user mode), that CPU' s prior read-side critical sections must have finished. After each CPU has passed through a quiescent state, the corresponding RCU grace period ends.

The key challenge is to know when all quiescent states are reached. Recording quiescent states in one single location would result in extreme contention on large systems. To achieve excellent performance and scalability, Tree RCU uses a hierarchy of data structures, each leaf of which records the corresponding CPU's quiescent states. Once a node's children have recorded a full set of quiescent states, that node propagates the quiescent states up toward the root. When the root is reached, a grace period has ended and notification is propagated down. Shortly after a leaf receives this notification, `synchronize_rcu()` calls on the corresponding CPU will return.

In the remainder of this section, we discuss the implementation of the non-preemptible Tree RCU in the Linux kernel version 4.3.6. We first briefly discuss the implementation of read/write-side primitives. We then explain Tree RCU's hierarchical data structure which records quiescent states while maintaining bounded lock contention. Finally, we discuss how RCU uses this data structure to detect quiescent states and communicate them among all the CPUs, as well as to detect a grace periods without individually tracking readers.

6.4.2 Read/Write-Side Primitives

In a non-preemptible kernel, any region of kernel code that does not voluntarily block is implicitly an RCU read-side critical section. Therefore, the implementations of `rcu_read_lock()` and `rcu_read_unlock()` need do nothing at all, and in fact in production kernel builds that do not have debugging enabled, these two primitives have absolutely no effect on code generation.

In the common case where there are multiple CPUs running, the update-side primitive `synchronize_rcu()` calls `wait_rcu_gp()`, which is an internal function that uses a callback mechanism to invoke `wakeme_after_rcu()` at the end of some later grace period. As its name suggests, `wakeme_after_rcu()` function wakes up `wait_rcu_gp()`, which returns, in turn allowing `synchronize_rcu()` to return control to its caller.

6.4.3 Data Structures of Tree RCU

RCU's global state is recorded in the `rcu_state` structure, which consists of a tree of `rcu_node` structures with a child count of up to 64 (32 in a 32-bit system). Every leaf node can have at most 64 `rcu_data` structures (again 32 on a 32-bit system), each representing a single CPU, as illustrated in Figure 6.3. Each `rcu_data` structure records its CPU's quiescent states, and the `rcu_node` tree propagates these states up

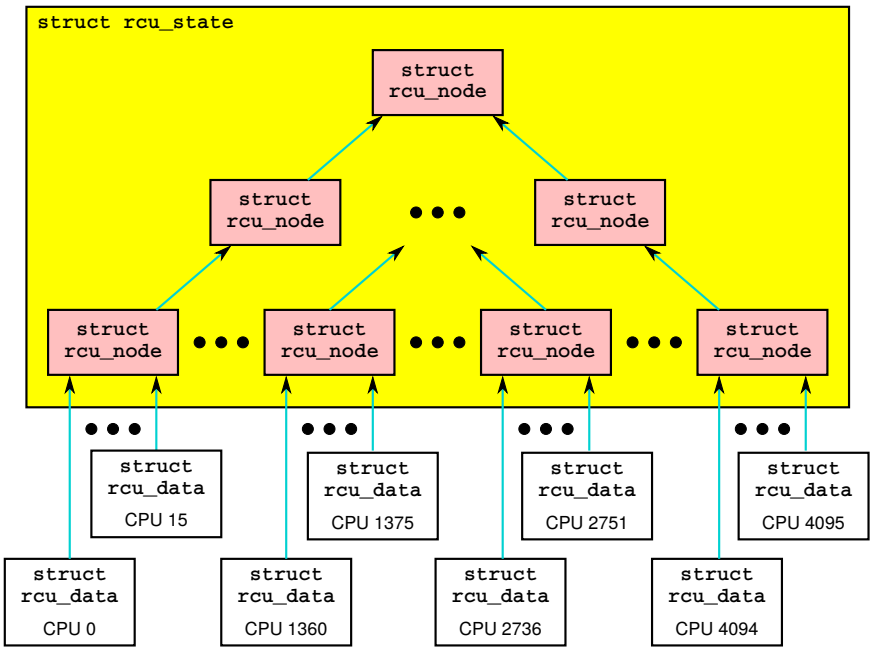


Figure 6.3: Tree RCU hierarchy

to the root, and then propagates grace-period information back down to the leaves. Quiescent-state information does not propagate upwards from a given node until a quiescent state has been reported by each CPU covered by the subtree headed by that node. This propagation scheme dramatically reduces the lock contention experienced by the upper levels of the tree. For example, consider a default `rcu_node` tree for a 4,096-CPU system, which will have have 256 leaf nodes, four internal nodes, and one root node. During a given grace period, each CPU will report its quiescent states to its leaf node, but there will only be 16 CPUs contending for each of those 256 leaf nodes. Only 256 of the CPUs will report quiescent states to the internal nodes, with only 64 CPUs contending for each of the four internal nodes. Only four CPUs will report quiescent states to the root node, resulting in extremely low contention on the root node's lock, so that contention on any given `rcu_node` structure is sharply bounded even in very large configurations. The current RCU implementation in the Linux kernel supports up to a four-level tree, and thus in total $64^4 = 16,777,216$

CPUs in a 64 bit machine.[1]

The `rcu_state` Structure

Each flavour of RCU has its own global `rcu_state` structure. The `rcu_state` structure includes an array of `rcu_node` structures, `struct rcu_node node[NUM_RCU_NODES]`, organized as a tree with `rcu_data` structures connected to the leaves. Given this organization, a breadth-first traversal is simply a linear scan of the array.

struct rcu_node node[NUM_RCU_NODES];
unsigned long gpnum;
unsigned long completed;

The `rcu_state` structure uses `unsigned long` fields `->gpnum` and `->completed` to track RCU's grace periods. The `->gpnum` field records the most recently started grace period, whereas `->completed` records the most recently ended grace period. If the two numbers are equal, then corresponding flavour of RCU is idle. If `gpnum` is one greater than `completed`, then RCU is in the middle of a grace period. All other combinations are invalid.

The `rcu_node` Structure

The tree of `rcu_node` structures records and propagates quiescent-state information from the leaves to the root, and also propagates grace-period information from the root to the leaves. The `rcu_node` structure has a spinlock `->lock` to protect its fields. The `->parent` field references the parent `rcu_node` structure, and is `NULL` for the root. The `->level` field indicates the level in the tree, counting from zero at the root. The `->grpmask` field identifies this node's bit in the `->qsmask` field of its parent. The `->grplo` and `->grphi` fields indicates the lowest and highest numbered CPU that are covered by this `rcu_node` structure, respectively.

[1] Four-level trees are only used in stress testing, but three-level trees are used in production by 4096-CPU systems.

```
raw_spinlock_t lock;
struct rcu_node *parent;
u8 level;
unsigned long grpmask;
int grplo;
int grphi;
unsigned long qsmask;
unsigned long gpnum;
unsigned long completed;
```

the `->qsmask` field indicates which of this node's children still need to report quiescent states for the current grace period. As with `rcu_state`, the `rcu_node` structure has `->gpnum` and `->completed` fields that have values identical to those of the enclosing `rcu_state` structure, except at the beginnings and ends of grace periods when the new values are propagated down the tree. Each of these fields can be smaller than its `rcu_state` counterpart by at most one.

The `rcu_data` structure

The `rcu_data` structure detects quiescent states and handles RCU callbacks for the corresponding CPU. The structure is accessed primarily from the corresponding CPU, thus avoiding synchronization overhead. As with the `rcu_state` structure, different flavours of RCU maintain their own per-CPU `rcu_data` structures. The `->cpu` field identifies the corresponding CPU, the `->rsp` field references the corresponding `rcu_state` structure, and the `->mynode` field references the corresponding leaf `rcu_node` structure. The `->grpmask` field identifies this `rcu_data` structure's bit in the `->qsmask` field of its leaf `rcu_node` structure.

```
int cpu;
struct rcu_state *rsp;
struct rcu_node *mynode;
unsigned long grpmask;
unsigned long qsmask;
bool qs_pending;
bool passed_quiesce;
unsigned long gpnum;
unsigned long completed;
```

The `rcu_data` structure's `->qs_pending` field indicates that RCU needs a quiescent state from the corresponding CPU, and the `->passed_quiesce` indicates that the CPU has already passed through a quiescent state. The `rcu_data` also has `->gpnum` and `->completed` fields, which can lag arbitrarily behind their counterparts in the `rcu_state` and `rcu_node` structures on idle CPUs. However, on the non-idle CPUs that are the focus of this thesis, these two fields can lag at most one grace period behind their leaf `rcu_node` counterparts.

The `rcu_state` structure's `->gpnum` and `->completed` fields represent the most current values, and are tracked closely by those of the `rcu_node` structure, which allows the `->gpnum` and `->completed` fields in the `rcu_data` structures to be are compared against their counterparts in the corresponding leaf `rcu_node` to detect a new grace period. This scheme allows CPUs to detect beginnings and ends of grace periods without incurring lock- or memory-contention penalties. The `rcu_data` structure manages RCU callbacks using a four-segment list [Jia08].

6.4.4 Quiescent State Detection

RCU has to wait until all pre-existing read-side critical sections have finished before it can safely allow a grace period to end. The performance and scalability of RCU rely on its ability to efficiently detect quiescent states and determine whether the set of quiescent states detected thus far allows the grace period to end. If each CPU (or, in the case of preemptible RCU, each task) has passed through a quiescent state, a

grace period has elapsed.

Different flavours of RCU use different sets of quiescent states. The non-preemptible RCU-sched flavour's quiescent states apply to CPUs, and are user-space execution, context switch, idle, and offline state. RCU-bh's quiescent states are those of RCU-sched plus any execution in which bottom-half (i.e. softirq) is enabled, along with transitions from one softirq handler to another. RCU-preempt's quiescent states are any execution outside of an RCU read-side critical sections. Therefore, RCU-sched and RCU-bh need only track CPU states as they need only track tasks and interrupt handlers that are actually running (blocked and preempted tasks are always in quiescent states). By contrast, RCU-preempt must track tasks states.

In this section, we focus on the quiescent-state detection of RCU-sched in a non-preemptible kernel.

Scheduling-Clock Interrupt

The `rcu_check_callbacks()` is invoked from the scheduling-clock interrupt handler, which allows RCU to periodically check whether a given busy CPU is in the user-mode or idle-loop quiescent states. If the CPU is in one of these quiescent states, `rcu_check_callbacks()` invokes `rcu_sched_qs()`, which sets the per-CPU `rcu_sched_data.passed_quiesce` fields to 1.

The `rcu_check_callbacks()` function invokes `rcu_pending()` to determine whether a recent event or current condition means that RCU requires attention from this CPU. If so, `rcu_check_callbacks()` invokes `raise_softirq()`, which will cause `rcu_process_callbacks()` to be invoked once the CPU reaches a state where it is safe to do so (roughly speaking, once the CPU has interrupts, preemption, and bottom halves enabled). This function is discussed in detail in Chapter 6.4.5.

