The Quest for Precision: A Layered Approach for Data Race Detection in Static Analysis

Jakob Mund[1⋆], Ralf Huuck[2], Ansgar Fehnker[2,3], and Cyrille Artho[4]

[1] Technische Universität München, Munich, Germany
mund@in.tum.de
[2] NICTA, University of New South Wales, Sydney, Australia
ralf.huuck@nicta.com.au
[3] University of the South Pacific, Suva, Fiji
ansgar.fehnker@usp.ac.fj
[4] Research Institute for Secure Systems, AIST, Amagasaki, Japan
c.artho@aist.go.jp

Abstract. Low level data-races in multi-threaded software are hard to detect, especially when requiring exhaustiveness, speed and precision. In this work, we combine ideas from run-time verification, static analysis and model checking to balance the above requirements. In particular, we adopt a well-known dynamic race detection algorithm based on calculating lock sets to static program analysis for achieving exhaustiveness. The resulting data race candidates are in a further step investigated by model checking with respect to a formal threading model to achieve precision. Moreover, we demonstrate the effectiveness of the combined approach by a case study on the open-source TFTP server OpenTFTP, which shows the trade-off between speed and precision in our two-stage analysis.

Keywords: Software verification, static analysis, concurrency, lock sets

1 Introduction

Multi-core architectures are the de facto industry standard for recent processors. To make full use of the hardware, however, software for multi-core processors often has to manage threads in the application code by itself. Such concurrency carries the risk of introducing subtle but serious defects that might show up only sporadically and are extremely hard to debug.

Concurrency issues have been studied extensively in the computer science community. However, common embedded programming languages such as C provide only basic primitives for concurrency in terms of threading, requiring careful thinking and experience from the developer while at the same time offering only limited tool support for debugging and bug prevention.

In this work we present a new way of detecting data races in embedded source code. We combine ideas from run-time verification, static analysis and

⋆ This work was carried out while visiting NICTA.

software model checking by balancing their strengths and weaknesses. Run-time verification provides a good means to detect certain race conditions, but can only reason over program executions that have actually been observed, limiting coverage. Static analysis is strong in covering all potential execution paths, but is prone to (over-)approximations leading to false positives. On the other hand, software model checking offers a precise analysis of the program semantics, but with limitations regarding scalability to larger code bases.

We propose a layered approach: In a first step we develop a path-sensitive static implementation of the well-known *Eraser* algorithm [16] typically used in run-time verification. Our static version is able to detect data races in C programs with a complete path coverage. While the algorithm is applicable to large-scale software, it is also prone to false positives. Therefore, in a second step we take those data races as *candidates* for a deeper model checking approach. The model checking phase is confined to candidates, abstracts from non-essential data and instructions, and takes the threading model into account.

In this way we avoid to apply traditional software model checking to the full multi-threaded source code, but rather treat its application as a false-positive elimination step on selected code parts only. As a result we obtain a solution that can deal with real software systems, has a higher degree of coverage than run-time verification, but is more precise than traditional static analysis.

The remainder of this paper is organized as follows: In Section 2 we give a brief introduction to data races and the objectives of this work, together with related work. We then present a semantics framework for multi-threading in Section 3, which is the basis of our analysis. Our two-step analysis approach is described in Section 4, covering the Eraser lock set analysis and its combination with software model checking. An evaluation based on a number of benchmarks including corner cases and open source software is presented in Section 5, followed by our conclusions in Section 6.

2 Data Races in Multi-threaded Programs

Threads are concurrent streams of program execution that can be created, merged and deleted at run-time. Threads might have access to shared resources. A *data race* occurs if two or more threads can simultaneously and non-atomically access a shared resource with at least one access being a write operation.

An example data race is given in Figure 1. A `reader` thread reads a shared variable (lines 3 – 6); a `writer` thread writes to it (lines 8 – 11). If these accesses are not synchronized using locks or other coordination mechanisms, then their effects are not well-defined. The update of the writer thread may become visible to other threads immediately or at any time after it has been issued, due to memory caches and other optimizations in modern hardware. Reading `shared_var` may thus yield different results depending on thread scheduling and hardware, which is why it is desirable to avoid data races in concurrent software.

