Sound Static Deadlock Analysis for C/Pthreads

Daniel Kroening
University of Oxford
Oxford, UK
kroening@cs.ox.ac.uk

Daniel Poetzl
University of Oxford
Oxford, UK
daniel.poetzl@cs.ox.ac.uk

Peter Schrammel
University of Sussex
Brighton, UK
p.schrammel@sussex.ac.uk

Björn Wachter
SSW-Trading GmbH
Germany
bjoern.wachter@gmail.com

ABSTRACT

We present a static deadlock analysis for C/Pthreads. The design of our method has been guided by the requirement to analyse real-world code. Our approach is sound (i.e., misses no deadlocks) for programs that have defined behaviour according to the C standard and the Pthreads specification, and is precise enough to prove deadlock-freedom for a large number of such programs. The method consists of a pipeline of several analyses that build on a new context- and thread-sensitive abstract interpretation framework. We further present a lightweight dependency analysis to identify statements relevant to deadlock analysis and thus speed up the overall analysis. In our experimental evaluation, we succeeded to prove deadlock-freedom for 292 programs from the Debian GNU/Linux distribution with in total 2.3 MLOC in 4 hours.

CCS Concepts

•**Theory of computation** → **Program analysis;**

Keywords

deadlock analysis, static analysis, abstract interpretation

1. INTRODUCTION

Locks are the most frequently used synchronisation mechanism in concurrent programs to guarantee atomicity, prevent undefined behaviour due to data races, and hide weak-memory effects of the underlying architectures. However, locks, if not correctly used, can cause a *deadlock*, where one thread holds a lock that the other one needs and vice versa.

In small programs, deadlocks may be spotted easily. However, this is not the case in larger software systems. Locking disciplines aim to prevent deadlocks but are difficult to maintain as the system evolves, and every extension bears the risk of introducing deadlocks. For example, if the order in which locks are acquired is different in different parts of the code this may cause a deadlock.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE'16, September 03-07, 2016, Singapore, Singapore

© 2016 ACM. ISBN 978-1-4503-3845-5/16/09…$15.00

DOI: http://dx.doi.org/10.1145/2970276.2970309

The problem is exacerbated by the fact that deadlocks are difficult to discover by means of testing. Even a test suite with full line coverage is insufficient to detect all deadlocks, and similar to other concurrency bugs, triggering a deadlock requires a specific thread schedule and a set of particular program inputs. Therefore, static analysis is a promising candidate for a thorough check for deadlocks.

We hypothesise that static deadlock detection can be performed with a sufficient degree of precision and scalability and without sacrificing soundness. To this end, this paper presents a new method for static deadlock analysis. Our approach is *sound*[1] (i.e., misses no deadlocks) for programs that have defined semantics according to the C standard [22] and the Pthreads specification [40] (and thus in particular do not contain data races). To quantify scalability, we have applied our implementation to a large body of real-world concurrent code from the Debian GNU/Linux project.

Specifically, this paper makes the following contributions:

1. A static deadlock analysis for C/Pthreads that is sound (for defined programs) and can handle real-world code.
2. A new context- and thread-sensitive abstract interpretation framework that forms the basis of our analyses. The framework unifies contexts, threads, and program locations via the concept of a place.
3. 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 that are relevant to deadlock analysis.
4. We show how to build a lock graph that soundly captures a variety of sources of imprecision, such as may-point-to information and thread creation in loops/recursions, and how to combine the cycle detection with a non-concurrency check to prune infeasible cycles.
5. A thorough experimental evaluation on 997 programs from Debian GNU/Linux with 11.4 MLOC in total and up to 50 KLOC per program.

2. OVERVIEW

The design of our analyses 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, as for such programs the compiler is allowed to do anything, and may in particular produce a program containing a deadlock. This could happen in

[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.