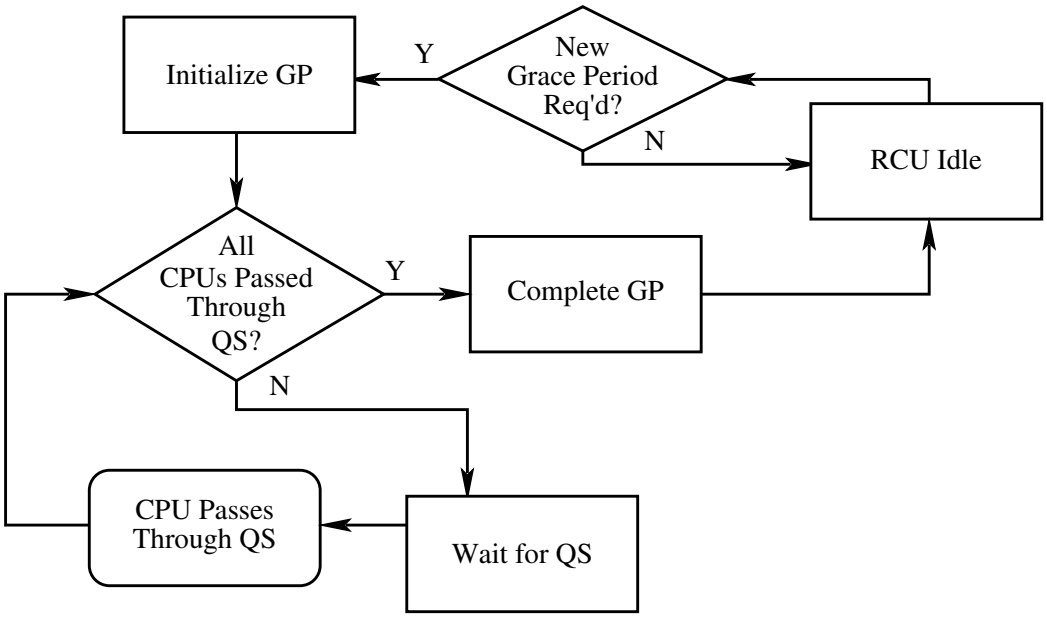


Figure 6.4: Grace-period detection state diagram

Context-Switch Handling

The context-switch quiescent state is recorded by invoking `rcu_note_context_switch()` from `__schedule()` (and, for the benefit of virtualization, also from `rcu_virt_note_context_switch()`). The `rcu_note_context_switch()` function invokes `rcu_sched_qs()` to inform RCU of the context switch, which is a quiescent state of the CPU.

6.4.5 Grace Period Detection

Once each CPU has passed through a quiescent state, a grace period for RCU has completed. As discussed in Chapter 6.4.3, Tree-RCU uses a hierarchy of `rcu_node` structures to manage quiescent state and grace period information. Quiescent-state information is passed up the tree from the leaf per-CPU `rcu_data` structures. Grace-period information is passed down from the root. We focus on grace-period detection for busy CPUs, as illustrated in Figure 6.4.

Softirq Handler for RCU

RCU's busy-CPU grace period detection relies on the `RCU_SOFTIRQ` handler function `rcu_process_callbacks()`, which is scheduled from the scheduling-clock interrupt. This function first calls `rcu_check_quiescent_state()` to report recent quiescent states on the current CPU. Then `rcu_process_callbacks()` starts a new grace period if needed, and finally calls `invoke_rcu_callbacks()` to invoke any callbacks whose grace period has already elapsed.

Function `rcu_check_quiescent_state()` first invokes `note_gp_changes()` to update the CPU-local `rcu_data` structure to record the end of previous grace periods and the beginning of new grace periods. Any new values for these fields are copied from the leaf `rcu_node` structure to the `rcu_data` structure. If an old grace period has ended, `rcu_advance_cbs()` is invoked to advance all callbacks, otherwise, `rcu_accelerate_cbs()` is invoked to assign a grace period to any recently arrived callbacks. If a new grace period has started, `->passed_quiesce` is set to zero, and if in addition RCU is waiting for a quiescent state from this CPU, `->qs_pending` is set to one, so that a new quiescent state will be detected for the new grace period.

Next, `rcu_check_quiescent_state()` checks whether `->qs_pending` indicates that RCU needs a quiescent state from this CPU. If so, it checks whether `->passed_quiesce` indicates that this CPU has in fact passed through a quiescent state. If so, it invokes `rcu_report_qs_rdp()` to report that quiescent state up the combining tree.

The `rcu_report_qs_rdp()` function first verifies that the CPU has in fact detected a legitimate quiescent state for the current grace period, and under the protection of the leaf `rcu_node` structure's `->lock`. If not, it resets quiescent-state detection and returns, thus ignoring any redundant quiescent states belonging to some earlier grace period. Otherwise, if the `->qsmask` field indicates that RCU needs to report a quiescent state from this CPU, `rcu_accelerate_cbs()` is invoked to assign a grace-period number to any new callbacks, and then `rcu_report_qs_rnp()` is invoked to

report the quiescent state to the rcu_node combining tree.

The rcu_report_qs_rnp() function traverses up the rcu_node tree, at each level holding the rcu_node structure's ->lock. At any level, if the child structure's ->qsmask bit is already clear, or if the ->gpnum changes, traversal stops. Otherwise, the child structure's bit is cleared from ->qsmask, after which, if ->qsmask is non-zero, traversal stops. Otherwise, traversal proceeds on to the parent rcu_node structure. Once the root is reached, traversal stops and rcu_report_qs_rsp() is invoked to awaken the grace-period kthread (kernel thread). The grace-period kthread will then clean up after the now-ended grace period, and, if needed, start a new one.

Grace-Period Kernel Thread

The RCU grace-period kthread invokes rcu_gp_kthread(), which contains an infinite loop that initializes, waits for, and cleans up after each grace period.

When no grace period is required, the grace-period kthread sets its rcu_state structure's ->flags field to RCU_GP_WAIT_GPS, and then waits within an inner infinite loop for that structure's ->gp_state field to be set. Once set, rcu_gp_kthread() invokes rcu_gp_init() to initialize a new grace period, which rechecks the ->gp_state field under the root rcu_node structure's ->lock. If the field is no longer set, rcu_gp_init() returns zero. Otherwise, it increments rsp->gpnum by 1 to record a new grace period number. Finally, it performs a breadth-first traversal of the rcu_node structures in the combining tree. For each rcu_node structure rnp, we set the rnp->qsmask to indicate which children must report quiescent states for the new grace period (Chapter 6.4.3), and set rnp->gpnum and rnp->completed to their rcu_state counterparts. If the rcu_node structure rnp is the parent of the current CPU's rcu_data, we invoke __note_gp_changes() to set up the CPU-local rcu_data state. Other CPUs will invoke __note_gp_changes() after their next scheduling-clock interrupt.

During a grace period, the grace-period kthread periodically calls force_qs_rnp()

to detect idle and offline CPUs. For each such CPU, `force_qs_rnp()` invokes `rcu_report_qs_rnp()` to report a quiescent state on its behalf, thus avoiding the degraded energy efficiency that would be incurred should RCU awaken idle CPUs. CPUs that fail to report quiescent states will be sent an inter-processor interrupt (IPI), and if that fails, warning messages will be emitted.

To clean up after a grace period, `rcu_gp_kthread()` calls `rcu_gp_cleanup()` after setting the `rcu_state` field `rsp->gp_state` to `RCU_GP_CLEANUP`. After the function returns, `rsp->gp_state` is set to `RCU_GP_CLEANED` to record the end of the old grace period. Function `rcu_gp_cleanup()` performs a breadth-first traversal of `rcu_node` combining-tree. It first sets each `rcu_node` structure's `->completed` field to the `rcu_state` structure's `->gpnum` field. It then updates the current CPU's CPU-local `rcu_data` structure by calling `__note_gp_changes()`. For other CPUs, the update will take place when they handle the scheduling-clock interrupts, in a fashion similar to `rcu_gp_init()`. After the traversal, it marks the completion of the grace period by setting the `rcu_state` structure's `->completed` field to that structure's `->gpnum` field, and invokes `rcu_advance_cbs()` to advance callbacks. Finally, if another grace period is needed, we set `rsp->gp_flags` to `RCU_GP_FLAG_INIT`. Then in the next iteration of the outer loop, the grace-period kthread will initialize a new grace period as discussed above.

6.5 Verification Scenario

We use the example in Figure 6.2 to demonstrate how Tree RCU guarantees that all pre-existing read-side critical sections finish before it allows a grace period to end. This example also drives the verification described in Chapter 7, which checks for violations of the assertion that follows the code.

We focus on the non-preemptible RCU-sched flavour. We assume there are only

two CPUs, and that CPU 0 runs `rcu_reader()` and CPU 1 runs `rcu_updater()`. When the system boots, the Linux kernel calls `rcu_init()` to initialize RCU, which includes making the tree of `rcu_node` and `rcu_data` structures via `rcu_init_geometry()` and initializing the fields of the nodes in the tree for each RCU flavour via `rcu_init_one()`. Function `rcu_spawn_gp_kthread()` is also called to initialize and spawn the RCU grace-period kthread for each RCU flavour.

Our example has a one-level tree, with one `rcu_node` root and two `rcu_data` children for each CPU. As discussed in this chapter, the `rcu_node` structure records and propagates quiescent-state information from leaves to the root, and also propagates grace-period information from the root to the leaves. The per-CPU `rcu_data` structure detects quiescent states and handles RCU callbacks for that CPU.

Suppose `rcu_reader()` begins execution on CPU 0 while `rcu_updater()` concurrently sets `x` to 1 and then invokes `synchronize_rcu()` on CPU 1. As discussed in Chapter 6.4.2, `synchronize_rcu()` invokes `wait_rcu_gp()`, an internal function that uses callbacks to invoke `wakeme_after_rcu()` some time after `rcu_reader()` exits its critical section, i.e. after a grace period. As its name suggests, `wakeme_after_rcu()` wakes up `wait_rcu_gp()`; this returns, allowing `synchronize_rcu()` to return control to its caller.

This critical-section exit has no immediate effect. A later context switch will invoke `rcu_note_context_switch()`, which invokes `rcu_sched_qs()` to record the quiescent state in the `->passed_quiesce` field of the corresponding CPU's `rcu_data` structure. Later, a scheduling-clock interrupt will invoke `rcu_check_callbacks()`, which calls `rcu_pending()` noting that the `->passed_quiesce` field is set. So `rcu_pending()` returns `true`, which in turn causes `rcu_check_callbacks()` to invoke `invoke_rcu_core()`. `invoke_rcu_core()` then invokes `raise_softirq(RCU_SOFTIRQ)`, which, once the CPU has interrupts, preemption, and bottom halves enabled, calls `rcu_process_callbacks()`.

