Lock Correctness

Daniel Poetzl

Department of Computer Science

University of Oxford

This dissertation is submitted for the degree of

Doctor of Philosophy

Linacre College May 2018

Acknowledgements

I would like to thank my supervisor Daniel Kroening. It was a great pleasure working with him and being part of his research group. I am thankful to my co-supervisor Peter Schrammel, for the enjoyable collaboration and the numerous helpful technical discussions.

I am grateful to Marta Kwiatkowska and Eran Yahav for being my thesis examiners. The discussions during the viva and their comments have improved the quality of this thesis.

Finally, I would like to thank all my collaborators, colleagues, friends, and family for their support and the fun during my studies.

Abstract

Locks are a frequently used synchronisation mechanism in shared memory concurrent programs. They are used to enforce atomicity of certain code portions, avoid undefined behaviour due to data races, and hide weak memory effects of the underlying hardware architectures (i.e., they provide the illusion of interleaved execution). To provide these guarantees, the correct interplay of a number of subsystems is required. We distinguish between the application level, the transformation level, and the hardware level.

On the application level, the programmer is required to correctly use the locks. This amounts to avoiding data races, deadlocks, and other errors in using the locking primitives, such as unlocking a lock that is not currently held.

On the transformation level, the compiler needs to correctly optimise the program and correctly map its operations to machine code. This requires knowing, for example, when it is safe to move a code statement in a thread past a lock operation such that the resulting thread is a refinement of the original thread.

On the hardware level, the lock operations themselves need to be implemented correctly, by usage of low-level primitives such as memory fences and read-modify-write operations. This requires knowing the relaxations of memory ordering that could occur on the target hardware, and the effect of the primitives that can be used to restore consistency (such as memory fences).

In this thesis, we address an aspect of each of the three levels of correctness mentioned above. On the application level, we provide a sound static approach for deadlock analysis of C/Pthreads programs. The approach is based on a context- and thread-sensitive abstract interpretation framework, and uses a lightweight dependency analysis to identify statements relevant to deadlock analysis. To quantify scalability, we have applied our approach to a large number of concurrent programs from the Debian GNU/Linux distribution.

On the transformation level, we provide a new theory of refinement between threads, which is phrased in terms of state transitions between lock operations. We show that the theory is more precise than existing approaches, and that its application in a compiler testing setting leads to large performance gains compared to a previous approach.

On the hardware level, we provide a toolchain to test the memory model of GPUs and the behaviour of code running on them. We automatically generate short concurrent code snippets that, when run on hardware, reveal interesting properties about the underlying memory model. These code snippets include idioms that typically appear in implementations of synchronisation operations. We further manually test several GPU locking primitives. Our testing has revealed surprising hardware behaviours and bugs in lock implementations.

Table of contents

1 Introduction **1**

1.1 Contributions 6

1.1.1 Thesis 6

1.1.2 Publications 7

2 Memory Model Testing **9**

2.1 Introduction 9

2.2 Introduction to GPUs 10

2.2.1 GPU Hardware Architecture 10

2.2.2 GPU Programming Model 11

2.2.3 PTX Assembly 13

2.3 Testing Framework 14

2.3.1 Test Generation 14

2.3.2 Running Tests 17

2.4 Checking for Optimisations 18

2.4.1 Example 20

2.4.2 Manufacturing Dependencies 22

2.5 Test Results 23

2.5.1 Automatically Generated Tests 23

2.5.2 Manual Tests 25

2.6 Related Work 30

3 Thread Refinement **33**

3.1 Introduction 33

3.2 Transition-Based and Event-Based Refinement 35

3.2.1 Example 36

3.2.2 Contexts that Write 39
3.2.3 Summary 39
3.3 Formalisation of Transition-Based Refinement 40
3.3.1 Program Model 40
3.3.2 Interleaving Semantics 45
3.3.3 Refinement 47
3.3.4 Transition Traces 48
3.3.5 Refinement Condition 50
3.3.6 Transitivity and Reflexivity 52
3.3.7 Soundness 55
3.4 Refinement Examples and Supported Optimisations 59
3.4.1 Roach Motel Reorderings 60
3.4.2 Inverse Roach Motel Reorderings 61
3.4.3 Counterexamples 62
3.5 Formalisation with Nested Locks 64
3.6 Compiler Testing 70
3.6.1 Implementation 70
3.6.2 Experiments 71
3.7 Related Work 73

4 Deadlock Analysis **77**
4.1 Introduction 77
4.1.1 Definition of Deadlocks 78
4.2 Analysis Overview 81
4.3 Analysis Framework 86
4.3.1 Program Representation 87
4.3.2 Analysis Framework – Overview 89
4.3.3 Analysis Framework – Details 91
4.3.4 Analysis Framework – Flow-Insensitivity 95
4.3.5 Pointer Analysis 97
4.3.6 Lockset Analysis 97
4.3.7 Lockset Correctness 98
4.4 Dependency Analysis 102
4.5 Non-Concurrency Analysis 109
4.6 Lock Graph Analysis 114

4.6.1 Lock Graph Construction 114
4.6.2 Checking Cycles in the Lock Graph 115
4.6.3 Lock Graph Correctness 116
4.7 Experiments 120
4.8 Related Work 123

5 Conclusions **127**

References **129**

Appendix A GPU Testing Tools **137**
A.1 Generating Tests 137
A.2 Testing 139

Appendix B Deadlock Analysis Tool **141**

Chapter 1

Introduction

Locks are a frequently used mechanism to synchronise code in concurrent programs that share memory. Locks are manipulated by the two main functions `lock()` and `unlock()`. By calling `lock(l)` a thread can attempt to acquire the lock `l`. If it is already held by another thread, the thread is blocked until the lock becomes available. A thread that is holding a lock can release it by calling `unlock(l)`. That way, locks can be used to ensure that code portions protected by the same lock are not executed concurrently (i.e., they are serialised). Thus, if potentially interfering code portions are protected by the same lock, they will appear to execute atomically.

The definition of a concurrent programming model essentially consists of two parts: a description of the effect of individual operations (such as assignment statements or lock operations), and a description of how the operations execute concurrently. The semantics of concurrent execution in shared memory systems is defined by a *memory model*. The simplest memory model is that of *Sequential Consistency* (SC) [66]. In this model, "the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program" [66]. In the SC model, a program thus behaves as if the operations of the individual threads were simply interleaved, and executions can therefore be modelled as interleavings. In the following, we thus use the terms SC execution and interleaving synonymously.

A downside of the SC model is that it disallows many program optimisations that would be valid in a sequential setting. This pertains both to compiler

optimisations and hardware optimisations. For example, a core in a multicore CPU would not be allowed to reorder two reads from different memory locations, as it could potentially result in an execution that has a result that differs from the result of any interleaving. These downsides have been remedied by the SC-for-DRF[1] memory model [9]. In this model, programs for which any SC execution (i.e., interleaving) has a data race have undefined semantics, and programs free from data races behave according to SC.

A *data race* in an interleaving is a pair of memory accesses (either reads or writes), at least one of them being a write, which are adjacent in the interleaving, and which access the same memory location (we later define data races formally in Section 3.3.2). In the SC-for-DRF memory model, the programmer only has to reason about interleavings: data races are defined in terms of interleavings, and if a program does not have data races then it behaves according to SC as well.

Data races can be avoided by the use of synchronisation primitives such as locks. When reasoning about data races, the synchronisation operations are treated as primitive operations. That is, they are treated according to their abstract semantics (as described informally for locks in the first paragraph above), irrespective of the concrete implementation on a given platform.

Since the programmer is required to write data-race-free code, the compiler can assume that a program given to it does not have data races. This allows for sequential reasoning between synchronisation operations [25, 40, 58]. For example, the compiler is free to reorder two instructions in a thread that are not separated by a synchronisation operation, as any other thread that could observe this reordering would necessarily have a data race with that thread. Virtually all programming languages with a shared memory concurrency model implement the SC-for-DRF memory model for programs using locks (such as C/Pthreads [101], C11 [54], or C++11 [46]).

In this thesis, we are concerned with the correctness of programs using locks. This pertains both to the correct use of locks by the programmer, as well as the correctness of the subsystems that are required in order to present the programmer with the SC-for-DRF model. We distinguish between three levels: the *application level*, the *transformation level*, and the *hardware level*. On the application level, the programmer is required to correctly use the locks. On the transformation

[1] The abbreviation stands for Sequential Consistency for Data-Race-Free programs.

```
1 void thread1() {
2   pthread_mutex_lock(&m1)
3   pthread_mutex_lock(&m2)
4   x = 1;
5   pthread_mutex_unlock(&m2)
6   pthread_mutex_unlock(&m1)
7 }
```

```
1 void thread2() {
2   pthread_mutex_lock(&m2)
3   pthread_mutex_lock(&m1)
4   x = 2;
5   pthread_mutex_unlock(&m1)
6   pthread_mutex_unlock(&m2)
7 }
```

Fig. 1.1: A program containing a deadlock

level, the compiler needs to correctly optimise the program and correctly map its operations to machine code. On the hardware level, the lock operations themselves need to be implemented correctly. We present effective and practical methods that address a correctness issue on each of those levels.

Application level

On the application level, we are concerned with the correct use of locks. In particular, we consider the issue of deadlocks. We address the following problem:

> **(P1)** Let P be a program with defined behaviour according to the SC-for-DRF model. Is P free from deadlocks?

The problem is illustrated in Figure 1.1. The figure gives two threads which each attempt to acquire locks `m1` and `m2` via calls to `pthread_mutex_lock()`. However, the threads attempt to acquire the locks in opposite order, which can lead to both threads being blocked at the lock operation in line 3. To address this problem, in Chapter 4 we present a static deadlock analysis approach for C/Pthreads that is sound and can handle real-world code.

Transformation level

On the transformation level, we are concerned with the correct compilation and optimisation of a given concurrent program. We address the problem of refinement between threads:

> **(P2)** Let T_1 and T_2 be two threads. Is T_1 a refinement of T_2 in the SC-for-DRF model?

```
1 void thread_orig() {
2   pthread_mutex_lock(&m)
3   x = 1
4   pthread_mutex_unlock(&m)
5   y = 1
6 }
```

```
1 void thread_trans() {
2   pthread_mutex_lock(&m)
3   x = 1
4   y = 1
5   pthread_mutex_unlock(&m)
6 }
```

Fig. 1.2: Original thread and transformed thread

Informally, a thread T_1 is a refinement of another thread T_2 if, in any program containing T_2, when T_2 is replaced by T_1 the resulting program does not have possible new behaviours. The problem is illustrated in Figure 1.2. The figure gives an original thread and a transformed thread. In the transformed thread, the write to variable y has been moved into the lock-protected section. This is a sound transformation, as it cannot introduce data races or change the behaviour of data-race-free programs. The transformed thread is thus a refinement of the original thread.

To reason about refinement between threads, we have developed a new refinement theory that is based on matching state transitions between lock operations. The theory is more precise than existing approaches and can be efficiently implemented. We have put our theory to use in a compiler testing application, addressing the following problem:

> **(P3)** Does a given compiler correctly optimise programs such that given a thread T, the optimised thread T' is a refinement of T in the SC-for-DRF model?

Our compiler testing approach based on our refinement theory significantly outperforms a previous method. We describe our refinement theory and compiler testing method in Chapter 3.

Hardware level

On the hardware level, we are concerned with the correct implementation of lock operations such that a program using those locks behaves according to the SC-for-DRF model. An issue is the fact that hardware memory models are often poorly documented [14]. We thus resort to testing the hardware to reveal potential behaviours. The problem can be phrased as follows:

```
1 lock(l):
2   L:
3   cas r, [l], 0, 1
4   jnz r, L
```

```
1 unlock(l):
2   st [l], 0
```

Fig. 1.3: Implementations of `lock()` and `unlock()`

> **(P4)** Given a language (e.g., a hardware assembly language) with an unknown (or underspecified) memory model, and given implementations of the `lock()` and `unlock()` operations in that language, does every program using those operations only exhibit behaviours possible according to the SC-for-DRF model?

Figure 1.3 gives a `lock()` and an `unlock()` operation in an assembly-like language. This is a typical spinlock implementation where the `lock()` operation spins in a loop until the lock is available (represented by the value 0). The compare-and-swap (CAS) operation atomically checks whether the lock `l` has value 0 and if yes sets it to 1. It returns the value read in register `r`. The loop is implemented via a `jnz` (jump if not zero) instruction conditionally jumping to label `L` when the lock could not be acquired.

We are interested in whether any program in the language above that does not have data races, while treating locks as primitive operations according to their abstract semantics, behaves according to SC when the given concrete lock implementations are used. If the lock operations are not correct we want to fix them. This can require the addition of operations like memory fences, e.g., before the store operation in line 2 of `unlock()`.

We address the problem for Nvidia GPUs. We extend the memory model testing approach previously presented by Alglave et al. [14] for CPUs. We automatically generate short concurrent code snippets that embody various concurrency idioms (similar to, e.g., the lock implementations shown in Figure 1.3). The outcome of executing these snippets on hardware reveals properties of the underlying hardware memory model. We discuss several interesting behaviours and bugs that our testing has revealed, and discuss the implications for the implementation of locks on Nvidia GPUs.

1.1 Contributions

We address the problems mentioned above in this thesis. We provide theories, algorithms, and tools to aid in answering the given questions. The overall goal is to improve the correctness of programs using locks, from the application level down to the hardware level.

1.1.1 Thesis

We summarise below the main contributions of this thesis:

1. Memory model testing (Chapter 2)

    - An extension of the diy [14] approach (originally developed for CPUs) to generate tests for GPUs that when executed on hardware reveal properties of the underlying memory model.
    - A method to enable the use of an optimising assembler to compile the tests. We embed a test specification into the test code which is checked against the generated binary code after compilation.
    - We use the testing framework to run both automatically generated and handwritten tests on several Nvidia GPUs. Our testing has revealed several interesting hardware behaviours and bugs in lock implementations.

2. Thread refinement (Chapter 3)

    - A new theory of thread refinement based on matching state transitions between lock operations. The theory is more precise than existing approaches and provides the basis for efficient implementations.
    - An application of our theory in a compiler testing setting. We show that the approach of matching state transitions leads to significant performance gains compared to a previous approach.

3. Deadlock analysis (Chapter 4)

    - The first static deadlock analysis approach for C/Pthreads that is sound (for defined programs) and can handle real-world code. It consists of a

pipeline of several static analyses, implemented on top of a new context- and thread-sensitive abstract interpretation framework.

- A lightweight dependency analysis for identifying statements that could affect a given set of expressions. We use it to speed up the pointer analysis by focusing it to statements relevant to deadlock analysis.
- We show how to build a lock graph that soundly captures various sources of imprecision, such as may-point-to information or thread creation in loops/recursions, and how to combine the cycle detection with a non-concurrency check to prune infeasible cycles in the lock graph.
- An evaluation of our approach on a large number of programs from the Debian/GNU Linux distribution.

Chapter 2 is based on parts of our ASPLOS paper [13], Chapter 3 is based on our TACAS paper [89], and Chapter 4 is based on our ASE paper [65].

1.1.2 Publications

We summarise below the contributions to the joint publications on which this thesis is based on. The items include both the theoretical development and implementation.

1. J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, 2015

   - Extension of the diy test generation approach to GPUs
   - Testing the memory model of Nvidia GPUs using automatically generated tests (with Tyler Sorensen)
   - Toolchain to analyse and compare testing logs of different GPUs
   - Approach to check GPU assembly code to verify that the compiler did not perform certain optimisations
   - Investigation of the correctness of two GPU spinlock implementations (given by He and Yu [51])

2. D. Poetzl and D. Kroening. Formalizing and checking thread refinement for data-race-free execution models. In *TACAS*, 2016

   - Theory of thread refinement in the SC-for-DRF memory model based on matching state transitions
   - Application and evaluation of the theory in a compiler testing application

3. D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound static deadlock analysis for C/Pthreads. In *ASE*, 2016

   - Context- and thread-sensitive abstract interpretation framework (with Peter Schrammel)
   - May lockset analysis and must lockset analysis
   - Dependency analysis to identify statements that could affect a given set of expressions
   - Non-concurrency analysis to identify program statements that cannot run concurrently
   - Experimental evaluation on programs from the Debian/GNU Linux distribution (with Peter Schrammel)

Chapter 2

Memory Model Testing

2.1 Introduction

An aspect of the correct implementation of a given concurrency model (such as SC-for-DRF with locks) is the correct implementation of the synchronisation primitives it provides. To that end, it is important to know which relaxations of memory consistency might occur on hardware (i.e., from which writes a given read could possibly read [8]). Moreover, one needs to know the effect of the various low-level primitives, such as memory fences or read-modify-write operations, that can be used to prevent certain relaxations.

Unfortunately, hardware vendor documentation is often not sufficient to answer these questions. The documentation with regard to memory consistency is often incomplete, ambiguous, or wrong [14]. In previous work, testing has been applied successfully to reveal possible hardware behaviours of CPUs (see, e.g.,[14, 15, 17, 35, 92, 93]. In contrast, the issue of testing the memory model of GPUs has received little attention. We thus investigate in the following the memory model of Nvidia GPUs.

We extend the CPU memory model testing approach of Alglave et al. [14] to GPUs. We automatically generate short concurrent code snippets (termed *litmus tests*) that embody different concurrency idioms. The outcome of executing the litmus tests on hardware reveals interesting properties of the underlying hardware memory model. We show how to employ the high-level assembly language PTX, which might be subject to optimisations, to test the hardware behaviour. This requires checking that the produced binary code conforms to the high-level PTX

code. We further discuss several interesting hardware behaviours and bugs in lock implementations that our testing has revealed.

2.2 Introduction to GPUs

GPU hardware architectures and programming models have a set of features that set them apart from their multicore CPU counterparts. We present the Nvidia Maxwell hardware architecture (introduced in 2014) and the CUDA programming platform as an example, and use Nvidia terminology. Architectures and programming platforms from other vendors share similar characteristics.

2.2.1 GPU Hardware Architecture

A Maxwell GPU consists of a set of *streaming multiprocessors* (SMs) (e.g., 5 on the GeForce GTX 750 Ti). Figure 2.1 shows a block diagram of a Maxwell GPU, depicting one SM, and the L2 cache (on-chip) and DRAM (off-chip) shared by all SMs. Each SM is divided into four separate processing blocks. A processing block contains (among other things left off in the diagram) a warp scheduler and a set of processing elements (PEs). There are three different kinds of processing elements in a processing block: 32 CUDA cores (for, e.g., integer and floating-point operations), 8 load-store units (for loading/storing data from/to memory), and 8 special function units (for computing functions such as sine and square root). The processing elements in all four processing blocks have access to a common, explicitely-addressable shared memory (64 KiB). As the shared memory is on-chip and part of the SM, it is much faster than the DRAM. Hence, it can be used for fast intra-SM communication. Two processing blocks share a common L1 cache.

Scheduling within an SM is done in groups of 32 threads called *warps*. The threads in a warp share the same instruction stream (they can be thought of as executing in lockstep). The warp scheduling is performed by the four warp schedulers that are part of an SM.

As context switches are performed in hardware by the warp schedulers, they are much faster than on CPUs (where they are handled by the operating system kernel). As a result, GPUs can (and do) switch the context when, e.g., a thread issues an expensive DRAM access. Thus, if there is enough parallel computational work available, the memory latency can be hidden in that way. Consequently,

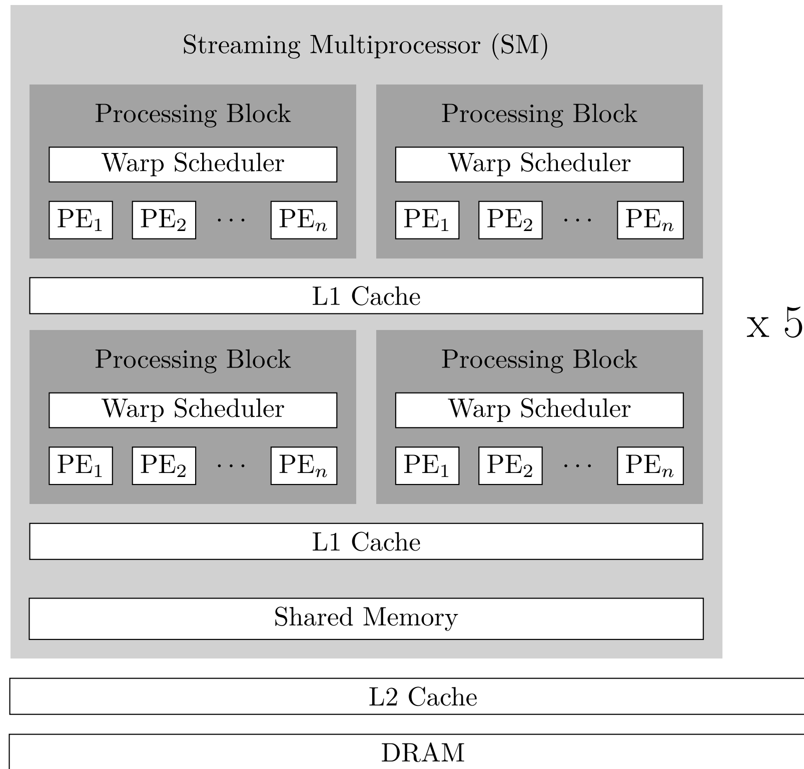


Fig. 2.1: Block diagram of the GeForce GTX 750 Ti (Maxwell architecture). A more detailed block diagram is given in the corresponding whitepaper [87].

GPUs typically have smaller caches than multicore CPUs. Another aspect is that the individual cores are simpler than on multicore CPUs. For instance, while CPUs have sophisticated logic to leverage instruction-level parallelism within a single thread, GPUs rely on the programmer to exploit the parallelism inherent in a problem by manually dividing the work among threads.

2.2.2 GPU Programming Model

The programming models of GPUs derive from their hardware architectures. CUDA (Compute Unified Device Architecture) is Nvidia's platform for general-purpose GPU (GPGPU) computing. Several languages can be used for writing CUDA applications, among them CUDA C (a C-like language), and PTX assembly (described in Section 2.2.3).

In CUDA, a workload for the GPU is specified by a *kernel*, which is essentially a function to be executed on the GPU. Several kernels can be executed in parallel

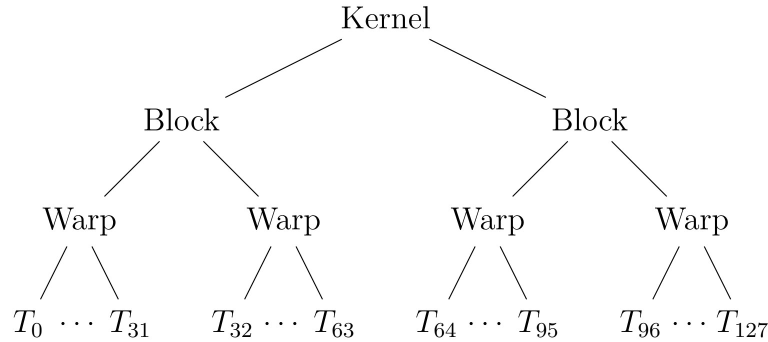


Fig. 2.2: Thread hierarchy in a program utilising 128 threads

(in different *streams*). A kernel consists of several *blocks* (also called *cooperative thread arrays* (CTAs) when programming in PTX). Threads in the same block are executed on the same SM, and can communicate via the shared memory that is part of the SMs. Blocks are in turn split into *warps* of 32 threads (which are scheduled by the warp schedulers that are part of the SMs). Thus, in CUDA the threads are hierarchically organized. The division of a program into kernels and blocks has to be explicitely specified by the programmer. To write correct programs, it is thus important to know when, e.g., a store becomes visible to another thread in the same block, or when it becomes visible to a thread in a different block. Figure 2.2 shows an example of how the threads in a program consisting of a single kernel may be organised hierarchically. In general, the farther apart two threads are in the tree, the weaker the memory consistency between them.

In addition to the hierarchical organization of threads into kernels, blocks and warps as described above, another distinguishing feature of GPU programming models is the division of memory into several segments (*state spaces* in CUDA terminology). All of these segments aside from the shared memory reside in DRAM. The state spaces available are `global`, `const`, `local`, `param`, and `shared`. Even though the first four state spaces all reside in DRAM, different caching policies apply to them, and hence memory accesses can behave differently based on which state space they target.

2.2.3 PTX Assembly

On Nvidia GPUs, the lowest supported and documented level to program on is the level of PTX assembly. PTX stands for *parallel thread execution*, and is the name for both a conceptual virtual machine and its corresponding instruction set architecture (ISA). PTX programs are translated to the (undocumented) target hardware instruction set (referred to as SASS by Nvidia) by the PTX optimizing assembler (ptxas). In contrast, there is no publicly available assembler for SASS. PTX differs from lower-level assembly languages in that instructions and instruction operands are typed and variables and registers have to be declared before use. There is an unbounded number of virtual registers which are mapped to actual hardware registers by ptxas. Most instructions have an optional predicate guard that controls conditional execution.

The actual machine-level assembly produced for a program written in PTX assembly can be viewed via the tool cuobjdump [84]. The machine-level assembly differs between architectures. The cuobjdump manual provides a list of the different machine-level instructions available, but does not describe their semantics. However, the PTX code and the associated machine-level code for the different classes of memory loads and stores are reasonably close. Hence, assuming that the PTX assembler does not perform optimisations such as reordering of memory accesses, the memory behaviour of PTX programs allows to make judgements about the memory behaviour of the machine-level instructions.

Figure 2.3a shows a PTX code example. The first 4 lines are register and variable declarations, with the variable `x` being in the `global` state space, and `y` being in the `shared` state space. The `.s32` and `.u32` specifications stand for signed 32-bit and unsigned 32-bit integer types. The last 4 lines contain actual code. `st` is a store, `ld` a load, and `membar.sys` a full memory fence. The `.ca` and `.wb` specifiers are cache operators (there are 5 cache operators for loads, and 4 cache operators for stores in PTX). They allow to specify, e.g., whether a load first consults the L1 cache, or bypasses it and considers L2 first. Figure 2.3b shows the corresponding machine-level assembly produced by ptxas for the GeForce GTX 680. As can be seen, there is a close correspondence between the PTX assembly and the machine-level assembly.

```
.reg .s32 r1;
.reg .s32 r2;
.global .s32 x;
.shared .u32 y;
...
mov.s32 r1, 1;
st.s32.wb [x], r1;
membar.gl;
ld.s32.ca r2, [y];
```

(a) PTX code snippet

```
MOV32I R0, 0x1;
MOV32I R4, 0x0;
MOV32I R5, 0x0;
MOV R4, R4;
MOV R5, R5;
ST.E [R4], R0;
MEMBAR.GL;
MOV R0, RZ;
LDS R0, [R0];
```

(b) SASS code snippet

Fig. 2.3: PTX code and corresponding machine-level assembly

2.3 Testing Framework

We have developed a framework to test the memory behaviour of Nvidia GPUs. It consists of the tools gpu-diy, gpu-litmus[1], optcheck, and logcheck. An overview of the framework is given in Figure 2.4. The tool gpu-diy generates PTX tests in the `.litmus` format (explained in Section 2.3.1). These tests can then be executed by gpu-litmus on the hardware. Additionally, one can also write tests by hand and then execute them with gpu-litmus. The optcheck tool is responsible for checking that in the compiled test the compiler and/or assembler did not reorder memory access instructions, such as to guarantee that the observed behaviour indeed reveals the hardware behaviour (see Section 2.4). The obtained testing logs can finally be analysed with the logcheck tool.

2.3.1 Test Generation

The gpu-diy tool generates PTX litmus tests, i.e., short concurrent PTX code snippets for which the result of executing them on hardware reveals interesting properties about the underlying memory model. It is based on the diy tool originally developed by Alglave et al. [14] to generate tests for CPUs.

We generate tests with instructions `ld`, `st` (load/store from/to memory), `atom.cas` (atomic compare-and-swap), `membar.cta`, `membar.gl` (memory fences), and various non-memory instructions acting only on registers such as `xor` or `and`. The memory fences `membar.cta` and `membar.gl` provide ordering relative to

[1]developed by Tyler Sorensen

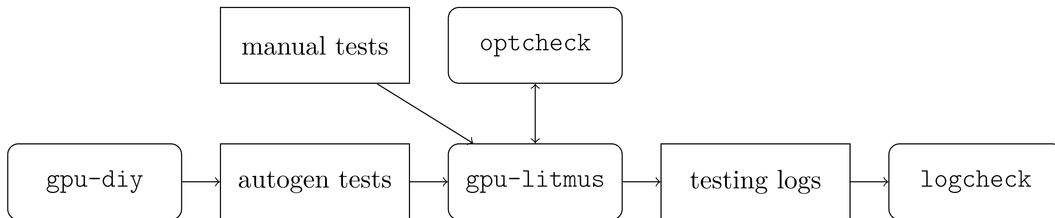


Fig. 2.4: Testing framework

instructions in other threads. When two instructions in one thread are separated by a memory fence, they will appear to execute in that order to instructions in another thread, given that those instructions in the other thread are separated by a memory fence as well. The `membar.cta` fence provides ordering relative to threads in the same CTA, whereas `membar.gl` provides ordering relative to all threads on the same GPU. Consequently, `membar.gl` is more costly than `membar.cta`.

Figure 2.5 shows an example of a test generated by gpu-diy. The test consists of six segments. Compared to CPU litmus tests, the second, fourth, and fifth segments are new in GPU litmus tests. Line 1 states the language (here `GPU_PTX`) and test name (here `LB` for “load buffering”). Lines 3–10 declare and initialise registers. Those prefixed with `0:` denote registers of thread 0, and those prefixed with `1:` denote registers of thread 1. A symbolic initialiser (i.e., `x` or `y` in the given test) denotes that the register is initialised to hold the *address* of the corresponding variable. Registers that are declared without initialisers are initialised to 0. Lines 12–15 give the test program with each column describing the code to be executed by a thread. Each thread starts with an identifier (e.g., `P0`), followed by a sequence of PTX instructions. Line 17 specifies the location of the threads in the thread hierarchy (cf. Figure 2.2) by means of what we call a *scope tree*. The scope tree is an S-expression (a notation borrowed from the Lisp programming language), which is a way to specify a tree as a string. The given scope tree specifies that `P0` and `P1` are in different CTAs but in the same kernel. Line 19 specifies the state spaces in which variables `x` and `y` reside. It specifies that both variables are in global memory. Line 21 gives an assertion about the final state of registers or memory. The given expressions asserts the registers `r0` in thread 0 and `r0` in thread 1 both have value 1 at the end of the execution.

By inspecting the test in Figure 2.5, we see that the final condition is not satisfied in any of the interleavings of the two threads. However, on several GPUs

```
1  GPU_PTX LB
2
3  0:.reg .s32 r0;
4  0:.reg .s32 r2;
5  0:.reg .b64 r1 = x;
6  0:.reg .b64 r3 = y;
7  1:.reg .s32 r0;
8  1:.reg .s32 r2;
9  1:.reg .b64 r1 = y;
10 1:.reg .b64 r3 = x;
11
12 P0                 : P1
13 ld.cg.s32 r0,[r1] : ld.cg.s32 r0,[r1]
14 mov.s32   r2,1    : mov.s32   r2,1
15 st.cg.s32 [r3],r2 : st.cg.s32 [r3],r2
16
17 (device (kernel (cta (warp P0))(cta (warp P1))))
18
19 x: global, y: global
20
21 0:r0=1 /\ 1:r0=1
```

Fig. 2.5: GPU PTX litmus test

(such as the GTX Titan), when we ran the test with gpu-litmus (as described in Section 2.3.2), we could observe the final outcome with both registers holding value 1. This suggests that the order between read operations and later write operations to different memory locations is relaxed on these machines. The way to prevent the weak behaviour is by putting a memory fence (`membar.gl`) between the read and the write on both threads. After we applied this transformation, we could not observe the weak behaviour anymore.

The gpu-diy tool generates the PTX code snippets and the assertions on the final state that make up the litmus tests (cf. Figure 2.5) according to the algorithm given by Alglave et al. [14]. The algorithm enumerates cycles formed from edges that represent potential relaxations of memory consistency. It then generates a code snippet and an assertion on the final state from each cycle. Then, the gpu-diy tool adds the register declarations (segment 2), enumerates the different scope trees (segment 4) that are possible with the given number of threads, and enumerates the different memory maps (segment 5) that are possible with the given shared variables. It then outputs a test for each possible combination.

Extending the original diy tool to gpu-diy to support the generation of GPU litmus tests required us to add support for generating PTX code, add new edge types representing relaxations that may occur on GPUs, add the enumeration of scope trees and memory maps, and add the generation of the register initialisations.

We generate tests where the shared variables are either in the `global` or the `shared` state space. The other state spaces (`local`, `param`, and `const`) are not suitable for inter-thread communication. Moreover, we restrict our testing to cases where all instructions use the cache operator `.cg` (which stands for cache globally). According to the PTX manual [83], this has the effect that the instruction bypasses the L1 cache and only uses the L2 cache. The manual further states that cache operators are performance hints only and do not change the memory consistency behaviour of programs.

2.3.2 Running Tests

The gpu-litmus tool is an extension of the litmus tool for CPUs by Alglave et al. [15]. It takes as input a litmus test specification as shown in Figure 2.5 (either generated by gpu-diy or written by hand) and generates a CUDA harness from it. The harness contains the PTX code snippet as inline assembly, wrapped in a loop

to repeatedly execute it a configurable number of times (100k in our experiments). The harness further contains initialisation code (to initialise variables and place threads according to the concurrency hierarchy) and code to stress the memory system (which we term *incantations*).

The incantations are an important component of gpu-litmus and crucial to observe weak behaviours. For example, one strategy employed by gpu-litmus is to create additional threads (in addition to those present in the litmus test) which continuously write to random memory locations (but different from those in the litmus test). The hypothesis is that the memory system may be more likely to service requests out of order when it is under heavy stress. The list of strategies and an evaluation of their effectiveness is given in our paper [13].

Once gpu-litmus has created the CUDA harness, it compiles it to binary code. It then passes the binary to optcheck, which checks that the instructions part of the litmus test appear in the same order in the binary as in the original test specification (cf. Figure 2.5). Finally, gpu-litmus executes the program on the GPU. It records in a logfile, among other statistics, how often the specified final state was observed (last line in Figure 2.5).

The tool logcheck aids in analysing the produced logs. It takes as input logfiles which contain the results of executing many different litmus tests. Then, for example, it allows to compare the logfiles produced when running the tests on different GPUs. That way, we can answer questions like whether a certain GPU has a memory model that is at least as strong as that of another GPU (this is the case if all weak behaviours observed on one could also be observed on the other). Moreover, logcheck can produce the various result tables given on our website [1].

2.4 Checking for Optimisations

We now discuss how we guard against unwanted optimisations when compiling the litmus tests. Recall from Section 2.2.3 that we write our tests in PTX. We compile the tests to SASS with the ptxas assembler, which optimises the code for efficiency.

If we invoke the assembler with minimal optimisations (-O0), we find that although each PTX load or store has a corresponding SASS load or store, instructions that were adjacent in the PTX code are separated by several instructions in

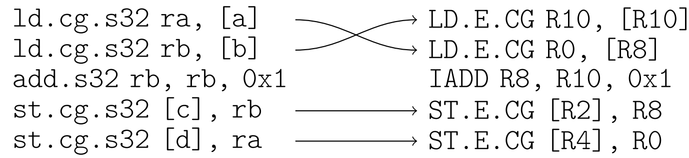


Fig. 2.6: PTX code (left) and optimised SASS code (right)

the SASS code. This is undesirable for testing the memory model: it can make the difference between observing weak behaviours or not.

If we invoke the assembler with maximal optimisations (`-O3`), most intermediate instructions are optimised away. However, we found that on rare occasions some instructions were reordered. This is again harmful for testing, as we could wrongly attribute weak behaviours to the hardware, when in fact they were introduced by the compiler. Figure 2.6 gives a PTX code snippet in which memory accesses were reordered by ptxas. The two loads were reorderd in an attempt to decrease the wait time for the result of the load of `b` by the subsequent `add`.