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

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 }
```

Figure 1: Lock and create wrappers

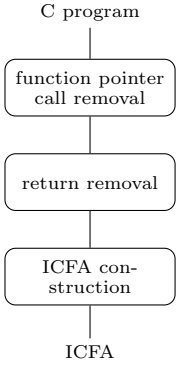

Figure 2: ICFA constr.

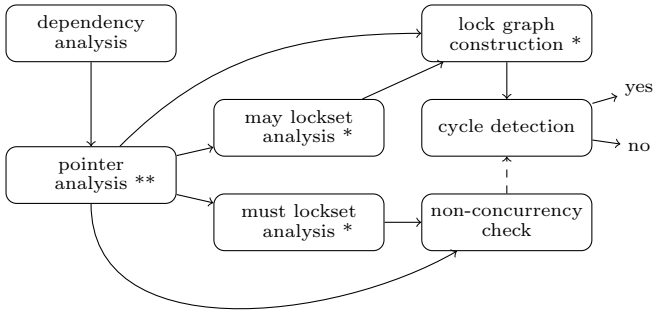

Figure 3: Analysis pipeline

practice, for example, if the compiler removes an if-branch that contains an unlock operation if it can determine that the branch always invokes an undefined operation (such as one resulting in an integer overflow).

Fig. 3 gives an overview of our analysis pipeline. 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). All of the analyses operate on a graph representation of the program (introduced in Sec. 3.1). The exception is the cycle detection phase, which only uses the lock graph computed in the lock graph construction phase.

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 Sec. 3.2). To enable trade-off between precision and cost, the framework comes in a flow-insensitive and a flow-sensitive version. The pointer analysis was implemented on top of the former (thus marked with ** in Fig. 3), and the may- and must-lockset analysis and the lock graph construction on top of the latter (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. Fig. 1, for example, shows a lock wrapper from the VLC project. 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 identified 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()`. Fig. 1 gives the wrapper for `pthread_create()` from the memcached project. The wrapper then 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