As discussed in Chapter 6.4.5, RCU's softirq handler `rcu_process_callbacks()` first calls `rcu_check_quiescent_state()` to report any recent quiescent states on the current CPU (CPU 0). Since a quiescent state has been recorded for CPU 0, `rcu_report_qs_rnp()` is invoked to traversal up the combining tree. It clears the first bit of the root `rcu_node` structure's `qsmask` field, which indicates which of this node' s children still need to report quiescent states for the current grace period. As the second bit for CPU 1 has not been cleared, the function returns.

Since `synchronize_rcu()` blocks in CPU 1, it will result in a context switch. This triggers a sequence of events similar to that described above for CPU 1, which results in the clearing of the second bit of the root `rcu_node` structure's `->qsmask` field, the value of which is now 0, indicating the end of the current grace period. CPU 1 therefore invokes `rcu_report_qs_rsp()` to awaken the grace-period kthread, which will clean up the ended grace period, and, if needed, start a new one (Chapter 6.4.5).

Lastly, `rcu_process_callbacks()` calls `invoke_rcu_callbacks()` to invoke any callbacks whose grace period has already elapsed, for example, `wakeme_after_rcu()`, which will allow `synchronize_rcu()` to return.

Summary

This chapter explains the concept of RCU and its tree-based implementation in the Linux kernel. The following chapter will describe our efforts to model and verify the Tree RCU implementation using the tool CBMC.

Chapter 7

Modelling and Verification of Tree RCU

This chapter studies the modelling and verification problem of the tree-based implementation of Read-Copy Update (Tree RCU) in the Linux kernel. We first describe how to construct a model from Tree RCU's source code. Then we explain how to use the C Bounded Model Checker (CBMC) [CKL04] to verify its safety and liveness properties. Lastly, we report our experimental results. This is the first attempt to verify the core components of the Tree RCU implementation. We hope it will be a step towards integrating formal verification into RCU's regression test suite in the Linux kernel.

7.1 Modelling RCU for CBMC

In this section we describe how to build a model from the source code of Tree RCU in the Linux kernel version 4.3.6, which we verified with CBMC. Model construction entailed stubbing out calls to other parts of the kernel, removing irrelevant functionality (such as idle-CPU detection), removing irrelevant data (such as statistics), and adding preprocessor directives to conditionally inject bugs (described in Section 7.2.1).

The Linux kernel environment and most of the source-code changes are made through macros in separate files, reusable across different versions of the Tree RCU implementation. The biggest change in the source files is to use arrays to model per-CPU data, which can be scripted [Roy17]. The resulting model has 8,626 lines of C code. Around 900 lines model the Linux kernel environment, much smaller than the actual Linux kernel code used by the Tree RCU implementation. The resulting model contains assertions and can be also run as a user program, providing important validation of the model itself.

Initialization

Our model first invokes `rcu_init()`. This then invokes `rcu_init_geometry()` to compute the `rcu_node` tree geometry, `rcu_init_one` to initialize the `rcu_state` structure (which includes an array of rcu node structures organized as a tree with rcu data structures at the leaves [LMKM16]), and `rcu_cpu_notify()` to initialize each CPU's `rcu_data` structure. This initialization tunes the RCU data structures to match the specific hardware at hand. For example, a large-system tree may resemble the one shown in Figure 6.3, while a small configuration has a single `rcu_node` "tree". The model then calls `rcu_spawn_gp_kthread()` to spawn the grace-period kernel threads discussed below.

Per-CPU Variables and State

RCU uses per-CPU data to provide cache locality and to reduce contention and synchronization overhead. For example, the per-CPU structure `rcu_data` records quiescent states and handles RCU callbacks (Section 6.4.3). We model per-CPU `rcu_data` as an array, indexed by CPU ID.

It is also necessary to model per-CPU state, including the currently running task and whether or not interrupts are enabled. Identifying the running task requires a

(trivial) model of the Linux-kernel scheduler, which uses an integer array `cpu_lock`, indexed by CPU ID. Each element of this array models an exclusive lock. When a task schedules on a given CPU, it acquires the corresponding CPU lock, and releases it when scheduling away. We currently do not model preemption, so need model only voluntary context switches.

A pair of integer arrays `local_irq_depth` and `irq_lock` is used to model CPUs enabling and disabling interrupts. Both arrays are indexed by CPU ID, with the first recording each CPU's interrupt-disable nesting depth and the second recording whether or not interrupts are disabled.

Update-Side API `synchronize_sched()`

Our model omits CPU hotplug and callback handling, so we cannot use Tree RCU's normal callback mechanisms to detect the end of a grace period. We therefore use a global variable `wait_rcu_gp_flag`, initialized to 1 in `wait_rcu_gp()` before the grace period. Because `wait_rcu_gp()` blocks, it can result in a context switch; so the model invokes `rcu_note_context_switch()`, followed by a call to `rcu_process_callbacks()` to inform RCU of the resulting quiescent state. When the resulting quiescent states propagate to the root of the tree, the grace-period kernel thread is awakened. This kthread then invokes `rcu_gp_cleanup()`, the modelling of which is described below. Then `rcu_gp_cleanup()` calls `rcu_advance_cbs()`, which invokes `pass_rcu_gp()` to clear the `wait_rcu_gp_flag` flag. Inserting `__CPROVER_assume(wait_rcu_gp_flag == 0)` in `wait_rcu_gp()` prevents CBMC from continuing execution until `wait_rcu_gp_flag` is equal to 0, thus modelling the needed grace-period wait.

Scheduling-Clock Interrupt and Context Switch

The `rcu_check_callbacks()` function detects idle and usermode execution, as well as invokes RCU core processing in response to state changes. Because we model neither

idle nor usermode execution, the only state changes are context-switches and the beginnings and ends of grace periods (Section 6.4.4). So we dispense with `rcu_check_callbacks()` and instead directly call `rcu_note_context_switch()` just after releasing a CPU, which in turn calls `rcu_sched_qs()` to record the quiescent state. Finally, we call `rcu_process_callbacks()`, which notes grace-period beginnings and ends and reports quiescent states up RCU's combining tree.

Grace-Period Kthread

As discussed in Section 6.4.5, `rcu_gp_kthread()` invokes `rcu_gp_init()`, `rcu_gp_fqs()`, and `rcu_gp_cleanup()` to initialize, wait for, and clean up after each grace period, respectively. To reduce the size of the formula CBMC generates, instead of spawning a separate thread, we directly invoke `rcu_gp_init()` from `rcu_spawn_gp_kthread()` and `rcu_gp_cleanup()` from `rcu_report_qs_rsp()`. Because we model neither idle nor usermode execution, we do not need to call `rcu_gp_fqs()`.

Kernel Spin Locks

CBMC's `__CPROVER_atomic_begin()`, `__CPROVER_atomic_end()`, and `__CPROVER_assume()` built-in primitives are used to construct atomic test-and-set for `spinlock_t` and `raw_spinlock_t` acquisition and atomic reset for release. We use GCC atomic builtins for user-space execution: `while (__sync_lock_test_and_set(lock, 1))` acquires a lock and `__sync_lock_release(lock)` releases it.

Limitations

We model only the fundamental components of Tree RCU, excluding quiescent-state forcing, grace-period expediting, and callback handling. We assume all CPUs are busy executing RCU related tasks, so we do not model CPU hotplug, dyntick-idle, RCU priority boosting, or thread-migration failure modes in the Linux kernel involving

per-CPU variables. Nonetheless, we model real-world server-class RCU code paths and data layout on systems with up to 16 CPUs (default configurations) or up to either 32 or 64 CPUs (nondefault configurations on either 32-bit or 64-bit CPUs). We also model scheduling-clock interrupts as function calls; as discussed later, this results in failure to model one of the bug-injection scenarios. Finally, our test harness passes through only one grace period, so cannot detect failures involving multiple grace periods.

7.2 Verification of Tree RCU

In this section, we discuss our experimental studies of verifying the Tree RCU implementation in the Linux kernel. We first describe several bug-injection scenarios used in our experiments. Next, we report results of user-space runs of the RCU model. Then we describe how we verify the RCU model using CBMC. Finally, we discuss the experimental results. We performed our experiments on a 64-bit machine running Linux 3.19.8 with eight Intel Xeon 3.07 GHz cores and 48 GB of memory. The source code of our RCU model and the experimental data are available at https://github.com/lihaol/verify-treercu/releases/tag/date18-camera-ready.

7.2.1 Bug-Injection Scenarios

We model non-preemptible Tree RCU, so each CPU runs exactly one RCU task as a separate thread. On completion, each task increments a global counter `thread_cnt`, enabling the parent thread to verify the completion of all RCU tasks using a statement `__CPROVER_assume(thread_cnt == 2)`. The base case is the example in Figure 6.2, including the assertion `assert(r2 == 0 || r1 == 1)`, which does not hold when RCU's fundamental safety guarantee is violated: read-side critical sections cannot span grace periods.We also verify a *weak form* of liveness by inserting `assert(0)`

after the `__CPROVER_assume(thread_cnt == 2)` statement above. This assertion cannot hold, so it will be violated if any grace period completes. This "failure" is expected behavior of a correct RCU implementation. But if the assertion is *not* violated, grace periods never complete, indicating a liveness bug.

To validate our verification, we also run CBMC with the bug-injection scenarios described below. These are simplified versions of bugs encountered in actual practice. Bugs 2–6 are liveness checks and so use the aforementioned `assert(0)`; the others are safety checks, which use the assertion in Figure 6.2.

Bug 1 This bug-injection scenario makes the RCU update-side primitive `synchronize_rcu()` return immediately (line 523 in `tree_plugin.h`). With this injected bug, updaters never wait for readers, which should violate the safety check in Figure 6.2.

Bug 2 The idea behind this bug-injection scenario is to stop individual CPUs realizing that quiescent states are needed, preventing the CPUs from from recording them. As a result, grace periods do not complete. Specifically, in `rcu_gp_init()`, for each `rcu_node` structure in the combining tree, we set the field `rnp->qsmask` to 0 instead of `rnp->qsmaskinit` (line 1889 in `tree.c`). When `rcu_process_callbacks()` is called, `rcu_check_quiescent_state()` will invoke `__note_gp_changes()` that sets `rdp->qs_pending` to 0, indicating that RCU needs no quiescent state from the corresponding CPU. So `rcu_check_quiescent_state()` will return without calling `rcu_report_qs_rdp()`, preventing grace periods from completing. This liveness violation should fail to trigger a violation of the end-of-execution `assert(0)`.

Bug 3 This bug-injection scenario is a variant of Bug 2, in which each CPU remains aware that quiescent states are required but incorrectly believes that it has already reported a quiescent state for the current grace period. To accomplish this, in `__note_gp_changes()`, we clear `rnp->qsmask` by adding `rnp->qsmask &= ~rdp->grpmask;`

in the last `if` code block (line 1739 in `tree.c`). So `rcu_report_qs_rnp()` never walks up the `rcu_node` tree, which results in a liveness violation as in Bug 2.

Bug 4 This is an alternative code change that gets the same effect as Bug 2. In `__note_gp_changes()`, we set the `rdp->qs_pending` field to 0 directly (line 1749 in `tree.c`), resulting in a liveness violation.