From looking at the SASS code for a litmus test (including the code added by gpu-litmus to stress the memory system), it is difficult to determine which SASS instruction corresponds to which PTX instruction. The registers in PTX are virtual and may be mapped to arbitrary registers on the hardware. Moreover, the code added to a test to stress the memory system increases the size of the program, resulting in a SASS binary with thousands of instructions. Thus, by inspecting the SASS code as is we cannot readily determine whether any memory access instructions have been reordered.

We thus add additional instructions (we call them specification instructions) to the PTX code that are linked to the instructions in the litmus test (i.e., the instructions given in the `.litmus` file, cf. Figure 2.5). We link the specification instructions and the litmus test instructions by making them use the same register. Additionally, the specification instructions include an easily recognisable immediate integer operand (starting with the prefix `0x07f3a`). When the test is then compiled to SASS, it contains both the litmus test instructions and the specification instructions. We can then find the litmus test instructions by looking for the instructions with an immediate value starting with prefix `0x07f3a`, and then looking for the instructions that use the same register. Furthermore, in the

immediate integer operand of the specification instructions we encode the sequence number and type of the corresponding litmus test instructions.

The specification code we add to a litmus test consists of a sequence of add instructions, placed at the end of each thread. A single specification instruction has the following shape:

```
linking register ─┐              ┌── instruction type
                  │             ─┴─
    add.s32 r2, rb, 0x07f3a001
                    ─────────
constant ─────────────┘      └── position
```

Each add instruction corresponds to exactly one memory access instruction. The integer literal of an add instruction (last operand) specifies two properties of the corresponding access: what type of instruction it is (e.g., 00 for a load with cache operator .cg), and its position in the order of memory access instructions within that thread. The constant serves to distinguish these specification instructions from any add instructions that appear in the code. In the litmus tests we generate, the accesses within a thread use different registers, so we can always create a one-to-one correspondence between memory access instructions and add instructions.

We implemented the approach described above in the optcheck tool. It takes a binary, obtains the SASS code via cuobjdump, and then checks that the litmus test code conforms to the embedded specification.

2.4.1 Example

We next give an example of our approach for detecting optimisations. Consider the code snippet in Figure 2.7a. The code portion before the empty line is the one for which we want to check if it is optimised. The three add instructions are added by gpu-litmus for that purpose. They have increasing immediate values in the order field. The add instruction with the lowest value 0 uses the same register as the first memory access instruction, the add instruction with value 1 uses the same register as the second memory access, and the add instruction with value 2 uses the same register as the third memory access instruction.

Figure 2.7b shows the cuobjdump output for the corresponding binary program. The additional information provided by the add instructions (IADD32I) allows us to check for optimisations as follows. We conclude that no optimisations have been performed if, in a small window of instructions around the add instructions,

```
mov.s32 r1, 1
st.s32.wb [r2], r1
ld.s32.ca r4, [r3]
mov.s32 r5, 1
st.s32.wb [r6], r5

add.s32 r7, r1, 0x07f3a_00_0
add.s32 r8, r4, 0x07f3a_01_1
add.s32 r9, r5, 0x07f3a_00_2
```

(a) PTX code

```
...
ST.E [R4], R16
MOV R4, R14
MOV R5, R15
LD.E R0, [R4]
MOV32I R4, 0x1
MOV R12, R12
MOV R13, R13
ST.E [R12], R4
IADD32I R5, R16, 0x07f3a000
IADD32I R12, R0, 0x07f3a011
IADD32I R4, R4, 0x07f3a002
...
```

(b) SASS code

Fig. 2.7: PTX code snippet including a specification consisting of three `add` instructions, and the corresponding SASS code as output by cuobjdump.

- there is an instruction using the same register as the add with value 0 in the order field, and the instruction is a store with cache operator `.wb` (specified by value 00 in the type field), followed by
- an instruction using the same register as the add with value 1 in the order field, and the instruction is a load with cache operator `.ca` (specified by value 01 in the type field), followed by
- an instruction using the same register as the add with value 2 in the order field, and the instruction is a store with cache operator `.wb` (specified by value 00 in the type field).

Thus, for the example in Figure 2.7, we conclude that no unwanted optimisations have been performed.

While the approach described above works well, it could (theoretically) yield false positives and false negatives. For example, as there is an infinite supply of virtual registers in PTX assembly, it could become necessary in the machine-level code to spill a register to memory, and then the spilled value could later be read back to a different register. This way, the correspondence between a memory access instruction and an `add` instruction could be broken.

2.4.2 Manufacturing Dependencies

We also want to test whether dependencies between memory accesses have an effect on memory consistency. For example, if there is a data dependency from a load to a later store this could prevent the reordering of the two instructions, as the result of the load is required to perform the store. For CPUs, litmus tests that check such properties use false dependencies [14]: ones that have no effect on the computed values. For example, in the PTX code snippet in Figure 2.8a, there is an address dependency between the load in line 1 and the load in line 5, since the result of the first load is used to compute the address of the memory location accessed by the second load. The dependency is a false dependency as the result of the `xor` is always 0, so the subsequent `add` never changes the value of the address register `r4`.

```
1 ld.s32 r1, [r0]
2 xor.b32 r2, r1, r1
3 cvt.u64.u32 r3, r2
4 add.u64 r4, r4, r3
5 ld.s32 r5, [r4]
```

(a) Optimised by ptxas (-O3)

```
1 ld.s32 r1, [r0]
2 and.b32 r2, r1, 0x80000000
3 cvt.u64.u32 r3, r2
4 add.u64 r4, r4, r3
5 ld.s32 r5, [r4]
```

(b) Not optimised by ptxas (-O3)

Fig. 2.8: Load-load address dependencies

Since we compile our litmus tests with the highest optimisation settings, the PTX assembler would recognise that the result of the `xor` is always 0, and hence remove lines 2–4, thereby removing the dependency. Therefore, we use a different scheme for testing dependencies, exemplified in Figure 2.8b. It is based on `and`-ing with a constant that has just the high bit set. The result of this operation will always be 0, since in our litmus tests all memory locations are initialised to 0 and the store instructions only write small positive values (with the high bit being 0). However, determining that the result is 0 would require an inter-thread analysis (which the PTX assembler does not perform). Thus, the dependencies are left intact.

chip	arch	year	SDK	driver	options
GTX 540m	Fermi	2011	5.5	331.20	sm_21
Tesla C2075	Fermi	2011	5.5	334.16	sm_20
GTX 660	Kepler	2012	5.0	331.67	sm_30
GTX Titan	Kepler	2013	6.0	331.62	sm_35
GTX 750	Maxwell	2014	6.0	331.62	sm_50

Fig. 2.9: The GPUs we tested

init: `x = 0` **final:** `r0 = 1` $\wedge$ `r1 = 0` **threads:** intra-CTA

```
0.1  st.cg [x], 1          1.1  ld.cg r0,[x]
                           1.2  ld.cg r1,[x]
```

obs/100k	GTX 540m	Tesla C2075	GTX 660	GTX Titan	GTX 750
	11642	8879	9599	9787	0

Fig. 2.10: Coherent reads test

2.5 Test Results

In this section, we discuss some of the hardware behaviours that our testing has revealed. The full set of results is given online [1]. Figure 2.9 lists the GPUs we have tested. Each entry also lists the hardware architecture the GPU is based on, the year is was released, and the CUDA SDK version, driver version, and compilation options used for the tests.

2.5.1 Automatically Generated Tests

In this section, we discuss some interesting hardware behaviours we observed by running tests automatically generated by gpu-diy on the GPUs in Figure 2.9.

Figure 2.10 shows the coherent reads test. The figure gives the litmus test together with the test results. The bottom of the figure indicates how often we observed the final state `r0 = 1` $\wedge$ `r1 = 0` for each of the chips when the test was run 100 000 times. The top of the figure gives the initial state of the shared variable `x`, the final state for which we want to check whether it is observable, and the placement of the threads in the concurrency hierarchy (an abbreviated description of the scope tree, cf. Section 2.3.1). The given specification "intra-CTA" indicates that both threads are in the same CTA.

init: `x = 0 ∧ y = 0`	**final:** `r0 = 1 ∧ r1 = 0`	**threads:** inter-CTA

	Thread 0		Thread 1
0.1	`st.cg [x], 1`	1.1	`atom.cas r0,[y],1,2`
0.2	`membar.gl`	1.2	`ld.cg r1,[x]`
0.3	`st.cg [y], 1`		

obs/100k	GTX 540m	Tesla C2075	GTX 660	GTX Titan	GTX 750
	0	0	0	11	0

Fig. 2.11: Message passing test with CAS

The test checks whether it is possible for the first load to observe the new value 1, and for the second load to still observe the old value 0. If this were the case, the principle of single-location coherency would be violated: the principle that memory accesses to the same location appear as if they were interleaved [17]. This is guaranteed by almost all CPU memory models (except SPARC RMO) [17], and the violation has been deemed a bug on some ARM chips [18]. The behaviour thwarts the implementability of the OpenCL [62], C11 [54], or C++11 [46] memory models, which guarantee single-location coherency for accesses marked as atomic. Implementing this language feature would thus require the addition of a memory fence between any two atomic accesses that might access the same memory location.

We observed this behaviour on all chips based on the Fermi and Kepler architectures that we tested. On the newer GTX 750, which is based on the Maxwell architecture, the behaviour appears to be fixed: we did not observe the weak outcome anymore.

Figure 2.11 shows the message passing with CAS test. This test is of particular importance as it encodes an idiom that typically appears in spinlock implementations. A spinlock is a particular implementation of locks where the thread attempting to acquire the lock waits via a busy-wait loop for the lock to become available. In the given example, the left thread represents the thread releasing the lock, and the right thread represents the thread attempting to acquire the lock. The test checks whether it is possible for the CAS to succeed, and the load to still read stale data (i.e., the initial value 0 of `x` as opposed to the value 1 written to `x` by the first thread).

We observed the behaviour on the GTX Titan. This indicates that the CAS does not imply a memory fence (as for example the `cmpxchg` instruction on x86 CPUs). This has led to bugs in a published spinlock implementation given in the

book CUDA by Example [91]. To prevent the weak behaviour, a memory fence (`membar.gl`) needs to be placed between the CAS and the load.

2.5.2 Manual Tests

We next experimentally investigate the correctness of two GPU spinlock implementations given by He and Yu [51]. In summary, we found that both locks allow to read stale values and future values. That is, a critical section protected by the lock might not see the updates performed by the previous critical section, or might read values from the next critical section. Consequently, the locks do not guarantee that any data-race-free program using them behaves according to SC.

He and Yu [51] use the locks to ensure mutual exclusion between database transactions. In particular, a concurrent execution of transactions should satisfy the *isolation* property: the resulting system state of a concurrent execution of transactions should be the same as a state that can be obtained by a serial execution of the transactions. The weak behaviours of the locks described above could lead to violations of the isolation property.

The same concurrency idioms that appear in the spinlocks also appear in the automatically generated tests. However, it may not always be easy to see to which litmus test a part of a program corresponds. We thus manually translate the CUDA C code to PTX. In the PTX code, we include all the operations that appeared in the original CUDA C code (which may include additional instructions that are irrelevant for memory consistency). It is thus easy to see that the PTX code corresponds to the CUDA C code. Then, running the resulting litmus test on the hardware reveals the memory behaviour of the original CUDA C code.

In the following, we go into more detail about our experiments, and discuss ways of how the locks might be fixed to forbid the weak behaviours.

Spinlock

Figure 2.12a shows an Nvidia CUDA C spinlock implementation given by He and Yu [51]. The lock-protected section is indicated by the comment "`# critical section`". The while loop spins until the CAS returns 0, at which point the thread enters the critical section. After executing the critical section, the thread writes 0 to `*lockAddr`, thereby releasing the lock.

```
1 bool leaveLoop = false;
2 while(!leaveLoop) {
3   int lockValue =
4     atomicCAS(lockAddr,0,1);
5   if(lockValue == 0) {
6     leaveLoop = true;
7     # critical section
8     *lockAddr = 0;
9   }
10  __threadfence();
11 }
```

(a) Original [51]

```
1 bool leaveLoop = false;
2 while(!leaveLoop) {
3   int lockValue =
4    atomicCAS(lockAddr,0,1);
5   if(lockValue == 0) {
6     leaveLoop = true;
7     __threadfence();
8     # critical section
9     __threadfence();
10    atomicExch(lockAddr, 0);
11  }
12 }
```

(b) Corrected

Fig. 2.12: Spinlock implementations

Typically, the steps of attempting to acquire a lock and the step of releasing a lock are factored out into separate `lock()` and `unlock()` operations. In the shown implementation, those steps are instead implemented as a single code snippet with a hole in it (indicated by the comment) to plug in the code to be protected. The code executed before reaching line 7 can be seen as the implementation of `lock()`, and the code executed when continuing to execute after line 7 can be seen as the implementation of `unlock()`.

To run the code on a GPU, it must first be compiled using Nvidia's compilation toolchain. The toolchain first translates the CUDA C code to PTX code and the PTX code is then translated to machine code by the PTX assembler.

To test the correctness of the lock given in Figure 2.12a, we manually built a PTX version of it. Our PTX version represents one possible compilation of the CUDA C code to PTX code. We give a schematic view of our PTX code in Figure 2.13. The left and the right code portion represent two threads T_0 and T_1.

The code implements a possible scenario that might occur during the concurrent execution of two threads that use the lock. T_0 represents a thread that currently has the lock and is about to unlock it. T_1 represents a thread that attempts to acquire the lock. The initial state of the global variables at the start of the program is given at the top of the figure: `lock = 1` and `x = 0`. We are interested whether there is an execution of the program such that we end up in a final state that satisfies `r0 = 1` $\wedge$ `r2 = 0`. This corresponds to the scenario where thread T_0

reads value 1 from `x` (at access 0.1). But this value can only be produced by the write 1.4 in thread T_1. But this write can only happen when T_1 has successfully acquired the lock (and thus T_0 must have released the lock before). Hence, the scenario corresponds to reading a value from the *next* critical section, i.e., a value from the future.

We could observe the behaviour described in the previous paragraph on both the Nvidia Tesla C2075 card and the Nvidia GTX Titan card. For the former we observed it in 40 out of 100 000 tests runs, and for the latter in 478 out of 100 000 test runs. In addition to the test shown in Figure 2.13, we also produced a test that checks whether a critical section can read a stale value. That is, whether a critical section might not see the update of the previous critical section. Again, we observed this weak outcome.

Going back to Figure 2.12a, we explain why the lock allows the weak behaviours. The behaviours arise because the CAS at the entry of the critical section (line 4) does not provide any ordering. Unlike, e.g., the `cmpxchg` instruction on x86, CUDA read-modify-write (RMW) instructions (like `atomicCAS()`) do not imply any ordering. The `__threadfence()` in the lock has no effect on memory consistency, as it was placed *after* the lock release. Correctly, it would need to be placed *before* the lock release.

Figure 2.12b shows a proposed fix for the spinlock. We added a `__threadfence()` after the CAS at the entry of the critical section, and before the release of the lock. Moreover, the lock is now released via an `atomicExch()` operation. Using a normal store for the lock release might thwart the atomicity of the CAS which is used to acquire the lock. The PTX manual [83] states: "Atomic operations on shared memory locations do not guarantee atomicity with respect to normal store instructions to the same address. It is the programmer's responsibility to guarantee correctness of programs that use shared memory atomic instructions, e.g., by inserting barriers between normal stores and atomic operations to a common address, or by using `atom.exch` to store to locations accessed by other atomic operations." With the described fixes, we do not observe the weak behaviour anymore in our experiments.

init: `lock = 1 ∧ x = 0` **final:** `r0 = 1 ∧ r2 = 0` **threads:** inter-CTA

	Thread 0			Thread 1	
0.1	`ld.cg r0,[x]`	7*	1.1	`atom.cas r2,[lock],0,1`	4*
0.2	`st.cg [lock],0`	8	1.2	`setp.eq p,r2,0`	5
0.3	`membar.gl`	10	1.3	`@p mov r3,1`	6
			1.4	`@p st.cg [x],1`	7

**original line in Figure 2.12a*

obs/100k	GTX 540m	Tesla C2075	GTX 660	GTX Titan	GTX 750
	0	40	0	478	0

Fig. 2.13: Future value test for spinlock

Counter-Based Spinlock

Similarly as in the previous section, we also investigated the correctness of the counter-based spinlock given in [51].

The original code of the lock is shown in Figure 2.14a. We have again distilled a PTX test out of the CUDA code, shown in Figure 2.15. The test checks whether the lock allows to read stale values. In our experiments, we observed that access 1.4 can read 0, even when the lock was acquired. We observed this behaviour 22 times on 100 000 runs on the Nvidia GTX Titan. Additionally, the counter-based spinlock also allows to read future values.

In Figure 2.14b, we show a proposed fix for the counter-based spinlock. In an analogue to Figure 2.12b, we added a `__threadfence()` both at the entry and exit of the critical section. With these additions, we do not observe the weak outcome anymore in our experiments.

```
1  bool leaveLoop = false;
2  while(!leaveLoop) {
3    volatile int lockValue =
4     *lockAddr;
5    if(lockValue == keyValue) {
6      leaveLoop = true;
7      # critical section
8      if (flag == marked)
9        atomicAdd(lockAddr, 1);
10   }
11   __threadfence();
12 }
```

(a) Original [51]

```
1  bool leaveLoop = false;
2  while(!leaveLoop) {
3    int lockValue =
4      atomicAdd(lockAddr, 0);
5    if(lockValue == keyValue) {
6      leaveLoop = true;
7      __threadfence();
8      # critical section
9      if (flag == marked) {
10       __threadfence();
11       atomicAdd(lockAddr, 1);
12     }
13   }
14 }
```

(b) Corrected

Fig. 2.14: Counter-based lock implementation

init: `lock = 0 ∧ x = 0` **final:** `r3 = 1 ∧ r5 = 0` **threads:** inter-CTA

		*			*
0.1	`st.cg [x],1`	7	1.1	`ld.cg r3,[lock]`	4
0.2	`ld.cg r0,[flag]`	8	1.2	`setp.eq p,r3,1`	5
0.3	`ld.cg r1,[marked]`	8	1.3	`@p mov r4,1`	6
0.4	`setp.eq p,r0,r1`	8	1.4	`@p ld.cg r5,[x]`	7
0.5	`@p atom.inc r2,[lock],1`	9			
0.6	`membar.gl`	11			

**original line in Figure 2.14a*

obs/100k	GTX 540m	Tesla C2075	GTX 660	GTX Titan	GTX 750
	0	0	0	22	0

Fig. 2.15: Stale value test for counter lock

2.6 Related Work

Memory model testing Modern multiprocessors are not sequentially consistent [66]. The behaviours they can exhibit are typically illustrated by small concurrent code snippets (as, e.g., in [39, 107]) called litmus tests. The term litmus test first appeared in the DEC Alpha manual [95]. By running litmus tests on hardware, one can determine the behaviours it can exhibit. This is useful for testing whether a certain synchronisation idiom is subject to weak behaviours, for building a formal model of the hardware behaviour, or for testing whether the hardware conforms to a given model. The ARCHTEST tool [35] runs several fixed tests to check for weak behaviours. However, the number of tests it runs is small, and does not come near the rich sets of litmus tests needed to illustrate architectures such as POWER [92]. Alglave et al. [16] developed a method to automatically generate litmus tests based on an axiomatic modelling framework. They have implemented their approach in the DIY toolsuite [16], comprising among others the diy and litmus tools to generate and run litmus tests on CPUs. Wickerson et al. [109] use the Alloy modelling framework [56] to generate litmus tests in order to compare memory consistency models. Litmus tests have also been studied formally [75], and have been shown to describe important properties of memory systems such as model equivalence [74]. Sorensen [96] first used litmus tests to provide intuition about the memory model of Nvidia GPUs, and formulated a preliminary operational memory model. Sorensen and Donaldson [97] investigated how to provoke weak memory effects in full GPU applicatons (as opposed to for litmus tests only). They provide a testing environment to execute applications in while stressing the memory system in order to reveal weak behaviours.

Microbenchmarking Our approach of determining possible memory behaviours of GPUs by running litmus tests is related to *microbenchmarking*. While we are concerned with *functional* aspects, the purpose of microbenchmarking is typically to gather data to be used in the performance optimization of GPU programs. GPUBench [7] is a benchmark suite consisting of several programs to determine statistics such as memory bandwidth and instruction throughput of AMD and Nvidia GPUs. Wong et al. [112] developed a test suite to reveal microarchitectural aspects of Nvidia GeForce GT200 and GTX280 GPUs, such as cache parameters. Feng and Xiao [44] analyse the overhead of barrier synchronisation.

Checking for optimisations Our checking whether a litmus test has been optimised (see Section 2.4) is related to testing of compiler optimisations for concurrent programs. Eide and Regehr [41] check whether accesses to C volatile variables are compiled correctly. They compile a test case both with and without optimisations (e.g., `-O3` and `-O0`), then run both versions with the same input while logging the accesses to volatile variables. If the traces of the two versions differ, an invalid optimisation has been detected. Morisset et al. [79] extend this work to a subset of C++11. Our approach differs from these in that we do not make use of an unoptimised version of the code, but instead embed a specification of the expected instruction sequence into the optimised version. Moreover, we statically check whether the compiled code conforms to the specification. Finally, the methods have different aims: our aim is not to find compiler bugs but to detect unwanted reorderings due to compilation.

Fine-grained synchronisation on GPUs Several lock-free algorithms, synchronisation mechanisms and data structures for GPUs have been proposed in the literature. At present, most work targets Nvidia GPUs, due to the availability of the CUDA general-purpose GPU progamming framework [85] and the PTX virtual assembly language [86]. Stuart and Owens [100] describe several synchronisation primitives for Nvidia GPUs, employing instructions such as atomic exchange, but no explicit memory fences. CUDA by Example [91] presents a spinlock implementation for inter-block communication using compare-and-swap. Misra and Chaudhuri [78] describe lock-free GPU implementations of several concurrent data structures, such as a linked list and a hash table. Feng and Xiao [44] present a barrier implementation for inter-block communication. They argue that one need not use a memory fence, as the likelihood of observing incorrect behaviour on a GeForce GTX 280 is "infinitesimally small *in practice*." [44]. He and Yu [51] give spinlocks used in a GPU database implementation. Our testing framework allows to express the synchronisation idioms used in much of the above work. As we have shown, some of the above algorithms are incorrect on current hardware, allowing, e.g., a spinlock-protected critical section in a thread to read stale data.

Chapter 3

Thread Refinement

3.1 Introduction

A program P' is said to be a *refinement* of another program P if, for all inputs to the programs, all the observable behaviours of P' can also be observed with P. Different definitions of what constitutes program inputs and observable behaviours are possible. We consider the input to be the initial state of the program, and the observable behaviour to be the final state of the program upon termination.

A special form of refinement is contextual refinement [70]. A code portion P' is a *contextual refinement* of another code portion P if, for all inputs and for all environments C (termed *contexts*, which are code portions as well) with which P' and P could be composed, the observable behaviours of the composition of P' and C can also be observed with the composition of P and C. When we refer to refinement in the following we shall mean contextual refinement.

We consider the contextual refinement problem between threads in the SC-for-DRF memory model, with locks used for synchronisation (cf. Chapter 1). The refinement problem between threads appears in various settings, such as regression verification [29], the proving of correctness of compiler optimisation passes [106], or compiler testing [79]. Informally, we say a thread T' is a refinement of a thread T if for all possible concurrent contexts $C = T_1 \parallel \ldots \parallel T_n$ (which is a parallel composition of other threads, denoted by $\parallel$), for all initial states, the set of final states reachable by $T' \parallel C$ is a subset of the set of final states reachable by $T \parallel C$. Often the thread T' and the thread T are related in some way, e.g., T' might have

been obtained from T by applying a set of compiler optimisations. We thus refer to T' as the transformed thread and to T as the original thread.

Formalising Thread Refinement

The informal definition of thread refinement given above is not directly useful for automated or manual reasoning about thread refinement, as it would require the enumeration of all possible concurrent contexts C. We thus develop a new theory that is better suited for those tasks.

The theory represents a thread as the set of sequences of state transitions it can perform between synchronisation operations. This is similar to the representation used by Brookes in the work about a compositional semantics for shared memory programs with data races [25, 26]. Based on this representation of threads, we develop a refinement condition which implies that a thread T' is a refinement of another thread T in an arbitrary context C.

Complexity arises in such an approach with the desire to be able to show refinement in a large number of practically relevant cases. For example, compiler optimisations like roach motel reorderings (see, e.g., [105]) can alter the state transitions a thread can perform between synchronisation operations. Thus, requiring complete equality between state transitions of T' and T would be too strong. Consequently, in Sections 3.3 and 3.5 we develop a condition that is weaker while still being sound.

We improve over existing work both in terms of *precision* and *efficiency*. First, our theory allows us to show refinement in cases where others fail. For example, we also allow the reordering of shared memory accesses out of critical sections (under certain circumstances); a transformation that is unsupported by other theories. Second, we show that applying our new specification method in a compiler testing setting leads to large performance gains. We can check whether two thread execution traces match significantly faster than a previous approach of Morisset et al.[79].

3.2 Transition-Based and Event-Based Refinement

Current theories of refinement for language-level memory models (such as the Java Memory Model or SC-for-DRF) are phrased in terms of transformations on thread executions. Thread executions are represented as partial or total orders (i.e., traces) of memory events (reads or writes) and synchronisation events (lock or unlock). Examples include Manson et al.[76], Sevcik and Aspinall [105], Boehm [22], Morisset et al. [79], and Sevcik [104]. The transformations on the thread executions are then lifted to transformations on the program code.

We focus in the following on the SC-for-DRF memory model and assume that thread executions are represented as traces (denoted by the symbol r). The valid transformations are typically given as descriptions of which *reorderings*, *eliminations*, and *introductions* of memory events on a trace are allowed. Checking whether a trace r' is a correctly transformed version of a trace r then amounts to determining whether there is a sequence of valid transformations that turns trace r into trace r'. If each trace r' of T' is a transformed version of a trace r of T, it follows that T' is a refinement of T.

We show that instead of describing refinement via a sequence of valid transformations on traces, switching to a theory based on state transitions is beneficial. In essence, in the transition-based approach, we require that traces r' and r perform similar state transitions between corresponding synchronisation operations, and that r' does not allow more data races than r. Given a trace r, by representing the execution portions between successive lock operations as state transitions, we abstract over the concrete reads and writes the thread performs between the lock operations.

In the following, we informally explain the key reason for why thread refinement can be specified this way. In the SC-for-DRF memory model, in a data-race-free program, memory accesses to the same location by different threads are protected by the same lock. Those lock-protected sections cannot be interleaved with each other. Thus, given a certain memory location x, if a thread T writes to this location several times in a lock-protected section, other threads can only observe the *last* write of T to x in this section. Therefore, all the intermediate writes that occured before are irrelevant. Similarly, if, in a lock-protected section, a thread

reads a value from a certain memory location x that was written by another thread, then all prior reads from x in the same section must have returned the same value.

As we show in Section 3.6, by representing traces as sequences of state transitions, we can check whether a trace is a correctly transformed version of another trace significantly faster than a previous approach based on trace transformations [79].

In the next section, we illustrate the difference between the transition-based and the event-based refinement approaches on an example.

3.2.1 Example

Consider Figure 3.1, which shows an original thread T, a (correctly) transformed version T', and a concurrent context C in the form of another thread. The threads access shared variables x, y, z and local variables a, b. The context C outputs the value of variable z in the final state. By inspecting $T' \parallel C$ and $T \parallel C$ (assuming initial state $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$), we see that both combinations produce the same possible outputs (0 or 2). In fact, T' and T exhibit the same behaviour in any concurrent context C for which $T \parallel C$ is data-race-free.

Now let us look at two traces r' of T' and r of T, and how an event-based and our transition-based theory would establish refinement. We assume for now that T and T' are only composed with contexts that do not write any shared memory locations accessed by them (as is the case for, e.g., the context shown in Figure 3.1c). Figure 3.2a shows the execution traces of T (left trace) and T' (right trace) for initial state $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$.

Event-based refinement A theory based on trace transformations (see Figure 3.2a) would establish the refinement between the two traces by noting that the event `write x 1` can be removed ("overwritten write elimination"), the events `unlock m` and `write y 2` can be reordered ("roach motel reordering"), and the event `read y 2` can be introduced ("irrelevant read introduction"). Proof of refinement is significantly more complicated if longer traces and further optimisations are considered.

Transition-based refinement We specify trace refinement by requiring that r', r perform the same sequence of lock operations, that r' does not allow more

```
1  void thread_orig() {
2    int a, b;
3    lock(l);
4    x = 1;
5    x = 2;
6    unlock(l);
7    a = y;
8    lock(l);
9    y = 1;
10   unlock(l);
11   y = 2;
12 }
```

(a) Original thread

```
1  void thread_trans() {
2    int a, b;
3    lock(l);
4    x = 2;
5    unlock(l);
6    a = y;
7    lock(l);
8    y = 1;
9    y = 2;
10   unlock(l);
11   b = y;
12 }
```

(b) Transformed thread

```
1  void context() {
2    int a;
3    lock(l);
4    a = x;
5    z = a;
6    unlock(l);
7    join(thread_{orig|
8                 trans});
9    printf("%d\n", z);
10 }
```

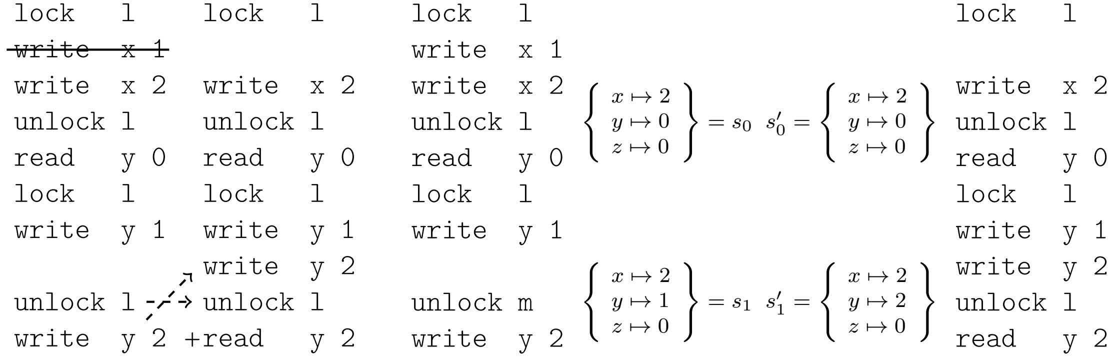
(c) Context

Fig. 3.1: Original thread T, transformed thread T', and concurrent context C

data races than r, and that r' and r end up in similar states at corresponding `unlock()` operations. The reason for why trace refinement can be specified that way is that any context C for which $T \,||\, C$ is data-race-free can for each shared variable only observe the *last write* to it before an `unlock()` operation. If it could observe any intermediate write, there necessarily would be a data race. Thus, only the values of the shared variables at `unlock()` operations are relevant.

Our approach is illustrated in Figure 3.2b. We depict above the two traces and below the conditions that need to be satisfied such that we consider r' a refinement of r. We denote by $M = \{x, y, z\}$ the set of all shared memory locations. We use s_0, s_1, s_0', s_1' do denote the state of r, r' at the two `unlock(l)` operations (assuming initial state $\{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$). We further denote by R_i, R_i' (W_i, W_i') the sets of memory locations read (resp. written) by r, r' between subsequent lock operations. For example, $W_0 = \{x\}$ denotes the set of memory locations written by r between the first `lock(l)` and the subsequent `unlock(l)` operation, and $R_1 = \{y\}$ denotes the set of memory locations read by r between the first `unlock(l)` and the subsequent `lock(l)`. We also use the abbreviations $A_i' = R_i' \cup W_i'$ and $A_i = R_i \cup W_i$.

In Figure 3.2b, we see that both traces are in state $\{x \mapsto 2, y \mapsto 0, z \mapsto 0\}$ at the first `unlock(l)` operation. At the second `unlock(l)` operation, r is in state $\{x \mapsto 2, y \mapsto 1, z \mapsto 0\}$ and r' is in state $\{x \mapsto 2, y \mapsto 2, z \mapsto 0\}$. However, the value of variable y is irrelevant at this point, as y is written to by r immediately



(a) Event-based matching

$$R'_0 \subseteq (A_0 \cup A_1) \qquad R'_2 \subseteq (A_1 \cup A_2 \cup A_3)$$
$$W'_0 \subseteq (W_0 \cup W_1) \qquad W'_2 \subseteq (W_1 \cup W_2 \cup W_3)$$

$$R'_1 \subseteq A_1 \qquad R'_3 \subseteq (A_2 \cup A_3)$$
$$W'_1 \subseteq W_1 \qquad W'_3 \subseteq (W_2 \cup W_3)$$

$$\forall x \in M - W_1 : s'_0(x) = s_0(x)$$
$$\forall x \in M - W_3 : s'_1(x) = s_1(x)$$

(b) Transition-based matching

Fig. 3.2: Trace matching

after the `unlock(l)` operation. Thus, any context that could observe the write to y before the `unlock(l)` operation would necessarily have a data race with T. We thus only require those shared variables to be equal that could be read without introducing a data race. The condition is formalised by the last two constraints in Figure 3.2b.

In addition to requiring that r' and r are in similar states, we also require that r' does not allow more data races than r. This requirement is specified by the subset constraints in Figure 3.2b. As an example, the condition $W'_0 \subseteq W_0 \cup W_1$ says that any memory location written by r' between the first `lock(l)` and the subsequent `unlock(l)` must also be written by r either between the first `lock(l)` and the subsequent `unlock(l)`, or between the first `unlock(l)` and the subsequent `lock(l)`. Since for $x \in W'_0$ we require only that $x \in W_0$ or $x \in W_1$, this allows a write to move into the critical section in r' compared to r. We will capture the set constraints more precisely in Section 3.3.

3.2.2 Contexts that Write

We now assume that a thread can be put in an arbitrary context that can also write to the shared state. Thus, when generating the traces of a thread we also need to take into account that a read of a variable `x` could yield a value that is both different from the initial value of `x`, and which the thread has not itself written (i.e., it was written by the context).

In an event-based theory this is typically handled by assuming that reads can return arbitrary values (see, e.g., [79]). However, this assumption is unnecessarily general. For example, if a thread reads the same variable twice in a row with no intervening lock operation, and it did not itself write to the variable, then both reads need to return the same value. Otherwise, this would imply that another thread has written to the variable and thus there would be a data race.

In fact, when generating the traces of a thread, it is sufficient to assume that a thread observes the shared state only at its `lock()` operations. The reason for this is that `lock()` operations synchronise with preceding `unlock()` operations of other threads. And those threads in turn make their writes available at their `unlock()` operations.

3.2.3 Summary

To summarise, we state the intuitive formulation of our refinement theory. We will formalise this notion in the next section.

> We say that thread T' is a refinement of thread T if for each trace r' of T' there is a trace r of T such that r' and r match.

> We say two traces r', r match if (1) they perform the same sequence of lock operations, (2) the sets of memory locations accessed by r' are subsets of corresponding sets of memory locations accessed by r, and (3) r' and r perform similar state transitions between lock operations.

$$P ::= S \mid S \parallel P$$
$$S ::= I \mid I; S$$
$$I ::= m := E \mid [E_l]\ \texttt{goto}\ n \mid \texttt{lock}(l) \mid \texttt{unlock}(l)$$

Fig. 3.3: Grammar of our programming language

```
1 if(c) {           1 [!c] goto 4
2   x = 1;          2 x = 1
3 } else {   ⇒      3 [true] goto 5
4   x = 2;          4 x = 2
5 }                 5 ...
```

```
1 while(a < 10) {        1 [!(a < 10)] goto 4
2   a = a + 1;     ⇒     2 a = a + 1
3 }                      3 [true] goto 1
                         4 ...
```

Fig. 3.4: Translation examples

3.3 Formalisation of Transition-Based Refinement

We now formalise the ideas from the previous section. We first formally define a program and execution model (i.e., the SC-for-DRF memory model). Then we define what it means for a thread to be a refinement of another thread in that model. Finally, we formulate our refinement condition that implies the definition of refinement.