Since threads can be created dynamically at run-time, there might be a large number of threads at any given time. The effect of data races may not be visible

```
1 int shared_var = 0;
2
3 void *reader() {
4   for(;;) {
5     t = shared_var;
6 } }
7
8 void *writer() {
9   for(;;) {
10    shared_var = compute();
11 } }
```

Fig. 1. Data Race Example

unless a particular interleaving of thread actions occurs at run-time; this makes the detection of data races difficult.

2.1 Scope and Contribution

This work considers detecting low-level data races by finding unprotected or inconsistent locking of variables shared across threads. A classical run-time verification approach to this problem is the Eraser algorithm. This algorithm monitors the *set of locks* that protect a shared variable during reads and writes to that variable. For every variable access, Eraser incrementally builds the intersection of the set of locks protecting a variable. Once the intersection is empty, i. e., there is no single lock consistently protecting a variable, a warning is issued.

Static Eraser Implementation. In this work we introduce a path-sensitive static implementation of Eraser. Unlike the monitored behavior in run-time verification we consider all paths statically, possibly over-approximating the set of feasible interleavings, but ensuring full coverage of inconsistencies. This approach finds all data races but may issue spurious warnings.

Model-checking Thread-Interleaving. We also propose another analysis that is more precise and can reduce false positives from the previous step. The second analysis creates the *thread-interleaving graph* of the program (with limited depth) that captures the call structure of the threads and their termination, as well as the read and write accesses to shared variables. Since the interleaving graph grows exponentially with the number of threads in the program, we restrict it to the *data-race candidates* as identified in the static Eraser approach.

As a result we are able to model-check the thread-interleaving graph for feasible data races. This approach is sound up to a bound on the number of threads. By abstracting from non-essential information we are able to apply this methods to some more realistic code sizes as shown in the evaluation in Section 5.

Formal Interleaving Semantics. As a basis for our approach we provide the thread-interleaving semantics used in the subsequent analysis. This semantics takes into account thread creation, join and cancelation as well as the acquisition and releases of locks. Moreover, we include the advanced concepts of waiting and signaling. These require a view of the global program state and are thus not covered by Eraser or other thread-modular approaches.

2.2 Related Work

Eraser [16] is the classical lock-set based algorithm that can approximate potential data races very well, while not having the overhead of more precise but heavy-weight approaches based on the happens-before relation [17].

Goldilocks is a newer algorithm that can compute data races precisely [7]. To be more accurate than Eraser, Goldilocks requires more elaborate data structures to be maintained. Furthermore, the precision of Goldilocks depends on its ability to recognize overlapping data of multiple software transactions. That data is readily available and precise when using run-time verification, but is difficult to approximate precisely enough in static analysis.

It has been shown that other concurrency error types still exist even when no data races (called low-level data races to compare them with similar concurrency problems) exist. High-level data races [3] and atomicity violations [2, 9] are two types of problems that remain even in the absence of low-level data races. High-level data races cover non-atomic accesses to sets of dependent variables (multiple memory locations). Atomicity violations concern the scope during which a lock is held, and thus the use of the data rather than its direct access. These two types of problems have recently been subsumed by the notion of *causality* in data flow, which can cover both accesses to data and its use [6].

Static analysis of such concurrency properties has been attempted in other work, in a static analyzer where the rules are hard-coded in the program [1], and in a framework that is specialized for concurrency properties [11]. In contrast to that tool, we build on top of a general static analysis framework, Goanna [8], that allows flexible rules to express a large range different properties.

The second part of our work is closely related to other software model checkers, e. g., Java PathFinder [19] for Java bytecode and inspect for C source code [20]. The key difference is that these software model checkers execute the full software at run time and explore alternative interleavings by rolling back the system to a previously stored state. This dynamic analysis is much more expensive than our approach, which works on an abstract model of the program.