```
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 }
```

Figure 4: Example of a deadlock-free program

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 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 for example Fig. 4. 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-, 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 Sec. 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 Fig. 4. The thread `main()` attempts to acquire the locks in the sequence m_1, m_3, m_2, and the thread `thread()` attempts to acquire the locks in the sequence m_1, m_2, m_3. There is an order inversion between m_2 and m_3, but there is no deadlock since the two sections 7–13 and 28–34 (and thus in particular the locations 8 and 30) are protected by the common lock m_1. The common locks criterion has first been described by Havelund [20] (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 Fig. 4, there is an order inversion between the locks of m_5 and m_4 by function `func2()`, and the locks of m_4, m_5 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 Sec. 5.

3. ANALYSIS FRAMEWORK

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

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 has been taken at some point in the code. This is illustrated in Fig. 5 (top). 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 [22]. In the second step, functions with multiple exit points (i.e., multiple return statements) are transformed such as to have only one exit points (illustrated in Fig. 5, bottom).

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 [21]. CFAs are similar to control flow graphs, but with the nodes representing program locations and the edges being labeled with operations. ICFAs have additional inter-function edges modeling function entry, function exit, thread entry, thread exit, and thread join. Fig. 4 shows a concurrent C program and Fig. 6 shows its corresponding ICFA (thread exit and thread join edges and the function `func1()` are omitted).

We denote by *Prog* a program (represented as an ICFA), by *Funcs* the set of identifiers of the functions, by $L = \{\ell_0, \ldots, \ell_{n-1}\}$ 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 Fig. 6, the edge between locations 49 and 52 is labeled with the operation `x=3`, and the edge between locations 18 and 46 is labeled 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 e.g. `pthread_create()` (as implemented in e.g. glibc on GNU/Linux systems). Instead, our analysis only tracks the semantic effect of the operation, i.e., creating a new thread.

Apart from intra-function edges we also have inter-function edges that can be labeled with the five operations func_entry, func_exit, thread_entry, thread_exit, and thread_join.

A function entry edge (func_entry) connects a 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., 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.

3.2 Analysis Framework – Overview

Our framework to perform context- and thread-sensitive analyses on ICFAs is based on abstract interpretation [14].

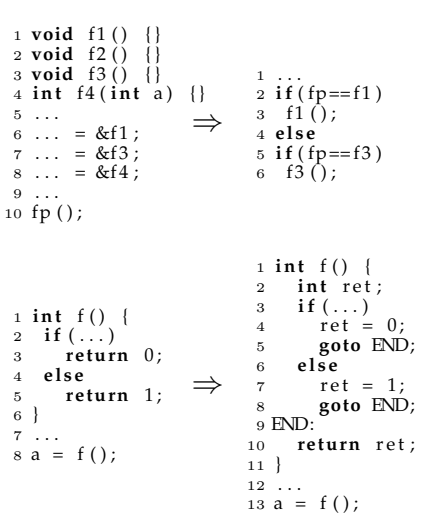

Figure 5: Function pointers and returns

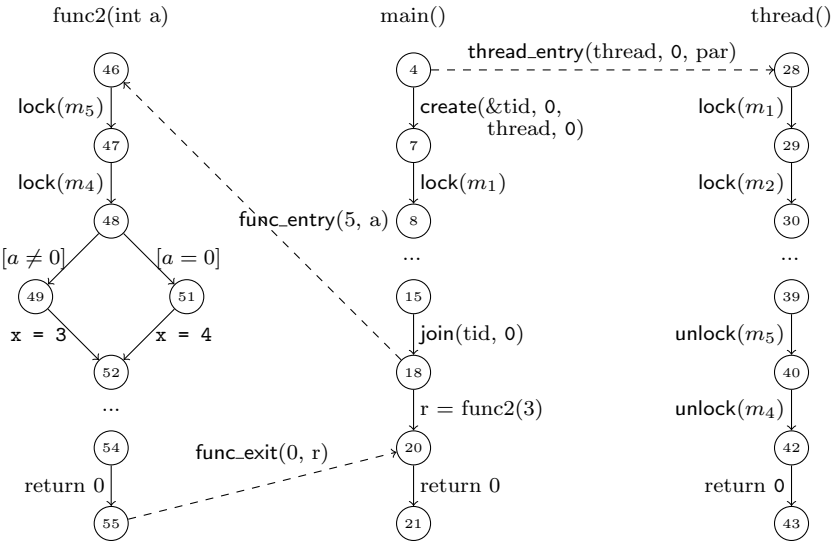

Figure 6: ICFA associated with the program in Fig. 4

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. We provide more details and a formalization of the framework in the next section.

Our analysis framework unifies contexts, 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., location 18 in Fig. 6). 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. 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 Fig. 6, 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). A function pointer fp might be mapped by fpm either to a function f or to the special value d (for "dirty") which indicates that the analysed function assigned to fp or took the address of fp. In this case we conservatively assume that the function pointer could point to any thread function.

3.3 Analysis Framework – Details

We now explain the formalisation of the analysis framework which is given in Fig. 7. The figure gives the flow-sensitive variant of our framework. We refer to the extended version of the paper [29] for the flow-insensitive version. The figure gives the domain, join function $\sqcup_s$ and transfer function $\mathcal{T}_s$, which are defined in terms of the join function $\sqcup_a$ and transfer function $\mathcal{T}_a$ of the parametrising analysis (such as the lockset analyses defined in the next section).

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 pre-

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

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

$$fpm^1 \sqcup_{fp} fpm^2 = fpm$$

$fpm(fp) =$

$$\begin{cases} d & fpm^1(fp) = d \vee fpm^2(fp) = d \\ v & (fpm^1(fp) = v \wedge (fp \notin \mathsf{dom}(fpm^2) \vee fpm^2(fp) = v)) \vee \\ & (fpm^2(fp) = v \wedge (fp \notin \mathsf{dom}(fpm^1) \vee fpm^1(fp) = v)) \\ \bot & \text{otherwise} \end{cases}$$

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
$s_s = (fpm, s_a)$:

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

$$fpm'(par_i) = \begin{cases} arg_i & \mathsf{is\_func}(arg_i) \\ fpm(arg_i) & \mathsf{is\_func\_pointer}(arg_i) \end{cases}$$

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