3.3.1 Program Model

We consider a simple programming language that includes parallel composition of threads, assignments to local and shared variables, guarded goto statements to alter the control-flow and implement loops, and `lock(l)` and `unlock(l)` operations for synchronisation between threads.

Syntax

Let M be a finite set of *shared* variables $x_1, \ldots, x_{|M|}$. Let M_l be a finite set of *local* variables $u_1, \ldots, u_{|M_l|}$. Let L be a finite set of locks $l_1, \ldots, l_{|L|}$. The grammar of our language is given in Figure 3.3. The symbol m denotes an element of M or M_l, E denotes an expression over M and M_l, E_l denotes an expression over M_l, n denotes an integer greater or equal to zero, and l denotes a lock.

$$\frac{\begin{array}{l} T \in \mathcal{T}(P) \\ T[pc(T)] = m \texttt{ := } E \\ pc' = pc[T/pc(T)+1] \\ m \in M \Rightarrow (s' = s[m/\mathsf{eval}(E, s_l(T), s)] \wedge s'_l = s_l) \\ m \in M_l \Rightarrow (s'_l = s_l[m/\mathsf{eval}(E, s_l(T), s)] \wedge s' = s) \end{array}}{(P, pc, s_l, s, ls) \rightarrow (P, pc', s'_l, s', ls)} \ \textbf{as}$$

$$\frac{\begin{array}{l} T \in \mathcal{T}(P) \\ T[pc(T)] = \texttt{[}E_l\texttt{] goto } n \\ \mathsf{eval}(E_l, s_l(T), s) = 0 \Rightarrow pc' = pc[T/pc(T)+1] \\ \mathsf{eval}(E_l, s_l(T), s) \neq 0 \Rightarrow pc' = pc[T/n] \end{array}}{(P, pc, s_l, s, ls) \rightarrow (P, pc', s_l, s, ls)} \ \textbf{go}$$

$$\frac{\begin{array}{l} T \in \mathcal{T}(P) \\ T[pc(T)] = \texttt{lock}(l) \\ pc' = pc[T/pc(T)+1] \\ l \notin ls \\ ls' = ls \cup \{l\} \end{array}}{(P, pc, s_l, s, ls) \rightarrow (P, pc', s_l, s, ls')} \ \textbf{lo} \qquad \frac{\begin{array}{l} T \in \mathcal{T}(P) \\ T[pc(T)] = \texttt{unlock}(l) \\ pc' = pc[T/pc(T)+1] \\ ls' = ls - \{l\} \end{array}}{(P, pc, s_l, s, ls) \rightarrow (P, pc', s_l, s, ls')} \ \textbf{un}$$

Fig. 3.5: Transition relation

We assume that an assignment instruction accesses at most one shared variable. That is, it is either a shared write (i.e., the lhs is a shared variable and the rhs accesses only local variables), a shared read (i.e., the lhs is a local variable and the rhs contains one variable from M), or it does not access shared variables.

A program according to the grammar is a parallel composition of threads (denoted by $\|$), each of which is a sequence of instructions. We assume the instructions in a thread are indexed starting with 0. The goto statement jumps to the instruction with the given index n when the expression E_l evaluates to a nonzero value.

In the remainder of this chapter, we continue to give code examples in C-like notation. The translation to the programming language defined in Figure 3.3 is straightforward. Figure 3.4 gives an example of how if-statements and while-loops can be expressed in our language. We chose our program notation for a number of reasons. First, the language is simple in that it consists of only four instructions. The semantics definition (Figure 3.5) consequently also consists of

four rules only. Second, the language is close to assembly and the intermediate languages on which compilers typically perform code optimisation. Third, there is a simple correspondence between instructions and execution steps. An execution step corresponds to the execution of exactly one instruction.

Semantics

We next define the semantics of our language by means of a transition relation $\rightarrow$ between program configurations. The formal definition of the transition relation is given in Figure 3.5. We use $\mathcal{T}(P)$ to denote the set of threads in program P. We refer to the threads in a program by $T_1, \ldots, T_n$. We use those symbols both to refer to the code portions of the threads as well as the corresponding thread IDs. The distinction will be clear from context.

A program configuration is a five-tuple (P, pc, s_l, s, ls). Given a configuration c, we write $P(c), pc(c), s_l(c), s(c)$, and $ls(c)$ for the respective components. The first component of a configuration denotes the program.

The second component of a configuration denotes the vector of program counters, which has one component for each thread in the program. A program counter is an integer greater or equal to zero which indexes into the list of instructions of the corresponding thread. It points to the next instruction to be executed. We write $pc(T)$ for the component of the vector of program counters corresponding to the program counter of thread T. We write pc_s for the initial vector of program counters with all components being zero. We write pc_f for the final vector of program counters with all threads having terminated.

We use the predicate $\mathsf{lock}(T, pc)$ (resp. $\mathsf{unlock}(T, pc)$) to denote that the next operation to be executed by thread T is a `lock(l)` (resp. `unlock(l)`) operation. We use $\mathsf{term}(T, pc)$ to denote that thread T has terminated.

The third component of a configuration denotes the vector of local states of the threads. We write $s_l(T)$ for the local state of thread T. A local state $s_l(T)$ is a total function $s_l(T)\colon M_l \rightarrow V$ from the set of local variables M_l to the set of values V. We write $s_{l,s}$ for the initial local state with all variables being zero.

The fourth component of a configuration denotes the shared state. A shared state is a total function $s\colon M \rightarrow V$ from the set of shared variables M to the set of values V. We denote the set of all shared states by S.

The fifth component of a configuration denotes the set of locks that are currently locked (by any thread). On program startup the lockset is always empty.

Instruction Semantics

We denote by $T[i]$ the instruction at index i of thread T. Thus, $T[pc(T)]$ denotes the instruction pointed to by program counter $pc(T)$. We denote the updating of a vector or a mapping by $f[x/v]$. Thus, $pc[T/pc(T)+1]$ denotes the vector of program counters pc with the component corresponding to T increased by one. We denote by $\mathsf{eval}(E, s_l(T), s)$ the value of expression E when evaluated with local state $s_l(T)$ and shared state s.

An assignment instruction updates a local or shared variable with the value of the rhs expression. We assume that an assignment instruction accesses at most one shared variable.

A guarded goto instruction updates the program counter of the thread with the given value n when the expression E_l evaluates to a nonzero value, and increments the program counter by one if it evaluates to zero.

A `lock(l)` operation adds the lock `l` to the lockset ls. When the lock is already in the lockset, the thread is blocked at the lock operation, i.e., it cannot make a step until the lock becomes available. When all threads are blocked at a `lock()` operation, we say the configuration is *stuck*.

An `unlock(l)` operation removes the given lock `l` from the lockset ls. According to this semantics the lock will be removed from the lockset even if it was initially locked by a different thread. We will later define a well-lockedness criterion which will ensure that a thread only unlocks a lock it has previously locked.

Program Executions

We next define a notion of program executions and thread executions based on the transition relation.

Definition 1 (executions)**.** An *execution fragment* of a program P is a sequence of pairs of configurations $(c_0, c_1)(c_1, c_2)\ldots(c_{n-1}, c_n)$ such that $n \in \mathbb{N}$ or $n = \omega$ and $\forall\, 0 \leq i < n\colon c_i \rightarrow c_{i+1} \wedge P(c_i) = P$. An *execution* is an execution fragment that starts in a configuration with initial PC vector pc_s, initial local state vector $s_{l,s}$, and an empty lockset, and for which either $n = \omega$ (the execution is infinite),

$n \in \mathbb{N}$ and $pc(c_n) = pc_f$ (the execution terminates), or $n \in \mathbb{N}$ and c_n is a stuck configuration (i.e., all threads are blocked on a `lock()` operation). An *execution prefix* is a finite execution fragment that starts in a configuration with initial PC vector pc_s, initial local state vector $s_{l,s}$, and the empty lockset.

Given an execution e, we refer to the pairs of configurations it is made up of as *steps*. We further write $e[i]$ for the step of e with index i. We assume indices are zero-based. We next define thread executions. Thread executions are like ordinary executions except that they only consist of steps of a single thread and the shared state and lockset may change arbitrarily between the steps of the thread.

Definition 2 (thread executions)**.** A *thread execution fragment* of a thread T is a sequence of pairs of configurations $(c_0, c'_0)(c_1, c'_1) \dots (c_{n-1}, c'_{n-1})$ such that $n \in \mathbb{N}$ or $n = \omega$ and $\forall\, 0 \leq i < n\colon c_i \to c'_i \wedge P(c_i) = T$ and $\forall\, 0 \leq i < n-1\colon pc(c'_i) = pc(c_{i+1}) \wedge s_l(c'_i) = s_l(c_{i+1})$. The PC vectors and local state vectors consist in this case of only a single component for thread T. A *thread execution* is a thread execution fragment that starts in a configuration with initial PC vector pc_s and initial local state vector $s_{l,s}$, and for which either $n = \omega$ (the thread execution is infinite), $n \in \mathbb{N}$ and $pc(c_{n-1}) = pc_f$ (the thread execution terminates), or $n \in \mathbb{N}$ and c'_{n-1} is a stuck configuration (i.e., the thread is blocked on a `lock()` operation). A *thread execution prefix* is a finite thread execution fragment that starts in a configuration with initial PC vector pc_s and initial local state vector $s_{l,s}$.

We further write $e|_T$ for the thread execution obtained from execution e by projecting it to the steps of T, and adjusting the configurations to only mention the components corresponding to T (e.g., the vector of program counters will only contain the component for T).

In Figure 3.6 we define several predicates and functions on execution fragments and thread execution fragments. We usually leave the execution e off when it is clear from context. The expression $\mathsf{src}(e, i)$ (resp. $\mathsf{tgt}(e, i)$) refers to the configuration to the left (resp. right) of $\to$ of the transition corresponding to step i of e.

Well-Locked Executions

For the remainder of this section, we assume that the threads in the programs we consider are *well-locked*. Most notably, we assume that the threads do not acquire and release locks in a nested fashion. This means that `lock(l)` and `unlock(l)`

operations occur alternately on each thread execution. In Section 3.5 we later adapt the formalisation to also handle nested locks.

We further assume that `lock()` and `unlock()` operations occur infinitely often on any infinite thread execution. This implies that a thread cannot get stuck, e.g., in an infinite loop without reaching a next lock operation. The assumption will slightly simplify the formalisation in the following. Since we represent thread executions as sequences of state transitions between lock operations, it ensures that there is always a transition to another state (or the thread has terminated). The assumption could be lifted by introducing a special state which indicates that the thread does not terminate and does not execute another lock operation.

We also assume that the first operation in a thread is a `lock()`, and the last *lock* operation in a thread is an `unlock()`. Our assumptions are formalised by the $\mathsf{well\text{-}locked}(e)$ predicate below.

Definition 3 (well-locked executions)**.** We say a thread execution e of a thread T is *well-locked* if the following holds (lines are assumed to be conjunctively connected):

$$
\begin{aligned}
&\mathsf{well\text{-}locked}(e) \Leftrightarrow \\
&\quad \textbf{let } n = |e| \textbf{ in} \\
&\quad n \geq 2 \\
&\quad \mathsf{lock}(0) \\
&\quad \forall\, 0 \leq i < n\colon \\
&\qquad \mathsf{lock}(i) \Rightarrow \exists\, i < j < n\colon \mathsf{unlock}(j) \wedge \mathsf{loc}(i) = \mathsf{loc}(j) \wedge \forall\, i < k < j\colon \neg\mathsf{sync}(k) \\
&\qquad \mathsf{unlock}(i) \Rightarrow \exists\, i < j < n\colon (\mathsf{lock}(j) \vee (j = n - 1 \wedge \neg\mathsf{sync}(j))) \wedge \\
&\qquad\qquad\qquad\qquad \forall\, i < k < j\colon \neg\mathsf{sync}(k)
\end{aligned}
$$

We say an execution e of a program P is well-locked if all the threads are well-locked: $\forall\, T \in \mathcal{T}(P)\colon \mathsf{well\text{-}locked}(e|_T)$.

3.3.2 Interleaving Semantics

We now define the semantics of a program according to interleaving semantics as the set of its initial/final state pairs (cf. Chaki et al. [29]).

$\mathsf{wr}(e,i)$:	step i of e is a shared write
$\mathsf{rd}(e,i)$:	step i of e is a shared read
$\mathsf{mem}(e,i)$:	$\mathsf{wr}(e,i) \vee \mathsf{rd}(e,i)$
$\mathsf{lock}(e,i)$:	step i of e is a lock
$\mathsf{unlock}(e,i)$:	step i of e is an unlock
$\mathsf{sync}(e,i)$:	$\mathsf{lock}(e,i) \vee \mathsf{unlock}(e,i)$
$\mathsf{loc}(e,i)$:	memory loc./lock accessed by step i of e, $\perp$ if none accessed
$\mathsf{conflict}(e,i,j)$:	$\mathsf{loc}(e,i) = \mathsf{loc}(e,j) \wedge (\mathsf{wr}(e,i) \vee \mathsf{wr}(e,j))$
$\mathsf{th}(e,i)$:	thread that performed step i of e
$\mathsf{src}(e,i)$:	source configuration of step i of e
$\mathsf{tgt}(e,i)$:	target configuration of step i of e
$\mathsf{initial}(e)$:	initial shared state of e
$\mathsf{final}(e)$:	final shared state of e, or $\perp$ if e is infinite or stuck

Fig. 3.6: Notation

Definition 4 (partial interleaving semantics)**.** $\mathbb{M}(P) = \{(s, s') \mid$ there exists an execution e of P such that $\mathsf{initial}(e) = s \wedge \mathsf{final}(e) = s' \wedge s' \neq \perp\}$.

Only finite executions are relevant for the program semantics as defined above. Consequently, two programs P', P for which $\mathbb{M}(P') = \mathbb{M}(P)$ might have different behaviour. For example, P' might have a nonterminating execution while P might always terminate. The programs P' and P are thus only *partially equivalent*.

Definition 5 (interleaving semantics)**.** $\mathbb{M}_t(P) = \{(s, s') \mid$ there exists an execution e of P such that $\mathsf{initial}(e) = s \wedge \mathsf{final}(e) = s'\}$.

We next define the sequenced-before (sb), synchronises-with (sw), and happens-before (hb) relation for a given execution e (with $|e| = n$). It holds that $(i,j) \in \mathsf{sb}$ if $0 \leq i < j < n$ and $\mathsf{th}(i) = \mathsf{th}(j)$. It holds that $(i,j) \in \mathsf{sw}$ if $0 \leq i < j < n$, $\mathsf{unlock}(i)$, $\mathsf{lock}(j)$, and $\mathsf{loc}(i) = \mathsf{loc}(j)$. The happens before relation hb is then the transitive closure of $\mathsf{sb} \cup \mathsf{sw}$.

Definition 6 (hb race)**.** We say an execution e (with $|e| = n$) contains an *hb data race*, written $\mathsf{hb\text{-}race}(e)$, if there are $0 \leq i < j < n$ such that $\mathsf{th}(i) \neq \mathsf{th}(j)$, $\mathsf{loc}(i) = \mathsf{loc}(j)$, $\mathsf{wr}(i)$ or $\mathsf{wr}(j)$, and $(i,j) \notin \mathsf{hb}$.

Definition 7 (adjacent access race)**.** We say an execution e (with $|e| = n$) contains an *adjacent access data race*, written $\mathsf{adj\text{-}race}(e)$, if there are $0 \leq i < j < n$ with $i = j - 1$, $\mathsf{th}(i) \neq \mathsf{th}(j)$, $\mathsf{loc}(i) = \mathsf{loc}(j)$, and $\mathsf{wr}(i)$ or $\mathsf{wr}(j)$.

The following lemma shows that these two data race definitions are equivalent when they are lifted to the level of programs. For a proof see, e.g., Boehm and Adve [23].

Lemma 8. A program has an execution that contains an hb data race if and only if it has an execution that contains an adjacent access data race.

We write $\mathsf{race}(P)$ to indicate that program P has an execution that contains a data race, and $\mathsf{race\text{-}free}(P)$ to indicate that it does not have an execution that has a data race.

3.3.3 Refinement

We are now in a position to define thread refinement.

Definition 9 (contextual thread refinement)**.** We say that T' is a refinement of T, written $\mathsf{ref}(T', T)$, if the following holds:

$$\forall\, C\colon \mathsf{race\text{-}free}(T \,\|\, C) \Rightarrow (\mathsf{race\text{-}free}(T' \,\|\, C) \wedge \mathbb{M}(T' \,\|\, C) \subseteq \mathbb{M}(T \,\|\, C))$$

The definition says that for all contexts C with which T is data-race-free, T' is also data-race-free, and the set of initial/final shared state pairs of $T' \,\|\, C$ is a subset of the set of initial/final shared state pairs of $T \,\|\, C$.

Two basic properties that we would expect of a refinement theory are reflexivity and transitivity. According to the above definition, the predicate $\mathsf{ref}(T', T)$ is clearly reflexive. Below we show that it is also transitive. The predicate $\mathsf{ref}(T', T)$ thus forms a preorder on the set of all threads.

Theorem 10 (transitivity)**.** Let T'', T', and T be threads, and let $\mathsf{ref}(T'', T')$ and $\mathsf{ref}(T', T)$. Then $\mathsf{ref}(T'', T)$.

Proof. Let C be an arbitrary context. Let $\mathsf{ref}_{ctx}(T', T, C) \Leftrightarrow (\mathsf{race\text{-}free}(T \,\|\, C) \Rightarrow (\mathsf{race\text{-}free}(T' \,\|\, C) \wedge \mathbb{M}(T' \,\|\, C) \subseteq \mathbb{M}(T \,\|\, C)))$. Then by the definition of $\mathsf{ref}()$ and $\mathsf{ref}(T'', T')$ and $\mathsf{ref}(T', T)$ it holds that $\mathsf{ref}_{ctx}(T'', T', C)$ and $\mathsf{ref}_{ctx}(T', T, C)$. We need to show that then also $\mathsf{ref}_{ctx}(T'', T, C)$ holds.

(1) Let $\neg\mathsf{race\text{-}free}(T \,\|\, C)$. Then $\mathsf{ref}_{ctx}(T'', T, C)$ as the premise of the implication is false.

(2) Let $\mathsf{race\text{-}free}(T \,\|\, C)$. Then by $\mathsf{ref}_{ctx}(T', T, C)$ it follows that $\mathsf{race\text{-}free}(T' \,\|\, C)$ and $\mathbb{M}(T' \,\|\, C) \subseteq \mathbb{M}(T \,\|\, C)$. Then by $\mathsf{ref}_{ctx}(T'', T', C)$ and the definition of $\subseteq$ it

follows that $\mathsf{race\text{-}free}(T'' \,\|\, C)$ and $\mathbb{M}(T'' \,\|\, C) \subseteq \mathbb{M}(T \,\|\, C)$. Therefore, it holds that $\mathsf{ref}_{ctx}(T'', T, C)$. □

The above definition of refinement (Definition 9) is not directly suited for automated refinement checking, as it would require implementing the ∀ quantifier (and hence enumerating all possible contexts C). We thus develop in the following our transition-based refinement condition that implies $\mathsf{ref}(T', T)$, and which is more amenable to automated and manual reasoning about refinement.

3.3.4 Transition Traces

We next define the transition relation $\rightarrow_L$, which is more coarse-grained than $\rightarrow$. It will form the basis of the refinement specification.

Definition 11 ($\rightarrow_L$)**.** $(P, pc, s_l, s, ls) \xrightarrow{l,(R_a,W_a),(R_b,W_b)}_L (P, pc', s_l', s', ls')$ if and only if there exists an execution fragment $e = (c_0, c_1), (c_1, c_2), \ldots, (c_{k-1}, c_k), \ldots, (c_{n-1}, c_n)$ such that $\mathsf{th}(0) = \mathsf{th}(1) = \ldots = \mathsf{th}(n-1) = T$ for some thread T of P, $\mathsf{lock}(0)$, $\mathsf{mem}(1), \ldots, \mathsf{mem}(k-2), \mathsf{unlock}(k-1)$, $\mathsf{mem}(k), \ldots, \mathsf{mem}(n-1)$, either $\mathsf{lock}(T, pc(c_n))$ or $\mathsf{term}(T, pc(c_n))$, $\mathsf{loc}(0) = l$, and $pc(c_0) = pc, s_l(c_0) = s_l, s(c_0) = s, ls(c_0) = ls$ and $pc(c_n) = pc', s_l(c_n) = s_l', s(c_n) = s', ls(c_n) = ls'$. The set R_a (resp. W_a) is the set of memory locations read (resp. written) by steps 1 to $k-2$. The set R_b (resp. W_b) is the set of memory locations read (resp. written) by steps k to $n-1$.

We also use the abbreviations $A_a = R_a \cup W_a$ and $A_b = R_b \cup W_b$. The relation $\rightarrow_L$ embodies uninterrupted execution of a thread T of P from a `lock()` to the next `lock()` (or the thread terminates). Since we have excluded nested locks, this means the thread executes exactly one `unlock()` in between. For example, in Figure 3.2b (left trace), the execution from the first `lock()` in line 1 to immediately before the second `lock()` in line 6 corresponds to a transition of $\rightarrow_L$. If we assume the thread starts in a state with all variables being 0, we have $s = \{x \mapsto 0, y \mapsto 0, z \mapsto 0\}$ and $s' = \{x \mapsto 2, y \mapsto 0, z \mapsto 0\}$. The corresponding access sets are $R_a = \{\}, W_a = \{x\}$, and $R_b = \{y\}, W_b = \{\}$.

We now define the semantics of a single thread T as the set of its *transition traces*. A transition trace is a finite sequence of the form $t = (l_0, s_0, R_0, W_0)$ $(R_1, W_1, s_1)(l_2, s_2, R_2, W_2)(R_3, W_3, s_3) \ldots (l_{n-1}, s_{n-1}, R_{n-1}, W_{n-1})(R_n, W_n, s_n)\mathit{term}$. Two items i, $i+1$ (with i being even) of a transition trace belong together. The

item i corresponds to execution starting in state s_i at a `lock()` and executing up to the next `unlock()`, with the thread reading the variables in R_i and writing the variables in W_i. The subsequent item $i+1$ corresponds to execution continuing at the `unlock()` and executing until the next `lock()` reaching state s_{i+1}, with the thread reading the variables in R_{i+1} and writing the variables in W_{i+1}. The final indicator *term* can be **true** or **false**. It indicates whether the transition trace corresponds to a terminated thread execution. We define the predicate $\mathsf{term}(t)$ to be true when t corresponds to a terminated thread execution.

We denote by $|t|$ the length of a transition trace (counting the number of tuples and not including the indicator *term*). Thus, for example, for the trace t in the previous paragraph we have $|t| = n + 1$. We denote by $t[0\!:\!i]$ the slice of a trace containing the tuples from index 0 to index i (exclusive). The final indicator is adjusted to **false** if $i < n+1$ since in this case the resulting transition trace cannot correspond to a terminated thread execution anymore.

The formal definition of the transition trace set $\mathbb{S}(T)$ is given in Figure 3.7. We denote by even_n^+ the set of all even indices i such that $1 \leq i \leq n$. Intuitively, the transition trace set of a thread T embodies all interactions it could potentially have with a context C for which $\mathsf{race\text{-}free}(T \,\|\, C)$. A thread might observe writes by the context at a `lock()` operation. This is modeled in $\mathbb{S}(T)$ by the state changing between transitions. For example, the target state s_1 of the first transition is different from the source state s_2 of the second transition. The last line of the definition of $\mathbb{S}(T)$ constrains how the state may change between transitions. It says that those memory locations that the thread T accesses in an execution portion from an `unlock()` to the next `lock()` (i.e., those in A_{i-1}) do not change at this `lock()`. The reason for this is that if those memory locations would be written by the context then there would be a data race. But since $\mathbb{S}(T)$ only models the potential interactions with race-free contexts, the last line excludes those transition traces.

Previously we stated that we are interested in the states of a thread at `lock()` and `unlock()` operations, but $\mathbb{S}(T)$ embodies transitions from a `lock()` to the next `lock()`. However, since we know the state at a `lock()`, and we know the set of memory locations W_i written between the previous `unlock()` and that `lock()`, we know the state of the memory locations $M - W_i$ at the `unlock()`. This is sufficient for phrasing the refinement in the following.

$$
\begin{aligned}
\mathbb{S}(T) = \{ & (l_0, s_0, R_0, W_0)(R_1, W_1, s_1)(l_2, s_2, R_2, W_2)(R_3, W_3, s_3) \ldots (R_n, W_n, s_n)\ \mathit{term} \mid \\
& \exists pc_0, pc_2, \ldots, pc_{n+1}, s_{l,0}, s_{l,2}, \ldots, s_{l,n+1} \colon \\
& (T, pc_0, s_{l,0}, s_0, \emptyset) \xrightarrow{l_0,(R_0,W_0),(R_1,W_1)}_L (T, pc_2, s_{l,2}, s_1, \emptyset) \wedge \\
& (T, pc_2, s_{l,2}, s_2, \emptyset) \xrightarrow{l_2,(R_2,W_2),(R_3,W_3)}_L (T, pc_4, s_{l,4}, s_3, \emptyset) \wedge \\
& \ldots \\
& (T, pc_{n-1}, s_{l,n-1}, s_{n-1}, \emptyset) \xrightarrow{l_{n-1},(R_{n-1},W_{n-1}),(R_n,W_n)}_L (T, pc_{n+1}, s_{l,n+1}, s_n, \emptyset) \wedge \\
& pc_0 = pc_s \wedge s_{l,0} = s_{l,s} \wedge \\
& \mathit{term} \Leftrightarrow pc_{n+1} = pc_f \wedge \\
& \forall i \in \mathsf{even}_n^+ \colon \forall x \in A_{i-1} \colon s_{i-1}(x) = s_i(x) \}
\end{aligned}
$$

Fig. 3.7: Definition of the transition trace set of a thread

3.3.5 Refinement Condition

In this section we will phrase our refinement condition. We will first define the predicate $\mathsf{match}_a(t', t)$, then extend it to the predicate $\mathsf{match}_b(t', t)$, and finally define the refinement condition $\mathsf{check}(T', T)$.

The $\mathsf{match}_a(t', t)$ predicate indicates whether a transition trace $t' \in \mathbb{S}(T')$ matches a transition trace $t \in \mathbb{S}(T)$. The formal definition is shown in Figure 3.8. Primed symbols refer to components of t', and unprimed symbols refer to components of t. We denote by even_n (odd_n) the set of all even (odd) indices i such that $0 \leq i \leq n$. Intuitively, the constraints in lines 3–6 specify that t' must not allow more data races than t. The constraints in lines 3–4 correspond to an execution portion from a `lock()` to the next `unlock()`, and lines 5–6 correspond to an execution portion from the `unlock()` to the next `lock()`. Since we have $R'_i \subseteq A_{i-1} \cup A_i \cup A_{i+1}$ and $W'_i \subseteq W_{i-1} \cup W_i \cup W_{i+1}$, the specification allows an access in t to move into a critical section in t' (we further investigate this in Section 3.4). The constraint in line 7 specifies that t' and t receive the same new values at `lock()` operations (modelling writes by the context). The constraint in line 8 specifies that those memory locations that are accessed by t but not by t' do not receive new values in t'. The constraint at line 9 specifies that the

$\mathsf{match}_a(t', t) \Leftrightarrow$

1 $|t'| = |t|$
2 $\textbf{let } n = |t| - 1 \textbf{ in}$

\# race constraints
3 $\forall i \in \mathsf{even}_n \colon R'_i \subseteq (A_{i-1} \cup A_i \cup A_{i+1})$
4 $\forall i \in \mathsf{even}_n \colon W'_i \subseteq (W_{i-1} \cup W_i \cup W_{i+1})$
5 $\forall i \in \mathsf{odd}_n \colon R'_i \subseteq A_i$
6 $\forall i \in \mathsf{odd}_n \colon W'_i \subseteq W_i$

\# state at locks constraints
7 $\forall i \in \mathsf{even}_n \colon \forall x \in M - A_{i-1} \colon s'_i(x) = s_i(x)$
8 $\forall i \in \mathsf{even}_n \colon \forall x \in A_{i-1} - A'_{i-1} \colon s'_{i-1}(x) = s'_i(x)$

\# state at unlocks constraints
9 $\forall i \in \mathsf{odd}_n \colon \forall x \in M - W_i \colon s'_i(x) = s_i(x)$

\# same locks constraint
10 $\forall i \in \mathsf{even}_n \colon l'_i = l_i$

Fig. 3.8: Definition of matching transition traces

values written by t' and t before `unlock()` operations must be the same. The last constraint specifies that t' and t perform the same sequence of lock operations.

We next define the $\mathsf{match}_b(t', t)$ predicate.

Definition 12.

$$\begin{aligned}\mathsf{match}_b(t', t) \Leftrightarrow\ & \textbf{let } n = \mathsf{min}(|t'|, |t|) - 1 \textbf{ in} \\ & ((\mathsf{term}(t') \Rightarrow \mathsf{term}(t)) \wedge \mathsf{match}_a(t', t)) \vee \\ & \exists i \in \mathsf{even}_n^+ \colon \mathsf{match}_a(t'[0\colon i], t[0\colon i]) \wedge \\ & \qquad \exists x \in (A_{i-1} - A'_{i-1}) \colon s'_{i-1}(x) \neq s'_i(x)\end{aligned}$$

The definition says that either t' and t match and if t' corresponds to a terminated thread execution then so does t, or there are prefixes that match, and at the subsequent `lock()` a memory location in t' changes that is accessed by t but not

by t' ($x \in A_{i-1} - A'_{i-1}$). Thus, a context that could implement the change of the memory location that t' observes would have a data race with t. Since when t is involved in a data race we have undefined behaviour, any behaviour of t' is allowed. Hence, we consider the traces t' and t matched.

We can now define our refinement condition $\mathsf{check}(T', T)$. In Section 3.3.7 we later show that it implies the refinement specification $\mathsf{ref}(T', T)$ of Definition 9.

Definition 13 (check)**.**

$$\mathsf{check}(T', T) \Leftrightarrow \forall t' \in \mathbb{S}(T') \colon \exists t \in \mathbb{S}(T) \colon \mathsf{match}_b(t', t)$$

3.3.6 Transitivity and Reflexivity

By inspecting Definition 13 we can see that it is reflexive. We show below via the Lemmas 14–16 and Theorem 17 that it is also transitive.

Lemma 14. Let $n = |t| - 1$ and $i \in \mathsf{even}^+_n$. Then:

(a) $\mathsf{match}_a(t', t) \Rightarrow \mathsf{match}_a(t'[0:i], t[0:i])$.
(b) $\mathsf{match}_b(t', t) \Rightarrow \mathsf{match}_b(t'[0:i], t[0:i])$.

The lemma directly follows from the definition of $\mathsf{match}_a(t', t)$ in Figure 3.8, and Definition 12.

Lemma 15. Let $\mathsf{match}_a(t'', t')$, $\mathsf{match}_a(t', t)$, $\mathsf{term}(t'') \Rightarrow \mathsf{term}(t')$, and $\mathsf{term}(t') \Rightarrow \mathsf{term}(t)$. Then $\mathsf{match}_b(t'', t)$.

Proof. We show that $\mathsf{match}_b(t'', t)$ holds by showing that $\mathsf{term}(t'') \Rightarrow \mathsf{term}(t)$ and the conjuncts in lines 1, 3–6, 7, and 9–10 in the definition of $\mathsf{match}_a(t'', t)$ are true, and that if the conjunct in line 8 is false, then the second disjunct in the definition of $\mathsf{match}_b(t'', t)$ is true.

Termination: By $\mathsf{term}(t'') \Rightarrow \mathsf{term}(t')$ and $\mathsf{term}(t') \Rightarrow \mathsf{term}(t)$ it follows that $\mathsf{term}(t'') \Rightarrow \mathsf{term}(t)$.

Line 1: We have $|t''| = |t'|$ and $|t'| = |t|$ and thus also $|t''| = |t|$. In the following we set $n = |t| - 1$.

Lines 5–6: Let $i \in \mathsf{odd}_n$. Recall that $A_i = R_i \cup W_i$. Then by the transitivity of $\subseteq$ we get $R''_i \subseteq A_i$ and $W''_i \subseteq W_i$.

Lines 3–4: Let $i \in \mathsf{even}_n$. Then we get:

$$R''_i \subseteq A'_{i-1} \cup A'_i \cup A'_{i+1} \subseteq A'_{i-1} \cup (A_{i-1} \cup A_i \cup A_{i+1}) \cup A'_{i+1}$$
$$W''_i \subseteq W'_{i-1} \cup W'_i \cup W'_{i+1} \subseteq W'_{i-1} \cup (W_{i-1} \cup W_i \cup W_{i+1}) \cup W'_{i+1}$$

Since we have already shown lines 5–6 and since $i-1$ and $i+1$ are odd we have that

$$A'_{i-1} \subseteq A_{i-1} \quad A'_{i+1} \subseteq A_{i+1} \quad W'_{i-1} \subseteq W_{i-1} \quad W'_{i+1} \subseteq W_{i+1}$$

Therefore we get

$$R''_i \subseteq A_{i-1} \cup A_i \cup A_{i+1} \quad W''_i \subseteq W_{i-1} \cup W_i \cup W_{i+1}$$

Line 7: Let $i \in \mathsf{even}_n$. Since by lines 5–6 it holds that $R'_{i-1} \subseteq A_{i-1}$ and $W'_{i-1} \subseteq W_{i-1}$ we have that $A_{i-1} \supseteq A'_{i-1}$. Hence it follows that $\forall x \in M - A_{i-1} \colon s''(x) = s(x)$.

Line 9: Let $i \in \mathsf{odd}_n$. Since by lines 5–6 it holds that $W_i \supseteq W'_i$ it follows that $\forall x \in M - W_i \colon s''_i(x) = s_i(x)$.

Line 10: Let $i \in \mathsf{even}_n$. Then by $l''_i = l'_i$ and $l'_i = l_i$ we get $l''_i = l_i$.

Line 8: If $\forall x \in A_{i-1} - A''_{i-1} \colon s''_{i-1}(x) = s''_i(x)$ does not hold for an i, then there is a $j \leq i$ such that $\mathsf{match}_a(t''[0\colon j], t[0\colon j])$ and $\exists x \in A_{j-1} - A''_{j-1} \colon s''_{j-1}(x) \neq s''_j(x)$. Thus, the second disjunct in the definition of $\mathsf{match}_b(t'', t)$ is satisfied.

□

Lemma 16. Let $\mathsf{match}_b(t'', t')$ and $\mathsf{match}_b(t', t)$. Then $\mathsf{match}_b(t'', t)$.

Proof. The proof works by a case distinction. For each case we show that $\mathsf{match}_b(t'', t)$ is satisfied. We make use of the following abbreviation:

$$\mathsf{match}_{b,idx}(t', t, i) \Leftrightarrow \mathsf{match}_a(t'[0\colon i], t[0\colon i]) \wedge \exists x \in A_{i-1} - A'_{i-1} \colon s'_{i-1}(x) \neq s'_i(x)$$

(1) Let i, j be even indices such that:

(a) $\mathsf{match}_{b,idx}(t'', t', i)$
(b) $\mathsf{match}_{b,idx}(t', t, j)$

Let further $l = \mathsf{min}(i, j)$. By Lemma 14a we get $\mathsf{match}_a(t''[0\colon l], t'[0\colon l])$ and $\mathsf{match}_a(t'[0\colon l], t[0\colon l])$. Then by Lemma 15 we get $\mathsf{match}_b(t''[0\colon l], t[0\colon l])$.

Now if there is a $k \in \mathsf{even}_{l-2}$ such that $\mathsf{match}_a(t''[0\colon k], t[0\colon k]) \wedge \exists x \in A_{k-1} - A''_{k-1}\colon s''_{k-1}(x) \neq s''_k(x)$, then by Definition 12 it follows that $\mathsf{match}_b(t'', t)$ and we are done.