Software model checkers working on a higher level of abstraction exist as well, such as SLAM, which analyzes device drivers against a given environment model [4], or SATABS, which can analyze programs using a subset of the Pthreads library [5]. In comparison, our work is not limited to certain domains (such as device drivers) and covers the full Pthreads library.

3 Semantics of Multi-Theaded Programs

As a basis for our data race analysis we first establish a formal semantics for multi-threaded programs. The semantics are given in terms of *structural operational semantics* (SOS) and *labeled transition systems* (LTS). We formally define basic concepts such as thread creation and locking, independently of a particular threading framework. In the evaluation we map POSIX threads (Pthreads) to this model.

Threads	θ	$\in \mathbf{Tid}$
Locks	m	$\in \mathbf{Locks}_\star$
Signals	c	$\in \mathbf{Signals}_\star$
Variables	v	$\in \mathbf{Variables}_\star$
LockState	Λ	$\in$ LockState: $\mathbf{Locks}_\star \to (\mathbf{Tid}_\star \cup \{\bot\})$
(Abstract) Local States	s, s'	$\in \mathbf{LocalState}$
(Global) ProgramCounter	κ	$\in$ ProgramCounter: $\mathbf{Tid} \rightharpoonup \mathbf{LocalState}$
Procedure Names	pn	$\in \mathbf{ProcName}_\star$
Procedure Environment	Π	$\in \mathit{ProcEnv} : \mathbf{ProcName}_\star \to \mathbf{LocalState}$
Actions	a	$\in \mathbf{Action} = \{\texttt{create}(\theta, pn), \texttt{join}(\theta), \texttt{exit}, \texttt{cancel}(\theta), \texttt{acq}(m), \texttt{acq?}(m), \texttt{rel}(m), \texttt{wait}(c, m), \texttt{signal}(c), \texttt{signal!}(c), \texttt{write}(v), \texttt{read}(v), \tau\}$

Fig. 2. Semantic domains

3.1 Thread Actions, Configurations, and Transitions

Semantic Domains. A multi-threaded program consists of an (unbounded) number of concurrently running threads, each identified by a unique thread identifier **Tid**, a finite number of locks $\mathbf{Locks}_\star$ (also called *mutexes*), signals (sometimes also referred to as *conditions*) and shared variables $\mathbf{Variables}_\star$.

Furthermore, a *LockState* characterizes the thread currently holding a specific lock. We use $\bot$ to denote that a lock is not held by any thread. An *abstract* **LocalState** represents the thread-local *control location* of a thread; we use a *global* program counter *ProgramCounter* as a partial function from a thread to its local state in the interleaving. The *ProgramCounter* is defined only for threads that have been created but not yet terminated. Each thread is considered a procedure with a name from $\mathbf{ProcName}_\star$; *ProcEnv* maps each name to its initial local state. Figure 2 summarizes the domains.

Actions. Actions describe thread activities. A thread can be created by `create`, terminated regularly by `exit`, or canceled by `cancel`, which terminates a thread in in its current state without further execution. Moreover, a thread can be suspended until the completion of a different thread by `join`. Locks can be acquired through either blocking (`acq`) or non-blocking actions (`acq?`) and can be released by `rel`. The action `wait` suspends the invoking thread until it receives a signal that has been sent to *one* thread by `signal` or to *all* threads by `signal!`. Variables are accessed by `read` and `write`. Other actions not related to threading are considered silent and denoted τ.

Transitions. Local transitions describe the atomic actions or steps of a thread. Global system progress is represented as choosing non-deterministically an action from one of the active threads (see below). We assume a finite local transition relation $\longrightarrow_L \subseteq \mathbf{LocalState} \times \mathbf{Action} \times (\mathbf{LocalState} \cup \{\varepsilon\})$, where ε represents successful termination. We write $(s, a, s') \in \longrightarrow_L$ as $s \xrightarrow{a}_L s'$.