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

$\mathsf{match\_fp}(fpm, thr, f) =$
$(\mathsf{is\_func\_pointer}(thr) \wedge fpm(thr) \in \{\bot, d, f\}) \vee$
$(\mathsf{is\_func}(thr) \wedge thr = f)$

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 $s_s = (fpm, s_a)$:

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

Figure 7: Context-, thread-, and flow-sensitive framework

fix 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 next_s), 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 $s \in P \to (Fpms \times \mathcal{D}_a)$. The result is defined via a fixpoint equation [14]. 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):

$$s = s_0 \sqcup \lambda p. \bigsqcup_{p', e \text{ s.t.} \mathsf{np}(p, p', e)}^{s} \mathcal{T}_s[\![e, p']\!](s(p'))$$
$$\text{with } \mathsf{np}(p, p', (\ell_1, \ell_2)) = \ell_1 = \mathsf{top}(p') \wedge$$
$$\ell_2 = \mathsf{top}(p) \wedge$$
$$\mathsf{next}_s((\ell_1, \ell_2), p') = p$$
$$\text{with } s \sqcup s' = \lambda p.\, s(p) \sqcup_s s'(p)$$

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 (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, its value is set to d in fpm, thus indicating that it could point to any thread function. This case is omitted from Fig. 7 for lack of space.

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 Fig. 3) 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. We give further details in the extended version [29].

3.4 Pointer Analysis

We use a standard points-to analysis that is an instantiation of the flow-insensitive version of the above framework

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

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

With $\mathsf{op}(e) \in \{\mathsf{thread\_entry}, \mathsf{thread\_exit}, \mathsf{thread\_join}\}$:

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

Figure 8: May lockset analysis. We denote by $vs(p,a)$ the value set of pointer expression a at place p (see Sec. 3.4)

(see the extended version of the paper [29]). It computes for each place an element of $Vars \to (2^{Objs} \cup \{\{\star\}\})$. 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 [37].

3.5 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 (computing 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.

The may lockset analysis is formalised in Fig. 8 as a custom analysis to parametrise the flow-sensitive framework with. The must lockset analysis is given in the extended version [29]. Both the may and must lockset analyses makes use of the previously computed points-to information by means of the function $vs()$. In both cases, care must be taken to compute sound information from the may-point-to information provided by $vs()$. For example, for the may lockset analysis on an $\mathsf{unlock}(a)$ operation, we cannot just remove all elements in $vs(p,a)$ from the lockset, as an unlock can only unlock one lock. We use $ls_a(p), ls_u(p)$ to denote the may and must locksets at place p.

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-, create-, and join-statements). The purpose of the analysis is to speed up the following pointer analysis phase (cf. Fig. 3).

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 over-approximation 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/assignment. 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 Sec. 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 C 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 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.

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 Sec. 7.

Evaluation.

We have evaluated the dependency analysis on a subset of 100 benchmarks of the benchmarks given in Sec. 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 results are given in the table below.

	runtime	sign. assign.	sign. func.
25th percentile	0.03 s	0.3%	43.3%
arithmetic mean	0.18 s	40.0%	63.3%
75th percentile	0.34 s	72.2%	86.5%

Figure 9: Dependency analysis runtime and effectiveness

The table shows that the average time (over all benchmarks) to perform the dependency analysis was 0.18 s. The first and last line give the 25th and 75th percentile. This indicates for example that for 25% of the benchmarks it took 0.03 s or less to perform the dependency analysis. The third and fourth column evaluate the effectiveness of the analysis. On average, 40% of the assignments in a program were classified as significant (i.e., potentially affecting the starting expressions). The data also shows that often the number of significant assignments was very low (in 25% of the cases it was 0.3% or less). This happens when the lock usage patterns in the program are simple, such as using simple lock expressions (like `pthread_mutex_lock(&mutex)`) that refer to global locks with simple initialisations (such as using static initialization via `PTHREAD_MUTEX_INITIALIZER`).

The average number of functions classified as significant was 63.3%. This means that on average 36.7% of the functions that occur in a program were identified as irrelevant by the dependency analysis and thus do not need to be analysed by the following pointer analysis.

Overall, the data shows that the analysis is cheap and able to prune a significant number of assignments and functions.

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
```