Otherwise we have $\mathsf{match}_a(t''[0\colon l], t[0\colon l])$. We further have $A''_{l-1} \subseteq A'_{l-1} \subseteq A_{l-1}$. Now we make a case distinction over $i > j$ and $i \leq j$.

$i > j$: By (b) we have:

$\forall x \in M - A_{i-1}\colon s'_i(x) = s_i(x)$
$\forall x \in M - A_{i-1}\colon s'_{i-1}(x) = s_{i-1}(x)$

Thus, since by (a) we have $\exists x \in A'_{i-1} - A''_{i-1}\colon s''_{i-1}(x) \neq s''_i(x)$ it follows that also $\exists x \in A_{i-1} - A''_{i-1}\colon s''_{i-1}(x) \neq s''_i(x)$ holds. Thus $\mathsf{match}_b(t'', t)$ holds.

$i \leq j$: By (a) we have:

$\forall x \in M - A'_{j-1}\colon s''_j(x) = s'_j(x)$
$\forall x \in M - A'_{j-1}\colon s''_{j-1}(x) = s'_{j-1}(x)$

Thus, since by (b) we have $\exists x \in A_{j-1} - A'_{j-1}\colon s'_{j-1}(x) \neq s'_j(x)$ it follows that also $\exists x \in A_{j-1} - A''_{j-1}\colon s''_{j-1}(x) \neq s''_j(x)$ holds. Thus $\mathsf{match}_b(t'', t)$ holds.

(2) Let j be an even index such that:

(a) $\mathsf{match}_a(t'', t') \wedge \mathsf{term}(t'') \Rightarrow \mathsf{term}(t')$
(b) $\mathsf{match}_{b,idx}(t', t, j)$

This case is analogous to (1) with $i > j$.

(3) Let i be an even index such that:

(a) $\mathsf{match}_{b,idx}(t'', t', i)$
(b) $\mathsf{match}_a(t', t) \wedge \mathsf{term}(t') \Rightarrow \mathsf{term}(t)$

This case is analogous to (1) with $i < j$.

(4)

(a) $\mathsf{match}_a(t'', t') \wedge \mathsf{term}(t'') \Rightarrow \mathsf{term}(t')$
(b) $\mathsf{match}_a(t', t) \wedge \mathsf{term}(t') \Rightarrow \mathsf{term}(t)$

By Lemma 15 it follows that $\mathsf{match}_b(t'', t)$ holds.

□

We get the following theorem:

Theorem 17 (transitivity)**.** Let $\mathsf{check}(T'', T')$ and $\mathsf{check}(T', T)$. Then $\mathsf{check}(T'', T)$.

The theorem directly follows from Definition 13 and Lemma 16.

3.3.7 Soundness

In this section, we prove the soundness of our refinement condition $\mathsf{check}(T', T)$. We first prove two lemmas (Lemma 18 and Lemma 19), which we will then use in the proof of the soundness theorem (Theorem 20).

Lemma 18 (coarse-grained interleaving)**.** Let e (with $|e| = n$) be an execution prefix of P with $\neg\mathsf{hb\text{-}race}(e)$ and $\mathsf{final}(e) = s$. Then there is an execution prefix e' of P with $\neg\mathsf{hb\text{-}race}(e')$ and $\mathsf{final}(e') = s$, such that execution portions from a `lock()` to the next `lock()` of a thread are not interleaved with other threads. Formally:

$$\forall\, 0 \le i < n\colon \mathsf{lock}(i) \Rightarrow \exists j > i\colon (\mathsf{lock}(\mathsf{th}(i), pc(\mathsf{tgt}(j))) \vee \mathsf{term}(\mathsf{th}(i), pc(\mathsf{tgt}(j))) \wedge \forall i < k < j\colon \mathsf{th}(k) = \mathsf{th}(i))$$

Proof. Let i be a step of e with $\neg\mathsf{lock}(i)$. Let j be a step of e with $j < i$, $\mathsf{th}(j) = \mathsf{th}(i)$, such that $\forall j < k < i\colon \mathsf{th}(k) \neq \mathsf{th}(i)$. It holds that $\forall j < k < i\colon (k, i) \notin \mathsf{hb}$. Therefore, $\forall j < k < i\colon \neg\mathsf{conflict}(k, i)$. Thus, step i can be moved over the steps k and right after j without changing the values read by any read operation. We thus get a new execution prefix e' with $\mathsf{final}(e') = \mathsf{final}(e)$. Moreover, moving step i cannot introduce a data race as it is moved "upwards" only and not past a `lock()` operation.

The repeated application of picking a step i with $\neg\mathsf{lock}(i)$ and moving it right after the previous step of the same thread finally yields an execution prefix in which portions from a `lock()` to the next `lock()` are not interleaved with other threads. □

Lemma 19 (race refinement)**.** $\mathsf{check}(T', T) \Rightarrow \forall\, C\colon (\mathsf{race}(T' \parallel C) \Rightarrow \mathsf{race}(T \parallel C))$

Proof. Let $\mathsf{race}(T' \parallel C)$. Then there is an execution that contains a data race. A data race can either be between two threads in C, or between T' and a thread C' in C. We assume the latter case. We further assume that data races are between two writes on variable x. The other cases are analogous.

Since $\mathsf{race}(T' \,\|\, C)$, there is an execution e such that thread T' and thread C' of C are involved in an adjacent access data race. Further, there is an (hb and adjacent access) race-free prefix e' of e such that the next operation to be executed by each thread is a `lock()`, and the next execution portions from a `lock()` to the next `lock()` of both T' and C' are those involved in a data race.

Since the prefix e' is data-race-free, by Lemma 18 there is an execution prefix e'' which ends in the same state as e', and for which the execution portions of a thread from a `lock()` to the next `lock()` are not interleaved. Moreover, the execution of T' and C' can be continued from e'' such that they are involved in an adjacent access data race. We denote this continuation of e'' by e'''.

The sequence of execution portions of T' in e'' corresponds to an element $t' \in \mathbb{S}(T')$. The next execution portion of T' from a `lock()` to the next `lock()` after e'' is the one involved in the data race. Thus, t' can be continued to $u' = t'(l'_k, s'_k, R'_k, W'_k)(R'_{k+1}, W'_{k+1}, s'_{k+1})$[1] such that $u' \in \mathbb{S}(T')$ and $x \in W'_k \cup W'_{k+1}$ (recall that we assumed that data races are between two writes on variable x).

Then, by the definition of $\mathsf{check}(T', T)$, there is a $u \in \mathbb{S}(T)$ such that either (1) $(\mathsf{term}(u') \Rightarrow \mathsf{term}(u)) \wedge \mathsf{match}_a(u', u)$ or (2) $\exists i \in \mathsf{even}_n^+ \colon \mathsf{match}(u'[0\colon i], u[0\colon i]) \wedge \exists x \in (A_{i-1} - A'_{i-1}) \colon s'_{i-1}(x) \neq s'_i(x)$.

(1) Let u be of the form $t(R_{k-1}, W_{k-1}, s_{k-1})(l_k, s_k, R_k, W_k)(R_{k+1}, W_{k+1}, s_{k+1})$. Since $t(R_{k-1}, W_{k-1}, s_{k-1})$ describes the same state transitions as t', the steps of T' in e'' can be replaced by steps of T. Then if this new execution prefix q contains a data race we are done (as we have $\mathsf{race}(T \,\|\, C)$). We need to show that if the new execution prefix q does not contain a data race, then the next steps taken by T and C' give rise to a data race.

By the definition of $\mathsf{match}(u', u)$, we have $W'_k \subseteq W_{k-1} \cup W_k \cup W_{k+1}$ and $W'_{k+1} \subseteq W_{k+1}$. In e''', the access of T' involved in the adjacent access data race might either occur (a) between a `lock()` and the subsequent `unlock()` (i.e., $x \in W'_k$), or (b) between an `unlock()` and the subsequent `lock()` (i.e., $x \in W'_{k+1}$).

(a) In this case the portion of e''' containing the data race has the following shape (portions denoted by an ellipsis $(\ldots)$ contain only memory accesses

[1] We leave the termination indicator *term* off for the remainder of this proof as it is irrelevant when reasoning about data races.

and no lock operations):

$$\ldots, T'\colon \texttt{lock(l)}, \ldots, T'\colon Wx, C'\colon Wx, \ldots, T'\colon \texttt{unlock(l)}, \ldots$$

It further holds that $W'_k \subseteq W_{k-1} \cup W_k \cup W_{k+1}$. Thus, when continuing to execute T from q a write to x might occur either (i) before the next `lock()` ($x \in W_{k-1}$), (ii) between the next `lock()` and `unlock()` ($x \in W_k$), or (iii) after the next `unlock()` ($x \in W_{k+1}$).

(i): In this case there is a continuation q' of q that contains an execution fragment of the following form:

$$\ldots, T\colon Wx, \ldots, T\colon \texttt{lock(l)}, \ldots, C'\colon Wx, \ldots$$

By the definition of the happens-before relation (hb), we see that there is no hb edge between the steps "$T\colon Wx$" and "$C'\colon Wx$". Therefore, there is a data race between the two steps.

(ii): We have the following execution portion:

$$\ldots, T\colon \texttt{lock(l)}, \ldots, T\colon Wx, \ldots, C'\colon Wx, \ldots$$

There is no hb edge between "$T\colon Wx$" and "$C'\colon Wx$", and thus there is a data race.

(iii): We have the following execution portion:

$$\ldots, C'\colon Wx, \ldots, T\colon \texttt{unlock(l)}, \ldots, T\colon Wx, \ldots$$

There is no hb edge between "$C'\colon Wx$" and "$T\colon Wx$", and thus there is a data race.

(b) In this case the portion of e''' containing the data race has the following shape:

$\ldots, T'\colon$ `unlock(l)`$, \ldots, T'\colon Wx, C'\colon x, \ldots$

It holds that $W'_{k+1} \subseteq W_{k+1}$. Thus, when continuing to execute T from q a write to x occurs after the `unlock(l)` just the same. In this case there is a continuation q' of q that contains an execution fragment of the following form:

$\ldots, T\colon$ `unlock(l)`$, \ldots, T\colon Wx, \ldots, C'\colon Wx, \ldots$

There is no hb edge between "$T\colon Wx$" and "$C'\colon Wx$", and thus there is a data race.

(2) Since $\mathsf{match}_a(u'[0\colon i], u[0\colon i])$, the first i state transitions described by u are the same as those described by u'. Thus, we can replace the first i execution portions of T' in e'' by execution portions of T. The last execution portion of T accesses a memory location x that was not accessed by the corresponding execution portion of T' (since we have $\exists x \in A_{i-1} - A'_{i-1}$). Moreover, by $s'_{i-1}(x) \neq s'_i(x)$ it follows that this memory location is written by the context C. Thus, we have $\mathsf{race}(T \parallel C)$.

□

The following theorem establishes the soundness of our refinement specification $\mathsf{check}(T', T)$.

Theorem 20 (soundness)**.** $\mathsf{check}(T', T) \Rightarrow \mathsf{ref}(T', T)$

Proof. Let C be an arbitrary context C such that $\mathsf{race\text{-}free}(T \parallel C)$. Let further (s, s') in $\mathbb{M}(T' \parallel C)$. Thus, there is an execution e of $T' \parallel C$ that starts in state s and ends in state s'. By Lemma 19, $\mathsf{race\text{-}free}(T' \parallel C)$. Thus, by Lemma 18, there is an execution e' for which portions from a `lock()` to the next `lock()` of a thread are not interleaved with other threads. The sequence of those execution portions of T' corresponds to an element of $t' \in \mathbb{S}(T')$. Then, by the definition of $\mathsf{check}(T', T)$, there is an element $t \in \mathbb{S}(T)$ such that either (a) $(\mathsf{term}(t') \Rightarrow \mathsf{term}(t)) \wedge \mathsf{match}_a(t', t)$, or (b) $\exists i \in \mathsf{even}^+_n\colon \mathsf{match}_a(t'[0\colon i], t[0\colon i]) \wedge \exists x \in (A_{i-1} - A'_{i-1})\colon s'_{i-1}(x) \neq s'_i(x)$.

(a) Since execution e' terminates, we have $\mathsf{term}(t')$. It thus follows that $\mathsf{term}(t)$. Moreover, t' and t embody the same state transitions. This is ensured by constraints 7 and 9 of the definition of $\mathsf{match}_a(t', t)$. Constraint 7 specifies that the starting states of a transition match, and constraint 9 specifies that the resulting states of a transition match. Taking a closer look at constraints 7 and 9 reveals that the corresponding states of t' and t do not need to be completely equal (only those memory locations in $M - A_{i-1}$ resp. $M - W_i$ need to have the same value). The reason for this is that if a thread would observe those memory locations it would give rise to a data race. Since we have both $\mathsf{race\text{-}free}(T' \parallel C)$ and $\mathsf{race\text{-}free}(T \parallel C)$, it follows that the values of the memory locations A_{i-1} resp. W_i can be arbitrary. Therefore, T can make the same state transitions as T'. Thus, we can replace the steps of T' in e' by steps of T, and get a valid execution e'' of $T \parallel C$ ending in the same state. Therefore, $(s, s') \in \mathbb{M}(T \parallel C)$.

(b) Since $\mathsf{match}_a(t'[0\colon i], t[0\colon i])$, the first i state transitions of t are the same as those of t'. Thus, we can replace the first i execution portions of T' in e' by execution portions of T. The last execution portion of T accesses a memory location x that was not accessed by the corresponding execution portion of T' (since we have $\exists x \in A_{i-1} - A'_{i-1}$). Moreover, by $s'_{i-1}(x) \neq s'_i(x)$ it follows that this memory location is written by the context C. Thus, we have $\mathsf{race}(T \parallel C)$, which contradicts the premise $\mathsf{race\text{-}free}(T \parallel C)$.

□

3.4 Refinement Examples and Supported Optimisations

In this section, we discuss in which cases our theory allows us to show refinement between threads. We discuss which code transformations can be applied to a thread T to obtain a thread T' such that T' is a refinement of T. Such a transformation may be performed manually or by an optimising compiler. We further give several examples of threads between which the refinement relation $\mathsf{ref}(T', T)$ (see Definition 9) holds and investigate whether our refinement condition $\mathsf{check}(T', T)$ (see Definition 13) allows us to show refinement.

By inspecting the definition of $\mathsf{match}_a(t', t)$ we see that it requires that t' and t perform similar state transitions between lock operations, and that the sets of memory locations accessed between lock operations of t' must be subsets of corresponding sets of memory locations accessed by t. Together with the definitions of $\mathsf{match}_b(t', t)$ and $\mathsf{check}(T', T)$, this implies that if an optimisation only performs transformations that do not change the state transitions between lock operations, and does not introduce accesses to new memory locations, then the optimised thread T' will be a refinement of the original thread T. This includes all the transformations shown to be sound by Boehm [22] and Morisset et al. [79] (considering programs using `lock()` and `unlock()` for synchronisation).

Our theory also allows the reordering of shared memory accesses into and out of critical sections (under certain circumstances). The former are called roach motel reorderings and have been studied for example in the context of the Java memory model (see, e.g., Sevcik and Aspinall [105]). The latter have not been previously described in the literature. In analogy to the former we term them *inverse roach motel reorderings*. Our theory enables the proof of both transformations.

3.4.1 Roach Motel Reorderings

Consider Figure 3.9. Both x and y are shared variables. Figure 3.9a shows the original thread T, and Figure 3.9b a correctly transformed version T'. Two transformations have been applied to obtain T'. The statement `y = 2` has been moved into the critical section ("roach motel reordering") and `y = 3` has been removed. Those two transformations cannot introduce data races.

The transition trace sets of the threads are shown in Figure 3.9d. Let t' be a transition trace of T' starting in some initial state s_{init}. Then there is a transition trace t of T starting also in s_{init}. The state s_{init} corresponds to the state at the first `lock(l)` for both threads. At termination they are in states $s' = \{x \mapsto 1, y \mapsto 3\}$ resp. $s = \{x \mapsto 1, y \mapsto 2\}$. The access sets of the two transition traces are $R'_0 = R'_1 = R_0 = R_1 = \{\}$ (we ignore the read sets in the following as they are empty), and $W'_0 = W_0 = \{x, y\}, W'_1 = \{\}, W_1 = \{y\}$.

At termination, according to the definition of $\mathsf{match}_a(t', t)$, the constraint $\forall x \in M - W_1 \colon s'(x) = s(x)$ needs to be satisfied. This is the case as the variable y for which s' and s differ is in W_1. Moreover, for $\mathsf{match}_a(t', t)$ to be satisfied, for the write sets the following must hold: $W'_0 \subseteq W_0 \cup W_1$ and $W'_1 \subseteq W_1$. This also holds.

```
1 lock(l);
2 x = 1;
3 y = 1;
4 unlock(l);
5 y = 2;
6 y = 3;
```

(a) Original T

```
1 lock(l);
2 x = 1;
3 y = 1;
4 y = 2;
5 unlock(l);
```

(b) Transformed T'

```
1 lock(l);
2 x = 1;
3 unlock(l);
4 y = 1;
5 y = 2;
```

(c) Transformed T''

$$\mathbb{S}(T) = \{\ (l, s, \emptyset, \{x, y\})(\emptyset, \{y\}, s[x/1][y/3])\ \mathbf{true} \mid s \in S\}$$
$$\mathbb{S}(T') = \{\ (l, s, \emptyset, \{x, y\})(\emptyset, \emptyset, s[x/1][y/2])\ \mathbf{true} \mid s \in S\}$$
$$\mathbb{S}(T'') = \{\ (l, s, \emptyset, \{x\})(\emptyset, \{y\}, s[x/1][y/2])\ \mathbf{true} \mid s \in S\}$$

(d) Transition traces of T, T', T''

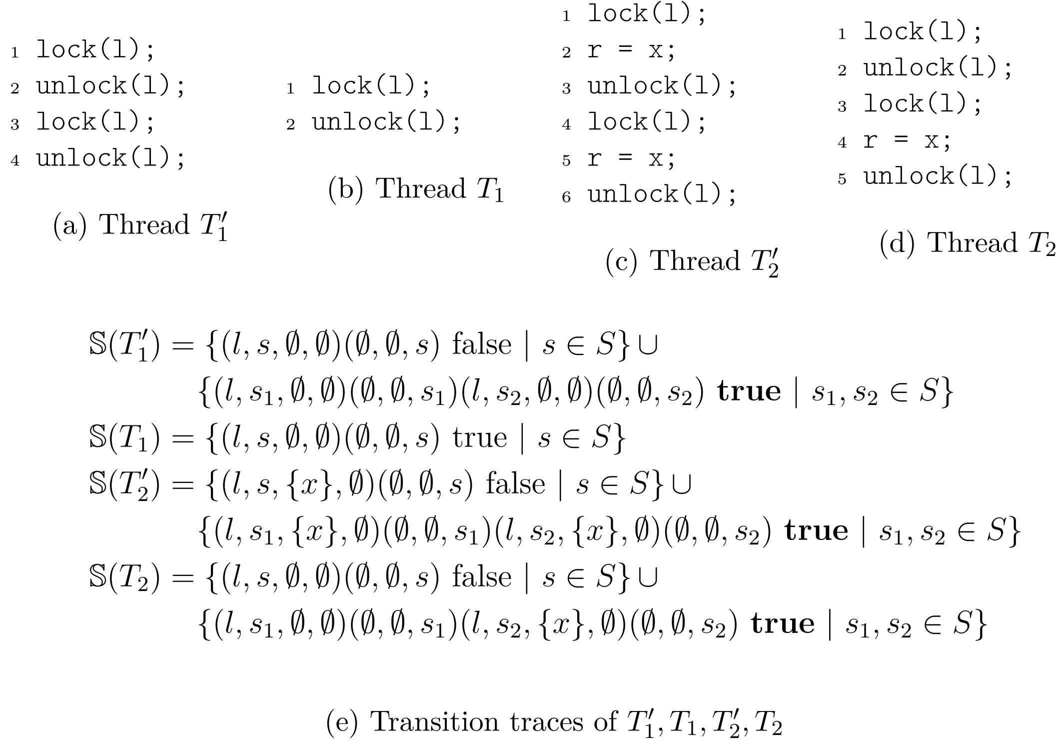
Fig. 3.9: Original, roach motel reordering, inverse roach motel reordering

Hence, $\mathsf{match}_a(t', t)$ holds. Both transition trace sets only contain terminated transition traces, hence $\mathsf{term}(t') \Rightarrow \mathsf{term}(t)$ also holds. Consequently, we also have $\mathsf{match}_b(t', t)$. Thus, we get $\mathsf{check}(T', T)$ which implies $\mathsf{ref}(T', T)$ according to Theorem 20. T' is thus a correctly transformed version of T.

3.4.2 Inverse Roach Motel Reorderings

Consider now the example in Figure 3.9c which again shows a correctly optimised version T'' of the thread T. Two transformations have been applied to obtain T'. The statement `y = 1` has been moved out of the critical section ("inverse roach motel reordering") and `y = 3` has been removed. In order to get defined behaviour of $T \,\|\, C$, the context C must in particular avoid data races with $\mathtt{y} = 2$ and $\mathtt{y} = 3$. But this implies that the context cannot observe the write $\mathtt{y} = 1$, for if it could, there would be a data race with $\mathtt{y} = 2$ and $\mathtt{y} = 3$. Moreover, moving $\mathtt{y} = 1$ downwards out of the critical section cannot introduce data races, as a write to y already occurs in this section. Consequently, $\mathtt{y} = 1$ can be moved downwards out of the critical section.

We can use a similar argument as in the previous section to show within our theory that T'' is a correctly optimised version of T. Let t'', t be again two transition traces starting in the same initial state s_{init}. Upon termination they are



(a) Thread T_1'

(b) Thread T_1

(c) Thread T_2'

(d) Thread T_2

(e) Transition traces of T_1', T_1, T_2', T_2

Fig. 3.10: Pairs of contextually equivalent threads

in states $s'' = \{x \mapsto 1, y \mapsto 2\}$ resp. $s = \{x \mapsto 1, y \mapsto 3\}$. Again the constraints $\forall x \in M - W_1 \colon s''(x) = s(x)$, and $W_0'' \subseteq W_0 \cup W_1$ and $W_1'' \subseteq W_1$ are satisfied. Thus, $\mathsf{match}_a(t', t)$ is satisfied. Both transition trace sets only contain terminated transition traces, hence $\mathsf{term}(t'') \Rightarrow \mathsf{term}(t)$ also holds. We thus can conclude that $\mathsf{match}_b(t', t)$, $\mathsf{check}(T', T)$, and finally $\mathsf{ref}(T', T)$ hold.

3.4.3 Counterexamples

In this section, we give two examples of pairs of threads (see Figure 3.10) between which the refinement relation $\mathsf{ref}(T', T)$ holds but for which our refinement condition $\mathsf{check}(T', T)$ is not sufficient to show refinement. The threads T_1', T_1, and T_2', T_2 are in fact contextually equivalent, i.e., $\mathsf{ref}(T_1', T_1)$, $\mathsf{ref}(T_1, T_1')$, and $\mathsf{ref}(T_2', T_2)$, $\mathsf{ref}(T_2, T_2')$ hold.

Counterexample 1

As the first example we consider the threads T_1', T_1 in Figures 3.10a and 3.10b. We first show that $\mathsf{ref}(T_1', T_1)$ indeed holds and then that our refinement condition $\mathsf{check}(T_1', T_1)$ does not hold.

Let C be an arbitrary context and let e' be an execution of $T_1' \parallel C$ with $\mathsf{initial}(e') = s$. Then by the definition of the transition relation $\rightarrow$ (Figure 3.5), and the definition of executions (Definition 1), there is an execution e of $T_1 \parallel C$ with $\mathsf{initial}(e) = s$ and which consists of the same interleaving of steps as e', but with the steps corresponding to the second `lock(l)` and `unlock(l)` operations of T_1' (lines 3, 4) left off. Hence, if e' contains an adjacent access data race, then e also contains an adjacent access data race. We thus have (1) $\mathsf{race}(T_1' \parallel C) \Rightarrow \mathsf{race}(T_1 \parallel C)$. Moreover, $\mathsf{final}(e') = \mathsf{final}(e)$. Since we have chosen e' arbitrarily it thus follows that (2) $\mathbb{M}(T_1' \parallel C) \subseteq \mathbb{M}(T_1 \parallel C)$. Therefore, by combining (1) and (2) we get $\mathsf{ref}(T_1', T_1)$.

We next show that $\mathsf{check}(T_1', T_1)$ does not hold. For this, we need to show that there is a transition trace $t' \in \mathbb{S}(T_1')$, such that for all transition traces $t \in \mathbb{S}(T_1)$, $\mathsf{match}_b(t', t)$ does not hold. To show $\neg\mathsf{match}_b(t', t)$, we need to show that both disjuncts in the definition of $\mathsf{match}_b(t', t)$ (Definition 12) are false.

Let $t' = (l, s_1, \emptyset, \emptyset)(\emptyset, \emptyset, s_1)(l, s_1, \emptyset, \emptyset)(\emptyset, \emptyset, s_1)\,\mathbf{true}$ be a transition trace of T_1'. Then for all $t \in \mathbb{S}(t)$, $|t'| \neq |t|$. Therefore, $\mathsf{match}_a(t', t)$ does not hold (see line 1 in Figure 3.8) and hence the first disjunct in the definition of $\mathsf{match}_b(t', t)$ is false. Moreover, the sets A_i', A_i that are part of the transition traces t', t are all empty as the two threads do not access any shared variables. Therefore, $\exists x \in (A_i - A_i')\colon \ \ldots$ is always false and hence the second disjunct in the definition of $\mathsf{match}_b(t', t)$ is also false.

Counterexample 2

A second example is given in Figures 3.10c and 3.10d. We again first show that $\mathsf{ref}(T_2', T_2)$ holds and then show that $\mathsf{check}(T_2', T_2)$ does not hold.

Let C be an arbitrary context and let e' be an execution of $T_2' \parallel C$ that contains an adjacent access data race. We consider the case of an execution e' where the race occurs between the step corresponding to the first assignment `r = x` (line 2) and the context C. The other cases are analogous. Then we can construct an execution e of $T_2 \parallel C$ with a similar prefix to e' up to the first lock operation of T_2',

T_2. Then when in e' the thread T_2' executes only the lock operation in line 1, in e the thread T_2 can execute the three lock operations in lines 1–3. Then $T_2' \,\|\, C$ and $T_2 \,\|\, C$ can continue executing similar steps up to the step corresponding to the first assignment `r = x` which is then involved in a data race for both e' and e. We thus have (1) $\mathsf{race}(T_2' \,\|\, C) \Rightarrow \mathsf{race}(T_2 \,\|\, C)$.

Now let e' be an execution of $T_2' \,\|\, C$ with $\mathsf{initial}(e') = s$. Then there is an execution e of $T_2 \,\|\, C$ with $\mathsf{initial}(e) = s$ and which consists of the same interleaving of steps as e', but with the steps corresponding to the first assignment `r = x` (line 2) left off. Hence, $\mathsf{final}(e') = \mathsf{final}(e)$. Since we have chosen e' arbitrarily we get (2) $\mathbb{M}(T_2' \,\|\, C) \subseteq \mathbb{M}(T_2 \,\|\, C)$. By combining (1) and (2) we get $\mathsf{ref}(T_2', T_2)$.

We next show that $\mathsf{check}(T_2', T_2)$ is false. Let $t' = (l, s, \{x\}, \emptyset)$ **false** be a transition trace of T_2'. Then for all transition traces $t \in \mathbb{S}(T)$, there does not exist an $i \in \mathsf{even}_n^+$ such that $\mathsf{match}_a(t'[0\colon i], t[0\colon i])$. The reason is that we have $R_0' = \{x\} \nsubseteq \emptyset = R_0$, which implies that $\mathsf{match}_a(t'[0\colon i], t[0\colon i])$ does not hold. Thus, both the first and the second disjunct in the definition of $\mathsf{match}_b(t', t)$ are false.

3.5 Formalisation with Nested Locks

We now adapt the formalisation from Section 3.3 to also allow nested locks. To that end, we define a new coarse-grained transition relation $\rightarrow_n$, the transition trace set $\mathbb{S}_n(T)$, the $\mathsf{match}_{n,a}(t', t)$ and $\mathsf{match}_{n,b}(t', t)$ predicates, and finally the $\mathsf{check}_n(T', T)$ predicate.

The definitions are similar to those in Section 3.3, but slightly more complex, due to the fact that now `lock()` and `unlock()` operations not only occur alternatingly, but several `lock()` or `unlock()` operations might occur in a row. In such a setting, for example, a valid optimisation is to reorder a memory access “upwards” across several `unlock()` operations, or “downwards” across several `lock()` operations. We want to be able to show refinement also in such cases.

We first introduce some additional notation. We use $\mathsf{sync}(T, pc)$ ($\Leftrightarrow \mathsf{lock}(T, pc) \vee \mathsf{unlock}(T, pc)$) to indicate that the next operation to be executed by thread T is a `lock()` or `unlock()` operation. The function $\mathsf{type}(e, i)$ returns the kind of step i of execution fragment e. This is one of lock, unlock, rd, or wr.

We first adapt our notion of well-locked executions to allow nested locks.

Definition 21 (well-locked executions)**.** We say a thread execution e of a thread T is *well-locked* if `lock()` and `unlock()` operations on the *same* lock occur alternately, it starts with a `lock()` operation, and there occur infinitely many lock operations if it is infinite. We say an execution e of a program P is well-locked if all the threads are well-locked: $\forall T \in \mathcal{T}(P)\colon \mathsf{well\text{-}locked}_n(e|_T)$.

We now define the new coarse-grained transition relation $\rightarrow_n$. It embodies uninterrupted execution from a lock operation to the next lock operation. Formally:

Definition 22 ($\rightarrow_n$)**.** $(P, pc, s_l, s, ls) \xrightarrow{l,t,R,W}_n (P, pc', s_l', s', ls')$ if and only if there exists an execution fragment $e = (c_0, c_1)(c_1, c_2)\ldots(c_{n-1}, c_n)$ such that $\mathsf{th}(0) = \mathsf{th}(1) = \ldots = \mathsf{th}(n-1) = T$ for some thread T of P, $\mathsf{sync}(0)$, $\mathsf{mem}(1)$, ..., $\mathsf{mem}(n-1)$, $\mathsf{sync}(T, pc')$ or $\mathsf{term}(T, pc')$, $\mathsf{loc}(0) = l$, $\mathsf{type}(0) = t$, $pc(c_0) = pc, s_l(c_0) = s_l, s(c_0) = s, ls(c_0) = ls$ and $pc(c_n) = pc', s_l(c_n) = s_l', s(c_n) = s', ls(c_n) = ls'$. The set R (resp. W) is the set of memory locations read (resp. written) by steps 1 to $n-1$.

The previous transition relation $\rightarrow_L$ for the case of non-nested locks embodied uninterrupted execution from a `lock()` operation to the next `lock()` operation, skipping over the single `unlock()` operation occuring in between the two `lock()` operations. The new transition relation $\rightarrow_n$ goes from a lock operation to the next lock operation, without skipping over any lock operations.

The set $\mathbb{S}_n(T)$ denotes the transition trace set of a thread T with nested locks. A transition trace has the form $(l_0, t_0, s_0, R_0, W_0, s_0^*)(l_1, t_1, s_1, R_1, W_1, s_1^*)\ldots(l_n, t_n, s_n, R_n, W_n, s_n^*)$ *term*. Each tuple corresponds to a transition from a lock operation to immediately before the next lock operation. The first component of a tuple denotes the lock operated on, the second component denotes the type of the lock operation (either lock or unlock), the third component denotes the starting state of the transition, the fourth and fifth components denote the sets of memory locations read or written during the transition, and the sixth component denotes the target state of the transition. The final part *term* indicates whether the transition trace corresponds to a terminated thread execution.

In Figure 3.11 we define two predicates on transition traces. Given a transition trace t and an index i, they return the index of the next transition that starts at a `lock()`, or the most recent transition that started in an `unlock()` (if existing). We leave the transition trace t off when it is clear from context.

$$\begin{array}{ll} \mathsf{next\text{-}lock}(t,i) = j \Leftrightarrow & \mathsf{prev\text{-}unlock}(t,i) = j \Leftrightarrow \\ \quad i < j & \quad j < i \\ \quad \mathsf{type}(t_j) = \mathsf{lock} \vee j = |t| & \quad \mathsf{type}(t_j) = \mathsf{unlock} \vee j = 0 \\ \quad \forall\, i < k < j \colon \mathsf{type}(t_k) \neq \mathsf{lock} & \quad \forall\, j < k < i \colon \mathsf{type}(t_k) \neq \mathsf{unlock} \end{array}$$

Fig. 3.11: Next lock and most recent unlock

Figure 3.12 defines the transition trace set $\mathbb{S}_n(T)$ of a thread T. Like in the case with non-nested locks, it embodies all interactions the thread T may perform with a context with which it is data-race-free. Line 8 specifies that the state does not change at `unlock()` operations, and lines 9–12 restrict how the state may change at `lock()` operations. The memory locations in $A_j \cup A_{j+1} \cup \ldots \cup A_{i-1}$ are defined to not change at a `lock()` operation since a context that could implement these changes would necessarily have a data race with T.

The $\mathsf{match}_{n,a}(t', t)$ predicate between transition traces t', t is shown in Figure 3.13. The constraints correspond to those of the previous $\mathsf{match}_a(t', t)$ predicate for the case without nested locks. The main differences are as follows. Previously, given a transition i starting in a `lock()` operation, we had sets W_{i-1} (resp. A_{i-1}) denoting the memory locations written (resp. accessed) between the previous `unlock()` operation and the current `lock()` operation, and sets W_{i+1} (resp. A_{i+1}) denoting the memory locations written (resp. accessed) between the next `unlock()` operation and the subsequent `lock()` operation.

Now, given a transition i, we have sets $W_l, \ldots, W_{i-1}$ (resp. $A_l, \ldots, A_{i-1}$) whose union corresponds to the set of memory locations accessed by the transitions starting at the most recent `unlock()` up to the current transition i, and sets $W_{i+1}, \ldots, W_{j-1}$ (resp. $A_{i+1}, \ldots, A_{j-1}$) whose union corresponds to the set of memory locations accessed by the transitions following transition i up to the next transition starting in a `lock()` operation.

Looking at the race constraints, we see that they allow memory accesses to move downwards over several `lock()` operations or upwards over several `unlock()` operations (lines 7–8). For example, the constraint $W'_i \subseteq (W_l \cup \ldots \cup W_i \cup \ldots \cup W_{j-1})$ states that any memory location written to by transition i of t' must have also been written to by one of the transitions $l, \ldots, i, \ldots, j-1$ of t. Thus, a memory

$$
\begin{array}{rl}
\multicolumn{2}{l}{\mathbb{S}_n(T) = \{(l_0, t_0, s_0, R_0, W_0, s_0^*)(l_1, t_1, s_1, R_1, W_1, s_1^*) \ldots (l_n, t_n, s_n, R_n, W_n, s_n^*)\ \mathit{term} \mid} \\
1 & \exists pc_0, \ldots, pc_{n+1}, s_{l,0}, \ldots, s_{l,n+1}, ls_0, \ldots, ls_{n+1} : \\
2 & \quad (T, pc_0, s_{l,0}, s_0, ls_0) \xrightarrow{l_0, t_0, R_0, W_0} (P, pc_1, s_{l,1}, s_0^*, ls_1) \wedge \\
3 & \quad (T, pc_1, s_{l,1}, s_1, ls_1) \xrightarrow{l_1, t_1, R_1, W_1} (P, pc_2, s_{l,2}, s_1^*, ls_2) \wedge \\
4 & \quad \ldots \\
5 & \quad (T, pc_n, s_{l,n}, s_n, ls_n) \xrightarrow{l_n, t_n, R_n, W_n} (T, pc_{n+1}, s_{l,n+1}, s_n^*, ls_{n+1}) \wedge \\
6 & \quad pc_0 = pc_s \wedge s_{l,0} = s_{l,s} \wedge ls_0 = ls_s \wedge \\
7 & \quad \mathit{term} \Leftrightarrow pc_{n+1} = pc_f \wedge \\
8 & \forall\, 0 < i \leq n \colon (t_i = \mathsf{unlock} \Rightarrow s_i = s_{i-1}^*) \wedge \\
9 & \forall\, 0 < i \leq n \colon t_i = \mathsf{lock} \Rightarrow \\
10 & \quad \textbf{let } j = \mathsf{prev\text{-}unlock}(i) \textbf{ in} \\
11 & \quad \forall x \in (M - (A_j \cup A_{j+1} \cup \ldots \cup A_{i-1})) \colon \\
12 & \qquad s_i(x) = s_{i-1}^*(x) \\
\multicolumn{2}{l}{\}}
\end{array}
$$

Fig. 3.12: Definition of the transition trace set of a thread

location written to by, e.g., transition l of t might be written to by transition i of t', corresponding to the respective write operation having moved downwards over several `lock()` operations in T' compared to T.