Bug 5 In this bug-injection scenario, CPUs remain aware of the need for quiescent states but are prevented from recording them, so grace periods do not complete. To achieve this, we modify function `rcu_sched_qs()` to return immediately (line 246 in `tree.c`), so that quiescent states are not recorded. Grace periods therefore never complete, which is a liveness violation.

Bug 6 In this bug, CPUs are aware of the need for quiescent states and also record them locally. However, CPUs are prevented from reporting quiescent states up the `rcu_node` tree, which again prevents grace periods from completing. This bug modifies function `rcu_report_qs_rnp()` to return immediately (line 2227 in `tree.c`). This prevents RCU from walking up the `rcu_node` tree, thus preventing grace periods from ending. It is again a liveness violation similar to Bug 2.

Bug 7 Where Bug 6 prevents quiescent states from being reported up the `rcu_node` tree, this bug causes quiescent states to be reported up the tree prematurely, before the CPUs covered by a given subtree have all reported quiescent states. To inject this bug, in `rcu_report_qs_rnp()`, we remove the `if`-block checking for `rnp->qsmask != 0 || rcu_preempt_blocked_readers_cgp(rnp)` (line 2251 in `tree.c`). Tree walking will then not stop until it reaches the root, resulting in too-short grace periods. This is therefore a safety violation similar to Bug 1.

7.2.2 Validating the RCU Model in User-Space

Scenario	#Successful Runs	#Failing Runs	#Timeouts	Max VM	Runtime	Result
Prove	1,000 (100.0%)	0 (0.0%)	0 (0.0%)	361.5 MB	3mins 51s	Safe
Prove-GP	0 (0.0%)	1,000 (100.0%)	0 (0.0%)	361.5 MB	5mins 9s	GP Completed
Bug 1	461 (46.1%)	539 (53.9%)	0 (0.0%)	361.5 MB	5mins 26s	Assertion Violated
Bug 2	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	GP Hung
Bug 3	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	GP Hung
Bug 4	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	GP Hung
Bug 5	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	GP Hung
Bug 6	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	GP Hung
Bug 7	0 (0.0%)	0 (0.0%)	1,000 (100.0%)	361.5 MB	16h 40mins	**Safe (Bug Missed)**
Bug 7 (2 readers)	758 (75.8%)	242 (24.2%)	0 (0.0%)	369.7 MB	4mins 40s	Assertion Violated

Table 7.1: Experimental results of testing the RCU model in user-space

We ran our RCU model in user space before formal verification by CBMC. We performed 1000 runs for each scenario described in Section 7.2.1 with a 60 s timeout for each run to wait for the end of a grace period and a random delay between 0 to 1 s in the RCU reader task.

The results are reported in Table 7.1. Column 1 gives the verification scenarios. Scenario Prove tests our RCU model without bug injection. Scenario Prove-GP tests a weak form of liveness by replacing Figure 6.2's assertion with `assert(0)` as described in Section 7.2.1. The next three columns present the number and the percentage of successful, failing, and timeout runs, respectively. The following two columns give the maximum memory consumption and the total runtime. The last column explains the results.

As expected, for the Prove scenario, testing the model without bug injection always ran to completion successfully. Testing a weak form of liveness using `assert(0)` (scenario Prove-GP) evidenced the end of a grace period by triggering an assertion violation in all runs. For Bug 1, an assertion violation was triggered in 539 out of 1000 runs. For Bugs 2–6, the user program timed out in all the runs, thus a grace period did not complete. For Bug 7 with one reader thread, the testing harness failed to trigger an assertion violation. But we were able to observe a failure in 242 out of 1000 runs with two reader threads.

7.2.3 Getting CBMC to work on Tree RCU

Getting CBMC to work on our RCU model is non-trivial, owing to Tree RCU's complexity combined with CBMC's bit-precise verification. Early attempts resulted in SAT formulas that were so large that CBMC ran out of memory. After the optimizations described in this section, the largest formula contained around 90 million variables and 450 million clauses, enabling CBMC to run to completion.

First, instead of placing the scheduling-clock interrupt in its own thread (as the thread instrumentation in Section 3.3 and the partial-order encoding in Chapter 4), we invoke functions `rcu_note_context_switch()` and `rcu_process_callbacks()` directly, as described in Section 7.1. Also, we invoke `__note_gp_changes()` from `rcu_gp_init()` to notify each CPU of a new grace period, instead of invoking `rcu_process_callbacks()`.

Second, the support for linked lists in CBMC version 5.4 is limited, resulting in unreachable code in CBMC's symbolic execution. Thus, we stubbed all list-related code in our RCU model, including that for callback handling.

Third, CBMC's structure-pointer and array encodings result in large formulas and long formula-generation times. Our focus on the RCU-sched flavor allowed us to eliminate data structures of other RCU flavors and trivialize the `for_each_rcu_flavor()` flavor-traversal loops. Our focus on small numbers of CPUs meant that RCU-sched's `rcu_node` tree contained only a root node, so we also trivialized the `rcu_for_each_node_breadth_first()` loops traversing this tree.

Fourth, CBMC unwinds each loop to the depth specified in its command line option `--unwind`, even when the actual loop depth is smaller. This unnecessarily increases formula size, especially for loops containing intricate RCU code. Since loops in our model can be decided at compile time, we used the command line option `--unwindset` to specify unwinding depths for each individual loop.

Finally, since our test harness only requires one `rcu_node` structure and two `rcu_`

`data` structures, we can use 32-bit encodings for `int`, `long`, and pointers by using the command line option `--ILP32`. This reduces CBMC's formula size by half compared to the 64-bit default.

7.2.4 Results and Discussion

Since the Tree RCU implementation in the Linux kernel is sophisticated, it has a non-trivial test suite [McK05], comprising several thousand lines of code. So it comes as little surprise that its verification is challenging.

Table 7.2 presents the results of our experiments using CBMC version 5.4 to verify our RCU model. Scenario Prove verifies our RCU model without bug injection over Sequential Consistency (SC). We also exercise the model over the weak memory models TSO and PSO with Prove-TSO and Prove-PSO, respectively. Scenario Prove-GP performs the same reachability check as in Section 7.2.2 over SC. We perform the same reachability verification over TSO and PSO in scenarios Prove-GP-TSO and Prove-GP-PSO, respectively. Scenarios Bug 1–7 are the bug-injection scenarios discussed in Section 7.2.1 and are verified over SC, TSO and PSO. Columns 2–4 give the number of constraints (symbolic program expressions and partial orders), variables, and clauses of the generated formula. The next three columns give the maximum (virtual) memory consumption, solver runtime, and total runtime of our experiments. The final column gives the verification results.

In our experiments, CBMC returned all the expected results except for Bug 7, for which it failed to report a violation of the assertion `assert(r2 == 0 || r1 == 1)` with one RCU reader thread running over SC. This was due to approximation of the scheduling-clock interrupt by a direct function call, as described in Section 7.1. However, CBMC did report a violation of the assertion either when two RCU reader threads were present or when run over TSO or PSO. All of these cases decrease determinism, which in turn more faithfully model non-deterministic scheduling-clock

Scenario	#Constraints	#Variables	#Clauses	Max VM	Solver Time	Total Time	Result
Prove	5,279,600	30,085,337	149,758,548	23.27 GB	9h 24mins	9h 36mins	Safe
Prove-TSO	5,646,959	42,042,386	210,708,442	34.00 GB	10h 51mins	11h 4mins	Safe
Prove-PSO	5,617,154	41,327,066	207,042,629	33.76 GB	11h 23mins	11h 36mins	Safe
Prove-GP	5,476,540	30,655,428	152,743,545	23.90 GB	3h 52mins	4h 5mins	GP Completed
Prove-GP-TSO	5,646,940	42,041,740	210,705,615	34.00 GB	13h 1mins	13h 14mins	GP Completed
Prove-GP-PSO	5,617,135	41,326,420	207,039,802	33.76 GB	8h 24mins	8h 37mins	GP Completed
Bug 1	1,343,449	11,719,966	56,027,980	8.24 GB	31mins	33mins	Assertion Violated
Bug 1-TSO	1,540,645	17,120,555	83,392,397	12.60 GB	53mins	56mins	Assertion Violated
Bug 1-PSO	1,514,657	16,548,819	80,481,851	12.42 GB	46mins	48mins	Assertion Violated
Bug 2	5,279,584	30,056,615	149,643,492	23.26 GB	4h 25mins	4h 37mins	GP Hung
Bug 2-TSO	5,646,940	42,013,372	210,592,015	34.01 GB	9h 57mins	10h 10mins	GP Hung
Bug 2-PSO	5,617,135	41,298,052	206,926,202	33.75 GB	8h 51mins	9h 4mins	GP Hung
Bug 3	6,374,373	34,856,577	174,131,331	28.04 GB	7h 11mins	7h 25mins	GP Hung
Bug 3-TSO	6,805,631	48,788,433	245,157,184	41.18 GB	19h 40mins	19h 55mins	GP Hung
Bug 3-PSO	6,773,763	48,023,601	241,237,629	40.95 GB	19h 19mins	19h 35mins	GP Hung
Bug 4	4,847,980	27,804,363	138,197,043	22.18 GB	4h 3mins	4h 14mins	GP Hung
Bug 4-TSO	5,170,928	38,480,891	192,605,939	31.49 GB	8h 18mins	8h 30mins	GP Hung
Bug 4-PSO	5,141,123	37,765,571	188,940,126	31.27 GB	8h 14mins	8h 26mins	GP Hung
Bug 5	5,161,874	29,510,828	146,787,005	23.02 GB	4h 6mins	4h 18mins	GP Hung
Bug 5-TSO	5,522,168	41,239,083	206,569,643	33.65 GB	5h 46mins	5h 59mins	GP Hung
Bug 5-PSO	5,492,607	40,529,619	202,933,839	33.04 GB	5h 42mins	5h 55mins	GP Hung
Bug 6	1,410,495	13,165,176	63,302,559	9.03 GB	19mins	21mins	GP Hung
Bug 6-TSO	1,541,937	17,286,058	84,131,818	12.59 GB	1h 32mins	1h 33mins	GP Hung
Bug 6-PSO	1,518,307	16,766,198	81,485,361	12.44 GB	1h 22mins	1h 24mins	GP Hung
Bug 7	5,022,249	29,242,760	145,389,516	22.87 GB	8h 48mins	9h	Safe (Bug Missed)
Bug 7-TSO	5,201,744	40,139,251	200,857,404	31.93 GB	11h 6mins	11h 18mins	Assertion Violated
Bug 7-PSO	5,172,720	39,442,675	197,287,644	31.71 GB	11h 32mins	11h 44mins	Assertion Violated
Bug 7 (2 readers) *	15,165,557	71,205,400	359,021,922	59.07 GB	19h 2mins	19h 40mins	Assertion Violated
Bug 7-TSO (2 readers) *	15,691,102	90,444,903	456,973,933	74.80 GB	78h 12mins	78h 53mins	Assertion Violated
Bug 7-PSO (2 readers) *	15,647,504	89,398,551	451,611,664	74.51 GB	84h 21mins	85h 2mins	Solver Out of Memory

* This experiment was performed on a 64-bit machine running Linux 3.19.8 with twelve Intel Xeon 2.40 GHz cores and 96 GB of main memory

Table 7.2: Experimental results of CBMC

interrupts, allowing the assertion to be violated.