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 labeled) cannot execute concurrently in the contexts embodied by p_1, p_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 [20], 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 to ℓ_1). This check is performed by the

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
```

procedure *unwind*(), the full details of which we give in the extended version [29].

We evaluated the non-concurrency analysis with respect to what fraction of all the pairs of places p_1, p_2 of a program it classifies as non-concurrent. We found that on a subset of 100 benchmarks of the benchmarks of Sec. 7, it classified 60% of the places corresponding to different threads as non-concurrent on average. We give more data in the extended version [29].

6. LOCK GRAPH ANALYSIS

Our lock graph analysis consists of two phases. First, we build a lock graph based on the lockset analysis. In the second phase, we prune cycles that are infeasible due to information from the non-concurrency analysis.

6.1 Lock Graph Construction

A *lock graph* is a directed graph $L \in 2^{Objs^* \times P \times Objs^*}$ (with $Objs^* = Objs \cup \{\star\}$). Each node is a $lock \in Objs^*$, and an edge $(lock_1, p, lock_2) \in Objs^* \times P \times Objs^*$ from $lock_1$ to $lock_2$ is labelled with the place p of the 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. Fig. 10 gives the lock graph for the example program in Fig. 4.

We use the result of the may lockset analysis (Sec. 3.5) to build the lock graph. Fig. 11 gives the lock graph domain

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 ls_u^1 ∩ ls_u^2 ≠ ∅ then
4      return true
5  i ← |common_prefix(p_1, p_2)|
6  r_1, r_2 ← true, true
7  ℓ_1, ℓ_2 ← p_1[i], p_2[i]
8  if has_path(ℓ_1, ℓ_2) then
9      r_1 ← unwind(i, p_1, ℓ_1, ℓ_2)
10 if has_path(ℓ_2, ℓ_1) then
11     r_2 ← unwind(i, p_2, ℓ_2, ℓ_1)
12 return r_1 ∧ r_2
```

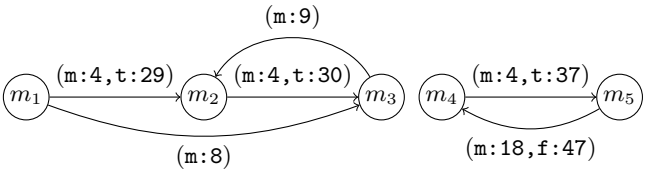

Figure 10: Lock graph for the program in Fig. 4 (`t`, `m` and `f` are shorthand for `thread`, `main` and `func2`, respectively).

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 w.r.t. 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.

6.2 Checking Cycles in the Lock Graph

The final step is to check the cycles in the lock graph. We look for deadlocks involving two or more threads. Each cycle c in the lock graph could be a potential deadlock. A cycle c is a set of (distinct) edges; there is a finite number of such sets. A cycle is a potential deadlock if $|c| > 1 \wedge \mathsf{all\_concurrent}(c)$ where

$$\begin{aligned}&\mathsf{all\_concurrent}(c) \Leftrightarrow \\ &\quad \forall (lock_1, p, lock_2), (lock_1', p', lock_2') \in c : \\ &\quad \neg \mathsf{non\_concurrent}(p, p') \vee \\ &\quad (\mathsf{get\_thread}(p) = \mathsf{get\_thread}(p') \wedge \\ &\quad \mathsf{multiple\_thread}(\mathsf{get\_thread}(p)))\end{aligned}$$

and $\mathsf{multiple\_thread}(t)$ means that t was created in a loop or recursion. Owing to the use of our non-concurrency analysis we do not require any special treatment for gate locks or thread segments as in [3].

7. EXPERIMENTS

We implemented our deadlock analyser as a pipeline of static analyses in the CPROVER framework.The tool and benchmarks are available online at http://www.cprover.org/deadlock-detection.

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

$s_1 \sqcup s_2 = s_1 \cup s_2$

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

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

$$cl(s) = s \cup \{(lock_1, p, lock) \mid (lock_1, p, \star) \in s, \; lock \in \mathsf{get\_locks}(s) \setminus \{lock_1, \star\}\} \cup \{(lock, p, lock_2) \mid (\star, p, lock_2) \in s, \; lock \in \mathsf{get\_locks}(s) \setminus \{lock_2, \star\}\}$$

Figure 11: Lock graph construction

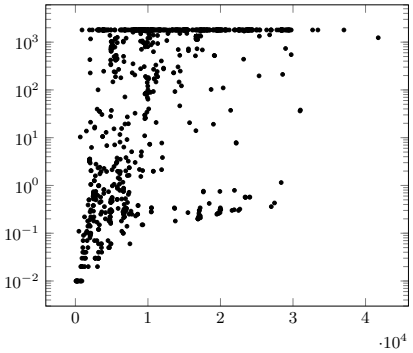

Figure 13: LOC vs. analysis time (timeout 1800 s)

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 Fig. 14.[2] 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 a 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.

Results.

We correctly report (potential) deadlocks in the 8 benchmarks with known deadlocks. The results for the deadlock-free programs are shown in Fig. 12 grouped by benchmark size (t/o. . . timed out, m/o. . . out of memory).