We can now define the $\mathsf{match}_{n,b}(t', t)$ predicate.

Definition 23.

$$\begin{aligned}\mathsf{match}_{n,b}(t',t) \Leftrightarrow\ & \textbf{let } n = \mathsf{min}(|t'|,|t|) - 1 \textbf{ in}\\ & ((\mathsf{term}(t') \Rightarrow \mathsf{term}(t)) \wedge \mathsf{match}_{n,a}(t',t)) \vee\\ & \exists\, 0 \le i \le n:\\ & \quad t_i = \mathsf{lock} \wedge\\ & \quad \mathsf{match}_{n,a}(t'[0:i], t[0:i]) \wedge\\ & \quad \textbf{let } j = \mathsf{prev\text{-}unlock}(i) \textbf{ in}\\ & \quad \exists x \in (A_j \cup A_{j+1} \cup \ldots \cup A_{i-1}) - (A'_j \cup A'_{j+1} \cup \ldots \cup A'_{i-1}):\\ & \qquad s'^{*}_{i-1}(x) \ne s'_i(x)\end{aligned}$$

Similarly to Definition 12, the definition says that either t' and t match and if t' corresponds to a terminated thread execution then so does t, or there are prefixes that match, and at the subsequent `lock()` a memory location in t' changes that is accessed by t but not by t' ($x \in (A_j \cup A_{j+1} \cup \ldots \cup A_{i-1}) - (A'_j \cup A'_{j+1} \cup \ldots \cup A'_{i-1})$). Thus, a context that could implement the change of the memory location that t' observes would have a data race with t. Since when t is involved in a data race we have undefined behaviour, any behaviour of t' is allowed. Hence, we consider the traces t' and t matched.

Finally, in Definition 24 we define the $\mathsf{check}_n(T', T)$ predicate. It implies $\mathsf{ref}(T', T)$ also in the case when T' and T contain nested locks.

Definition 24.

$$\mathsf{check}_n(T', T) \Leftrightarrow \forall t' \in \mathbb{S}_n(T'): \exists t \in \mathbb{S}_n(T): \mathsf{match}_{n,b}(t', t)$$

$\mathsf{match}_{n,a}(t', t) \Leftrightarrow$

1 $|t'| = |t|$

2 **let** $n = |t| - 1$ **in**

\# `same locks constraint`

3 $\forall\, 0 \leq i \leq n\colon l'_i = l_i \wedge t'_i = t_i$

\# `race constraints`

4 $\forall\, 0 \leq i \leq n\colon$

5 **let** $l = \mathsf{prev\text{-}unlock}(i)$ **in**

6 **let** $j = \mathsf{next\text{-}lock}(i)$ **in**

7 $W'_i \subseteq (W_l \cup \ldots \cup W_i \cup \ldots \cup W_{j-1})$

8 $R'_i \subseteq (A_l \cup \ldots \cup A_i \cup \ldots \cup A_{j-1}))$

\# `state at locks constraints`

9 $\forall\, 0 \leq i \leq n\colon$

10 $t_i = \mathsf{lock} \Rightarrow$

11 **let** $j = \mathsf{prev\text{-}unlock}(i)$ **in**

12 $\forall\, x \in (M - (A_j \cup A_{j+1} \cup \ldots \cup A_{i-1}))\colon$

13 $s_i(x) = s'_i(x)$

14 $\forall\, x \in (A_j \cup A_{j+1} \cup \ldots \cup A_{i-1}) - (A'_j \cup A'_{j+1} \cup \ldots \cup A'_{i-1})\colon$

15 $s'_i(x) = s'^{*}_{i-1}(x)$

\# `state at unlocks`

16 $\forall\, 0 \leq i \leq n\colon$

17 $t_i = \mathsf{unlock} \Rightarrow$

18 **let** $j = \mathsf{next\text{-}lock}(i)$ **in**

19 $\forall x \in (M - (W_i \cup W_{i+1} \cup \ldots \cup W_{j-1}))\colon$

20 $s_i(x) = s'_i(x)$

Fig. 3.13: Definition of matching transition traces

3.6 Compiler Testing

Previously we have argued that our specification efficiently captures thread refinement in the SC-for-DRF memory model, as it abstracts over the way in which a thread implements the state transitions between lock operations. In this section we provide experimental evidence, showing that the application of our transition-based theory in a compiler testing setting leads to large performance improvements compared to using an event-based theory.

Eide and Regehr [41] pioneered an approach to test that a compiler correctly optimises programs that involves repeatedly (1) generating a random C program, (2) compiling it both with and without optimisations (e. g., `gcc -O0` and `gcc -O3`), (3) collecting a trace from both the original and the optimised program, and (4) checking whether the traces match. If two traces do not match a compiler bug has been found. Morisset et al. [79] extended this approach to a fragment of C11 and implemented it in their cmmtest tool.

The cmmtest tool consists of the following components: an adapted version of csmith [113] (we call it "csmith-sync" in the following) to generate random C threads, a tool to collect execution traces of a thread ("pin-interceptor"), and a tool to check whether two given traces match ("cmmtest-check"). The csmith-sync tool generates random C threads with synchronisation operations such as `pthread_mutex_lock()`, `pthread_mutex_unlock()`, or the C11 primitives `release()` and `acquire()`. We only consider programs containing lock operations. The pin-interceptor tool is based on the Pin binary instrumentation framework [72]. It executes a program and instruments the memory accesses and synchronisation operations in order to collect a trace of those operations. The cmmtest-check tool takes two execution traces (produced by pin-interceptor) of an optimised and an unoptimised thread, and checks whether the traces match.

3.6.1 Implementation

We use the existing csmith-sync and pin-interceptor tools, and implemented our own trace checker tracecheck. It takes two thread execution traces (such as those depicted in Figure 3.2b) and their common known states at `lock()` operations, and then determines the state transitions between lock operations, and the sets of memory locations accessed between lock operations. That is, in essence, for a trace

it constructs its corresponding transition trace (i.e., an element of $\mathbb{S}_n(T)$). Then, it checks whether the two transition traces match by implementing the $\mathsf{match}_{n,b}(t', t)$ predicate. This way of checking thread execution traces is very efficient as it can be implemented in runtime *linear* in the length of the traces.

This can be seen as follows. First, the trace generation step ensures that both traces start in the same state at `lock()` operations. Second, at each `unlock()` operation not the complete states have to be checked for equality, but only the memory locations that have been written to since the last lock operation. Thus, checking the states at `unlock()` operations (corresponding to the "states at unlock" constraints of the $\mathsf{match}_{n,a}(t', t)$ predicate) is a linear operation.

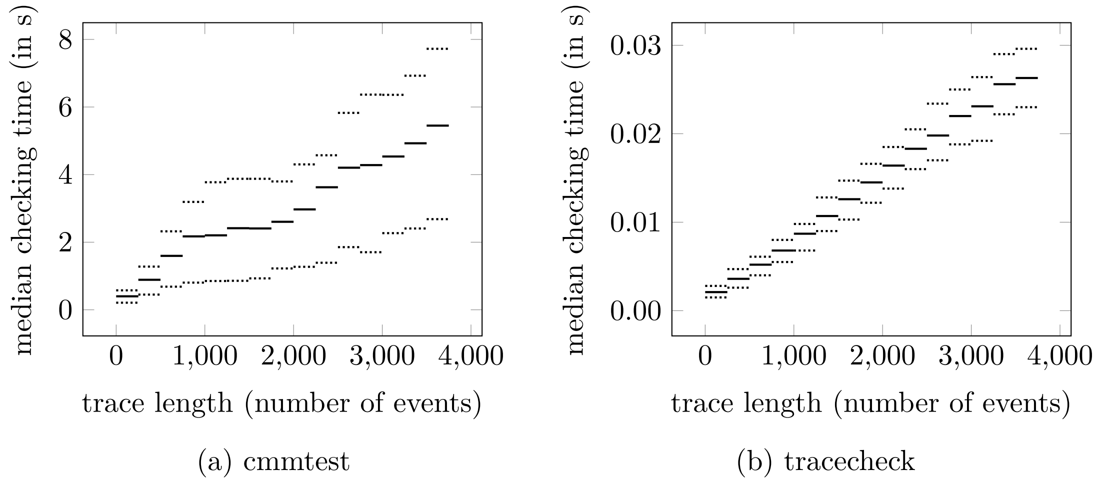
The race constraints can also be checked in linear time. First, the size of the sets is bounded by the number of memory locations accessed between the two corresponding lock operations. Second, checking whether a set A is a subset of another set B can be implemented in linear time in the size of A.[2] In summary, we have a linear procedure for checking whether two traces match.

By contrast, cmmtest-check attempts to match traces by finding a sequence of valid transformations that transforms one trace into the other. Different sequences are explored in a tree-like fashion [79], suggesting exponential runtime in the worst case.

3.6.2 Experiments

We evaluated tracecheck on in total 40,000 randomly generated C threads. We compiled each with `gcc -O0` and `gcc -O3` and collected a trace from each. The length of the traces was in the range of 1 to 4,000 events. Our tool outperformed cmmtest-check on all traces. On average, tracecheck was 210 X faster. As outlined in the previous section, tracecheck and cmmtest-check have different asymptotic complexity. Thus, the relative performance between the two will likely depend on the length of the traces considered. During our testing, we did not find new bugs in `gcc`. Moreover, both tracecheck and cmmtest-check did not yield any false positives.

[2] If A and B are represented as hash sets, then $A \subseteq B$ can be checked by iterating over the elements of A, and for each one performing a lookup in B (which has constant time). If all elements are found, A is a subset of B.



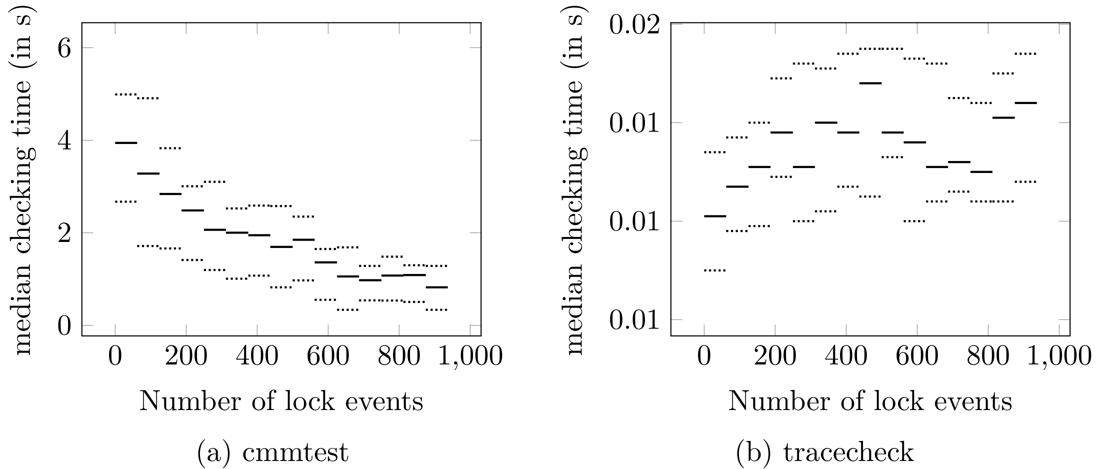
(a) cmmtest

(b) tracecheck

Fig. 3.14: Median checking time over length of traces

Figure 3.14 shows the median time it took to match two traces of a certain length, for cmmtest-check (Figure 3.14a) and tracecheck (Figure 3.14b). Along the x axis, we classify the pairs of traces t', t into bins according to the length of the unoptimised trace t. Each bin i contains 100 pairs t', t such that the length of t is in the range $[250 \cdot i, 250 \cdot (i+1))$. For example, bin 5 contains the pairs with the length of the unoptimised trace being in the range $[1250, 1500)$. The y axis shows the median time it took to match two traces t', t in the respective bin. The dotted lines represent the 20th and 80th percentile to indicate the spread of the times.

Figure 3.15 shows the effect of the number of lock operations in the two traces on the time it takes to check if they match. We have evaluated this on pairs of traces t', t with the unoptimised trace t having length in the range of $[1900, 2100)$. Along the x axis, we classify the pairs of traces t', t into bins according to the number of lock operations they contain. The y axis again indicates the median matching time. As can be seen in Figure 3.15a, cmmtest-check is sensitive to the number of locks in a trace. That is, matching traces generally takes longer the fewer locks they contain. The reason for this is that cmmtest-check considers lock operations as "barriers" against transformations: it does not try to reorder events across lock operations. Thus, the more lock operations there are in a trace, the fewer potential transformations it tries, and thus the lower the checking time. Our tool tracecheck on the other hand is largely insensitive to the number of locks in a trace.



(a) cmmtest (b) tracecheck

Fig. 3.15: Median checking time over number of locks in a trace

3.7 Related Work

Refinement approaches can be classified based on whether they handle language-level memory models (such as SC-for-DRF or C11) [22, 38, 76, 79, 104, 105], hardware memory models (such as TSO) [57, 106], or idealised models (typically SC) [24, 71].

The approaches for language-level models typically describe refinement by giving valid transformations on thread execution traces. These trace transformations are then lifted to the program code level. An example is the theory of valid optimisations of Morisset et al. [79]. They handle the fragment of C11 with lock/unlock and release/acquire operations. The theory is relatively restrictive in that they do not allow the reordering of memory accesses across synchronisation operations (such as the roach model reorderings described in Section 3.4).

The approaches of Brookes [24] (for SC) and Jagadeesan et al. [57] (for TSO) are closer to ours in that they also specify refinement in terms of state transitions rather than transformations on traces. They provide a sound and complete denotational specification of refinement. However, their completeness proofs rely on the addition of an unrealistic `await()` statement which provides strong atomicity.

Brookes [25] was the first to introduce a semantics for concurrent programs which includes data races and abstracts over execution steps between synchronisation operations by means of a single step. Based on the semantics he also gave a definition of equivalence between programs (and thus in particular threads) in

arbitrary contexts based on equality of the corresponding trace sets. Our work differs in that we consider refinement as opposed to equivalence (with mutual refinement being equivalent to equivalence). Moreover, as we have outlined in Section 3.4, requiring trace set equality does not allow the proof of several common compiler optimisations (in particular roach motel reorderings). Finally, Brookes' semantics only considers terminated traces, and thus cannot detect data races that only occur on non-terminating executions.

Brookes [26] extended his semantics with a more refined approach to race detection. In this work, execution steps are annotated with a flow relation which indicates which reads affect which writes. Then, data races do not lead to program abortion (as in [25]) but instead taint the variables which are affected by the variable involved in the data race. This leads to a larger number of traces for a given command. Thus, while the semantics allows us to distinguish more threads, it validates less compiler optimisations than the previous semantics.

Liang et al. [68] presented a rely-guarantee-based approach to reason about thread refinement. Starting from the assumption of arbitrary concurrent contexts, they allow us to add constraints that capture knowledge about the context in which the threads run in. They later extended their approach to also allow reasoning about whether the original and the refined thread exhibit the same termination behaviour [69].

Lochbihler [71] provides a verified non-optimising compiler for concurrent Java guaranteeing equivalence between the threads in the source program and the bytecode. It is however based on SC semantics rather than the Java memory model. Sevcik et al. [106] developed the verified CompCertTSO compiler for compilation from a C-like language with TSO semantics to x86 assembly.

The compiler testing method based on checking traces of randomly generated programs on which we evaluated our refinement specification in Section 3.6 was pioneered by Eide and Regehr [41]. They used this approach to check the correct compilation of volatile variables. It was extended to a fragment of C11 by Morisset et al. [79].

Chaki et al. [29] extended regression verification [48, 49] (i.e., deciding whether two programs are equivalent) to concurrent programs under sequential consistency. They define a notion of partial equivalence and present two proof rules that allow us to infer that two concurrent programs are equivalent under arbitrary contexts.

The premises of their proof rules essentially require that corresponding functions in the original and the refined program, when given the same input vector, read and write the same values to and from shared variables. They later extended their approach to programs with locks and dynamic thread creation [30].

Chakraborty and Vafeiadis [31] presented an approach to validate optimisations for concurrent C11 and LLVM IR programs. Their approach works on the CFG of the programs, as opposed to on concrete executions. They provide two matching algorithms that are sound for loop-free programs (programs with loops are handled heuristically). One algorithm is compiler-independent, whereas the other makes use of compiler-provided metadata about performed optimisations for more efficient matching. They applied their approach in compiler testing by means of checking the correct compilation of short, automatically generated programs.

Dodds et al. [38] describe a refinement approach for C11/C++11 (without relaxed atomics and SC atomics). They represent the possible interactions of a sequential code block with an arbitrary context via a set of histories. A history consists of a set of memory accesses together with a guarantee relation which models the possible happens-before relations with the context. Their approach is sound and complete for programs that only have finite executions. They also provide a tool Stellite to verify refinement between code blocks.

Chapter 4

Deadlock Analysis

4.1 Introduction

In the previous two chapters, we were concerned with the correctness of lock implementations, and the validity of compiler optimisations in the presence of locks. In constrast, in this chapter, we are concerned with the correct use of locks. In particular, we are concerned with deadlocks and present a static approach for deadlock analysis.

A deadlock is an erroneous situation where each of a set of threads is blocked on a `lock()` operation, with each thread awaiting the release of a lock that another thread in that set has. Consequently, the threads are stuck and cannot continue execution. Deadlocks are difficult to discover by means of testing. Similar to other concurrency bugs, triggering a deadlock requires a specific thread schedule and a set of particular program inputs. In contrast to testing, formal verification approaches like model checking and deductive verification typically do not miss bugs, but may suffer from limited scalability. We thus turn to static analysis to analyse a program for deadlocks.

Our deadlock analysis approach consists of a pipeline of several static analyses, such as may-hold-lock analysis, implemented on top of a new context- and thread-sensitive abstract interpretation framework. We show how to soundly handle a variety of sources of imprecision, such as may-point-to information or thread creation in loops or recursions. We further present a lightweight dependency analysis for identifying statements that could affect a given set of expressions. We use it to speed up the pointer analysis by focusing it to statements relevant to

deadlock analysis. To quantify scalability, we have applied our implementation to a large body of real-world concurrent C code from the Debian GNU/Linux distribution.

Our approach is *sound*[1] (i.e., misses no deadlocks) for programs that have defined semantics according to the C standard [54] and the Pthreads specification [101] (and thus in particular do not contain data races).

4.1.1 Definition of Deadlocks

There are a multitude of ways in which a program can contain a bug that can cause a thread to be blocked erroneously, depending on the programming language, concurrency model, and the involved language constructs (e.g., synchronisation primitives or system calls). Thus, in the remainder of this section, we set out to delineate the type of deadlocks we aim to detect. We first survey the classical definitions of deadlocks and then give the definition of deadlocks that we will use in the rest of the chapter, which is phrased in terms of lock assignment graphs. We focus on deadlocks involving locks in C/Pthreads programs.

Classical deadlock definitions Informally, a *deadlock* is a situation in a program where a set of threads is blocked, each awaiting an event (typically the freeing up of a requested resource, such as a lock) that can only be triggered by another thread in that set of threads [99].

Much of the early work on deadlocks is phrased generally in terms of *resources*. From the viewpoint of deadlock analysis, a resource is any program entity a program needs in order to be able to continue execution. Examples of resources are files, network ports, semaphores, or locks. At a particular step in the program execution, a thread can hold a certain resource (i.e., the resource is assigned to the thread) or it may wait for a resource currently not available (i.e., it has requested a resource currently assigned to another thread).

Coffman et al. [34] introduced a graph representation (termed *state graphs*) to depict the resources allocated and requested by the threads in the program. A state graph is associated with a particular step in the execution of the program. That

[1] We use the term *soundness* in the static analysis/verification sense, i.e., a sound analysis does not miss any bugs. This differs from the usage in dynamic analysis, where it means that an analysis does not yield false bug reports.

is, if concurrent executions are represented as interleavings, each prefix of a given interleaving has an associated state graph which depicts the resources currently allocated and requested by the different threads after the execution of that prefix. A program is then defined to contain a deadlock at a certain execution step if the associated state graph has a cycle. This implies that the threads involved in the cycle cannot continue execution.

Holt [53] introduced a different graph representation termed *general resource graphs*. Like state graphs, they are associated with a particular step in the program execution. However, general resource graphs have nodes for both threads and resources (whereas state graphs only have nodes for resources), and resources can be consumable or non-consumable and can be annotated with the number of available units. Then, similarly as above, deadlocks are defined as cycles in the general resource graphs.

Lock assignment graphs To define deadlocks involving locks, we use a graph representation which we call *lock assignment graphs* (LAGs). LAGs are a special case of general resource graphs, with each resource (i.e., lock) being non-consumable and having one available unit.

To illustrate the definition of deadlocks, we give a simple example. Figure 4.1 shows an example program containing a deadlock. The two threads attempt to acquire the locks `m1` and `m2` in opposite order which could result in a deadlock with the threads being blocked at lines 5 and 13, respectively.

Figure 4.2a shows a concrete interleaving of the two threads, and Figure 4.2b shows the LAG after the final step of the interleaving. The square nodes represent threads and the circle nodes represent locks. A solid arrow indicates that the lock at the source of the edge has been assigned to the thread at the target of the arrow, and a dashed arrow indicates that the thread at the source of the arrow has requested the lock at the target of the arrow.

At the start of the program, the LAG contains only the nodes for the locks and threads and no arrows (assuming statically allocated locks and threads here). Executing a `pthread_mutex_lock()` operation creates an assignment edge when the lock is available, and creates a request edge when the lock is assigned to a thread already. Executing a `pthread_mutex_unlock()` operation removes an assignment edge from the graph.

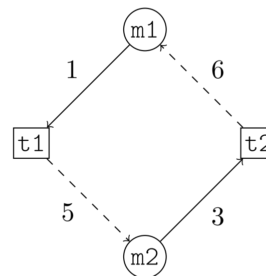
```
1 void *thread1()
2 {
3   pthread_mutex_lock(&m1);
4   x = 1;
5   pthread_mutex_lock(&m2);
6
7   return 0;
8 }
```

```
9  void *thread2()
10 {
11   pthread_mutex_lock(&m2);
12   y = 2;
13   pthread_mutex_lock(&m1);
14
15   return 0;
16 }
```

Fig. 4.1: A simple deadlocking program

```
1 pthread_mutex_lock(&m1)
2 x = 1
3    pthread_mutex_lock(&m2)
4    y = 2
5 pthread_mutex_lock(&m2)
6    pthread_mutex_lock(&m1)
```

(a) Interleaving (thread1 in bold)



(b) Lock assignment graph (LAG)

Fig. 4.2: Interleaving of the program in Figure 4.1 and the LAG after the final step of the interleaving. Solid arrows represent lock assignments and dashed arrows represent lock requests. The arrows are annotated with the steps in the interleaving that created them.

In Figure 4.2b we have annotated the edges with the steps in the interleaving in Figure 4.2a that create the respective edges. After the final step of the interleaving the associated LAG has a cycle. Thus, there is a cyclic relationship of lock assignments and lock requests and the threads involved in the cycle cannot continue execution.

Other blocking situations involving locks The definition of deadlocks given above in terms of cycles in LAGs does not cover all situations in which a thread is erroneously blocked on a lock. Consider the example in Figure 4.3a. If the lock in line 10 is executed before the lock in line 3, then thread2 exits while still holding the lock. Consequently, when thread1 tries to acquire the lock in line 3, it will be blocked indefinitely. However, there is no corresponding cycle in the LAG.

Another example is given in Figure 4.3b. Here thread1 first acquires lock m and then waits to join with thread2. However, thread2 also requires m. Thus,

```
1 void *thread1()
2 {
3   pthread_mutex_lock(&m);
4   pthread_mutex_unlock(&m);
5
6   return 0;
7 }

8 void *thread2()
9 {
10   pthread_mutex_lock(&m);
11
12   return 0;
13 }
```

(a) Thread holding a lock exits

```
1 void *thread1()
2 {
3   pthread_t tid;
4   pthread_create(
5     &tid, NULL, thread2, NULL);
6
7   pthread_mutex_lock(&m);
8   pthread_join(tid, NULL);
9   pthread_mutex_unlock(&m);
10
11   return 0;
12 }

13 void *thread2()
14 {
15   pthread_mutex_lock(&m);
16   pthread_mutex_unlock(&m);
17
18   return 0;
19 }
```

(b) Thread holding a lock invokes join

Fig. 4.3: Other blocking situations

`thread1` is blocked at the join operation and `thread2` is blocked at the lock. Again, there is no cycle in the LAG.

4.2 Analysis Overview

The design of our deadlock analysis approach has been guided by the goal to analyse real-world concurrent C/Pthreads code in a sound way. For programs with undefined behaviour, we do not formally guarantee soundness. For such programs the compiler is allowed to do anything, and may in particular produce a program containing a deadlock. This could happen in practice, for example, if the compiler removes an if-branch that contains an unlock if it can determine that the branch always invokes an undefined operation (such as one resulting in an integer overflow).

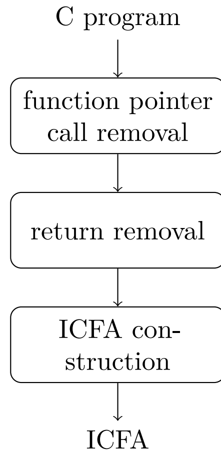


Fig. 4.4: ICFA construction

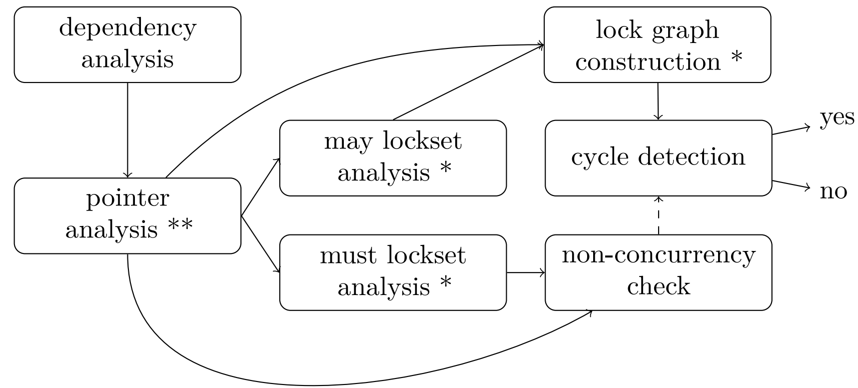


Fig. 4.5: Analysis pipeline

As discussed in the previous section, LAGs are associated with a particular step in an interleaving of the program. Thus, a simple approach to check a program for deadlocks would be to enumerate all interleavings, and for each step in the interleaving check whether the associated LAG has a cycle. However, due to the large number of interleavings, this approach would be infeasible for all but the simplest programs. We thus aim to abstract the information from all the LAGs into a single graph such that if it does not have a cycle, none of the LAGs associated with any of the execution steps in any of the interleavings have a cycle.

Figures 4.4 and 4.5 give an overview of our analysis approach. We take a C program and first transform it into a graph representation (see Section 4.3.1). Then the deadlock analysis is performed via a pipeline of several analyses. An arrow between two analyses indicates that the target uses information computed by the source. We use a dashed arrow from the non-concurrency analysis to the cycle detection to indicate that the required information is computed on-demand (i.e., the cycle detection may repeatedly query the non-concurrency analysis, which computes the result in a lazy fashion).

The pointer analysis, may- and must-lockset analysis, and the lock graph construction are implemented on top of our new generic context- and thread-sensitive analysis framework (described in detail in Sections 4.3.2–4.3.4). The framework comes in a flow-insensitive and a flow-sensitive version. This allows a trade-off between precision and cost. Furthermore, there can be differences

regarding the soundness of the result for concurrent programs. Flow-insensitive analyses are sound for concurrent programs as they do not take into account the order of statements [90]. Flow-sensitive analyses may be sound or unsound for concurrent programs (depending on the concrete analysis at hand). Consequently, the pointer analysis was implemented on top of the flow-insensitive version of the analysis framework (thus marked with ** in Figure 4.5), and the may- and must-lockset analysis and the lock graph construction were implemented on top of the flow-sensitive version of the analysis framework (marked with *). The dependency analysis and the non-concurrency analysis are separate standalone analyses.

Context and thread sensitivity Typical patterns in real-world C code suggest that an approach that provides a form of context-sensitivity is necessary to obtain satisfactory precision, as otherwise there would be too many false deadlock reports. For instance, many projects provide their own wrappers for the functions of the Pthreads API. Figure 4.6a, for example, shows a lock wrapper from the VLC project[2]. An analysis that is not context-sensitive would merge the points-to information for pointer `p` from different call sites invoking `vlc_mutex_lock()`, and thus yield many false alarms.

Thread creation causes a similar problem. For every call to `pthread_create()`, the analysis needs to determine which thread is created (i.e., the function pointed to by the pointer passed to `pthread_create()`). This is straightforward if a function identifier is given to `pthread_create()`. However, similar to the case of lock wrappers above, projects often provide wrappers for `pthread_create()`. Figure 4.6b gives the wrapper for `pthread_create()` from the memcached project[3]. The wrapper uses the function pointer that is passed to `create_worker()` to create a thread. Maintaining precision in such cases requires us to track the flow of function pointer values from function arguments to function parameters. This is implemented directly as part of the analysis framework (as opposed to in the full points-to analysis).

Dependency analysis Deadlock detection requires the information which lock objects an expression used in a `pthread_mutex_lock()` call may refer to. We

[2] http://www.videolan.org
[3] https://memcached.org

```
1 void vlc_mutex_lock (vlc_mutex_t *p) {
2   int val = pthread_mutex_lock(p);
3   VLC_THREAD_ASSERT("locking␣mutex");
4 }
```

(a) Lock wrapper

```
1  void create_worker(void *(*func)(void *),
2                     void *arg) {
3    pthread_attr_t attr;
4    int ret;
5    pthread_attr_init(&attr);
6    if ((ret=pthread_create(
7                &((THREAD*)arg)->thread_id,
8                &attr, func, arg)) != 0) {
9      fprintf(stderr, "Error:␣%s\n",
10               strerror(ret));
11     exit(1);
12   }
13 }
```

(b) Thread create wrapper

Fig. 4.6: Wrapper functions

compute this data using the pointer analysis, which is potentially expensive. However, it is easy to see that potentially many assignments and function calls in a program do not affect the values of lock expressions. Consider the example in Figure 4.7. The accesses to `x` cannot affect the value of the lock pointers `m1`–`m5`. Further, the code in function `func1()` cannot affect the values of the lock pointers, and thus in turn the call `func1()` in line 10 cannot affect the lock pointers.

We have developed a lightweight context-insensitive, flow-insensitive analysis to identify statements that may affect a given set of expressions. The result is used to speed up the pointer analysis. The dependency analysis is based on marking statements which (transitively) share common variables with the given set of expressions. In our case, the relevant expressions are those used in lock-, unlock-, create-, and join-statements. For the latter two we track the thread ID variable (first parameter of both) whose value is required to determine which thread is joined by a join operation. We give the details of the dependency analysis in Section 4.4.

Non-concurrency analysis A deadlock resulting from a thread 1 first acquiring lock m_1 and then attempting to acquire m_2 (at program location ℓ_1), and thread 2 first acquiring m_2 and then attempting to acquire m_1 (at program location ℓ_2) can only occur when in a concrete program execution the program locations ℓ_1 and ℓ_2 run concurrently. If we have a way of deciding whether two locations could potentially run concurrently, we can use this information to prune spurious deadlock reports. For this purpose we have developed a non-concurrency analysis

that can detect whether two statements cannot run concurrently based on two criteria.

Common locks. If thread 1 and thread 2 hold a common lock at locations ℓ_1 and ℓ_2, then they cannot both simultaneously reach those locations, and hence the deadlock cannot happen. This is illustrated in Figure 4.7. The thread `main()` attempts to acquire the locks in the sequence `m1`, `m3`, `m2`, and the thread `thread()` attempts to acquire the locks in the sequence `m1`, `m2`, `m3`. There is an order inversion between `m2` and `m3`, but there is no deadlock since the two sections 7–13 and 28–34 (and thus in particular the locations 9 and 30) are protected by the common lock `m1`. The common locks criterion was first described by Havelund [50] (common locks are called *gatelocks* there).

Create and join. Statements might also not be able to run concurrently because of the relationship between threads due to the `pthread_create()` and `pthread_join()` operations. In Figure 4.7, there is an order inversion between the locks of `m5` and `m4` by function `func2()`, and the locks of `m4` and `m5` of thread `thread()`. Yet there is no deadlock since the thread `thread()` is joined before `func2()` is invoked.

Our non-concurrency analysis makes use of the must lockset analysis (computing the locks that must be held) to detect common locks. To detect the relationship between threads due to create and join operations, it uses a search on the program graph for joins matching earlier creates. We give more details of our non-concurrency analysis in Section 4.5.

```
1  int main()
2  {
3    pthread_t tid;
4    pthread_create(&tid, 0,
5      thread, 0);
6
7    pthread_mutex_lock(&m1);
8    pthread_mutex_lock(&m3);
9    pthread_mutex_lock(&m2);
10   func1();
11   pthread_mutex_unlock(&m2);
12   pthread_mutex_unlock(&m3);
13   pthread_mutex_unlock(&m1);
14
15   pthread_join(tid, 0);
16
17   int r;
18   r = func2(5);
19
20   return 0;
21 }

22 void func1()
23 {
24   x = 0;
25 }

26 void *thread()
27 {
28   pthread_mutex_lock(&m1);
29   pthread_mutex_lock(&m2);
30   pthread_mutex_lock(&m3);
31   x = 1;
32   pthread_mutex_unlock(&m3);
33   pthread_mutex_unlock(&m2);
34   pthread_mutex_unlock(&m1);
35
36   pthread_mutex_lock(&m4);
37   pthread_mutex_lock(&m5);
38   x = 2;
39   pthread_mutex_unlock(&m5);
40   pthread_mutex_unlock(&m4);
41
42   return 0;
43 }

44 int func2(int a)
45 {
46   pthread_mutex_lock(&m5);
47   pthread_mutex_lock(&m4);
48   if(a)
49     x = 3;
50   else
51     x = 4;
52   pthread_mutex_unlock(&m4);
53   pthread_mutex_unlock(&m5);
54   return 0;
55 }
```

Fig. 4.7: Example of a deadlock-free program

4.3 Analysis Framework

In this section, we first introduce our program representation, then describe our context- and thread-sensitive analysis framework, and then describe the pointer analysis and lockset analyses that are implemented on top of the framework.

```
1  void f1() {}
2  void f2() {}
3  void f3() {}
4  int f4(int a) {}
5  ...
6  ... = &f1;
7  ... = &f3;
8  ... = &f4;
9  ...
10 fp();
```

⇒

```
1 ...
2 if(fp==f1)
3   f1();
4 else
5 if(fp==f3)
6   f3();
```

Fig. 4.8: Function pointer removal

```
1 int f() {
2   if(...)
3     return 0;
4   else
5     return 1;
6 }
7 ...
8 a = f();
```

⇒

```
1  int f() {
2    int ret;
3    if(...)
4      ret = 0;
5      goto END;
6    else
7      ret = 1;
8      goto END;
9  END:
10   return ret;
11 }
12 ...
13 a = f();
```

Fig. 4.9: Return removal

4.3.1 Program Representation

Preprocessing. Our tool takes as input a concurrent C program using the Pthreads threading library. In the first step, the calls to functions through function pointers are removed. A call is replaced by a case distinction over the functions the function pointer could refer to. Specifically, a function pointer can only refer to functions that are type-compatible and of which the address is taken at some point in the code. This is illustrated in Figure 4.8. We assume that the function pointer `fp` has type `void (*)(void)`. Function `f2()` (address not taken) and `f4()` (not type-compatible) do not have to be part of the case distinction. Calling `f4()` via `fp` would be undefined behaviour according to the C standard [54]. In the second step, functions with multiple exit points (i.e., multiple return statements) are transformed such as to have only one exit point (illustrated in Figure 4.9).