CBMC took over 9 hours to verify our model over SC (scenario Prove). The resulting SAT formulas have more than 5m constraints, 30m variables, and 149m clauses, occupying 23 GB of memory. The formulas for Prove-TSO and Prove-PSO are about 40% larger than for Prove. They have more than 40m variables and 200m clauses, and took more than 11 hours and 33 GB memory to solve. Although this verification consumed considerable memory and CPU, it verified all possible executions and reorderings permitted by TSO and PSO, a tiny subset of which are reached by the `rcutorture` test suite.

CBMC proved that grace periods can end (i.e., `assert(0)` is violated) over SC (Prove-GP), TSO (Prove-GP-TSO), and PSO (Prove-GP-PSO). The formula size and memory consumption are similar to those of the three Prove scenarios. However, it took CBMC only about 4, 13, and 8.5 hours to find an violation of `assert(0)` in

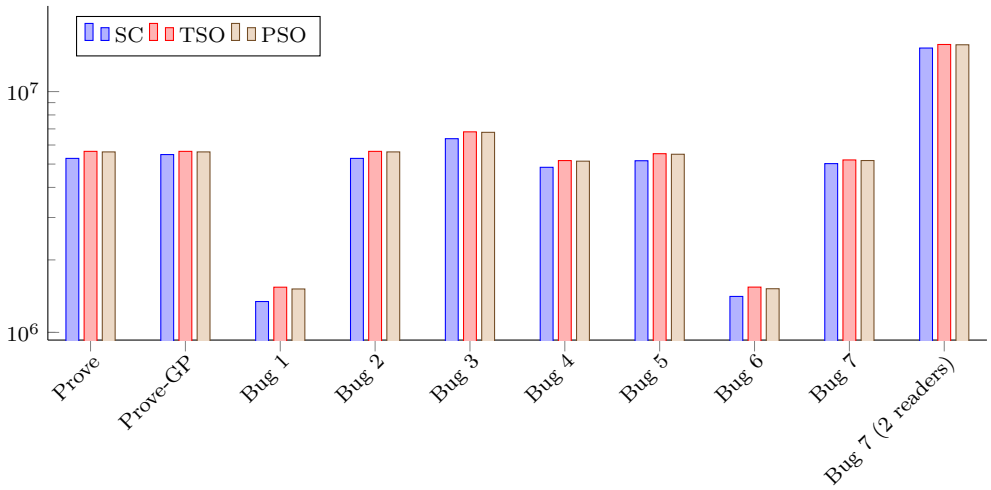


Figure 7.1: Number of constraints in the SAT formulas

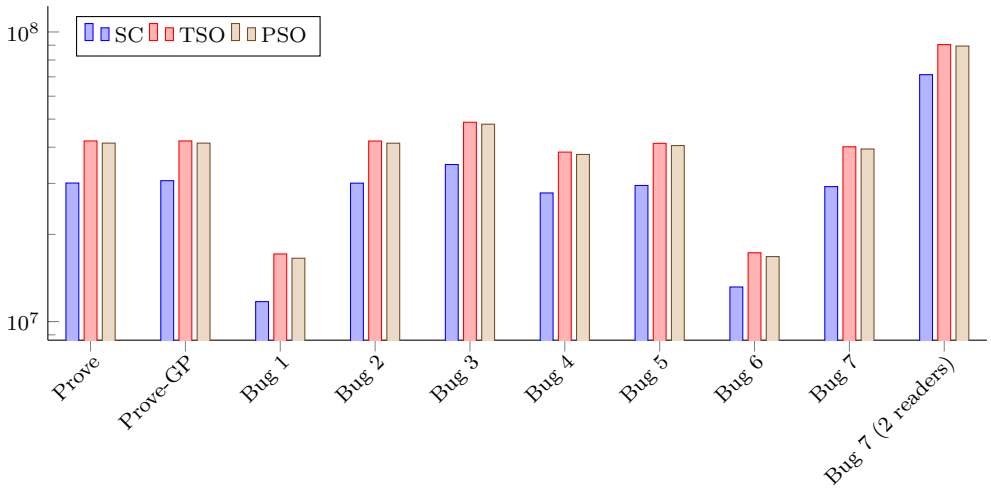


Figure 7.2: Number of variables in the SAT formulas

Prove-GP, Prove-GP-TSO, and Prove-GP-PSO, respectively.

For the bug-injection scenarios described in Section 7.2.1, CBMC was able to return the expected results in all scenarios over SC except for Bug 7, as noted earlier. The formula size varies from scenario to scenario, with 27m–35m variables and 138m–174m clauses. The runtime was 4–9 hours and memory consumption exceeded 22 GB. The exceptions are Bugs 1 and 6, which have fewer than 14m variables and 64m clauses, and took less than 35 mins and about 9 GB of memory to solve. This reduction was due to the large amount of code removed by the bug injections in these cases.

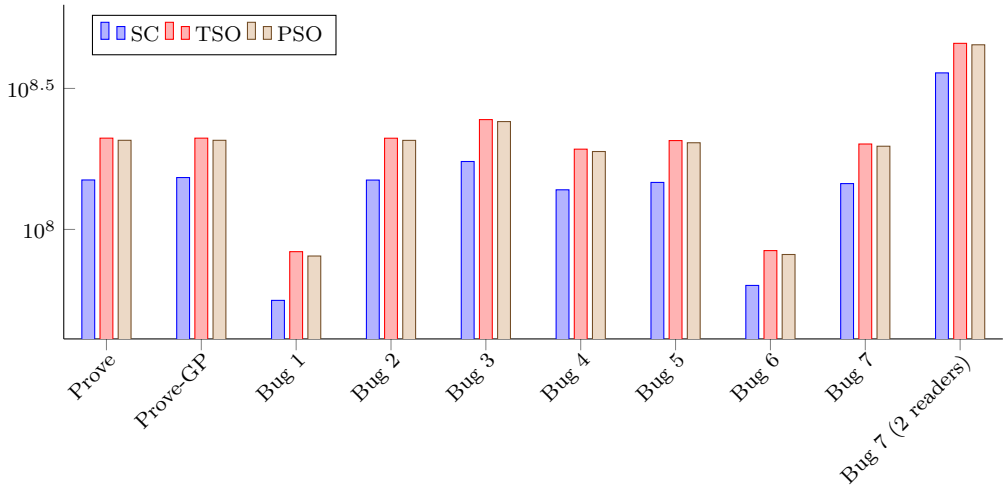


Figure 7.3: Number of clauses in the SAT formulas

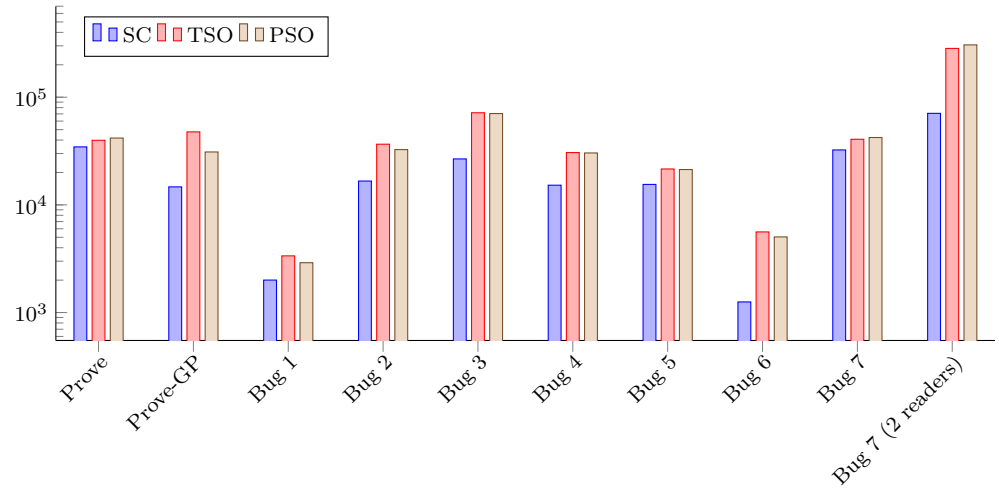


Figure 7.4: Total runtime in seconds

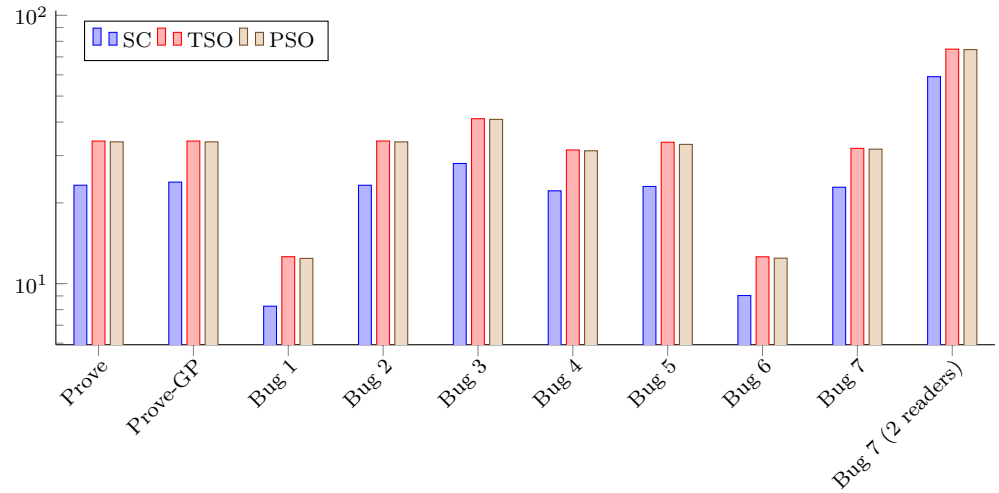


Figure 7.5: Maximum memory consumption in gigabytes

Figures 7.1–7.3 compare the formula size between SC, TSO and TSO. Comparison of runtime and memory can be found in Figures 7.4 and 7.5. As we can see, runtime and memory overhead for TSO and PSO are quite similar. The overheads of TSO are slightly higher than those of PSO in all bug-injection scenarios except for Bug 7 on which PSO had longer runtime.

However, the overhead of TSO and PSO significantly exceeds that of SC, with up to 340% (Bug 6 runtime) and 50% (Bug 1 memory) increases. The runtime was 5–19 hours and memory use exceeded 31 GB in all scenarios except Bug 1 and 6, owing to the large amount of code removed for these two scenarios. The numbers of variables and clauses are 37m–49m and 188m–245m, respectively, around 130% greater than for SC.