KLOC	analysed	**proved**	alarms	t/o	m/o
0–5	250	**131**	55	44	20
5–10	272	**91**	35	104	42
10–15	152	**22**	13	95	22
15–20	181	**28**	6	98	49
20–50	142	**20**	5	112	5

Figure 12: Results for the deadlock-free programs

For 114 deadlock-free benchmarks, we report alarms that are most likely spurious. The main reason for such false alarms is the imprecision of the pointer analysis with respect to dynamically allocated data structures. This leads to lock operations on indeterminate locks (see statistics in Fig. 14). For 65.1% of the lock operations on average the analysis was precise, i.e., the value set contained only a single lock.

The scatter plot in Fig. 13 illustrates how the tool scales in terms of running time with respect to the number of lines of code. The tool successfully analysed programs with up to 40K lines of code. As the plots show, the asymptotic behaviour is difficult to predict since it mostly depends on the complexity of the pointer analysis.

We evaluated the impact of the different analysis features 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, which is due to the often large number of general memory objects, including all heap and stack objects that may contain locks. 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.

8. THREATS TO VALIDITY

This section discusses the threats to internal and external validity of our results and the strategies we have employed to mitigate them [17].

The main threat to internal validity concerns the correctness of our implementation. To mitigate this threat we have continually tested our tool during the implementation phase, which has resulted in a testsuite of 122 system-, unit-, and regression-tests. To further test the soundness claim of our tool on larger programs, we introduced deadlocks into 8 previously deadlock-free programs, and checked that our tool correctly detected them. While we have reused existing locks and lock operations to create those deadlocks, they might nevertheless not correspond well to deadlocks inadvertently introduced by programmers.

The threats to external validity concern the generalisability of our results to other benchmarks and programming languages. Our benchmarks have been drawn from a collection of open-source C programs from the Debian GNU/Linux distribution [30] that use Pthreads and contain lock operations, from which we ran most of the smaller ones and some larger ones. We found that the benchmark set contains a diverse set of programs. However, we did not evaluate our tool on embedded (safety- or mission-critical) software, a field that we see as a prime application area of a sound tool like ours, due to the ability to verify the absence of errors.

During the experiments we have used a timeout of 1800 s. This in particular means that we cannot say how the tool would fare for users willing to invest more time and computing power; in particular, the false positive rate could increase as programs that take a long time to analyse are likely larger and could have more potential for deadlocks to occur.

Finally, our results might not generalise to other programming languages. For example, while Naik et al. [34]

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

	max	avg	min
# Lines of code	41,749	11,401.7	86
# Threads	163	3.8	1
# Threads in loop	162	5.8	0
# Locks	16	1.8	0
# 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	1.0
# Non-concurrency checks	11,043.0	132.8	0.0

	max	avg	min
Total analysis time (s)	1,800.0	80.54	0.0
Dependency analysis	2.8	0.02	0.0
Pointer analysis	1,710.7	2.15	0.0
May lockset analysis	346.3	0.16	0.0
Must lockset analysis	211.4	0.06	0.0
Lock graph construction	259.8	0.06	0.0
Cycles detection	318.8	0.00	0.0
Peak memory (GB)	24.0	7.7	0.007

Figure 14: Benchmark characteristics and analysis statistics

(analysing Java) found that the ability to detected common locks was crucial to lower the false positive rate, we found that it had little effect in our setting, since most programs do not acquire more than 2 locks in a nested manner (see Tab. 14, size of largest lockset).

9. RELATED WORK

Deadlock analysis is an active research area. A common deficiency of many of the existing tools is that they are neither sound nor complete, and produce false positives and negatives.

Dynamic tools.

The development of the Java PathFinder tool [20, 3] led to ground-breaking work over more than a decade to find lock acquisition hierarchy violation with the help of lock graphs, exposing the issue of gatelocks, and segmentation techniques to handle different threads running in parallel at different times. [4, 25, 39] try to predict deadlocks in executions similar to the observed one. DeadlockFuzzer [25] use a fuzzying technique to search for deadlocking executions. Multicore SDK [32] tries to reduce the size of lock graphs by clustering locks; and Magiclock [10] implements significant improvements on the cycle detection algorithms. Helgrind [1] 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 [2] is a semi-automatic lightweight approach that relies on user-supplied lock acquisition orders. RacerX [15] 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 [23] and MU-CSeq [41] use context-bounded sequentialisation, and analyse the resulting program with the bounded model checker CBMC [12]. ESBMC [13] 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 [45, 38] 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 [6], a tool similar to LockLint. The tool Jade [34] consciously uses a may analysis instead of a must analysis, which causes unsoundness. The tools presented in [44] and [42] do not consider gatelock scenarios.