Global Configurations. A configuration describes the global control state of a multi-threaded program. It is defined as the product of the local states given by the program counter κ of the active (denoted $\Theta \subseteq \mathbf{Tid}$) and waiting (denoted $\Delta \subseteq \mathbf{Tid}$) threads, and the currently held locks Λ. Formally:

$$\Sigma = \langle \kappa_\Theta, \kappa_\Delta, \Lambda \rangle \subseteq \mathit{ProgramCounter} \times \mathit{ProgramCounter} \times \mathit{LockState}$$

As a convention we will write $(\theta : s) \| \kappa$ (or simply $\theta : s \| \kappa$) to denote the configuration where thread θ is in state s and all other threads κ are unchanged.

Global Transitions. The *global* transition relation $\longrightarrow_G \subseteq \Sigma \times \mathbf{Tid} \times \mathbf{Action}_\star \times \Sigma$ is parametric in a *local* transition relation and a procedure environment. It manages threads and limits the steps of concurrent processes to respect synchronization primitives, e. g., locks. The SOS rules are specified in terms of

$$\longrightarrow_L, \Pi \vdash \langle \kappa_\Theta, \kappa_\Delta, \Lambda \rangle \xrightarrow{\theta, a}_G \langle \kappa'_\Theta, \kappa'_\Delta, \Lambda' \rangle$$

These transitions from configuration σ to σ' are executed by thread θ with action a, given the local transition relation $\longrightarrow_L$ and the procedure environment Π.

Interleaving Thread Semantics. The semantics are defined in terms of the labeled transition system $LTS = (\Sigma, \Sigma_0, \Omega, \longrightarrow_G)$ where

- Σ is the set of global configurations,
- $\Sigma_0 \subseteq \Sigma$ is the set of initial configurations,
- $\Omega = \mathbf{Tid} \times \mathbf{Action}$ is the alphabet consisting of pairs of thread identifiers and actions, and
- $\longrightarrow_G$ is the transition relation as it will be defined below.

3.2 Rules: Threads Creation and Thread Termination

The rules for multi-threading actions concerning thread management are formalized in Figure 3. Starting a thread is formalized in (rule create). That action, invoked by an active thread θ, adds a new thread with identifier θ' to the set of active threads. The initial local state of the procedure with name pn is determined by the procedure environment Π. Note, that if a thread with the same identifier as θ' already exists in κ_Θ, the number of threads is not increased but the thread is reset to its initial local state instead.

Thread termination can happen either explicitly by executing an `exit` action (rule exit_1) or implicitly by terminating successfully (rule exit_2). In both cases the executing thread is removed from the set of active threads; however, the rules differ in the action exposed by $\rightarrow_G$. Thread joining (rule join) contains a side condition ensuring that the join can only be performed when the thread waited for (i. e., θ') has already terminated (or has never been created).

The cancellation rules express that one thread may cancel another thread asynchronously, i. e., without allowing any further execution of the canceled thread. In this case the thread is removed from the program counter, so that other threads waiting to join the thread can continue. The need for two rules arises from the fact that the canceled thread can be either active (rule cancel_1) or delayed (rule cancel_2).

$$\text{(create)}\ \frac{\theta : s \xrightarrow{\texttt{create}(\theta', pn)}_L s'}{\to_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta, \texttt{create}(\theta', pn)}_G \langle \theta' : \Pi(pn) \| \theta : s' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}$$

$$\text{(exit}_1\text{)}\ \frac{\theta : s \xrightarrow{\texttt{exit}}_L s'}{\to_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta, \texttt{exit}}_G \langle \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}$$

$$\text{(exit}_2\text{)}\ \frac{}{\to_L, \Pi \vdash \langle \theta : \varepsilon \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta, \texttt{exit}}_G \langle \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}$$

$$\text{(join)}\ \frac{\theta : s \xrightarrow{\texttt{join}(\theta')}_L s'}{\to_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta, \texttt{join}(\theta')}_G \langle \theta : s' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}\ \text{if } \theta' \notin \Theta \cup \Delta$$