The two-reader variant of Bug 7 has by far the longest runtime, taking more than 19 hours and 78 hours over SC and TSO, respectively, compared to 9 hours and 11 hours with one reader. It also used about 75 GB memory, more than double the one-reader variant. For PSO, with two reader threads CBMC's solver ran out of memory after 85 hours, whereas with one reader it completed in less than 12 hours. The increased overhead is due to the additional RCU reader's call to `rcu_process_callbacks()`. This in turn results in more than a 125% increase in the number of constraints, variables, and clauses. For example, the two-reader TSO formula has triple the constraints and double the variables and clauses of the one-reader case.

Summary

In this chapter, we describe our experience of verifying the key components of the Tree RCU implementation in the Linux kernel using the CBMC verification tool. Modifications of the Tree RCU source code are required to get CBMC work on a model manually constructed from the Linux kernel, which could potentially be scripted in a real-world software engineering process [Roy17]. We also highlight limitations of

our Tree RCU model. In our experiments, CPU time and memory consumption are large on a Tree RCU model with three nodes without modelling the scheduling-clock interrupt as a separate thread. This is the main reason that we chose to use CBMC instead of i-CBMC, which would otherwise produce an even larger SAT formula (as demonstrated in the experiments of Chapter 5). We believe this verification effort sits on the current limit of CBMC and SAT solving in terms of scalability.

In the next and the final chapter, we conclude and discuss future research directions.

Chapter 8

Conclusion

This thesis addresses the research problem of interrupt-driven software validation. Interrupt-driven programs are notoriously difficult to write and debug. Testing is quite often not enough as interrupts, once enabled, can arrive at arbitrary times, leading to an exponential blow-up in the number of executions to consider. This situation becomes much worse when there are multiple interrupt types whose execution can be nested, and when highly concurrent software makes extensive use of interrupts, such as Read-Copy Update (RCU) in the Linux kernel.

The first part of this thesis proposes two approaches to verify software with nested interrupts. The first approach is based on source-to-source transformations to sequential and multi-threaded programs that can be handled by off-the-shelf verification tools. The second approach is based on symbolic execution. It leverages a partial-order encoding that more precisely models the interleaving semantics of nested interrupts than threads do. We extensively compare the proposed methods on benchmarks drawn from real-world programs.

Our experimental results presented in Chapter 5 have demonstrated that the partial-order encoding described in Chapter 4 is to date the most effective method to verify software with nested interrupts, followed by the thread instrumentation

introduced in Section 3.3. The interrupt sequentialisation presented in Section 3.2 performs the worst because of the recursion in its transformation.

The second part of this thesis demonstrates the nascent ability of SAT-based formal verification tools to handle real-world production-quality synchronization primitives, as exemplified by Linux-kernel Tree RCU on weakly ordered TSO and PSO systems. In Chapter 6, we give a detailed account of RCU and its tree-based implementation in the Linux kernel. Then Chapter 7 describes how to construct a model from the source code of Tree RCU. In this chapter we show, for the first time, how to use the tool CBMC to verify the core components of the Tree RCU implementation.

In our experiments, CPU time and memory consumption are large on a Tree RCU model with three nodes without modelling the scheduling-clock interrupt as a separate thread. We believe this verification effort sits on the current limit of CBMC and SAT solving in terms of scalability. Yet it demonstrates that RCU is a rich example to drive research: It is small enough to provide models that can just barely be verified by existing tools, but it also has sufficient concurrency and complexity to drive significant advances in techniques and tooling. It also confirms the tractability and practicality of the use of bug injection to validate both the model and the tools. We hope our work will be a step towards integrating verification into RCU' s regression test suite in the Linux kernel.

There are a number of research directions which we believe are promising to lead to such advances. We describe them below.

Future Work

For the verification of software with nested interrupts, our partial-order encoding is based on bounded model checking. It is worth implementing the method in the model checker Impara [WKO13], which is capable of proving unbounded safety.

There are also potential improvements for CBMC to better support future RCU verification:

- Better support of lists will be needed to verify RCU' s callback handling mechanism.
- A field-sensitive SSA encoding for structures and a thread-aware slicer will help reduce encoding size and therefore improve scalability.

For the modelling of RCU, there are quite a few interesting directions:

- Use more sophisticated test harnesses that pass through multiple grace periods and operate on a larger tree structure.
- Add quiescent-state forcing and grace-period expediting into our model and verify their safety and liveness properties.
- Model and verify the preemptible version of Tree RCU, which we expect to be quite challenging.
- Verify uses of RCU in the Linux kernel, for example, there is much fertile ground in the Virtual File System (VFS).

Bibliography

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AGC12a] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. Craig interpretation. In Antoine Miné and David Schmidt, editors, *Proceedings of the 19th International Symposium on Static Analysis (SAS), Deauville, France, September 11-13, 2012*, volume 7460 of *Lecture Notes in Computer Science*, pages 300–316. Springer, 2012.

[AGC12b] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. From under-approximations to over-approximations and back. In Cormac Flanagan and Barbara König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Tallinn, Estonia, March 24 - April 1, 2012*, volume 7214 of *Lecture Notes in Computer Science*, pages 157–172. Springer, 2012.

[AGC12c] Aws Albarghouthi, Arie Gurfinkel, and Marsha Chechik. WHALE: An interpolation-based algorithm for inter-procedural verification. In Viktor Kuncak and Andrey Rybalchenko, editors, *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), Philadelphia, PA, USA, January 22-24, 2012*, volume 7148 of *Lecture Notes in Computer Science*, pages 39–55. Springer, 2012.

[AKNR12] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing top-down interprocedural analyses. In Jan Vitek, Haibo Lin, and Frank Tip, editors, *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Beijing, China, June 11-16, 2012*, pages 217–228. ACM, 2012.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Proceedings of the 25th International Conference on Computer Aided Verification (CAV), Saint Petersburg, Russia, July 13-19, 2013*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013.

[ALGC12] Aws Albarghouthi, Yi Li, Arie Gurfinkel, and Marsha Chechik. UFO: A framework for abstraction- and interpolation-based software verification. In *Proceedings of the 24th International Conference on Computer Aided Verification*, volume 7358 of *Lecture Notes in Computer Science*, pages 672–678. Springer, 2012.

[AMSS12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, 2012.

[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In Jeanne Ferrante and P. Mager, editors, *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL), San Diego, California, USA, January 10-13, 1988*, pages 1–11. ACM, 1988.

[BBC+06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. Thorough static analysis of device drivers. In Yolande Berbers and Willy Zwaenepoel, editors, *Proceedings of the First European Conference on Computer Systems (EuroSys), Leuven, Belgium, April 18-21, 2006*, pages 73–85. ACM, 2006.

[BBKL10] Thomas Ball, Ella Bounimova, Rahul Kumar, and Vladimir Levin. SLAM2: static driver verification with under 4% false alarms. In Roderick Bloem and Natasha Sharygina, editors, *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Lugano, Switzerland, October 20-23, 2010*, pages 35–42. IEEE, 2010.

[BBLdM10] Thomas Ball, Ella Bounimova, Vladimir Levin, and Leonardo de Moura. Efficient evaluation of pointer predicates with Z3 SMT solver in SLAM2. Technical Report MSR-TR-2010-24, Microsoft Research, March 2010.

[BBT05] Bodil Biering, Lars Birkedal, and Noah Torp-Smith. BI hyperdoctrines and higher-order separation logic. In Shmuel Sagiv, editor, *Proceedings of the 14th European Symposium on Programming (ESOP), Edinburgh, UK, April 4-8, 2005*, volume 3444 of *Lecture Notes in Computer Science*, pages 233–247. Springer, 2005.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In Werner Damm and Holger Hermanns, editors, *Proceedings of the 19th International Conference on Computer Aided Verification (CAV), Berlin, Germany, July 3-7, 2007*, volume 4590 of *Lecture Notes in Computer Science*, pages 178–192. Springer, 2007.

[BCHS98] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Journal of Software Practice and Experience*, 28(8):859–881, 1998.

[BCI11] Josh Berdine, Byron Cook, and Samin Ishtiaq. SLAyer: Memory safety for systems-level code. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV), Snowbird, UT, USA, July 14-20, 2011*, volume 6806 of *Lecture Notes in Computer Science*, pages 178–183. Springer, 2011.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, 1992.

[BDP01] Dennis Brylow, Niels Damgaard, and Jens Palsberg. Static checking of interrupt-driven software. In Hausi A. Müller, Mary Jean Harrold, and Wilhelm Schäfer, editors, *Proceedings of the 23rd International Conference on Software Engineering, (ICSE), Toronto, Ontario, Canada, 12-19 May, 2001*, pages 47–56. IEEE Computer Society, 2001.

[BK11] Doina Bucur and Marta Z. Kwiatkowska. On software verification for sensor nodes. *Software and Systems*, 84(10):1693–1707, 2011.

[BLR11] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Communications of the ACM*, 54(7):68–76, 2011.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(6):1684–1698, 1994.

[BNRS08] Nels E. Beckman, Aditya V. Nori, Sriram K. Rajamani, and Robert J. Simmons. Proofs from tests. In Barbara G. Ryder and Andreas Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Seattle, WA, USA, July 20-24, 2008*, pages 3–14. ACM, 2008.

[BP04] Dennis Brylow and Jens Palsberg. Deadline analysis of interrupt-driven software. *IEEE Transactions on Software Engineering*, 30(10):634–655, 2004.

[BR00] Thomas Ball and Sriram K. Rajamani. Boolean programs: A model and process for software analysis. Technical Report MSR-TR-2000-14, Microsoft Research, February 2000.

[BR02a] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In John Launchbury and John C. Mitchell, editors, *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Portland, OR, USA, January 16-18, 2002*, pages 1–3. ACM, 2002.

[BR02b] Thomas Ball and Sriram K. Rajamani. SLIC: A specification language for interface checking (of C). Technical Report MSR-TR-2001-21, Microsoft Research, January 2002.

[Cal15] John Callaham. Google says there are now 1.4 billion active Android devices worldwide. http://www.androidcentral.com/google-says-there-are-now-14-billion-active-android-devices-worldwide, September 2015.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *Proceedings of the 7th NASA Formal Methods Symposium (NFM), Pasadena, CA, USA, April 27-29, 2015*, volume 9058 of *Lecture Notes in Computer Science*, pages 3–11. Springer, 2015.

[CDE08] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In Richard Draves and Robbert van Renesse, editors, *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI), San Diego, California, USA, December 8-10, 2008*, pages 209–224. USENIX Association, 2008.

[CF10] Lucas C. Cordeiro and Bernd Fischer. Bounded model checking of multi-threaded software using SMT solvers. *CoRR*, abs/1003.3830, 2010.