Other tools.

Some tools combine dynamic approaches and constraint solving. For example, CheckMate [24] model-checks a path along an observed execution of a multi-threaded Java program; Sherlock [16] uses concolic testing; and [39, 4] 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 [19, 11], for the modelling languages BIP (DFinder tool [7, 8]) or ABS (DECO tool [18]) that 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 [27, 28, 35] and alias analysis [9, 26]. 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 [43].

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 [15], and Havelund [3] 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. [5]. 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 [33] and Naumovich et al. [36] for ADA, and the work of Lee et al. [31] for async-finish parallelism.

10. CONCLUSIONS

We presented a new static deadlock analysis approach for concurrent C/Pthreads programs. We demonstrated that our tool can effectively prove deadlock freedom of 2.3 MLOC concurrent C code from the Debian GNU/Linux distribution. The experiments show that the pointer analysis takes the most time, and its imprecision is the primary source for false alarms. Future work will focus on addressing this limitation. Moreover, we will integrate the analysis of Pthreads synchronisation primitives other than mutexes, e.g. condition variables, and extend our algorithm to Java synchronisation constructs.

11. ACKNOWLEDGMENTS

This work is supported by ERC project 280053 and SRC task 2269.002. We are grateful to Michael Tautschnig for the infrastructure for extracting CFGs from Debian.

12. REFERENCES

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

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] M. Eslamimehr and J. Palsberg. Sherlock: scalable deadlock detection for concurrent programs. In *Foundations of Software Engineering*, pages 353–365. ACM, 2014.

[17] R. Feldt and A. Magazinius. Validity threats in empirical software engineering research – an initial survey. In *Software Engineering & Knowledge Engineering (SEKE)*, pages 374–379. Knowledge Systems Institute Graduate School, 2010.

[18] 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.

[19] 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.

[20] K. Havelund. Using runtime analysis to guide model checking of Java programs. In *SPIN*, volume 1885 of *LNCS*, pages 245–264. Springer, 2000.

[21] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*. ACM, 2002.

[22] International Organization for Standardization. C standard, 2011. ISO/IEC 9899:2011.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] J. Krinke. Static slicing of threaded programs. In *PASTE*, pages 35–42. ACM, 1998.

[28] J. Krinke. Context-sensitive slicing of concurrent programs. In *ESEC/FSE*, pages 178–187. ACM, 2003.

[29] D. Kroening, D. Poetzl, P. Schrammel, and B. Wachter. Sound static deadlock analysis for C/Pthreads (extended version). *CoRR*, abs/1607.06927, 2016.

[30] D. Kroening and M. Tautschnig. Automating software analysis at large scale. In *MEMICS*, volume 8934 of *LNCS*, pages 30–39. Springer, 2014.

[31] 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.

[32] 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.

[33] S. P. Masticola and B. G. Ryder. Non-concurrency analysis. In *PPoPP*, pages 129–138. ACM, 1993.

[34] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *International Conference on Software Engineering*, pages 386–396. IEEE, 2009.

[35] M. G. Nanda and S. Ramesh. Slicing concurrent programs. In *ISSTA*, pages 180–190. ACM, 2000.

[36] 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.

[37] M. C. Rinard. Analysis of multithreaded programs. In *SAS*, volume 2126 of *LNCS*, pages 1–19. Springer, 2001.

[38] 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.

[39] F. Sorrentino. PickLock: A deadlock prediction approach under nested locking. In *SPIN*, volume 9232 of *LNCS*, pages 179–199. Springer, 2015.

[40] The Open Group and IEEE. The open group base specifications issue 7, 2013. IEEE Std 1003.1-2008/Cor 1-2013.

[41] 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.

[42] C. von Praun. *Detecting Synchronization Defects in Multi-Threaded Object-Oriented Programs*. PhD thesis, 2004.

[43] M. Weiser. Program slicing. In *ICSE*, pages 439–449. IEEE, 1981.

[44] 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.

[45] 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.