Interprocedural CFAs. We transform the program into a graph representation which we term *interprocedural control flow automaton* (ICFA). The functions of the program are represented as CFAs [52]. CFAs are similar to control flow graphs, but with the nodes representing program locations and the edges being labelled with operations. ICFAs have additional inter-function edges modelling function entry, function exit, thread entry, thread exit, and thread join. Figure 4.10 shows the ICFA corresponding to the program in Figure 4.7 (thread exit and thread join edges and the function `func1()` are omitted).

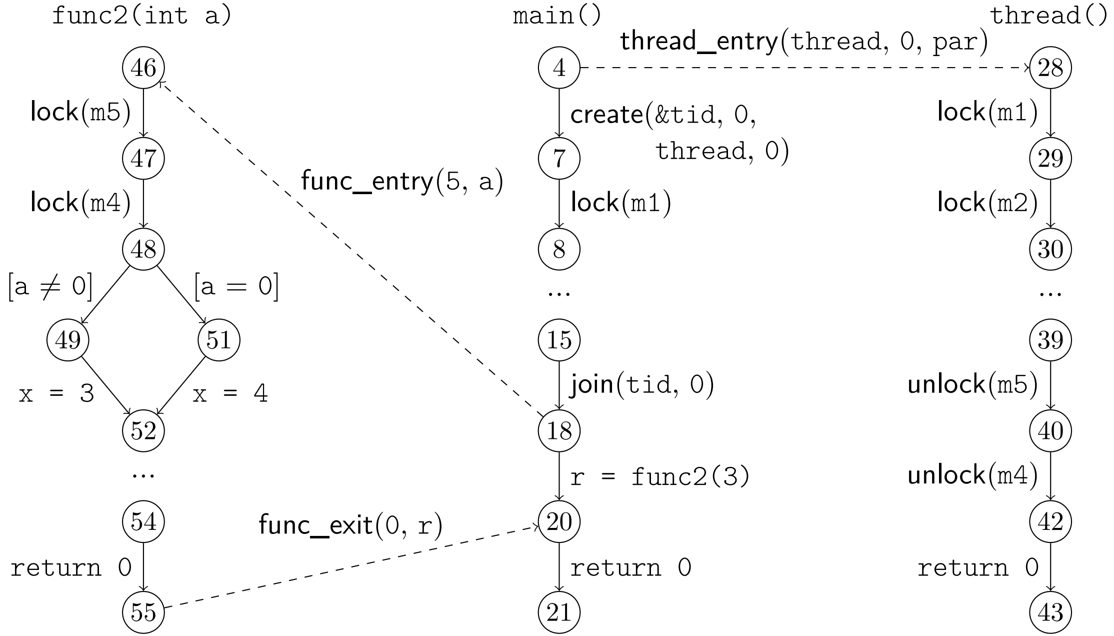


Fig. 4.10: ICFA associated with the program in Figure 4.7

We denote by *Prog* a program (represented as an ICFA), by *Funcs* the set of identifiers of the functions (with `main()` being the entry point of the program), by $L = \{\ell_0, \dots, \ell_{n-1}\}$ the set of program locations, by E the set of edges connecting the locations, and by $\mathsf{op}(e)$ a function that labels each edge with an operation. For example, in Figure 4.10, the edge between locations 49 and 52 is labelled with the operation `x = 3`, and the edge between locations 18 and 46 is labelled with the operation func_entry(`5, a`).

We further write $L(f)$ for the locations in function f. Each program location is contained in exactly one function. The function $\mathsf{func}(\ell)$ yields the function that contains ℓ. The set of variable identifiers in the program is denoted by *Vars*. We assume that all identifiers in *Prog* are unique, which can always be achieved by a suitable renaming of identifiers.

We treat lock, unlock, thread create, and thread join as primitive operations. That is, we do not analyse the body of the threading functions like `pthread_create()` (as implemented in, e.g., glibc on GNU/Linux systems). Instead, our analysis only tracks the semantic effect of the operation, e.g., creating a new thread.

Apart from intra-function edges, ICFAs also have inter-function edges. Each inter-function edge is labelled with one of the five operations func_entry, func_exit, thread_entry, thread_exit, and thread_join.

A function entry edge (func_entry) connects a function call site to the function entry point. The edge label also includes the function call arguments and the function parameters. For example, func_entry(5, a) indicates that the integer literal 5 is passed to the call as an argument, which is assigned to function parameter a. A function exit edge (func_exit) connects the exit point of a function to *every* call site calling the function. Our analysis algorithm filters out infeasible edges during the exploration of the ICFA. That is, if a function entry edge is followed from a function f_1 to function f_2, then the analysis algorithm later follows the exit edge from f_2 to f_1, disregarding exit edges to other functions.

A thread entry edge (thread_entry) connects a thread creation site to all potential thread entry points. It is necessary to connect to all potential thread entry points since often a thread creation site can create threads of different types (i.e., threads corresponding to different functions), depending on the value of the function pointer passed to `pthread_create()`. Analogous to the case of function exit edges, our analysis algorithm tracks the values of function pointers during the ICFA exploration. At a thread creation site it thus can resolve the function pointer, and only follows the edge to the thread entry point corresponding to the value of the function pointer.

A thread_exit edge connects the exit point of a thread to the location following all thread creation sites, and a thread_join edge connects a thread exit point to all join operations in the program.

4.3.2 Analysis Framework – Overview

Our framework to perform context- and thread-sensitive analyses on ICFAs is based on abstract interpretation [37]. It implements a flow-sensitive and flow-insensitive fixpoint computation over the ICFA, and needs to be parametrised with a custom analysis to which it delegates the handling of individual edges of the ICFA. For example, a custom analysis to compute the set of locks held at different program locations would implement the handling of edges corresponding to lock operations in order to add and remove locks to a lock set. We provide more details and a formalisation of the framework in the next section.

Our analysis framework unifies contexts (i.e., function call stacks), threads, and program locations via the concept of a *place*. A place is a tuple $(\ell_0, \ell_1, \ldots, \ell_n)$ of program locations. The program locations $\ell_0, \ldots, \ell_{n-1}$ are either function call sites or thread creation sites in the program (such as, e.g., locations 18 and 4 in Figure 4.10). The final location ℓ_n can be a program location of any type. The locations $\ell_0, \ldots, \ell_{n-1}$ model a possible function call and thread creation history that leads up to program location ℓ_n. We denote the set of all places for a given program by P. The set P may be infinite for programs containing recursion. We use the + operator to extend tuples, i.e., $(\ell_0, \ldots, \ell_{n-1}) + \ell_n = (\ell_0, \ldots, \ell_{n-1}, \ell_n)$. We further write $|p|$ for the length of the place. We write $p[i]$ for element i (indices are 0-based). We use slice notation to refer to contiguous parts of places; $p[i\,{:}\,j]$ denotes the part from index i (inclusive) to index j (exclusive), and $p[\,{:}\,i]$ denotes the prefix until index i (exclusive). We write $\mathsf{top}(p)$ for the last location in the place.

As an example, in Figure 4.10, place $(18, 49)$ denotes the program location 49 in function `func2()` when it has been invoked at call site 18 in the main function. If function `func2()` were called at multiple program locations $\ell_1, \ldots, \ell_m$ in the main function, we would have different places $(\ell_1, 49), \ldots, (\ell_m, 49)$ for location 49 in function `func2()`. Similarly, for the thread function `thread()` and, e.g., location 29, we have a place $(4, 29)$ with 4 identifying the creation site of the thread.

Each place has an associated abstract thread identifier, which we refer to as *thread ID* for short. Given a place $p = (\ell_0, \ldots, \ell_n)$, the associated thread ID is either $t = ()$ (the empty tuple) if no location in p corresponds to a thread creation site, or $t = \ell_0, \ldots, \ell_i$, such that ℓ_i is a thread creation site and all ℓ_j with $j > i$ are not thread creation sites. It is in this sense that our analysis is thread-sensitive, as the information computed for each place can be associated with an abstract thread that way. We write $\mathsf{get_thread}(p)$ for the thread ID associated with place p.

The analysis framework must be parametrised with a custom analysis. The framework handles the tracking of places, the tracking of the flow of function pointer values from function arguments to function parameters, and it invokes the custom analysis to compute dataflow facts for each place.

The domain, transfer function, and join function of the framework are denoted by $\mathcal{D}_s$, $\mathcal{T}_s$, and $\sqcup_s$, respectively, and the domain, transfer function, and join function of the parametrising analysis are denoted by $\mathcal{D}_a$, $\mathcal{T}_a$, and $\sqcup_a$. The custom

analysis has a transfer function $\mathcal{T}_a : E \times P \to (\mathcal{D}_a \to \mathcal{D}_a)$ and a join function $\sqcup_a : \mathcal{D}_a \times \mathcal{D}_a \to \mathcal{D}_a$. The domain of the framework (parametrised by the custom analysis) is then $\mathcal{D}_s = \mathit{Fpms} \times \mathcal{D}_a$, the transfer function is $\mathcal{T}_s : E \times P \to (\mathcal{D}_s \to \mathcal{D}_s)$, and the join function is $\sqcup_s : \mathcal{D}_s \times \mathcal{D}_s \to \mathcal{D}_s$.

The set *Fpms* is a set of mappings from identifiers to functions which map function pointers to the functions they point to. We denote the empty mapping by $\emptyset$. We further write $\mathit{fpm}(\mathit{fp}) = \bot$ to indicate that *fp* is not in $\mathsf{dom}(\mathit{fpm})$ (the domain of *fpm*).

4.3.3 Analysis Framework – Details

We now explain the formalisation of the flow-sensitive version of the analysis framework, which is given in Figures 4.11 and 4.12. Many of the definitions will be reused in the formalisation of the flow-insensitive version of the framework (described in Figure 4.3.4). The figures give the domain $\mathcal{D}_s$, join function $\sqcup_s$, and transfer function $\mathcal{T}_s$, which are defined in terms of the domain $\mathcal{D}_a$, join function $\sqcup_a$, and transfer function $\mathcal{T}_a$ of the parametrising analysis (such as the lockset analyses defined in Section 4.3.6).

The function $\mathsf{next}_s(e, p)$ defines how the place p is updated when the analysis follows the ICFA edge e. For example, on a func_exit edge, the last two locations are removed from the place (which are the exit point of the function, and the location of the call to the function), and the location to which the function returns to is added to the place (which is the location following the call to the function). The thread_entry and func_entry cases are delegated to $\mathsf{entry}_s(p, \ell)$. The first case of the function handles recursion. If the location ℓ of the called function is already part of the place, then the prefix of the place that corresponds to the original call to the function is reused (first case). If no recursion is detected, the entry location of the function is simply added to the current place (second case). For intra-function edges (last case of $\mathsf{next}_s(e, p)$), the last location is removed from the place and the target location of the edge is added.

The overall result of the analysis is a mapping $s\colon P \to (\mathit{Fpms} \times \mathcal{D}_a)$. The result is defined via a fixpoint equation [37]. We obtain the result by computing the least fixpoint (via a worklist algorithm) of the equation below (with s_0 denoting the initial state of the places):

Domain: $\mathcal{D}_s = Fpms \times \mathcal{D}_a$

$$\begin{aligned} s_s^1 \sqcup_s s_s^2 =& (fpm, s_a) \\ & \text{with } s_s^1 = (fpm^1, s_a^1) \\ & \text{with } s_s^2 = (fpm^2, s_a^2) \\ & \text{with } fpm = fpm^1 \sqcup_{fp} fpm^2 \\ & \text{with } s_a = s_a^1 \sqcup_a s_a^2 \end{aligned}$$

$$fpm^1 \sqcup_{fp} fpm^2 = \lambda fp. \begin{cases} v & fpm^1(fp) = v \wedge fpm^2(fp) = v \\ \bot & \text{otherwise} \end{cases}$$

With $e = (\ell_1, \ell_2)$, $\mathsf{top}(p) = \ell_1$, $f = \mathsf{func}(\ell_2)$, and $n = |p|$:

$$\mathsf{next}_s(e,p) = \begin{cases} \mathsf{entry}_s(p, \ell_2) & \mathsf{op}(e) \in \{\mathsf{thread_entry}, \mathsf{func_entry}\} \\ p[:n-2] + \ell_2 & \mathsf{op}(e) \in \{\mathsf{func_exit}, \mathsf{thread_exit}, \mathsf{thread_join}\} \\ p[:n-1] + \ell_2 & \text{otherwise} \end{cases}$$

$$\mathsf{entry}_s(p, \ell) = \begin{cases} p' + \ell & p = p' + \ell + p'' \\ p + \ell & \text{otherwise} \end{cases}$$

With $e = (\ell_{src}, \ell_{tgt})$, $\mathsf{op}(e) = \mathsf{func_entry}(arg_1, \ldots, arg_k, par_1, \ldots, par_k)$, and $s_s = (fpm, s_a)$:

$$\mathcal{T}_s[\![e,p]\!](s_s) = (fpm', \mathcal{T}_a[\![e,p]\!](s_a))$$

$$fpm'(par_i) = \begin{cases} arg_i & \mathsf{is_func}(arg_i) \\ fpm(arg_i) & \mathsf{is_func_pointer}(arg_i) \end{cases}$$

Fig. 4.11: Context-, thread-, and flow-sensitive abstract interpretation framework

$$s = s_0 \;\sqcup\; \lambda\, p. \bigsqcup_{p',e \text{ s.t. } \mathsf{np}(p,p',e)}^{s} \mathcal{T}_s[\![e,p']\!](s(p'))$$

With $e = (\ell_{src}, \ell_{tgt})$, $f = \mathsf{func}(\ell_{tgt})$, $\mathsf{op}(e) = \mathsf{thread_entry}(thr, arg, par)$, and $s_s = (fpm, s_a)$:

$$\mathcal{T}_s[\![e,p]\!](s_s) = \begin{cases} (fpm', \mathcal{T}_a[\![e,p]\!](s_a)) & \mathsf{match_fp}(fpm, thr, f) \\ (\emptyset, \bot_a) & \text{otherwise} \end{cases}$$

$$\begin{aligned} &\mathsf{match_fp}(fpm, thr, f) = \\ &\quad (\mathsf{is_func_pointer}(thr) \wedge fpm(thr) \in \{\bot, f\}) \vee \\ &\quad (\mathsf{is_func}(thr) \wedge thr = f) \end{aligned}$$

$$fpm'(par) = \begin{cases} arg & \mathsf{is_func}(arg) \\ fpm(arg) & \mathsf{is_func_pointer}(arg) \end{cases}$$

With $e = (\ell_{src}, \ell_{tgt})$, $\mathsf{op}(e) \in \{\mathsf{func_exit}, \mathsf{thread_exit}, \mathsf{thread_join}\}$, and $s_s = (fpm, s_a)$:

$$\mathcal{T}_s[\![e,p]\!](s_s) = (\emptyset, \mathcal{T}_a[\![e,p]\!](s_a))$$

With $e = (\ell_{src}, \ell_{tgt})$, $\mathsf{op}(e) = \mathsf{op}$, and $s_s = (fpm, s_a)$:

$$\mathcal{T}_s[\![e,p]\!](s_s) = (fpm, \mathcal{T}_a[\![e,p]\!](s_a))$$

Fig. 4.12: Context-, thread-, and flow-sensitive abstract interpretation framework (continued)

$$\begin{aligned} \text{with} \quad \mathsf{np}(p, p', (\ell_1, \ell_2)) \;\Leftrightarrow\; & \ell_1 = \mathsf{top}(p') \wedge \\ & \ell_2 = \mathsf{top}(p) \wedge \\ & \mathsf{next}_s((\ell_1, \ell_2), p') = p \end{aligned}$$

$$\text{with} \quad s \sqcup s' \;=\; \lambda\, p.\, s(p) \sqcup_s s'(p)$$

The equation involves computing the join over all places p' and edges e in the ICFA such that $\mathsf{np}(p, p', e)$.

We next describe the definition of the transfer function of the framework in more detail. The definition consists of four cases: (1) function entry, (2) thread entry, (3) function exit, thread exit, thread join, and (4) intra-function edges.

(1) When applying a function entry edge, a new function pointer map fpm' is created by assigning arguments to parameters and looking up the values of the arguments in the current function pointer map fpm. As in the following cases, the transfer function $\mathcal{T}_a$ of the custom analysis is applied to the state s_a.

(2) Applying a thread entry edge to a state s_s yields one of two outcomes. When the value of the function pointer argument *thr* matches the target of the edge (i.e., the edge enters the same function as the function pointer points to), then the function pointer map is updated with *arg* and *par* producing a new function pointer map fpm' (as in the previous case), and the transfer function of the custom analysis is applied. Otherwise, the result is the bottom element $\bot_s = (\emptyset, \bot_a)$.

(3) The function pointer map is cleared (as its domain contains only parameter identifiers, which are not accessible outside of the function), and the custom transfer function is applied.

(4) The custom transfer function is applied.

If a function pointer *fp* is assigned to or its address is taken, the mapping for *fp* is removed from *fpm*. We take the fact that the address of *fp* was taken as an indication that *fp* might be assigned to via a pointer somewhere in the program. If there is no mapping for *fp* in *fpm*, we assume that *fp* could point to any thread function. This case is omitted from Figures 4.11 and 4.12.

Implementation During the analysis we need to keep a mapping from places to abstract states (which we call the *state map*). However, directly using the places as keys for the state maps in all analyses can lead to high memory consumption. Our implementation therefore keeps a global two-way mapping (shared by all analyses in Figure 4.5) between places and unique IDs for the places (we call this the *place map*). The state maps of the analyses are then mappings from unique IDs to abstract states, and the analyses consult the global place map to translate between places and IDs when needed.

In the two-way place map, the mapping from places to IDs is implemented via a trie, and the mapping from IDs to places via an array that stores pointers back into the trie. The places in a program can be efficiently stored in a trie as many of them share common prefixes.

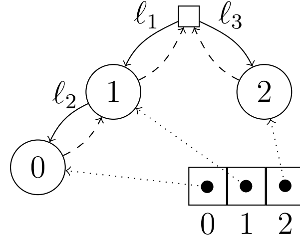


Fig. 4.13: Trie- and array-based place map data structure, representing the two way mapping $(\ell_1) \leftrightarrow 1$, $(\ell_1, \ell_2) \leftrightarrow 0$, $(\ell_3) \leftrightarrow 2$

Figure 4.13 gives an example of a place map which contains the mappings $(\ell_1) \leftrightarrow 1$, $(\ell_1, \ell_2) \leftrightarrow 0$, and $(\ell_3) \leftrightarrow 2$. Unlike in an ordinary trie, in our implementation the nodes also have pointers to their parent (dashed arrows). This allows to reconstruct a place from a pointer to a node in the trie. For example, by starting from the leaf node labelled with 0 we can traverse the parent edges backwards to get the place (ℓ_1, ℓ_2).

4.3.4 Analysis Framework – Flow-Insensitivity

In the flow-insensitive framework, the topmost location of a place always corresponds to the entry point of a function. That is, we associate a sound overapproximation of the data flow facts that hold for all locations in the function with the entry point of the function. Figure 4.14 gives the formalisation of the context- and thread-sensitive flow-insensitive framework. It reuses many of the definitions from the flow-sensitive formalisation in Figures 4.11 and 4.12.

The result of the flow-insensitive analysis is defined as the least fixpoint of the following equation:

$$s = s_0 \sqcup \lambda p.\mathcal{T}_i[\![\mathsf{func}(p), p]\!](s(p)) \sqcup_i \bigsqcup_{i \atop p', e \text{ s.t. } \mathsf{np}(p, p', e)} \mathcal{T}_i[\![e, p']\!](s(p'))$$

Domain: $\mathcal{D}_i = Fpms \times \mathcal{D}_a$

$$s_i^1 \sqcup_i s_i^2 = s_i^1 \sqcup_s s_i^2$$

With $e = (\ell_1, \ell_2)$, $\mathsf{top}(p) = \mathsf{entry_loc}(\ell_1)$, $f = \mathsf{func}(\ell_2)$, and $n = |p|$:

$$\mathsf{next}_i(e,p) = \begin{cases} \mathsf{entry}_s(p, \ell_2) & \mathsf{op}(e) \in \{\mathsf{thread_entry}, \mathsf{func_entry}\} \\ p[:n-2] + \mathsf{entry_loc}(\ell_2) & \mathsf{op}(e) \in \{\mathsf{func_exit}, \mathsf{thread_exit}, \\ & \qquad \mathsf{thread_join}\} \\ p[:n-1] & \text{otherwise} \end{cases}$$

With $E' = \{e_1, \ldots, e_n\}$:

$$\mathcal{T}_i[\![E', p]\!](s) =$$

$$\begin{cases} \mathcal{T}_i[\![E' - \{e_{|E'|}\}, p]\!](\mathcal{T}_i[\![e_{|E'|}, p]\!](s) \sqcup_i s) \sqcup_i (\mathcal{T}_i[\![e_{|E'|}, p]\!](s) \sqcup_i s) & |E'| \geq 1 \\ \bot & E' = \emptyset \end{cases}$$

$$\mathcal{T}_i[\![f, p]\!] = \mathcal{T}_i[\![\{e_i \mid e_i \in E(f)\}, p]\!]$$

$$\mathcal{T}_i[\![e, p]\!](s_i) = \mathcal{T}_s[\![e, p]\!](s_i)$$

Fig. 4.14: Context-, thread-, and flow-insensitive abstract interpretation framework

$$\text{with} \quad \mathsf{np}(p, p', (\ell_1, \ell_2)) \Leftrightarrow \mathsf{func}(\ell_1) \neq \mathsf{func}(\ell_2)$$
$$\mathsf{entry_loc}(\ell_1) = \mathsf{top}(p') \wedge$$
$$\mathsf{entry_loc}(\ell_2) = \mathsf{top}(p) \wedge$$
$$\mathsf{next}_i((\ell_1, \ell_2), p') = p$$

$$\text{with} \quad s \sqcup s' = \lambda p.\, s(p) \sqcup_i s'(p)$$

4.3.5 Pointer Analysis

We use a standard points-to analysis that is an instantiation of the flow-insensitive version of the above framework (see Figure 4.14). It computes for each place an element of $Vars \rightarrow (2^{Objs} \cup \{\{\star\}\})$. That is, the set of possible values of a pointer variable is either a finite set of objects it may point to, or $\{\star\}$ to indicate that it could point to any object. We use $vs(p, a)$ to denote the value set at place p of pointer expression a. The pointer analysis is sound for concurrent programs due to its flow-insensitivity [90].

4.3.6 Lockset Analysis

Our analysis pipeline includes a may lockset analysis (computing for each place the locks that may be held) and a must lockset analysis (computes for each place the locks that must be held). The former is used by the lock graph analysis, and the latter by the non-concurrency analysis. We use $ls_a(p)$, $ls_u(p)$ to denote the may and must locksets at place p.

The may and must lockset analyses are formalised in Figures 4.15 and 4.16 as custom analyses to parametrise the flow-sensitive framework with. We leave off the cases of the transfer function for when it is the identity function (such as for assignment statements). Both the may and must lockset analyses make use of the previously computed points-to information by means of the function $vs()$.

In a concrete program execution, a lock operation would add exactly one lock to the lockset, and an unlock operation would remove exactly one lock from the lockset. The imprecision of the lockset analyses comes from both the control flow joins and the imprecision of the pointer analysis.

For example, in the may lockset analysis, on a lock operation, all the locks that the argument of $\mathsf{lock}(a)$ could point to are added to the lockset (or $\star$ if it could point to any lock). On an $\mathsf{unlock}(a)$ operation, it may not be possible to soundly remove a lock from the lockset. For example, if the lockset before an unlock operation is $\{l_1, l_2\}$, and $\{l_1, l_2\}$ is the set of locks the argument of the unlock operation could point to, then no lock can be removed from the lockset, as it is not known which of the two locks the unlock releases.

The may locksets are later used in the lock graph construction (see Section 4.6.1). Having imprecise may locksets can lead to spurious edges in the lock graph which in

Domain: $2^{Objs} \cup \{\{\star\}\}$

$$s_1 \sqcup s_2 = \begin{cases} s_1 \cup s_2 & \text{if } s_1, s_2 \neq \{\star\} \\ \{\star\} & \text{otherwise} \end{cases}$$

With $\mathsf{op}(e) = \mathsf{lock}(a)$:

$$\mathcal{T}[\![e,p]\!](s) = \begin{cases} s \cup vs(p,a) & \text{if } s, vs(p,a) \neq \{\star\} \\ \{\star\} & \text{otherwise} \end{cases}$$

With $\mathsf{op}(e) = \mathsf{unlock}(a)$:

$$\mathcal{T}[\![e,p]\!](s) = \begin{cases} \emptyset & \text{if } |s| = 1 \wedge s \neq \{\star\} \\ s - vs(p,a) & \text{if } |s \cap vs(p,a)| = 1 \wedge s \neq \{\star\} \wedge vs(p,a) \neq \{\star\} \\ s & \text{otherwise} \end{cases}$$

With $\mathsf{op}(e) \in \{\mathsf{thread_entry}, \mathsf{thread_exit}, \mathsf{thread_join}\}$:

$$\mathcal{T}[\![e,p]\!](s) = \emptyset$$

Fig. 4.15: May lockset analysis. We denote by $vs(p,a)$ the value set of pointer expression a at place p (see Section 4.3.5)

turn can lead to spurious cycles and thus false deadlock reports. The must locksets are used by the non-concurrency analysis (see Section 4.5). Having imprecise must locksets can mean that we will not be able to prune certain infeasible cycles.

4.3.7 Lockset Correctness

In this section, we show the correctness of the may lockset analysis. We show that the may locksets computed for each place overapproximate the sets of locks a thread may hold in a concrete execution at that place.

Domain: $2^{Objs} \cup \{\{\star\}\}$

$$s_1 \sqcup s_2 = s_1 \cap s_2$$

With $\mathsf{op}(e) = \mathsf{lock}(a)$:

$$\mathcal{T}[\![e,p]\!](s) = \begin{cases} s \cup vs(p,a) & \text{if } |vs(p,a)| = 1 \wedge vs(p,a) \neq \{\star\} \\ s & \text{otherwise} \end{cases}$$

With $\mathsf{op}(e) = \mathsf{unlock}(a)$:

$$\mathcal{T}[\![e,p]\!](s) = \begin{cases} s - vs(p,a) & \text{if } vs(p,a) \neq \{\star\} \\ \emptyset & \text{otherwise} \end{cases}$$

With $\mathsf{op}(e) \in \{\mathsf{thread_entry}, \mathsf{thread_exit}, \mathsf{thread_join}\}$:

$$\mathcal{T}[\![e,p]\!](s) = \emptyset$$

Fig. 4.16: Must lockset analysis

Preliminaries

In Section 4.3.3 we have formulated our analyses as a fixpoint computation over the ICFA. An analysis computes data flow facts for each *place* (see the fixpoint equation in Section 4.3.3). The analysis can also be viewed in a slightly different way: as computing a fixpoint over a larger structure that we term *context-sensitive control-flow automaton* (CCFA). The CCFA for a program is just like the ICFA, but with the nodes being places rather than locations. Two places p, p' are connected by an edge if $\mathsf{top}(p)$ and $\mathsf{top}(p')$ are connected by an edge in the corresponding ICFA.

We denote a concrete *execution* (or execution prefix) of a program as an interleaving $E = (p_1, t_1, o_1)(p_2, t_2, o_2) \ldots (p_n, t_n, o_n)$. The p_i are places, the t_i are concrete thread IDs, and the o_i are execution instances of the operations with which the edges connecting the places p_i, p_{i+1} are labelled. We refer to the individual tuples that make up E as *steps*. We use array subscript notation (0-based) to refer

to individual elements of lists and tuples. For example, $E[0][2]$ refers to the third component of the first tuple of execution E. We further use slice notation (e.g., $E[n:m]$) for contiguous subsequences of executions (see Section 4.3.2).

We denote an *execution path* (or an execution path prefix) of a program as $E_p = (p_1, op_1)(p_2, op_2) \dots (p_n, op_n)$. An execution path of a program is a path through its CCFA, starting at the entry point. Here the op_i are the operations with which the edges connecting the places are labelled (rather than concrete instances of those operations).

Given a thread ID t (resp. abstract thread ID t'), we denote by $E|t$ (resp. $E_p|t'$) the executions (resp. execution paths) projected to the steps of the given thread t (resp. abstract thread t'). We further denote by $T(E) = \{t \mid (_, t, _) \in E\}$ the set of threads in execution E.[4] A thread always starts with a thread_entry operation, thus we have $(E|t)[0][2] = \mathsf{thread_entry}(\dots)$ and $(E_p|t')[0][1] = \mathsf{thread_entry}(\dots)$.

Property 25. Let E be an execution prefix and let $E' = E|t$ $(n = |E'|)$ be the execution of some thread t in E. Then, there is an execution path prefix E_p $(m = |E_p|)$ with

$$E_p[m-n:]|_{(p,_)\mapsto p} = E'|_{(p,_,_)\mapsto p}$$

The property holds by the shape of the CCFA (which directly derives from the shape of the ICFA). The property states that for each execution E and thread t in E, there is an execution path prefix E_p such that the sequence of places visited by t and the sequence of places visited by the suffix of E_p are the same.

We next define a concretisation function which states how the result of the pointer analysis (and hence also the static locksets) are interpreted.

Definition 26. Let A be the set of all locks that may be held in any execution of a program. Then:

$$c(ls) = \begin{cases} A & ls = \{\star\} \\ ls & \text{otherwise} \end{cases}$$

The function satisfies the property $c(ls_1) \cup c(ls_2) = c(ls_1 \cup ls_2)$. We are now in a position to prove the correctness of the may lockset analysis.

[4] We use the same notation ($\in$) to denote elements of sets and lists.

May Lockset Correctness

If $E = (p_1, t_1, o_1) \dots (p_n, t_n, o_n)$ is an execution prefix we denote by $ls_c(E)$ the concrete set of locks before executing the final step (p_n, t_n, o_n). This is the set of locks held by thread t_n at that step. A thread starts with an empty set of locks held.

Theorem 27. Let *Prog* be a program and let $E = (p_1, t_1, o_1) \dots (p_n, t_n, o_n)$ be an execution prefix of *Prog*. Then:

$$ls_c(E) \subseteq c(ls_a(p_n)).$$

Proof. Let $t = t_n$ and let $E' = E|t$. By Property 25 there is an execution path E_p that ends in a sequence E'_p which consists of the same sequence of places as E'. We write $ls_c(i)$ for the concrete lockset before executing step i of E'. These locksets result from the execution of the program. We write $ls_a(i)$ for the may lockset before handling step i of E'_p. These locksets are the result of applying the transfer function defined in Figure 4.15.

We next show by induction that the may locksets computed by our analysis for each step along E'_p overapproximate the concrete locksets of E' at the corresponding steps. That is, we show that for all $0 \leq i < |E'|$: $ls_c(i) \subseteq c(ls_a(i))$. Each thread starts with an empty lockset. In our analysis this is reflected by the clause (2) in Figure 4.15. Hence the base case $ls_c(0) = \emptyset \subseteq \emptyset = c(\emptyset) = c(ls_a(0))$ holds.

We next show the induction step via a case distinction based on whether step i is (1) a lock or (2) an unlock.

(1) Let $ls_c(i) \subseteq c(ls_a(i))$ and let $p = E'[i][0]$. We show that then also $ls_c(i+1) \subseteq c(ls_a(i+1))$ after an operation $\mathsf{lock}(a)$. We perform a case distinction over the cases of the definition of $\mathcal{T}[\![.]\!](ls_a(i))$ (see Figure 4.15).

(Case 1) Let l be the concrete lock acquired. By the correctness of the pointer analysis, we have $\{l\} \subseteq c(vs(p, a))$. Therefore:

$$ls_c(i+1) = ls_c(i) \cup \{l\} \subseteq c(ls_a(i)) \cup c(vs(p, a)) \subseteq c(ls_a(i) \cup vs(p, a)) = c(ls_a(i+1))$$

(Case 2) $ls_c(i+1) \subseteq A = c(\{\star\})$ holds since A is the set of all locks.

(2) Let $ls_c(i) \subseteq c(ls_a(i))$ and let $p = E'[i][0]$. We show that then also $ls_c(i+1) \subseteq c(ls_a(i+1))$ after an operation $\mathsf{unlock}(a)$. We perform a case distinction over the cases of the definition of $\mathcal{T}[\![.]\!](ls_a(i))$ (see Figure 4.15).

(Case 1) Since $|ls_a(i)| = 1$ and $ls_a(i) \neq \{\star\}$, we also have $|ls_c(i)| = 1$. Therefore, $ls_c(i+1) = \emptyset \subseteq \emptyset = c(\emptyset) = c(ls_a(i+1))$.

(Case 2) Let l be the concrete lock released. Thus we have $l \in c(ls_a(i))$ and $l \in c(vs(p,a))$. Since $|ls_a(i) \cap vs(p,a)| = 1$ we have that $c(ls_a(i) \cap vs(p,a)) = \{l\}$. Thus, $c(ls_a(i) - vs(p,a)) = c(ls_a(i)) - \{l\}$. Therefore, $ls_c(i+1) = ls_c(i) - \{l\} \subseteq c(ls_a(i)) - \{l\} = c(ls_a(i) - vs(p,a)) = c(ls_a(i+1))$.

(Case 3) $ls_c(i+1) \subseteq ls_c(i)$ and thus $ls_c(i+1) \subseteq c(ls_a(i)) = c(ls_a(i+1))$

Thus, we have shown that the may lockset is an overapproximation of the concrete lockset at any step along E'_p, and thus in particular also at the final step of E'_p (i.e., at the place associated with the final step of E'_p). We can now use the properties of data flow analyses to complete the proof. First, since the may lockset computed for the final place of E_p is an overapproximation of the concrete lockset, it follows that the "meet over all paths" (MOP) at this place is an overapproximation of the concrete locksets for all concrete executions that might reach that place.

Second, the analysis given in Figure 4.15 consists of a finite lattice with top element $\{\star\}$, join function $\sqcup$, and a monotonic transfer function. Consequently, the minimal fixpoint solution (MFP) of the data flow equations overapproximates the MOP solution. Therefore, since the MOP solution is sound, the MFP solution is also sound. □

4.4 Dependency Analysis

We have developed a context-insensitive, flow-insensitive *dependency analysis* to compute the set of assignments and function calls that might affect the value of a given set of expressions (in our case the expressions used in lock-, unlock-, create-, and join-statements). The purpose of the analysis is to speed up the following pointer analysis phase (cf. Figure 4.5).

Below we first describe a semantic characterisation of dependencies between expressions and assignments, and then devise an algorithm to compute dependencies based on syntax only (specifically, the variable identifiers occuring in the expressions/assignments).

Semantic characterisation of dependencies Let $AS = \{e \in E(Prog) \mid \mathsf{is_assign}(\mathsf{op}(e))\}$ be the set of assignment edges. Let *exprs* be a set of starting expressions. Let further $R(a), W(a)$ denote the set of memory locations that an expression or assignment a may read (resp. write) over *all* possible executions of the program. Let further $M(a) = R(a) \cup W(a)$. Then we define the immediate dependence relation *dep* as follows (with $*$ denoting transitive closure and ; denoting composition):

$$dep_1 \subseteq exprs \times AS, (a, b) \in dep_1 \Leftrightarrow R(a) \cap W(b) \neq \emptyset$$
$$dep_2 \subseteq AS \times AS, (a, b) \in dep_2 \Leftrightarrow R(a) \cap W(b) \neq \emptyset$$
$$dep = dep_1; dep_2^*$$

If $(a, b) \in dep_1$, then the evaluation of expression a may read a memory location that is written to by assignment b. If $(a, b) \in dep_2$, then the evaluation of the assignment a may read a memory location that is written to by the assignment b. If $(a, b) \in dep$, this indicates that the expression a can (transitively) be influenced by the assignment b. We say *a depends on b* in this case.