[CF11] Lucas C. Cordeiro and Bernd Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In Richard N. Taylor, Harald C. Gall, and Nenad Medvidovic, editors, *Proceedings of the 33rd International Conference on Software Engineering (ICSE), Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 331–340. ACM, 2011.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL), Austin, Texas, USA, January 11-13, 1989*, pages 25–35. ACM Press, 1989.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Proceedings of the 12th International Conference on Computer Aided Verification (CAV), Chicago,*

IL, USA, July 15-19, 2000, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 2001.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Proceedings of the 10th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Barcelona, Spain, March 29 - April 2, 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKS05] Byron Cook, Daniel Kroening, and Natasha Sharygina. Symbolic model checking for asynchronous boolean programs. In Patrice Godefroid, editor, *Proceedings of the 12th International SPIN Workshop on Model Checking of Software (SPIN), San Francisco, CA, USA, August 22-24, 2005*, volume 3639 of *Lecture Notes in Computer Science*, pages 75–90. Springer, 2005.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Edinburgh, UK, April 4-8, 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[CKY03] Edmund M. Clarke, Daniel Kroening, and Karen Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference (DAC), Anaheim, CA, USA, June 2-6, 2003*, pages 368–371. ACM, 2003.

[CMM+04] Krishnendu Chatterjee, Di Ma, Rupak Majumdar, Tian Zhao, Thomas A. Henzinger, and Jens Palsberg. Stack size analysis for interrupt-driven programs. *Information and Computation*, 194(2):144–174, 2004.

[Coq] The coq proof assistant. https://coq.inria.fr/.

[Cor10] Lucas Cordeiro. SMT-based bounded model checking for multi-threaded software in embedded systems. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, 1-8 May 2010*, pages 373–376. ACM, 2010.

[Cor14] Jonathan Corbet. The RCU-tasks subsystem. http://lwn.net/Articles/607117/, July 2014.

[CYC+01] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson R. Engler. An empirical study of operating system errors. In Keith Marzullo and M. Satyanarayanan, editors, *Proceedings of the 18th ACM Symposium on Operating System Principles (SOSP), Chateau Lake Louise, Banff, Alberta, Canada, October 21-24, 2001*, pages 73–88. ACM, 2001.

[DLW15] Matthias Dangl, Stefan Löwe, and Philipp Wendler. Cpachecker with support for recursive programs and floating-point arithmetic (competition contribution). In Christel Baier and Cesare Tinelli, editors, *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), London, UK, April 11-18, 2015*, volume 9035 of *Lecture Notes in Computer Science*, pages 423–425. Springer, 2015.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Budapest, Hungary, March 29 - April 6, 2008*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DMD13] Mathieu Desnoyers, Paul E. McKenney, and Michel R. Dagenais. Multi-core systems modeling for formal verification of parallel algorithms. *ACM SIGOPS Operating Systems Review*, 47(2):51–65, 2013.

[DMS+12] Mathieu Desnoyers, Paul E. McKenney, Alan S. Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23(2):375–382, 2012.

[EDE08] Dhammika Elkaduwe, Philip Derrin, and Kevin Elphinstone. Kernel design for isolation and assurance of physical memory. In *Proceedings*

of the 1st Workshop on Isolation and Integration in Embedded Systems (IIES), pages 35–40. ACM, 2008.

[Eme91] E. Allen Emerson. Temporal and modal logic. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics (B)*, pages 995–1072. The MIT Press, 1991.

[FSDG08] Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. Certifying low-level programs with hardware interrupts and preemptive threads. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI), Tucson, AZ, USA, June 7-13, 2008*, pages 170–182. ACM, 2008.

[GAJM15] Alex Groce, Iftekhar Ahmed, Carlos Jensen, and Paul E. McKenney. How verified is my code? Falsification-driven verification. In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, November 9-13, 2015*, pages 737–748. IEEE, 2015.

[GBC06] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In Kwangkeun Yi, editor, *Proceedings of the 13th International Symposium on Static Analysis (SAS), Seoul, Korea, August 29-31, 2006*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer, 2006.

[GHK+06] Bhargav S. Gulavani, Thomas A. Henzinger, Yamini Kannan, Aditya V. Nori, and Sriram K. Rajamani. SYNERGY: a new algorithm for property checking. In Michal Young and Premkumar T. Devanbu, editors, *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Portland, Oregon, USA, November 5-11, 2006*, pages 117–127. ACM, 2006.

[GNRT10] Patrice Godefroid, Aditya V. Nori, Sriram K. Rajamani, and SaiDeep Tetali. Compositional may-must program analysis: unleashing the power of alternation. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Madrid, Spain, January 17-23, 2010*, pages 43–56. ACM, 2010.

[God94] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*. PhD thesis, University of Liege, Computer Science Department, 1994.

[GPR11] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. Predicate abstraction and refinement for verifying multi-threaded programs. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), Austin, TX, USA, January 26-28, 2011*, pages 331–344. ACM, 2011.

[GRY13] Alexey Gotsman, Noam Rinetzky, and Hongseok Yang. Verifying concurrent memory reclamation algorithms with grace. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming (ESOP), Rome, Italy, March 16-24, 2013*, volume 7792 of *Lecture Notes in Computer Science*, pages 249–269. Springer, 2013.

[GSW95] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(1):85–122, 1995.

[GW91] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Kim Guldstrand Larsen and Arne Skou, editors, *Proceedings of the 3rd International Workshop on Computer Aided Verification (CAV), Aalborg, Denmark, July, 1-4, 1991*, volume 575 of *Lecture Notes in Computer Science*, pages 332–342. Springer, 1991.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transaction of Software Engineering*, 23(5):279–295, 1997.

[HTV+13] Alex Horn, Michael Tautschnig, Celina G. Val, Lihao Liang, Tom Melham, Jim Grundy, and Daniel Kroening. Formal co-validation of low-level hardware/software interfaces. In Barbara Jobstmann and Sandip Ray, editors, *Proceedings of the 13rd International Conference on Formal Methods in Computer-Aided Design (FMCAD), Portland, OR, USA, October 20-23, 2013*, pages 121–128. IEEE, 2013.

[INF+15] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In Myra B. Cohen, Lars Grunske, and Michael Whalen, editors, *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), Lincoln, NE, USA, November 9-13, 2015*, pages 807–812. IEEE, 2015.

[ITF+14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In Armin Biere and Roderick Bloem, editors, *Proceedings of the 26th International Conference on Computer Aided Verification (CAV), Vienna, Austria, July 18-22, 2014*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014.

[Jia08] Lai Jiangshan. [RFC][PATCH] rcu classic: new algorithm for callbacks-processing. Available: Linux-kernel git SHA-1 5127bed588a2 [Viewed July 9, 2016], June 2008.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP), Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009.

[KJV10] Nicholas Kidd, Suresh Jagannathan, and Jan Vitek. One stack to run them all – reducing concurrent analysis to sequential analysis under priority scheduling. In Jaco van de Pol and Michael Weber, editors, *Proceedings of the 17th International SPIN Workshop on Model Checking Software (SPIN), Enschede, The Netherlands, September 27-29, 2010*, volume 6349 of *Lecture Notes in Computer Science*, pages 245–261. Springer, 2010.

[KLM+15] Daniel Kroening, Lihao Liang, Tom Melham, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with nested interrupts. In Wolfgang Nebel and David Atienza, editors, *Proceedings of the 2015 Design, Automation and Test in Europe Conference*

(DATE), Grenoble, France, March 9-13, 2015, pages 229–234. ACM, 2015.

[KP88] Shmuel Katz and Doron Peled. An efficient verification method for parallel and distributed programs. In J. W. de Bakker, Willem P. de Roever, and Grzegorz Rozenberg, editors, *Proceedings of the Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency, Noordwijkerhout, The Netherlands, May 30 - June 3, 1988*, volume 354 of *Lecture Notes in Computer Science*, pages 489–507. Springer, 1988.

[KS17] Michalis Kokologiannakis and Konstantinos Sagonas. Stateless model checking of the Linux kernel's hierarchical Read-Copy-Update (Tree RCU). In Hakan Erdogmus and Klaus Havelund, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Model Checking of Software (SPIN), Santa Barbara, CA, USA, July 10-14, 2017*, pages 172–181. ACM, 2017.

[KSS11] Jonathan Kotker, Dorsa Sadigh, and Sanjit A. Seshia. Timing analysis of interrupt-driven programs under context bounds. In Per Bjesse and Anna Slobodová, editors, *Proceedings of the 11th International Conference on Formal Methods in Computer-Aided Design (FMCAD), Austin, TX, USA, October 30 - November 02, 2011*, pages 81–90. FMCAD Inc., 2011.

[Lam77] Leslie Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3(2):125–143, 1977.

[Lin] The Linux kernel. https://www.kernel.org/.

[Lin13] Where is the internet? http://www.whoishostingthis.com/blog/2013/12/06/internet-infographic/, December 2013.

[LJZ+16] Han Liu, Yu Jiang, Huafeng Zhang, Ming Gu, and Jiaguang Sun. Taming interrupts for verifying industrial multifunction vehicle bus controllers. In John S. Fitzgerald, Constance L. Heitmeyer, Stefania Gnesi, and Anna Philippou, editors, *Proceedings of the 21st International Symposium on Formal Methods*, volume 9995 of *Lecture Notes in Computer Science*, pages 764–771. Springer, 2016.

[LLV] The LLVM compiler infrastructure. https://llvm.org/.

[LMK+18] Lihao Liang, Tom Melham, Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Effective verification of low-level software with competing interrupts. *ACM Transactions on Embedded Computing Systems (TECS)*, 17(2), 2018.

[LMKM16] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of the tree-based hierarchical read-copy update in the linux kernel. *CoRR*, abs/1610.03052, 2016.

[LMKM18] Lihao Liang, Paul E. McKenney, Daniel Kroening, and Tom Melham. Verification of tree-based hierarchical read-copy update in the Linux kernel. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE)*, pages 61–66. IEEE, 2018.

[LO11] Lihao Liang and C.−H. Luke Ong. Formal analysis of low-level programs with interrupts. Technical report, University of Oxford, 2011.

[LR09] Akash Lal and Thomas W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. *Formal Methods in System Design (FMSD)*, 35(1):73–97, 2009.

[LRJ13] Chunxiao Li, Anand Raghunathan, and Niraj K. Jha. Improving the trustworthiness of medical device software with formal verification methods. *Embedded Systems Letters*, 5(3):50–53, 2013.

[MBWW13] Paul E. McKenney, Silas Boyd-Wickizer, and Jonathan Walpole. RCU usage in the Linux kernel: One decade later. Technical Report, 2013.

[McK05] Paul E. McKenney. [PATCH] RCU torture testing. Available: http://lkml.kernel.org/g/20051001182056.GA1613@us.ibm.com [Viewed July 9, 2016], October 2005.

[McK08a] Paul E. McKenney. Integrating and validating dynticks and preemptable RCU. https://lwn.net/Articles/279077/, April 2008.

[McK08b] Paul E. McKenney. RCU part 3: the RCU API. http://lwn.net/Articles/264090/, January 2008.

[McK15] Paul E. McKenney. Verification challenges. http://paulmck.livejournal.com/tag/verification%20challenge, April 2015.

[McM92] Kenneth L. McMillan. The SMV system. Technical Report MU-CS-92-131, Carnegie Mellon University, 1992.