$$\text{(cancel}_1\text{)}\ \frac{\theta : s \xrightarrow{\texttt{cancel}(\theta')}_L s'}{\to_L, \Pi \vdash \langle \theta : s \| \theta' : s'' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta, \texttt{cancel}(\theta')}_G \langle \theta : s' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}$$

$$\text{(cancel}_2\text{)}\ \frac{\theta : s \xrightarrow{\texttt{cancel}(\theta')}_L s'}{\to_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \theta' : s'' \| \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta, \texttt{cancel}(\theta')}_G \langle \theta : s' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}$$

Fig. 3. Thread rules

3.3 Rules: Locking and Synchronization

The rules concerning locking and synchronization are depicted in Figure 4. Acquiring a lock (rule acq) uses a blocking action $\texttt{acq}(m)$. Its side condition ensures that a thread cannot proceed until the lock m is available.[5] The non-blocking action in (rule acq?$_2$) allows the thread to proceed without acquiring the lock.

Releasing a lock (rule rel) relinquishes ownership of the lock if it is currently held. However, it behaves differently to the previous actions if the lock is not held by any thread, i. e., if $\Lambda(m) = \bot$. Although explicitly unspecified in Pthreads [14], we adapted the typical behavior to allow the thread to continue.

A thread can be suspended until it receives the appropriate signal c (rule wait). If the next step of an active thread is a `wait` action, thread θ is marked as delayed, added to the waiting threads κ_Δ and removed it from κ_Θ. Note, the program counter of the thread is not changed until the appropriate signal is received.

There are two cases for sending a signal: If there are waiting threads for the matching signal *only one* of them will receive the signal and be reactivated at a time (rule signal$_1$). This means if there is more than one thread waiting for a signal, there will be a global transition for every one of them, but each only signaling the one selected thread while all other waiting threads are unmodified. On the other hand, if no suspended thread is waiting for signal c, that signal

[5] Note the special case that the current thread itself holds the lock, and in this case, cannot proceed either. This behavior is desired and works according to the POSIX standard, though may be modified by using more advanced locks like recursive ones.

$$\text{(acq)}\ \frac{\theta : s \xrightarrow{\texttt{acq}(m)}_L s'}{\rightarrow_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta,\texttt{acq}(m)}_G \langle \theta : s' \| \kappa_\Theta, \kappa_\Delta, \Lambda[m \mapsto \theta]\rangle} \text{if } \Lambda(m) = \bot$$

$$\text{(acq?)}\ \frac{\theta : s \xrightarrow{\texttt{acq?}(m)}_L s'}{\rightarrow_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta,\texttt{acq?}(m)}_G \langle \kappa_\theta : s' \| \Theta, \kappa_\Delta, \Lambda[m \mapsto \theta]\rangle} \text{if } \Lambda(m) = \bot$$

$$\text{(acq?}_2\text{)}\ \frac{\theta : s \xrightarrow{\texttt{acq?}(m)}_L s'}{\rightarrow_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta,\texttt{acq?}(m)}_G \langle \kappa_\theta : s' \| \Theta, \kappa_\Delta, \Lambda\rangle} \text{if } \Lambda(m) \neq \bot$$

$$\text{(rel)}\ \frac{\theta : s \xrightarrow{\texttt{rel}(m)}_L s'}{\rightarrow_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta,\texttt{rel}(m)}_G \langle \kappa_\theta : s' \| \Theta, \kappa_\Delta, \Lambda[m \mapsto \bot]\rangle} \text{if } \Lambda(m) \in \{\theta, \bot\}$$

$$\text{(wait)}\ \frac{\theta : s \xrightarrow{\texttt{wait}(c,m)}_L s'}{\rightarrow_L, \Pi \vdash \langle \theta : s \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta,\texttt{wait}(c,m)}_G \langle \kappa_\Theta, \theta : s' \| \kappa_\Delta, \Lambda\rangle} \text{if } \Lambda(m) = \bot$$