The goal of our dependency analysis is to compute the set of assignments $A = dep|_{(_,a)\mapsto a}$ (the binary relation A projected to the second component). However, we cannot directly implement a procedure based on the definitions above as this would require the functions $R()$, $W()$ to return the memory locations accessed by the expressions/assignments. This in turn would require a pointer analysis–the very thing we are trying to optimise.

Thus, in the next section, we outline a procedure for computing the relation *dep* which relies on the symbols (i.e., variable identifiers) occuring in the expressions/assignments rather then the memory locations accessed by them.

Computing dependencies In this section, we outline how we can compute an overapproximation of the set of assignments A as defined above. Let $\mathsf{symbols}(a)$ be a function that returns the set of variable identifiers occuring in an expression/assign-

ment. For example, $\mathsf{symbols}(\texttt{a[i]->lock}) = \{\texttt{a}, \texttt{i}\}$ and $\mathsf{symbols}(\texttt{*p=q+1}) = \{\texttt{p}, \texttt{q}\}$. As stated in Section 4.3.1, in our program representation all variable identifiers in a program are unique. We first define the relation sym_2 which indicates whether two assignments have common symbols:

$$sym_2 \subseteq AS \times AS$$
$$(a, b) \in sym_2 \Leftrightarrow \mathsf{symbols}(a) \cap \mathsf{symbols}(b) \neq \emptyset$$

Our analysis relies on the following property: If two assignments a, b can access a common memory location (i.e., $M(a) \cap M(b) \neq \emptyset$), then $(a, b) \in sym_2^*$. This can be seen as follows. Whenever a memory region/location is allocated in the C programming language it initially has at most one associated identifier. For example, the memory allocated for a global variable `x` at program startup has initially just the associated identifier `x`. Similarly, memory allocated via, e.g., `a = (int *)malloc(sizeof(int) * NUM)` has initially only the associated identifier `a`. If an expression not mentioning `x`, such as `*p`, can access the associated memory location, then the address of `x` must have been propagated to `p` via a sequence of assignments such as `q=&x`, `s->f=q`, `p=s->f`, with each of the adjacent assignments having common variables. Thus, if a, b can access a common memory location, then both must be "connected" to the initial identifier associated with the location via such a sequence. Thus, in particular, a, b are also connected. Therefore, $(a, b) \in sym_2^*$.

We next define the *sym* relation which also incorporates the starting expressions:

$$sym_1 \subseteq exprs \times AS$$
$$(a, b) \in sym_1 \Leftrightarrow \mathsf{symbols}(a) \cap \mathsf{symbols}(b) \neq \emptyset$$
$$sym = sym_1; sym_2^*$$

As we will show below we have $dep \subseteq sym$ and thus also $A = dep|_{(_,a) \mapsto a} \subseteq sym|_{(_,a) \mapsto a}$. Thus, if we compute *sym* above we get an overapproximation of A.

The fact that $dep \subseteq sym$ can be seen as follows. Let $(a, b) \in dep$. Then there are $a_1, a_2, \ldots, a_n, b$ such that $(a_1, a_2) \in dep_1 \cup dep_2, (a_2, a_3) \in dep_2, \ldots, (a_n, b) \in dep_2$. Let (a', a'') be an arbitrary one of those pairs. Then $R(a) \cap W(b) \neq \emptyset$ by the definition of dep_1 and dep_2. Thus $M(a) \cap M(b) \neq \emptyset$. As we have already argued above, if two expressions/assignments can access the same memory location then

Algorithm 1: Dependency analysis

Input : ICFA *Prog*, start edges *start_edges*
Output : Set of affecting edges A

```
1  A ← affecting_edges(Prog, start_edges)
2  F ← {f | e ∈ A ∧ f = func(src(e))}
3  F_h ← ∅
4  while F ≠ ∅ do
5      remove f from F
6      F_h ← F_h ∪ {f}
7      E ← {e | func(tgt(e)) = f ∧
                 op(e) ∈ {func_entry, thread_entry}}
8      for e ∈ E do
9          A ← A ∪ {e}
10         f' ← func(src(e))
11         if f' ∉ F_h then
12             F ← F ∪ {f'}
13 return A
```

they must transitively share symbols. Thus $(a', a'') \in sym_1 \cup sym_2^*$ must hold. Therefore, since we have chosen (a', a'') arbitrarily, we have that all of the pairs above are contained in $sym_1 \cup sym_2^*$ and thus by the definition of *sym* and in particular the transitivity of sym_2^* we get $(a, b) \in sym$.

Thus, we can use the definition of *sym* above to compute an overapproximation of the set of assignments that can affect the starting expressions as defined semantically in the previous section.

Algorithm Algorithm 1 gives our dependency analysis. The first phase (line 1), Algorithm 2) is based on the ideas from the previous section. It computes the set of edges that can affect the given set of starting edges. It first computes the set of sets R which contains for each edge a set which contains the symbols mentioned by this edge (lines 4–9). Then line 10 assigns to NM a map that maps unique integers to sets in R. Then line 11 assigns to SM a map that maps symbols to those numbers corresponding to sets in R in which the respective symbols occur. For example, if we have $R = \{\{x, y\}, \{z\}\}$, then we would get $NM = \{0 \mapsto \{x, y\}, 1 \mapsto \{z\}\}$ and $SM = \{x \mapsto 0, y \mapsto 0, z \mapsto 1\}$. The purpose of this numbering is to have a compact representation of SM to guarantee linear runtime of the algorithm.

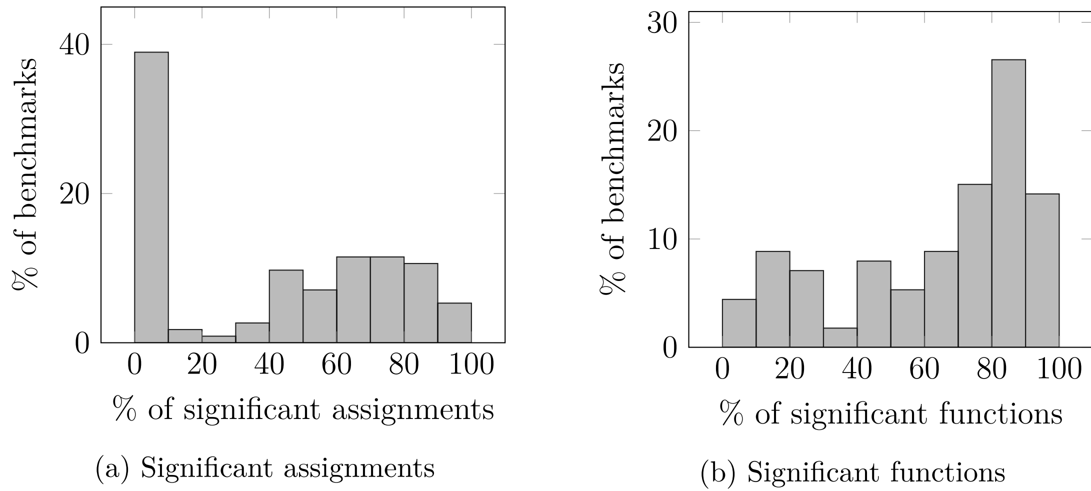
Then in lines 12–20, the algorithm propagates the set of symbols to compute transitive dependencies. The sets N_h, S_h store the numbers and symbols that have been handled already. Finally, lines 21–24 select the assignment-, func_exit-, and thread_join-edges that share symbols with those encountered during the propagation phase. This set of edges is returned.

In the second phase the algorithm additionally determines the func_entry and thread_entry edges that could lead to an edge determined in the previous phase. The ability to prune function calls has a potentially big effect on the performance of the analysis, as it can greatly reduce the amount of code that needs to be analysed. In the following section we evaluate the performance and effectiveness of the dependency analysis. Its effect on the overall analysis is evaluated in Section 4.7.

Evaluation We have evaluated the dependency analysis on a subset of 100 benchmarks of the benchmarks given in Section 4.7. For each benchmark the dependency analysis was invoked with the set of starting expressions *exprs* being those occuring in lock operations or as the first argument of create and join operations.

The average time (over the 100 benchmarks) to perform the dependency analysis was 0.18 s. The 25th and 75th percentile of the dependency analysis runtimes were 0.03 s and 0.34 s. Thus, for 25% of the benchmarks it took 0.03 s or less to perform the dependency analysis, and for 25% of the benchmarks it took 0.34 s or more to perform the dependency analysis. The data shows that the dependency analysis is very efficient in terms of runtime.

The histograms in Figure 4.17 shows the effectiveness of the dependency analysis. Figure 4.17a shows what percentage of assignments in the programs could affect the starting expressions. The x axis ranges from 0 to 100% with bins that are 10% wide each. The histogram can be interpreted as follows. The first bar in Figure 4.17a, for example, indicates that for 39% of the benchmarks (y axis), the number of assignments that could affect the starting expressions was between 0 and 10% (x axis). Thus, for 39% of the benchmarks, at least 90% of the assignments could be pruned from the program. Thus, the data shows that often the number of significant assignments was very low. This happens when the lock usage patterns in the program are simple, such as when using simple lock expressions (like `pthread_mutex_lock(&mutex)`) that refer



(a) Significant assignments

(b) Significant functions

Fig. 4.17: Dependency analysis effectiveness

to global locks with simple initialisations (such as using static initialisation via `PTHREAD_MUTEX_INITIALIZER`).

Figure 4.17b shows the percentage of functions classified as significant by the dependency analysis. For example, the rightmost bar indicates that for 14% of the benchmarks, between 90 and 100% of the functions were classified as significant. Overall, the data shows that the analysis is cheap and able to prune a significant number of assignments and functions.

Algorithm 2: Affecting edges

```
1  function affecting_edges(Prog, start_edges)
2      A ← start_edges
3      R ← ∅
4      for e ∈ E(Prog) do
5          op ← op(e)
6          if op = (a = b) ∨
               op = thread_entry(_, a, b) ∨
               op = func_exit(a, b) ∨
               op = thread_join(a, b) then
7              R ← R ∪ {symbols(a) ∪ symbols(b)}
8          else if op = func_entry(arg_1, ..., arg_n,
                                   par_1, ..., par_n) then
9              R ← R ∪ {symbols(arg_i) ∪ symbols(par_i) |
                        i ∈ {1, ..., n}}
10     NM ← number_map(R)
11     SM ← symbol_map(R)
12     S ← ⋃_{e∈start_edges} symbols(op(e))
13     N_h, S_h ← ∅, ∅
14     while S ≠ ∅ do
15         remove s from S
16         S_h ← S_h ∪ {s}
17         for n ∈ SM[s] do
18             if n ∉ N_h then
19                 N_h ← N_h ∪ {n}
20                 S ← S ∪ (NM[r] − S_h)
21     for e ∈ E(Prog) do
22         if op = (a = b) ∨
               op = func_exit(a, b) ∨
               op = thread_join(a, b) then
23             if ((symbols(a) ∪ symbols(b)) ∩ S_h) ≠ ∅ then
24                 A ← A ∪ {e}
25     return A
```

4.5 Non-Concurrency Analysis

We have implemented an analysis (Algorithm 3) to compute whether two places p_1, p_2 are non-concurrent. That is, the analysis determines whether the statements associated with the places p_1, p_2 (i.e., the operations with which the outgoing edges of $\mathsf{top}(p_1), \mathsf{top}(p_2)$ are labelled) cannot execute concurrently in the contexts embodied by p_1, p_2.

Algorithm 3: Non-concurrency analysis

Input : places p_1, p_2, must locksets ls_u^1, ls_u^2
Output : *true* if p_1, p_2 are non-conc., *false* otherwise

1 **if** $p_1 = p_2$ **then**
2   **return** *true*
3 **if** $c(ls_u^1) \cap c(ls_u^2) \neq \emptyset$ **then**
4   **return** *true*
5 $i \leftarrow |\mathsf{common_prefix}(p_1, p_2)|$
6 $r_1, r_2 \leftarrow \mathit{true}, \mathit{true}$
7 $\ell_1, \ell_2 \leftarrow p_1[i], p_2[i]$
8 **if** $\mathsf{has_path}(\ell_1,\ \ell_2)$ **then**
9     $r_1 \leftarrow \mathit{unwind}(i, p_1, \ell_1, \ell_2)$
10 **if** $\mathsf{has_path}(\ell_2,\ \ell_1)$ **then**
11     $r_2 \leftarrow \mathit{unwind}(i, p_2, \ell_2, \ell_1)$
12 **return** $r_1 \wedge r_2$

Whether the places are protected by a common lock is determined by computing the intersection of the must locksets (lines 3–4). If the intersection is non-empty they cannot execute concurrently and the algorithm returns *true*. This approach is similar to the one described by Havelund [50], except that we statically compute the set of locks that must be held at a place p, whereas Havelund does dynamic analysis and deals with exact locksets associated with concrete program executions.

If the common locks check yields *false*, the algorithm proceeds to check whether the places are non-concurrent due to create and join operations. This is done via a graph search in the ICFA. First the length of the longest common prefix of p_1 and p_2 is determined (line 5). This is the starting point for the ICFA exploration. If there is a path from ℓ_1 to ℓ_2, it is checked that all the threads that are created to reach place p_1 are joined before location ℓ_2 is reached (and same for a path from ℓ_2

```
1  int main()
2  {
3    pthread_t tid1;
4    pthread_t tid3;
5    pthread_create(
6      &tid1, 0, thread1, 0);
7    pthread_join(tid1);
8    pthread_create(
9      &tid3, 0, thread3, 0);
10   return 0;
11 }
```

```
12 void *thread1() {
13   pthread_t tid2;
14   pthread_create(
15     &tid2, 0, thread2, 0);
16   pthread_join(tid2, 0);
17   return 0;
18 }

19 void *thread2() {
20   x = 1;
21   return 0;
22 }

23 void *thread3() {
24   x = 2;
25   return 0;
26 }
```

Fig. 4.18: Statements `x = 1` and `x = 2` are non-concurrent

to ℓ_1). This check is performed by the procedure *unwind*(), given in Algorithm 4. It makes use of the function function in_loop(ℓ) which returns true when there is a path within the same function in the ICFA that starts and ends in ℓ.

The procedure *unwind*() uses a function *find*() to search for a join that joins a thread that was created earlier. It is shown in Algorithm 5. It makes use of two basic functions on directed graphs. The function has_path(ℓ_1, ℓ_2) returns true when there is a path in the ICFA within the same function from ℓ_1 to ℓ_2. The function on_all_paths(ℓ_1, ℓ_2, ℓ_3) returns true when all paths in the ICFA within the same function from ℓ_1 to ℓ_3 pass through ℓ_2. It is implemented by computing the set of dominators of ℓ_3 (assuming ℓ_1 to be the entry point) and then checking whether ℓ_2 is contained in that set.

Algorithm We explain the algorithm on an example (Figure 4.18). The example consists of four threads (including the main thread). We want to determine whether the statements `x = 1` and `x = 2` are non-concurrent. We see that they cannot run concurrently as `main()` joins with `thread1()` before starting `thread3()` and `thread1()` joins with `thread2()` before returning.

Algorithm 4: Unwind

1 **function** $unwind(i, p, \ell_1, \ell_2)$
2 $\quad create_locs \leftarrow \{\ell \mid \exists e = (\ell, _) \colon \mathsf{op}(e) = \mathsf{thread_entry}\}$
3 $\quad joined \leftarrow true$
4 $\quad$ **while** $|p| > i + 1$ **do**
5 $\quad\quad \ell \leftarrow \mathsf{top}(p)$
6 $\quad\quad f \leftarrow \mathsf{func}(\ell)$
7 $\quad\quad$ **if** $\ell \notin create_locs \wedge joined$ **then**
8 $\quad\quad\quad p \leftarrow \mathsf{pop}(p)$
9 $\quad\quad\quad$ **continue**
10 $\quad\quad$ **if** $\ell \in create_locs$ **then**
11 $\quad\quad\quad$ **if** $\neg joined$ **then**
12 $\quad\quad\quad\quad$ **return** *false*
13 $\quad\quad\quad joined \leftarrow false$
14 $\quad\quad\quad p_c \leftarrow p$
15 $\quad\quad p \leftarrow \mathsf{pop}(p)$
16 $\quad\quad joined \leftarrow find(p_c, p, \ell, \mathsf{exit_loc}(f))$
17 $\quad\quad$ **if** $\mathsf{in_loop}(\ell)$ **then**
18 $\quad\quad\quad loop_joined \leftarrow find(p_c, p, \ell, \ell)$
19 $\quad\quad\quad$ **if** $\neg joined \vee \neg loop_joined$ **then**
20 $\quad\quad\quad\quad$ **return** *false*
21 $\quad$ **if** $\ell_1 \in create_locs$ **then**
22 $\quad\quad$ **if** $\neg joined$ **then**
23 $\quad\quad\quad$ **return** *false*
24 $\quad\quad joined \leftarrow false$
25 $\quad\quad p_c \leftarrow p$
26 $\quad$ **if** $\neg joined$ **then**
27 $\quad\quad p \leftarrow \mathsf{pop}(p)$
28 $\quad\quad joined \leftarrow find(p_c, p, \ell_1, \ell_2)$
29 $\quad\quad$ **if** $\mathsf{in_loop}(\ell_1)$ **then**
30 $\quad\quad\quad loop_joined \leftarrow find(p_c, p, \ell_1, \ell_1)$
31 $\quad\quad\quad$ **return** $joined \wedge loop_joined$
32 $\quad$ **return** *joined*

Algorithm 5: Find join

```
1  function find(p_c, p, ℓ_1, ℓ_2)
2      f ← func(ℓ_1)
3      join_locs ← {ℓ ∈ L(f) | ∃e = (ℓ, _): is_join(op(e))}
4      foreach ℓ_join ∈ join_locs do
5          if on_all_paths(ℓ_1, ℓ_join, ℓ_2) ∧ match(p_c, p + ℓ_join) then
6              return true
7      entry_edges ← {e = (ℓ_src, ℓ_tgt) | func(ℓ_src) = f ∧ op(e) = func_entry}
8      foreach (ℓ_src, ℓ_tgt) ∈ entry_edges do
9          if on_all_paths(ℓ_1, ℓ_src, ℓ_2) then
10             p' ← p + ℓ_src
11             f' ← func(ℓ_tgt)
12             r ← find(
                       p_c,
                       p',
                       ℓ_tgt,
                       exit_loc(f'))
13             if r then
14                 return true
15     return false
```

Let us now look at how our algorithm establishes this fact. The algorithm is called with places $p_1 = (5, 14, 20)$ and $p_2 = (8, 24)$. We have no locks in the example and hence the must locksets are empty (line 3). Line 5 determines the length of the longest common prefix of p_1 and p_2 (which is 0 in this case). This is the starting point for the exploration.

The algorithm then checks whether there is a path from 5 to 8 (line 8). If there is no path from either 5 to 8 or 8 to 5, then 5 and 8 occur in conflicting branches (e.g., one in the then- and the other in the else-branch of an if statement) and thus the places cannot not be concurrent. In the current case there is a path from 5 to 8.

The algorithm then invokes *unwind*(), which checks that the threads that are created to reach place p_1 are all joined before location 8 is reached. It does so by iterating over p starting from the end. The operation $\mathsf{top}(p)$ returns the last element of p, and the operation $\mathsf{pop}(p)$ returns p with the last element removed.

The variable *joined* indicates whether the last thread that was created has been joined yet. If the top element of p corresponds to a create operation (line 10), then if *joined* is *false* the function returns *false*. If not, then *joined* is set to *false*, and the place corresponding to the create is recorded in p_c. Then, a matching join for the create is searched (line 16) by invoking the *find*() function.

The function $\mathit{find}(p_c, p, \ell_1, \ell_2)$ takes the place p_c, a place p, and locations ℓ_1 and ℓ_2. The locations ℓ_1 and ℓ_2 are in the same function (let $f = \mathsf{func}(l_1)$), and the place p has as top element the call to the function f. The function *find*() looks for a matching join to the create at place p_c. It does so by looking in the function f (lines 3–6), and (recursively) in the callees of f (lines 7–14). The join must occur on all paths between ℓ_1 and ℓ_2 (lines 5, 9). The call $\mathit{match}(p_c, p+\ell_{join})$ checks that the join at place $p + \ell_{join}$ matches the create at p_c (i.e., the thread ID returned by the `pthread_create()` is the same as the one passed to the `pthread_join()`). The function *match*() makes use of the pointer information computed earlier.

If the creation site is in a loop, we additionally ensure that the thread is joined also on each path that goes back to the same location (lines 17–20). The final lines 21–31 are like the loop body and handle the locations ℓ_1 and ℓ_2.

For our example, for the first iteration of the while loop in line 4 we have $p = (5, 14, 20)$. In this case, $20 \notin \mathit{create_locs} \wedge \mathit{joined}$ in line 7 and we thus continue with the next iteration. Now we have $p = (5, 14)$ and $14 \in \mathit{create_locs}$ and thus set *joined* to *false* and record $p_c = (5, 14)$. The invocation of *find*() (line 16) finds the join in line 16, and thus *joined* is set to *true*. The while loop then terminates as $|(5)| \leq i + 1$. Lines 21–31 then look for a matching join for the create at location 5. Again such a join is found and *unwind*() returns *true*. The algorithm thus overall returns *true*.

Evaluation We have evaluated the non-concurrency analysis on a subset of 100 benchmarks of the benchmarks described in Section 4.7. For each benchmark we randomly selected 1000 pairs of places (p_1, p_2) such that p_1 and p_2 correspond to different threads. We then performed the non-concurrency check for each of the 1000 pairs. The results are given in the table below.

	runtime	n.c. places	n.c. lock places
25th percentile	0.14 s	47.3%	10.8%
arithmetic mean	4.07 s	60.0%	42.7%
75th percentile	4.56 s	78.9%	66.7%

Fig. 4.19: Non-concurrency analysis runtime and effectiveness

The table shows that the average time (over all benchmarks) it took to perform 1000 non-concurrency checks was 4.07 s. The first and last line give the 25th and 75th percentile. This indicates for example that for 25% of the benchmarks it took 0.14 s or less to perform 1000 non-concurrency checks. The third and fourth column evaluate the effectiveness of the non-concurrency analysis. The third column shows that on average our analysis classified 60% of the place pairs as non-concurrent. The fourth column gives the same property while only regarding places that correspond to lock operations. The number of places classified as non-concurrent is lower in this case, which is expected as the code portions using locks are those that can run concurrently with others. Overall, the data shows that the non-concurrency analysis is both fast and effective.

4.6 Lock Graph Analysis

Our lock graph analysis consists of two phases. In the first phase, we build a lock graph based on the may lockset analysis and the pointer analysis. In the second phase, we search for cycles in the graph, while disregarding cycles that are infeasible due to information from the non-concurrency analysis.

4.6.1 Lock Graph Construction

A *lock graph* is a directed graph $L \in 2^{\mathit{Objs}^* \times P \times \mathit{Objs}^*}$ (with $\mathit{Objs}^* = \mathit{Objs} \cup \{\star\}$). Each node is a $\mathit{lock} \in \mathit{Objs}^*$, and an edge $(\mathit{lock}_1, p, \mathit{lock}_2) \in \mathit{Objs}^* \times P \times \mathit{Objs}^*$ from lock_1 to lock_2 is labelled with the place p of the $\mathsf{lock}()$ operation that acquired lock_2 while lock_1 was owned by the same thread $\mathsf{get_thread}(p)$. Hence, the directed edges indicate the order of lock acquisition. Figure 4.20 gives the lock graph for the example program in Figure 4.7.

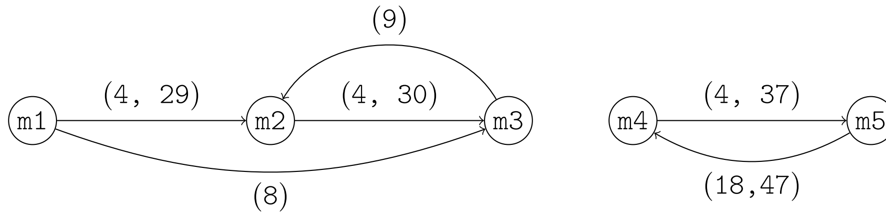


Fig. 4.20: Lock graph for the program in Figure 4.7 (`t`, `m`, and `f` are shorthand for `thread()`, `main()`, and `func2()`)

We use the result of the may lockset analysis (Section 4.3.6) to build the lock graph. Figure 4.21 gives the lock graph domain that is instantiated in our analysis framework. For each lock() operation in place p a thread may acquire a lock $lock_2$ corresponding to the value set of the argument to the lock operation. This happens while the thread may own any lock $lock_1$ in the lockset at that place. Therefore we add an edge $(lock_1, p, lock_2)$ for each pair $(lock_1, lock_2)$.

Finally, we have to handle the indeterminate locks, denoted by $\star$. We compute the closure $cl(L)$ of the graph with respect to edges that involve $\star$ by adding edges from all predecessors of the $\star$ node to all other nodes, and to each successor node of the $\star$ node, we add edges from all other nodes. The set get_locks(s) denotes all the locks in the lock graph s (not including $\star$).

As can be seen in Figure 4.21, the precision of the lock graph depends on the precision of the may lockset analysis and the precision of the pointer analysis. Overapproximate locksets $ls_a(p)$ and value sets $vs(p, a)$ can lead to spurious edges in the lock graph which in turn can lead to spurious cycles.

4.6.2 Checking Cycles in the Lock Graph

The final step is to check the lock graph for cycles. Each cycle could be a potential deadlock. Self-loops correspond to potential self-deadlocks, i.e., cases where a thread attempts to acquire a lock it already holds.

Cycles consisting of two or more edges correspond to deadlocks between two or more threads. A cycle c is a potential deadlock between two or more threads if

Domain: $2^{Objs^* \times P \times Objs^*}$

$$s_1 \sqcup s_2 = s_1 \cup s_2$$

With $\mathsf{op}(e) = \mathsf{lock}(a)$:

$$\mathcal{T}[\![e, p]\!](s) = s \cup \{(lock_1, p, lock_2) \mid lock_1 \in ls_a(p), lock_2 \in vs(p, a)\}$$

$$\begin{aligned} cl(s) = s \cup \{&(lock_1, p, lock) \mid (lock_1, p, \star) \in s, lock \in \mathsf{get_locks}(s)\} \cup \\ &\{(lock, p, lock_2) \mid (\star, p, lock_2) \in s, lock \in \mathsf{get_locks}(s)\} \end{aligned}$$

Fig. 4.21: Lock graph construction

$|c| > 1 \wedge \mathsf{all_concurrent}(c)$ where

$$\begin{aligned} &\mathsf{all_concurrent}(c) \Leftrightarrow \\ &\quad \forall (lock_1, p, lock_2), (lock'_1, p', lock'_2) \in c : \\ &\qquad \neg\mathsf{non_concurrent}(p, p') \vee \\ &\qquad (\mathsf{get_thread}(p) = \mathsf{get_thread}(p') \wedge \\ &\qquad \mathsf{multiple_thread}(\mathsf{get_thread}(p))) \end{aligned}$$

and $\mathsf{multiple_thread}(t)$ means that abstract thread t is created in a loop or recursion. Owing to the use of our non-concurrency analysis we do not require any special treatment of gate locks or thread segments as in the work of Agarwal et al. [10].

4.6.3 Lock Graph Correctness

We now show the correctness of our lock graph by showing that it overapproximates all the potential LAGs. That is, we show that for each execution prefix of the program for which there exists a cycle in the LAG, there also is a cycle in the lock graph. We assume the correctness of the pointer analysis, the non-concurrency analysis, and the may lockset analysis (which we have shown to be correct in Section 4.3.7).

As discussed in Section 4.1.1, during a concrete execution of a program, each step of the execution has an associated LAG. The LAG has two types of nodes: threads and locks. There is an edge from a lock node to a thread node if the lock is assigned to that thread (assignment edge). There is an edge from a thread node to a lock node if the thread has requested the lock (a request edge). If the LAG at a certain step in the execution has a cycle, then the involved threads have deadlocked. Those threads cannot make any more steps (but other threads might).

We next show that if the program has a self-deadlock (corresponding to a cycle in the LAG consisting of two edges and involving one thread and one lock), then our lock graph has a self-loop. We first show a lemma that we will use in the proof of the subsequent theorem:

Lemma 28. Let ls_1, ls_2 be nonempty static locksets with $l \in c(ls_1), l \in c(ls_2)$. Let further $L = \{(lock_1, p, lock_2) \mid lock_1 \in ls_1, lock_2 \in ls_2\}$. Then:

$$\exists lock\colon (lock, p, lock) \in cl(L)$$

Proof.

(1) Assume $lock \in ls_1, ls_2$ ($lock$ may be $\star$): Then by the definition of L above, $(lock, p, lock) \in L \subseteq cl(L)$.
(2) Assume $ls_1 = \{\star\}$, $lock \in ls_2$, $lock \neq \star$: Then $(\star, p, lock) \in L$ and by the definition of $cl()$, $(lock, p, lock) \in cl(L)$.
(3) $lock \in ls_1, lock \neq \star, ls_2 = \{\star\}$: analogous to the case above. □

Theorem 29. Let *Prog* be a program with an execution prefix E. Let further the LAG after the final step of E have a cycle involving one thread t and one lock l. Then the lock graph of *Prog* has a self-loop.

Proof. Let $(l, t), (t, l)$ be the two edges in the LAG forming the cycle. Let (p, t, o) be the final step of t in E corresponding to the $\mathsf{lock}(exp)$ operation on which t is blocked. By the correctness of the may lockset analysis and the pointer analysis we have $l \in c(ls_a(p))$ and $l \in c(vs(p, exp))$. Thus, the lock graph analysis adds the set of edges $L = \{(lock_1, p, lock_2) \mid lock_1 \in ls_a(p), lock_2 \in vs(p, exp)\}$ to the lock graph. By Lemma 28, it follows that there is a lock such that $(lock, p, lock) \in cl(L)$. Thus, the lock graph of *Prog* has a self-loop. □

We next show the whenever there is a cycle in the LAG involving two or more threads, then there also is a cycle c in the lock graph such that $\mathsf{all_concurrent}(c)$ holds.

Lemma 30. Let ls_1, ls_2, ls_3, ls_4 be nonempty static locksets, and let p, p' be places. Let further $l \in c(ls_2)$ and $l \in c(ls_3)$. Let $L = \{(lock_1, p'', lock_2) \mid (lock_1 \in ls_1 \wedge lock_2 \in ls_2 \wedge p'' = p) \vee (lock_1 \in ls_3 \wedge lock_2 \in ls_4 \wedge p'' = p')\}$. Then:

$$\forall lock_1 \in ls_1, lock_2 \in ls_4 \colon \exists lock \colon (lock_1, p, lock), (lock, p', lock_2) \in cl(L)$$

Proof.

(1) Assume $lock \in ls_2, lock \in ls_3$ ($lock$ may be $\star$). Then, by the definition of L above, for all locks $lock_1 \in ls_1, lock_2 \in ls_4, (lock_1, p, lock), (lock, p', lock_2) \in L \subseteq cl(L)$.
(2) Assume $ls_2 = \{\star\}, lock \in ls_3, lock \neq \star$. Then for all locks $lock_1 \in ls_1, lock_2 \in ls_4, (lock_1, p, \star), (lock, p', lock_2) \in L$. Then in $cl(L)$ there is an edge $(lock_1, p, lock)$ by the definition of $cl()$.
(3) Assume $lock \in ls_2, lock \neq \star, ls_3 = \{\star\}$. This case is symmetric to (2). □

We next show a lemma about the definition of $\mathsf{all_concurrent}()$.

Lemma 31. Let E be an execution prefix, let G be the LAG at its final step, and let c be a cycle in G. Let $t_1, \ldots, t_n$ be the threads involved in the cycle c, and let $(p_1, t_1, o_1), \ldots, (p_n, t_n, o_n)$ be last steps of each thread involved in c in E. Then for all p_i, p_j:

$$\neg\mathsf{non_concurrent}(p_i, p_j) \vee (\mathsf{get_thread}(p_i) = \mathsf{get_thread}(p_j) \wedge \mathsf{multiple_thread}(p_i))$$

Proof. Since in E all steps $(p_1, t_1, o_1), \ldots, (p_n), t_n, o_n)$ could reach a $\mathsf{lock}()$ operation on which they blocked, they must have been able to run concurrently in this execution. Now for two places $p_i, p_j, \mathsf{get_thread}(p_i) \neq \mathsf{get_thread}(p_j)$, it follows that $\neg\mathsf{non_concurrent}(p_i, p_j)$ by the correctness of the non-concurrency analysis.

Now assume $\mathsf{get_thread}(p_i) = \mathsf{get_thread}(p_j)$. In this case the places have the same abstract thread ID but they occur in *different* concrete threads. This occurs when a thread create operation occurs in a loop or a recursion (or the call to the function that invokes the thread creation operation occurs in a loop or recursion, etc.). Then we have $\mathsf{multiple_thread}(p_i)$. □

We can now show the main theorem stating the soundness of our lock graph and cycle search.

Theorem 32. Let *Prog* be a program with an execution prefix E. Let further the LAG after the final step of E have a cycle involving two or more threads. Then the lock graph of *Prog* has a cycle c' consisting of two or more edges and for which all_concurrent(c') holds.

Proof. Let c denote a cycle in the LAG after the final step of E. In this cycle, every thread and every lock occurs exactly once. Let t, t' be two threads involved in the cycle such that there are edges $(l_1, t), (t, l_2), (l_2, t'), (t', l_3)$, for locks l_1, l_2, l_3 (we might have that $l_1 = l_3$).

Let $n = |(E|t)|$, $m = |(E|t')|$, and let $(p, t, o) = (E|t)[n-1]$, $(p', t', o') = (E|t')[m-1]$ be the last steps of $E|t$, $E|t'$. The steps o, o' are lock() operations, i.e., $o = \mathsf{lock}(exp_1\colon l_2), o' = \mathsf{lock}(exp_2\colon l_4)$ with expression exp_1 referring to l_2 and expression exp_2 referring to l_4. That is, l_2 is the lock requested by o, and l_4 is the lock requested by o'. Moreover, l_1 is in the lockset ls_c at the last step of $E|t$, and l_2 is in the lockset ls'_c at the last step of $E|t'$.

Applying the transfer function to the lock edge starting at p adds edges from the elements of $ls_a(p)$ to the elements of $vs(p, exp_1)$ to the lock graph L. Applying the transfer function to the lock edge starting at p' adds edges from the elements of $ls_a(p')$ to the elements of $vs(p', exp_2)$ to the lock graph L. By the correctness of the may lockset analysis and the correctness of the pointer analysis we have $l_2 \in c(vs(p, exp_1)), l_2 \in c(ls_a(p')))$. Therefore, by Lemma 30, it follows that for all locks $lock_1 \in ls_a(p), lock_2 \in vs(p', exp_2)$, there is a lock *lock* such that $(lock_1, p, lock), (lock, p', lock_2) \in cl(L)$.