[McM06] Kenneth L. McMillan. Lazy abstraction with interpolants. In Thomas Ball and Robert B. Jones, editors, *Proceedings of the 18th International Conference on Computer Aided Verification (CAV), Seattle, WA, USA, August 17-20, 2006*, volume 4144 of *Lecture Notes in Computer Science*, pages 123–136. Springer, 2006.

[Min17] Antoine Miné. Static analysis of embedded real-time concurrent software with dynamic priorities. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 331:3–39, 2017.

[MRC+14] Jeremy Morse, Mikhail Ramalho, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22 (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Proceedings of the 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Grenoble, France, April 5-13, 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 405–407. Springer, 2014.

[MS98] Paul E. McKenney and John D. Slingwine. Read-copy update: Using execution history to solve concurrency problems. In *Proceedings of the 10th International Conference on Parallel and Distributed Computing and Systems, Las Vegas, NV, USA, October 28-31, 1998*, pages 509–518, 1998.

[MTD16] Pasquale Malacaria, Michael Tautschnig, and Dino Distefano. Information leakage analysis of complex C code and its application to openssl. In Tiziana Margaria and Bernhard Steffen, editors, *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA), Imperial, Corfu, Greece, October 10-14, 2016*, volume 9952 of *Lecture Notes in Computer Science*, pages 909–925, 2016.

[MW08] Paul E. McKenney and Jonathan Walpole. Introducing technology into the Linux kernel: a case study. *ACM SIGOPS Operating Systems Review*, 42(5):4–17, 2008.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[NR10] Aditya V. Nori and Sriram K. Rajamani. An empirical study of optimizations in YOGI. In Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel, editors, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE), Cape Town, South Africa, 1-8 May, 2010*, pages 355–364. ACM, 2010.

[NRTT09] Aditya V. Nori, Sriram K. Rajamani, SaiDeep Tetali, and Aditya V. Thakur. The yogiproject: Software property checking via static analysis and testing. In Stefan Kowalewski and Anna Philippou, editors, *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), York, UK, March 22-29, 2009*, volume 5505 of *Lecture Notes in Computer Science*, pages 178–181. Springer, 2009.

[OMLB16] Abdelraouf Ouadjaout, Antoine Miné, Noureddine Lasla, and Nadjib Badache. Static analysis by abstract interpretation of functional properties of device drivers in TinyOS. *Software and Systems*, 120:114–132, 2016.

[Pel93] Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Proceedings of the 5th International Conference on Computer Aided Verification (CAV), Elounda, Greece, June 28 - July 1, 1993*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[Pel94] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In David L. Dill, editor, *Proceedings of the 6th International Conference on Computer Aided Verification (CAV), Stanford, California, USA, June 21-23, 1994*, volume 818 of *Lecture Notes in Computer Science*, pages 377–390. Springer, 1994.

[PM02] Jens Palsberg and Di Ma. A typed interrupt calculus. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Proceedings of the 7th International Symposium the Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT), Oldenburg, Germany, September 9-12, 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 291–310. Springer, 2002.

[PR13] Corneliu Popeea and Andrey Rybalchenko. Threader: A verifier for multi-threaded programs (competition contribution). In Nir Piterman

and Scott A. Smolka, editors, *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Rome, Italy, March 16-24, 2013*, volume 7795 of *Lecture Notes in Computer Science*, pages 633–636. Springer, 2013.

[PTS+11] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia L. Lawall, and Gilles Muller. Faults in Linux: Ten years later. In Rajiv Gupta and Todd C. Mowry, editors, *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Newport Beach, CA, USA, March 5-11, 2011*, pages 305–318. ACM, 2011.

[QR05] Shaz Qadeer and Jakob Rehof. Context-bounded model checking of concurrent software. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Edinburgh, UK, April 4-8, 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2005.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In Tony Montgomery, Lori A. Clarke, and Carlo Ghezzi, editors, *Proceedings of the 14th International Conference on Software Engineering (ICSE), Melbourne, Australia, May 11-15, 1992*, pages 105–118. ACM Press, 1992.

[RC07] John Regehr and Nathan Cooprider. Interrupt verification via thread verification. *Electronic Notes in Theoretical Computer Science*, 174(9):139–150, 2007.

[RD05] John Regehr and Usit Duongsaa. Preventing interrupt overload. In Yunheung Paek and Rajiv Gupta, editors, *Proceedings of the 6th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), Chicago, Illinois, USA, June 15-17, 2005*, pages 50–58. ACM, 2005.

[RE08] Willem-Paul de Roever and Kai Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison.* Cambridge University Press, 1st edition, 2008.

[Reg05] John Regehr. Random testing of interrupt-driven software. In Wayne Wolf, editor, *Proceedings of the 5th ACM International Conference On*

Embedded Software (EMSOFT), Jersey City, NJ, USA, September 18-22, 2005, pages 290–298. ACM, 2005.

[Reg07] John Regehr. Safe and structured use of interrupts in real-time and embedded software. In Sang H. Son, Insup Lee, and Joseph Y.-T. Leung, editors, *Handbook of Real-Time and Embedded Systems.* Chapman and Hall/CRC, 2007.

[Roy17] Lance Roy. rcutorture: Add CBMC-based formal verification for SRCU. https://www.spinics.net/lists/kernel/msg2421833.html, 2017.

[RRW05] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(4):751–778, 2005.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In Jeanne Ferrante and P. Mager, editors, *Proceedings of the 15th ACM Symposium on Principles of Programming Languages (POPL), San Diego, California, USA, January 10-13, 1988*, pages 12–27. ACM Press, 1988.

[Shi91] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie Mellon University, 1991.

[SK07] Kohei Suenaga and Naoki Kobayashi. Type-based analysis of deadlock for a concurrent calculus with interrupts. In Rocco De Nicola, editor, *Proceedings of the 16th European Symposium on Programming (ESOP), Braga, Portugal, March 24 - April 1, 2007*, volume 4421 of *Lecture Notes in Computer Science*, pages 490–504. Springer, 2007.

[SKW] Chungha Sung, Markus Kusano, and Chao Wang. Modular verification of interrupt-driven software. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana-Champaign, Illinois, October 30 - November 3, 2017.*

[SMM12] Pavel Shved, Mikhail U. Mandrykin, and Vadim S. Mutilin. Predicate analysis with BLAST 2.7 (competition contribution). In Cormac Flanagan and Barbara König, editors, *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), Tallinn, Estonia, March 24 - April 1, 2012*, volume

7214 of *Lecture Notes in Computer Science*, pages 525–527. Springer, 2012.

[SNBB09] Bastian Schlich, Thomas Noll, Jörg Brauer, and Lucas Brutschy. Reduction of interrupt handler executions for model checking embedded software. In Kedar S. Namjoshi, Andreas Zeller, and Avi Ziv, editors, *Proceedings of the 5th International Haifa Verification Conference on Hardware and Software: Verification and Testing (HVC), Haifa, Israel, October 19-22, 2009*, volume 6405 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2009.

[SR08] Sanjit A. Seshia and Alexander Rakhlin. Game-theoretic timing analysis. In Sani R. Nassif and Jaijeet S. Roychowdhury, editors, *Proceedings of the International Conference on Computer-Aided Design (ICCAD), San Jose, CA, USA, November 10-13, 2008*, pages 575–582. IEEE Computer Society, 2008.

[SR12] Sanjit A. Seshia and Alexander Rakhlin. Quantitative analysis of systems using game-theoretic learning. *ACM Transactions on Embedded Computing Systems (TECS)*, 11(S2):55, 2012.

[SS07] Y. N. Srikant and Priti Shankar, editors. *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*. CRC Press, 2007.

[SSVA14] Martin D. Schwarz, Helmut Seidl, Vesal Vojdani, and Kalmer Apinis. Precise analysis of value-dependent synchronization in priority scheduled programs. In Kenneth L. McMillan and Xavier Rival, editors, *Proceedings of the 15th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI), San Diego, CA, USA, January 19-21, 2014*, volume 8318 of *Lecture Notes in Computer Science*, pages 21–38. Springer, 2014.

[SW10] Nishant Sinha and Chao Wang. Staged concurrent program analysis. In Gruia-Catalin Roman and Kevin J. Sullivan, editors, *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Santa Fe, NM, USA, November 7-11, 2010*, pages 47–56. ACM, 2010.

[SW11] Nishant Sinha and Chao Wang. On interference abstractions. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-*

SIGACT Symposium on Principles of Programming Languages (POPL), Austin, TX, USA, January 26-28, 2011, pages 423–434. ACM, 2011.

[TDV15] Joseph Tassarotti, Derek Dreyer, and Viktor Vafeiadis. Verifying read-copy-update in a logic for weak memory. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Portland, OR, USA, June 15-17, 2015*, pages 110–120. ACM, 2015.

[VHHK15] Thilo Vörtler, Benny Höckner, Petra Hofstedt, and Thomas Klotz. Formal verification of software for the contiki operating system considering interrupts. In Zoran Stamenkovic, Witold A. Pleskacz, Jaan Raik, and Heinrich Theodor Vierhaus, editors, *Proceedings of the 18th International Symposium on Design and Diagnostics of Electronic Circuits & Systems DDECS, Belgrade, Serbia, April 22-24, 2015*, pages 295–298. IEEE, 2015.

[WBKW07] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. Model checking concurrent Linux device drivers. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Atlanta, Georgia, USA, November 5-9, 2007*, pages 501–504. ACM, 2007.

[WCGY09] Chao Wang, Swarat Chaudhuri, Aarti Gupta, and Yu Yang. Symbolic pruning of concurrent program executions. In Hans van Vliet and Valérie Issarny, editors, *Proceedings of the 17th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), Amsterdam, The Netherlands, August 24-28, 2009*, pages 23–32. ACM, 2009.

[WCM+16] Xueguang Wu, Liqian Chen, Antoine Miné, Wei Dong, and Ji Wang. Static analysis of runtime errors in interrupt-driven programs via sequentialization. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(4):70, 2016.

[WKO13] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with Impact. In Barbara Jobstmann and Sandip Ray, editors, *Proceedings of the Internation Conference on Formal Methods in Computer-Aided Design (FMCAD), Portland, OR, USA, October 20-23, 2013*, pages 210–217. IEEE, 2013.

[WWC+13] Xueguang Wu, Yanjun Wen, Liqian Chen, Wei Dong, and Ji Wang. Data race detection for interrupt-driven programs via bounded model checking. In *Proceedings of the 7th International Conference on Software Security and Reliability (SERE), Gaithersburg, Maryland, USA, 18-20 June 2013*, pages 204–210. IEEE, 2013.

[YLB+08] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter W. O'Hearn. Scalable shape analysis for systems code. In Aarti Gupta and Sharad Malik, editors, *Proceedings of the 20th International Conference on Computer Aided Verification (CAV), Princeton, NJ, USA, July 7-14, 2008*, volume 5123 of *Lecture Notes in Computer Science*, pages 385–398. Springer, 2008.