$$\text{(signal}_1\text{)}\ \frac{\theta_1 : s_1 \xrightarrow{\texttt{signal}(c)}_L s_1' \quad \theta_2 : s_2 \xrightarrow{\texttt{wait}(c,m)}_L s_2'}{\rightarrow_L, \Pi \vdash \langle \theta_1 : s_1 \| \kappa_\Theta, \theta_2 : s_2 \| \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta_1,\texttt{signal}(c)}_G \langle \theta : s_1' \| \theta_2 : s_2' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle} \text{if } \exists \theta_2 \in \Delta_c$$

$$\text{(signal}_2\text{)}\ \frac{\theta_1 : s_1 \xrightarrow{\texttt{signal}(c)}_L s_1'}{\rightarrow_L, \Pi \vdash \langle \theta_1 : s_1 \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta_1,\texttt{signal}(c)}_G \langle \theta : s_1' \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle} \text{if } \Delta_c = \emptyset$$

$$\text{(signal!)}\ \frac{\theta : s \xrightarrow{\texttt{signal!}(c)}_L s' \quad (\theta' : s_{\theta',1} \xrightarrow{\texttt{wait}(c,m_{\theta'})}_L s_{\theta',2})_{\theta' \in \Delta_c}}{\rightarrow_L, \Pi \vdash \langle \theta : s_1 \| \kappa_\Theta, (\theta' : s_{\theta',1})_{\theta' \in \Delta_c} \| \kappa_\Delta, \Lambda\rangle \xrightarrow{\theta_1,\texttt{signal!}(c)}_G \langle \theta : s' \| (\theta' : s_{\theta',2})_{\theta' \in \Delta_c} \| \kappa_\Theta, \kappa_\Delta, \Lambda\rangle}$$

where $\Delta_c = \{\theta' \in dom(\kappa_\Delta) \mid \theta' : s_{\theta',1} \xrightarrow{\texttt{wait}(c,m_{\theta'})}_L s_{\theta',2} \text{ for some lock } m_{\theta'}\}$

Fig. 4. Rules for locking and synchronization

is lost (rule signal$_2$). A broadcast of signal c affects *all* threads waiting for that signal (rule signal!). Central to this rule is the set $\Delta_c \subseteq \Delta$, i. e., the set of all delayed threads waiting for a signal c. For every such thread $\theta \in \Delta_c$ the signal is received as in the signal rules.

4 A Layered Approach for Static Race Detection

In this section we describe a layered approach for detecting data races. Our approach first applies static analysis to obtain data race candidates (based on

the Eraser algorithm [16]) and then applies model checking on parts of the program involving those candidates (based on the LTS as defined in Section 3).

4.1 Static Data Race Analysis

A common way to prevent data races is to impose a *locking discipline* that requires any shared (write) variable to be protected by at least one distinct lock among *all* threads. Since each lock can only be held by one thread, data races are effectively prevented.

The Eraser approach is to monitor the dynamic execution paths of each tread and record for each shared variable the *set of locks* that are held. If the intersection of those sets of locks across threads for the same variable is empty, we assume a potential data race, i. e., an unprotected variable. An advantage of this approach is that it is thread-modular, i. e., each tread can be analyzed separately and only the intersection of the information needs to be globally tracked.

To achieve the same statically, we propose to check *all program paths* for each thread and then build the same intersection over all threats. Obviously, the static approach is an over-approximation as not all paths might be executable. We use the definition of a *lock set* [16, 15] as the mapping of shared variables to its potential set of locks,i.e., $\mathbf{Lockset} : \mathbf{Variables}_\star \rightarrow \wp(\mathbf{Locks}_\star)$. In the following, we show how to compute and check for emptiness of the **Lockset**.