Thus, in the previous paragraph, we have shown that for any portion of the cycle c consisting of adjacent threads t, t' and edges $(l_1, t), (t, l_2), (l_2, t'), (t', l_3)$, there is a portion $(lock_1, p, lock), (lock, p', lock_2)$ in the lock graph. Therefore, the lock graph also has a cycle c'. The places p, p' are the places associated with the final steps of the threads in E that are involved in the cycle c in the LAG. Hence, by Lemma 31, it follows that all_concurrent(c'). □

	max	avg	min
# Lines of code	41,749	11,401.7	86
# Threads	163	3.8	1
# Lock operations	30,773	417.4	0
# Precise lock operations	100%	65.1%	0%
# Indeterminate locking operations	100%	8.6%	0%
Size of largest lockset	8.0	1.5	0
# Non-concurrency checks	11,043.0	132.8	0.0

Fig. 4.22: Benchmark characteristics (max., min., and mean over all benchmarks)

4.7 Experiments

We implemented our deadlock analyser as a pipeline of static analyses in the CPROVER framework. The tool and benchmarks are available online [2]. We performed experiments to support the following hypothesis: *Our analysis handles real-world C code in a precise and efficient way.*

We used 997 concurrent C programs that contain locks from the Debian GNU/Linux distribution, with the characteristics shown in Figure 4.22.[5] The first line gives the number of lines of code, the second line gives the number of abstract threads, and the third line gives the number of lock operations encountered during the context- and thread-sensitive traversal of the ICFA. The table shows that the minimum number of different locks and lock operations encountered by our analysis was 0. We found that this is due to a small number of benchmarks on which the lock operations were not reachable from the main function of the program (i.e., they were contained in dead code).

We additionally selected 8 programs and introduced deadlocks in them. This gives us a benchmark set consisting of 1005 benchmarks with a total of 11.4 MLOC. Of these, 997 benchmarks are assumed to be deadlock-free, and 8 benchmarks are known to have deadlocks. The experiments were run on an Intel Xeon X5667 at 3 GHz running Fedora 20 with 64-bit binaries. Memory and CPU time were restricted to 24 GB and 1800 seconds per benchmark, respectively.

[5] Lines of code were measured using `cloc` 1.53.

KLOC	analy.	proved (self)	proved (≥ 2)	proved (all)	alarms (self)	alarms (≥ 2)	alarms (all)	t/o	m/o
0–5	250	93	131	93	93	55	93	44	20
5–10	272	67	91	67	59	35	59	104	42
10–15	152	18	22	18	17	13	17	95	22
15–20	181	26	28	26	8	6	8	98	49
20–50	142	16	20	16	9	5	9	112	5

Fig. 4.23: Results for the deadlock-free programs

Results We correctly report deadlocks for the 8 benchmarks with known deadlocks. The results for the deadlock-free programs are shown in Figure 4.23, grouped by benchmark size. The second column (labelled "analys.") indicates how many benchmarks we analysed in total in each group. The columns labelled "t/o" and "m/o" indicate on how many benchmarks our tool timed out or ran out of memory. We separately report the results for self-deadlocks (labelled "self") and deadlocks involving at least two threads (labelled ≥ 2). The columns labelled with "all" include both self-deadlocks and deadlocks involving two or more threads. For example, the first line in Figure 4.23, corresponding to programs consisting of less than 5000 lines of code, shows that we managed to prove absence of self-deadlocks for 93 out of 250 programs, and absence of deadlocks involving two or more threads for 131 out of 250 programs.

Over all the 997 deadlock-free benchmarks, we report spurious alarms regarding self-deadlocks in 186 cases, and spurious alarms regarding deadlocks involving two or more threads in 114 cases. A possible useful addition to our tool would be the ability to rank deadlock reports according to our confidence in that it corresponds to a real deadlock. This would increase the usability of our tool on programs on which it yields a large number of deadlock reports. Our analysis already computes some information that could be used for this purpose. For example, if a cycle involves only lock operations for which we have precise pointer information, then it is more likely that it corresponds to a real deadlock than if the cycle would include indeterminate lock operations.

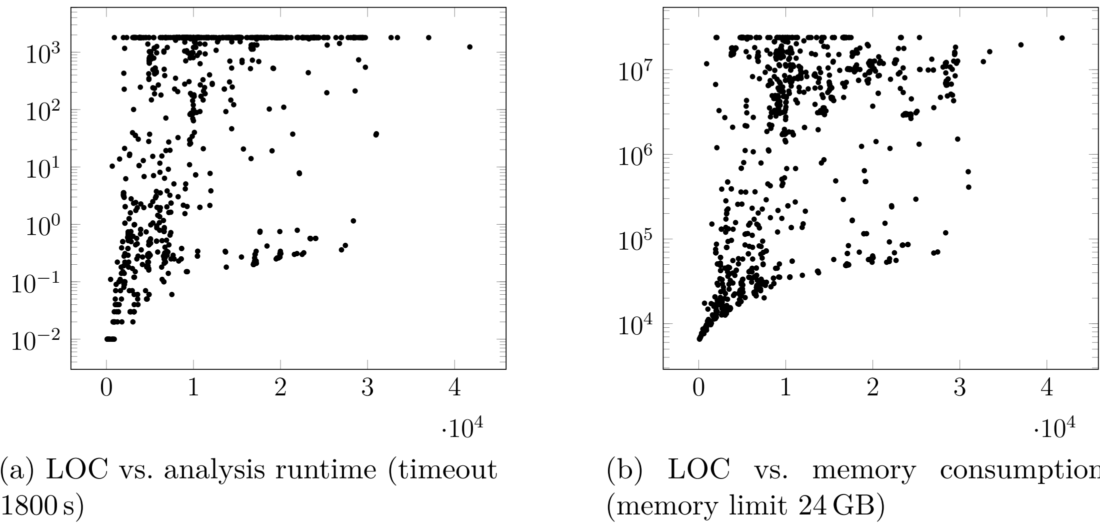
The scatter plots in Figures 4.25a and 4.25b illustrate how the tool scales in terms of runtime and memory consumption with respect to the number of lines of code. The runtime and memory consumption are shown on a logarithmic scale.

	max	avg	min
Total analysis time (s)	1,800.0	80.54	0.00
Dependency analysis	2.8	0.02	0.00
Pointer analysis	1,710.7	2.15	0.00
May lockset analysis	346.3	0.16	0.00
Must lockset analysis	211.4	0.06	0.00
Lock graph construction	259.8	0.06	0.00
Cycles detection	318.8	0.00	0.00
Peak memory (GB)	24.0	7.70	0.007

Fig. 4.24: Analysis statistics

As the plots show, the asymptotic behaviour of the analysis in terms of lines of code is difficult to predict. However, on average, the runtime and memory consumption appear to grow exponentially with the number of lines of code. A reason for this may be that our analysis is fully context-sensitive. It has been shown that a context-sensitive analysis can have exponentially worse runtime and memory consumption compared to a non-context-sensitive analysis [111]. A way of improving performance while retaining precision would be to employ selective context-sensitivity [88]. A pre-analysis tailored to the main analysis would estimate the precision loss of analysing certain parts of the code in a non-context-sensitive way. The main analysis would then use this information to decide when to turn on and off its context sensitivity during the analysis.

We evaluated the impact of the different features of our analysis on a random selection of 61 benchmarks and break down the running times into the different analysis phases on those benchmarks where the tool does not time out or goes out of memory. We found that the dependency analysis is effective at decreasing both the memory consumption and the runtime of the pointer analysis. It decreased the memory consumption by 67% and the runtime by 81% on average. We observed that still the vast majority of the running time (80%) of our tool is spent in the pointer analysis. May lock analysis (7.5%), must lock analysis (5.5%), and lock graph construction (5.5%) take less time. The run times for the dependency analysis and the cycle checking (up to the first potential deadlock) are negligible. In 89 benchmarks, the non-concurrency analysis refuted infeasible cycles. However, all of these programs had other feasible cycles.



(a) LOC vs. analysis runtime (timeout 1800 s)

(b) LOC vs. memory consumption (memory limit 24 GB)

Fig. 4.25: Experimental results

4.8 Related Work

Deadlocks were characterised as cycles in state graphs by Coffman et al. [34]. Holt [53] introduced a different graph representation with nodes for both threads and resources. Since the mid-1990s, numerous tools, mainly for C and Java, have been developed. One can distinguish dynamic and static approaches. A common deficiency of many of the existing tools is that they are neither sound nor complete, and thus produce both false positives and false negatives.

Dynamic tools Java PathFinder [10, 50] finds lock acquisition hierarchy violation with the help of lock graphs, detects common locks, and uses segmentation techniques to handle different threads running in parallel at different times. [11, 60, 98] try to predict deadlocks in executions similar to the observed one. DeadlockFuzzer [60] use a fuzzing technique to search for deadlocking executions. Multicore SDK [73] tries to reduce the size of lock graphs by clustering locks, and Magiclock [28] implements improvements of the cycle detection algorithms. Helgrind [3] is a popular open source dynamic deadlock detection tool, and there are many commercial implementations of dynamic deadlock detection algorithms.

Static tools There are few static analysis tools to find deadlocks in C programs. LockLint [4] is a semi-automatic lightweight approach that relies on user-supplied

lock acquisition orders. RacerX [42] focuses on fast analysis of large code bases rather than precision. It performs a path- and context-sensitive analysis, but its pointer analysis is very rudimentary. Several model checking tools also allow to find deadlocks (among other errors). The tools Lazy-CSeq [55] and MU-CSeq [102] use context-bounded sequentialisation, and analyse the resulting program with the bounded model checker CBMC [33]. ESBMC [36] also uses context-bounding and encodes each interleaving as an SMT formula. Both the versions of CSeq and ESBMC are unsound for defined programs due to context-bounding and loop unrolling, but in contrast to our tool are also able to detect some undefined behaviours. The CIVL model checker [94, 114] explores all execution paths and interleavings of a program. It symbolically represents the program state and queries a constraint solver to check state reachability.

For Java there is Jlint2 [19], a tool similar to LockLint. The tool Jade [80] consciously uses a may analysis instead of a must analysis, which causes unsoundness. The tools presented in [110] and [103] do not consider gatelock scenarios.

Other tools Some tools combine dynamic analysis and constraint solving. For example, CheckMate [59] applies model checking to a path along an observed execution of a multi-threaded Java program, Sherlock [43] uses concolic testing, and [11, 98] monitor runtime executions of Java programs. There are related techniques to detect synchronisation defects due to blocking communication, e.g., in message passing (MPI) programs [32, 47], or for the modelling languages BIP (DFinder tool [20, 21]) or ABS (DECO tool [45]) which use similar techniques based on lock graphs and may-happen-in-parallel information.

Dependency analysis Our dependency analysis is related to work on concurrent program slicing [63, 64, 81] and alias analysis [27, 61]. Our analysis is more lightweight than existing approaches as it works on the level of variable identifiers only, as opposed to more complex objects such as program dependence graphs (PDG) or representations of possible memory layouts. Moreover, our analysis disregards expressions occuring in control flow statements (such as if-statements) as these are not relevant to the following pointer analysis which consumes the result of the dependency analysis. The analysis thus does not produce an *executable subset* of the program statements as in the original definition of slicing by Weiser [108].

Non-concurrency analysis Our non-concurrency analysis is context-sensitive, works on-demand, and can classify places as non-concurrent based on locksets or create/join. Locksets have been used in a similar way in static data race detection [42], and Havelund [10] used locksets in dynamic deadlock detection to identify non-concurrent lock statements. Our handling of create/join is most closely related to the work of Albert et al. [12]. They consider a language with asynchronous method calls and an await statement that allows to wait for the completion of a previous call. Their analysis works in two phases, the second of which can be performed on-demand, and also provides a form of context-sensitivity. Other approaches, which however do not work on-demand, include the work of Masticola and Ryder [77] and Naumovich et al. [82] for ADA, and the work of Lee et al. [67] for async-finish parallelism.

Chapter 5

Conclusions

We have addressed several aspects of correctness in concurrent programs that use locks for synchronisation.

In Chapter 2, we have presented a framework to test the memory behaviour of GPU programs. Knowing which behaviours can occur on hardware is vital for the correct implementation of locks. Our approach is based on automatically generating tests that correspond to various concurrency idioms. Executing those tests on the hardware reveals interesting properties of the underlying memory model. We have shown how to employ a high-level language (in our case PTX), which might be subject to optimisations, to test the hardware behaviour. This requires checking that the produced binary code conforms to the high-level code. Our testing has revealed both hardware issues (such as a read coherency violation) and bugs in application code. We have studied two incorrect spinlock implementations from the literature and have used our framework to experimentally confirm that our changes fix the respective bugs.

While we targeted Nvidia GPUs and PTX in this thesis, the framework could be extended to support GPUs of other vendors and other assembly languages. Another possible line of future work would be to automatically extract the concurrency idioms present in a GPU application. Then, the test results would indicate whether the application is vulnerable to certain relaxations of memory consistency.

In Chapter 3, we have presented a new theory of thread refinement. The theory is based on matching state transitions between lock operations. We have shown that, in addition to optimisations that do not change the state transitions between lock operations, the theory also supports optimisations that reorder memory

accesses across lock operations (such as roach motel reorderings). We also have applied our theory in compiler testing. Our evaluation demonstrates that we can check whether two traces match significantly faster than a previous approach.

The theory of thread refinement could potentially be adapted to other memory models. For example, the release and acquire operations of C11/C++11 are semantically very similar to lock operations. It would thus also be possible to represent the semantics of a thread as a set of transition traces in this model. We have applied our theory in a testing tool in this thesis. However, it could also form the basis of verification approaches. For example, the transition traces and the matching of those traces could be encoded symbolically, similar to bounded model checking constraints.

In Chapter 4, we have presented an approach to verify deadlock-freedom of C/Pthreads programs. The approach is based on a pipeline of several static analyses. We have shown how to soundly handle a variety of sources of imprecision, such as may-point to information or thread creation in loops or recursions. We have applied our tool to a sizeable number of real-world concurrent C programs, and succeeded to prove deadlock-freedom in a large number of cases. We further demonstrated that employing a lightweight dependency analysis can significantly improve performance, reducing the analysis time by about factor five on average. We identified the pointer analysis as the main limitation of our approach: it takes most of the time and its imprecision is the cause for most of the false positives.

Our analysis is fully context-sensitive. A way to reduce both the memory consumption and runtime would be to decrease the number of calling contexts that the analysis explores. The challenge is to retain precision. A potential solution would be to employ selective context-sensitivity. A lightweight pre-analysis would determine the functions that could likely be analysed in a non-context-sensitive way without significant precision loss. Then, the analysis results for different invocations of those functions would be merged during the fixpoint computation.

References

[1] http://multicore.doc.ic.ac.uk/gpu-litmus.

[2] http://www.cprover.org/deadlock-detection.

[3] http://valgrind.org/info/tools.html#helgrind.

[4] http://developers.sun.com/solaris/articles/locklint.html.

[5] https://github.com/danpoe/gpu-tools.

[6] http://diy.inria.fr/doc/gen.html.

[7] GPUBench, June 2014. http://graphics.stanford.edu/projects/gpubench.

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

[9] S. V. Adve and M. D. Hill. Weak ordering – a new definition. In *ISCA*, 1990.

[10] R. Agarwal, S. Bensalem, E. Farchi, K. Havelund, Y. Nir-Buchbinder, S. D. Stoller, S. Ur, and L. Wang. Detection of deadlock potentials in multithreaded programs. *IBM Journal of Research and Development*, 54(5):3, 2010.

[11] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *Workshop on Parallel and Distributed Systems: Testing, Analysis*, pages 51–60. ACM, 2006.

[12] E. Albert, A. Flores-Montoya, and S. Genaim. Analysis of may-happen-in-parallel in concurrent objects. In *FMOODS/FORTE*, volume 7273 of *LNCS*, pages 35–51. Springer, 2012.

[13] J. Alglave, M. Batty, A. F. Donaldson, G. Gopalakrishnan, J. Ketema, D. Poetzl, T. Sorensen, and J. Wickerson. GPU concurrency: Weak behaviours and programming assumptions. In *ASPLOS*, 2015.

[14] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models. In *CAV*, 2010.

[15] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Litmus: Running tests against hardware. In *TACAS*, 2011.

[16] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, 2012.

[17] J. Alglave, L. Maranget, and M. Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *TOPLAS*, 36(2), 2014.

[18] ARM Ltd. *Cortex-A9 MPCore, Programmer Advice Notice, Read-after-Read Hazards*, 2011.

[19] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded Java programs. In *Australian Software Engineering Conference*, pages 68–75. IEEE, 2001.

[20] S. Bensalem, M. Bozga, T. Nguyen, and J. Sifakis. D-Finder: A tool for compositional deadlock detection and verification. In *CAV*, volume 5643 of *LNCS*, pages 614–619. Springer, 2009.

[21] S. Bensalem, A. Griesmayer, A. Legay, T. Nguyen, and D. Peled. Efficient deadlock detection for concurrent systems. In *Formal Methods and Models for Codesign*, pages 119–129. IEEE, 2011.

[22] H.-J. Boehm. Reordering constraints for pthread-style locks. In *PPoPP*, 2007.

[23] H.-J. Boehm and S. V. Adve. Foundations of the C++ concurrency memory model. In *PLDI*, 2008.

[24] S. Brookes. Full abstraction for a shared variable parallel language. In *LICS*, 1993.

[25] S. Brookes. A grainless semantics for parallel programs with shared mutable data. *ENTCS*, 155:277–307, 2006.

[26] S. Brookes. On grainless footprint semantics for shared-memory programs. *ENTCS*, 308:65–86, 2014.

[27] M. G. Burke, P. R. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC*, volume 892 of *LNCS*, pages 234–250. Springer, 1995.

[28] Y. Cai and W. K. Chan. Magiclock: Scalable detection ofpotential deadlocks in large-scale multithreaded programs. *IEEE Trans. Software Eng.*, 40(3):266–281, 2014.

[29] S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs. In *VMCAI*, pages 119–135, 2012.

[30] S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). *FMSD*, 47(3):287–301, 2015.

[31] S. Chakraborty and V. Vafeiadis. Validating optimizations of concurrent C/C++ programs. In *CGO*, pages 216–226, 2016.

[32] Z. Chen, X. Li, J. Chen, H. Zhong, and F. Qin. SyncChecker: detecting synchronization errors between MPI applications and libraries. In *International Parallel and Distributed Processing Symposium*, pages 342–353. IEEE, 2012.

[33] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[34] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[35] W. W. Collier. *Reasoning about parallel architectures*. Prentice-Hall, Inc., 1992.

[36] L. C. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE*, pages 331–340. ACM, 2011.

[37] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.

[38] M. Dodds, M. Batty, and A. Gotsman. Compositional verification of relaxed-memory program transformations. In *Under submission*, 2016.

[39] M. Dubois, C. Scheurich, and F. A. Briggs. Memory access buffering in multiprocessors. In *ISCA*, pages 434–442, 1986.

[40] L. Effinger-Dean, H.-J. Boehm, D. Chakrabarti, and P. Joisha. Extended sequential reasoning for data-race-free programs. In *MSPC*, 2011.

[41] E. Eide and J. Regehr. Volatiles are miscompiled, and what to do about it. In *Embedded Software (EMSOFT)*, pages 255–264. ACM, 2008.

[42] D. R. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *Symposium on Operating Systems Principles*, pages 237–252. ACM, 2003.

[43] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Foundations of Software Engineering*, pages 353–365. ACM, 2014.

[44] W. Feng and S. Xiao. To GPU synchronize or not GPU synchronize? In *International Symposium on Circuits and Systems (ISCAS*, pages 3801–3804, 2010.

[45] A. Flores-Montoya, E. Albert, and S. Genaim. May-happen-in-parallel based deadlock analysis for concurrent objects. In *FMOODS/FORTE*, volume 7892 of *LNCS*, pages 273–288. Springer, 2013.

[46] I. O. for Standardization. C++ standard, 2011. ISO/IEC 14882:2011.

[47] V. Forejt, D. Kroening, G. Narayanaswamy, and S. Sharma. Precise predictive analysis for discovering communication deadlocks in MPI programs. In *Formal Methods*, volume 8442 of *LNCS*, pages 263–278. Springer, 2014.

[48] B. Godlin and O. Strichman. Regression verification. In *DAC*, pages 466–471, 2009.

[49] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *STVR*, 23(3):241–258, 2013.

[50] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *SPIN*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.

[51] B. He and J. X. Yu. High-throughput transaction executions on graphics processors. In *VLDB*, 2011.

[52] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*. ACM, 2002.

[53] R. C. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3):179–196, Sept. 1972.

[54] International Organization for Standardization. C standard, 2011. ISO/IEC 9899:2011.

[55] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-programs. In *ASE*, pages 807–812. IEEE, 2015.

[56] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* MIT Press, 2012.

[57] R. Jagadeesan, G. Petri, and J. Riely. Brookes is relaxed, almost! In *FoSSaCS*, 2012.

[58] P. G. Joisha, R. S. Schreiber, P. Banerjee, H. J. Boehm, and D. R. Chakrabarti. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. In *POPL*, 2011.

[59] P. Joshi, M. Naik, K. Sen, and D. Gay. An effective dynamic analysis for detecting generalized deadlocks. In *Foundations of Software Engineering*, pages 327–336. ACM, 2010.

[60] P. Joshi, C. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI*, pages 110–120. ACM, 2009.

[61] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239. Springer, 2007.

[62] Khronos OpenCL Working Group. The OpenCL specification (version 2.0), November 2013.

[63] J. Krinke. Static slicing of threaded programs. In *PASTE*, pages 35–42. ACM, 1998.

[64] J. Krinke. Context-sensitive slicing of concurrent programs. In *ESEC/FSE*, pages 178–187. ACM, 2003.

[65] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound static deadlock analysis for C/Pthreads. In *ASE*, 2016.

[66] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *TC*, 100(9), 1979.

[67] J. K. Lee, J. Palsberg, R. Majumdar, and H. Hong. Efficient may happen in parallel analysis for async-finish parallelism. In *SAS*, volume 7460 of *LNCS*, pages 5–23. Springer, 2012.

[68] H. Liang, X. Feng, and M. Fu. A rely-guarantee-based simulation for verifying concurrent program transformations. In *POPL*, 2012.

[69] H. Liang, X. Feng, and Z. Shao. Compositional verification of termination-preserving refinement of concurrent programs. In *LICS*, 2014.

[70] H. Liang, J. Hoffmann, X. Feng, and Z. Shao. Characterizing progress properties of concurrent objects via contextual refinements. In *CONCUR*, 2013.

[71] A. Lochbihler. Verifying a compiler for Java threads. In *ESOP*, 2010.

[72] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[73] Z. D. Luo, R. Das, and Y. Qi. Multicore SDK: A practical and efficient deadlock detector for real-world applications. In *International Conference on Software Testing, Verification and Validation*, pages 309–318. IEEE, 2011.

[74] S. Mador-Haim, R. Alur, and M. M. K. Martin. Generating litmus tests for contrasting memory consistency models. In *CAV*, pages 273–287, 2010.

[75] S. Mador-Haim, R. Alur, and M. M. K. Martin. Litmus tests for comparing memory consistency models: how long do they need to be? In *DAC*, pages 504–509, 2011.

[76] J. Manson, W. Pugh, and S. V. Adve. The Java memory model. In *POPL*, 2005.

[77] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *PPoPP*, pages 129–138. ACM, 1993.

[78] P. Misra and M. Chaudhuri. Performance evaluation of concurrent lock-free data structures on GPUs. In *International Conference on Parallel and Distributed Systems, (ICPADS)*, pages 53–60, 2012.

[79] R. Morisset, P. Pawan, and F. Zappa Nardelli. Compiler testing via a theory of sound optimisations in the C11/C++11 memory model. In *PLDI*, 2013.

[80] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering*, pages 386–396. IEEE, 2009.

[81] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA*, pages 180–190. ACM, 2000.

[82] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *FSE*, pages 24–34. ACM, 1998.

[83] Nvidia. Parallel thread execution ISA, version 4.0. http://docs.nvidia.com/cuda/parallel-thread-execution.

[84] Nvidia. Cuobjdump - Application Note, 2012.

[85] Nvidia. CUDA C programming guide, version 5.5, July 2013.

[86] Nvidia. Parallel thread execution ISA, version 3.2, July 2013. http://docs.nvidia.com/cuda/pdf/ptx_isa_3.2.pdf.

[87] Nvidia. GeForce GTX 750 Ti - Whitepaper, 2014.

[88] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *PLDI*, 2014.

[89] D. Poetzl and D. Kroening. Formalizing and checking thread refinement for data-race-free execution models. In *TACAS*, 2016.

[90] M. C. Rinard. Analysis of multithreaded programs. In *SAS*, volume 2126 of *LNCS*, pages 1–19. Springer, 2001.

[91] J. Sanders and E. Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, chapter Appendix A.1: Dot Product Revisited. Addison Wesley, 2010.

[92] S. Sarkar, P. Sewell, J. Alglave, L. Maranget, and D. Williams. Understanding POWER multiprocessors. In *Programming Language Design and Implementation (PLDI)*, pages 175–186, 2011.

[93] S. Sarkar, P. Sewell, F. Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M. O. Myreen, and J. Alglave. The semantics of x86-CC multiprocessor machine code. In *POPL*, 2009.

[94] S. F. Siegel, M. Zheng, Z. Luo, T. K. Zirkel, A. V. Marianiello, J. G. Edenhofner, M. B. Dwyer, and M. S. Rogers. CIVL: The concurrency intermediate verification language. In *SC*, pages 61:1–61:12. ACM, 2015.

[95] R. L. Sites. *Alpha Architecture Reference Manual*. Digital Press, 1992.

[96] T. Sorensen. Towards shared memory consistency models for GPUs. Bachelor's thesis, University of Utah, 2013.

[97] T. Sorensen and A. F. Donaldson. Exposing errors related to weak memory in GPU applications. In *PLDI*, 2016.

[98] F. Sorrentino. PickLock: A deadlock prediction approach under nested locking. In *SPIN*, volume 9232 of *LNCS*, pages 179–199. Springer, 2015.

[99] W. Stallings. *Operating Systems: Internals and Design Principles*. Pearson, 9th edition, 2017.

[100] J. A. Stuart and J. D. Owens. Efficient synchronization primitives for GPUs. *Computing Research Repository (CoRR)*, abs/1110.4623, 2011. http://arxiv.org/pdf/1110.4623.pdf.

[101] The Open Group and IEEE. The open group base specifications issue 7, 2013. IEEE Std 1003.1-2008/Cor 1-2013.

[102] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. Verifying concurrent programs by memory unwinding. In *TACAS*, LNCS, pages 551–565. Springer, 2015.

[103] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, 2004.

[104] J. Ševčík. Safe optimisations for shared-memory concurrent programs. In *PLDI*, 2011.

[105] J. Ševčík and D. Aspinall. On validity of program transformations in the Java memory model. In *ECOOP*, 2008.

[106] J. Ševčík, V. Vafeiadis, F. Zappa Nardelli, S. Jagannathan, and P. Sewell. CompCertTSO: A verified compiler for relaxed-memory concurrency. *JACM*, 60(3), 2013.

[107] D. L. Weaver and T. Gremond. *The SPARC architecture manual.* PTR Prentice Hall Englewood Cliffs, NJ 07632, 1994.

[108] M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE, 1981.

[109] J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides. Automatically comparing memory consistency models. In *POPL*, 2017.

[110] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *European Conference on Object-Oriented Programming*, volume 3586 of *LNCS*, pages 602–629. Springer, 2005.

[111] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *PLDI*, 1995.

[112] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS)*, pages 235–246, 2010.

[113] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *PLDI*, 2011.

[114] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel. CIVL: formal verification of parallel programs. In *ASE*, pages 830–835. IEEE, 2015.

Appendix A

GPU Testing Tools

In the following, we give a brief tutorial on the usage of the toolchain to test the memory model of Nvidia GPUs. The tools gpu-diy, optcheck, and the log analysis tools are available online [5]. The gpu-litmus tool is available upon request from Tyler Sorensen.

A.1 Generating Tests

The gpu-diy tool can be used to generate litmus tests for GPUs. The gpu-diy tool builds on the diy tool for CPUs, and shares with it the concept of generating tests from cycles formed of edges which represent potential relaxations of memory consistency. A comprehensive tutorial of this approach can be found online [6]. The theoretical background of the diy approach is described in a paper by Alglave et al. [14].

When invoking gpu-diy, one needs to specify a set of edge types which represent potential relaxations. The tool then enumerates all the cycles that can be formed with those edges. Each cycle corresponds to an execution that is not sequentially consistent. Then, gpu-diy generates a test from each cycle. Each such test checks whether an execution exhibiting that cycle is possible on the tested hardware. Let us now generate a set of Nvidia PTX tests:

```
gpu-diy -gpu-safe 'Po-d-Wgl2-Rgl2 Fr-c-gl2l2'
```

The `-gpu-safe` option takes a set of relaxation edges. The invocation produces two variants of the classical store buffering (SB) test. One of those tests is shown in

```
1  GPU_PTX SB
2
3  0:.reg .s32 r0;
4  0:.reg .s32 r2;
5  0:.reg .b64 r1 = x;
6  0:.reg .b64 r3 = y;
7  1:.reg .s32 r0;
8  1:.reg .s32 r2;
9  1:.reg .b64 r1 = y;
10 1:.reg .b64 r3 = x;
11
12 P0                : P1
13 mov.s32 r0,1      : mov.s32 r0,1
14 st.cg.s32 [r1],r0 : st.cg.s32 [r1],r0
15 ld.cg.s32 r2,[r3] : ld.cg.s32 r2,[r3]
16
17 (device (kernel (cta (warp P0)) (cta (warp P1))))
18
19 x: global, y: global
20
21 0:r2=0 /\ 1:r2=0
```

Fig. A.1: Store buffering litmus test

Figure A.1. We refer to Section 2.3.1 for an explanation of the litmus test format. When executed with the gpu-litmus tool, the test checks whether the specified final state can be observed. This state can only be reached by a non-SC execution.

The relaxations supported by gpu-diy are similar to those of the CPU version (see the diy online tutorial [6] for more information), but each of them is enriched with additional information. This is to account for GPU features such as concurrency scopes and different memory spaces that are not present on CPUs. The possible relaxations can be listed via `gpu-diy -gpu-list`. It prints a comma separated list of relaxations.

Let us look at the relaxations given to the gpu-diy invocation above in more detail. `Po-d-Wgl2-Rgl2` is a program order relaxation (indicated by the prefix `Po`). It specifies that the memory accesses it relates target different memory locations (`d`). The source of the relaxation is a write (`W`), which targets global memory (`g`), and has a level 2 cache operator (`l2`; see the PTX manual [83] for more details

on cache operators). The target of the relaxation is a read (R), which also targets global memory (g), and has a level 2 cache operator (l2).

The Fr-c-gl2l2 is a from-read (Fr) relaxation. It belongs to the communication relations, which can relate events in different threads. On Nvidia GPUs, threads can be in different warps, CTAs, kernels, or devices. The c portion specifies that the relaxation relates two events in threads that are in different CTAs. An Fr relaxation always relates a read event to a write event, both of which target the same memory location. The g portion specifies that both events access global memory, and both the read and the write have a level 2 cache operator (l2 and l2, respectively).

A.2 Testing

The automatically generated litmus tests, or handwritten tests, can be run on a GPU with gpu-litmus. It adds a test harness around the litmus test code which runs the test many times on the hardware (100 000 times by default) while stressing the memory system in order to provoke weak behaviours [13]. For each generated binary, gpu-litmus calls optcheck.py and passes the binary to it. The optchec.py tool then verifies that the test code has not been transformed by the compiler in a way that could introduce weak behaviours. The gpu-litmus tool can be invoked as follows:

```
gpu-litmus -gpu sb.litmus -s 1000
```

This will run the litmus test in the file sb.litmus 1000 times and print the results. The tool has additional options that allow to specify how to stress the memory system. The full list of options can be viewed via gpu-litmus -h.

A typical use case of gpu-litmus is to run a large number of automatically generated litmus tests and append all output to a log file. The log files can then be processed with the two tools log2log.py and log2tbl.py. The former can be used to compare logs resulting from testing different GPUs. For example, the following invocation prints all the tests on which the first GPU (with testing log log1.txt) showed a weak outcome but the second GPU (with testing log log2.txt) did not:

```
log2log.py cmp -weaker log1.txt log2.txt
```

The full list of options can be viewed via `log2log.py -h`. The tool log2tbl.py can be used to produce HTML or LaTeX tables from testing logs. The list of options can be viewed via `log2tbl.py -h`.

Appendix B

Deadlock Analysis Tool

Our deadlock analysis tool and the set of benchmark programs are available online [2]. In the following we give a brief tutorial about how to use the tool. It is implemented on top of the CPROVER framework. To analyse a C program with a tool from the CPROVER toolsuite, it first has to be compiled to a goto binary. A goto binary is a file containing a goto program, the intermediate program representation of the CPROVER framework. A goto program consists of a set of functions, and each function consists of a list of instructions. Each instruction is identified by a location number.

Consider the program `test.c` in Figure B.1. We can compile the program to a goto binary using `goto-cc`:

```
goto-cc -o test.gb test.c
```

The resulting goto program can be viewed with `goto-instrument`:

```
goto-instrument --show-goto-functions test.gb
```

The `goto-instrument` tool of the CPROVER toolsuite has various options for static analysis as well as instrumentation of goto programs. The full list of options can be viewed via `goto-instrument -h`. Our deadlock analysis can be invoked via the new option `-show-deadlocks`:

```
goto-instrument --show-deadlocks test.gb
```

For the program given in Figure B.1, it produces the output given in Figure B.2. In lines 11 and 13, the two locks in the program are given. In line 26, the tool reports that the program contains one potential deadlock. The subsequent lines give the

```
1  #include <pthread.h>
2
3  pthread_mutex_t m1, m2;
4  int x;
5
6  void *thread1(void *arg);
7  void *thread2(void *arg);
8
9  int main()
10 {
11   pthread_t tid1, tid2;
12
13   pthread_create(&tid1, 0,
14     thread1, 0);
15   pthread_create(&tid2, 0,
16     thread2, 0);
17
18   pthread_join(tid1, 0);
19   pthread_join(tid2, 0);
20
21   return 0;
22 }
```

```
23 void *thread1(void *arg)
24 {
25   pthread_mutex_lock(&m1);
26   pthread_mutex_lock(&m2);
27   ++x;
28   pthread_mutex_unlock(&m2);
29   pthread_mutex_unlock(&m1);
30
31   return 0;
32 }

33 void *thread2(void *arg)
34 {
35   pthread_mutex_lock(&m2);
36   pthread_mutex_lock(&m1);
37   ++x;
38   pthread_mutex_unlock(&m1);
39   pthread_mutex_unlock(&m2);
40
41   return 0;
42 }
```

Fig. B.1: Program `test.c` containing a deadlock

associated cycle `0 -> 1 -> 0` in the lock graph. As described in Section 4.6.1, each edge is annotated with a place. A place is a sequence of program locations. In a goto program, locations are represented by location numbers. In the given example, the edge `0 -> 1` is annotated with a place consisting of the location numbers 14, 2, and 17. Our tool prints some additional information, namely the function a location is in (e.g., `main`), the C source location corresponding to the location number (e.g., `file test.c line 13 function main`), and whether the location corresponds to a function call or thread creation site (e.g., `thread_create`). The function name `_start` which appears in the output denotes the entry point to the goto program. A goto program has a function `_start()` which in turn calls the function `main()`.

```
1  Reading GOTO program from test.gb
2  Checking location tags
3  Sanitization was not necessary
4  Function Pointer Removal
5  Removing returns
6  Pointer Analysis (0.001s)
7  May Lock Set Analysis (0.001s)
8  Must Lock Set Analysis (0s)
9  ...
10 Construct Lock Graph (0s)
11 0 lock m1
12 ...
13 1 lock m2
14 ...
15 *** Statistics:
16   Number of threads:              3
17   Number of threads in loop:      0
18   Number of locks:                2
19   Number of lock operations:      4
20   Number of indeterminate lock ops: 0
21   Size of largest lock set:       2
22 Check Cycles (0s)
23   Number of cycles:               1
24   Number of non-concurrency checks: 1
25   Length of longest valid cycle:  2
26 * potential deadlocks    1
27   0 - ([(_start, (14, file test.c line 9), function_call),
28          (main, (2, file test.c line 13 function main), thread_create)],
29         17) ->
30   1 - ([(_start, (14, file test.c line 9), function_call),
31          (main, (3, file test.c line 15 function main), thread_create)],
32         24) ->
33   0
```

Fig. B.2: Deadlock analysis output