Path-Sensitive Lock Set Computation. A thread (procedure) π is defined as a procedure with name pn that occurs in an action $\texttt{create}(\theta, pn)$. Nodes in the control flow graph of π are denoted by $\mathbf{Nodes}_\pi$. For a given thread π we define a function $\mathbf{is\_locked} : \mathbf{Nodes}_\pi \times \mathbf{Locks}_\star \rightarrow \mathbb{B}$ that returns for each node n in π and each lock l, whether l is held along all paths leading to n by

$$\mathbf{is\_locked}(n, l) = \begin{cases} tt & \text{if } n = \mathbf{Lock\ l}, \\ ff & \text{if } n = \mathbf{Unlock\ l}, \\ \forall m \in pred(n) \bigwedge \mathbf{is\_locked}(m, l) & \text{otherwise.} \end{cases}$$

Here *pred* denotes the predecessors of a node; the conjunction ensures coverage of all potential paths. This notion captures a standard path-sensitive static program analysis to compute the must-hold locks for each node in a thread.[6] Based on the information about the held locks, the thread-local lock set for each shared variable $v \in \mathbf{Variables}_\star$ and thread $\pi_i \in \mathbf{Threads}_\star$ is computed by

$$\mathbf{LocalLockset}(v, \pi_i) = \begin{cases} \bigcap_{n \in N_v} \{l' \mid \mathbf{is\_locked}(n, l')\} & \text{if } N_v \neq \emptyset \text{ in } \pi_i, \\ \mathbf{Locks}_\star & \text{otherwise.} \end{cases}$$

where N_v denotes the set $\{n \in \mathbf{Nodes}_\pi \mid n = \mathbf{Write}_v \vee n = \mathbf{Read}_v\}$, i. e., the `Write` and `Read` nodes of thread π_i which access variable v. The second case

[6] Modern programming languages like Java support `synchronized` blocks which acquire (resp. release) a lock when entering (resp. exiting) the critical section, enabling path-insensitive approaches [13].

Algorithm 1: Static implementation of the lock set algorithm.

```
begin
    Lockset(v) ← Locks⋆;
    isReadOnly_v ← tt ;
    foreach π_i ∈ Threads⋆ do
        lockstate ⟵ MFP(Nodes_{π_i}, is_locked);
        foreach n ∈ Nodes_{π_i} do
            if n accesses v then
                LocalLockset_{π_i}(v) ← LocalLockset_{π_i}(v) ∩ lockstate(n) ;
                isReadOnly_v ← isReadOnly_v ∨ (v is modified in n);
        Lockset(v) ← Lockset(v) ∩ LocalLockset_{π_i}(v);
```

accounts for variables which are not accessed in π_i, mapping them to the set of all locks. Finally, the lock set for a program is the intersection of the lock sets for each thread occurring in a given program, i. e.,

$$\mathbf{Lockset}(v) := \bigcap_{\pi_i \in \mathbf{Threads}_\star} \mathbf{LocalLockset}(v, \pi_i)$$

The fact that a program is free of data races can be then be stated as

$$\mathit{racefree} \iff \forall v \in \mathbf{Variables}_\star \,.\, \mathbf{Lockset}(v) \neq \emptyset.$$

Our algorithm [12] uses the maximal-fix-point worklist algorithm MFP [10] (see Algorithm 1). However, similar fix-point solutions popular in static program analysis are also applicable.

Soundness and Completeness Given the semantics of Section 3 and the assumption that shared variables (as well as locks and signals) are not aliases, the static lock set algorithm for low-level data race detection presented here is *sound*, hence false negatives (i. e., missed races) are not possible. The argument here is that a non-empty lock set means that there exists at least one distinct lock that has to be held by every thread accessing a shared variable. Since a lock can only be held by one thread at a time, a simultaneous access from two or more threads to a variable is not permitted according to the semantics.

However, our analysis is *incomplete* as false positives (i. e., spurious warnings) are possible, because the analysis does not consider the *temporal* (also called *happens-before*) relation among events in different threads. Furthermore, warnings may be spurious if data races are avoided by other synchronization primitives like signals, or other more fine-grained accesses of variables[18].

4.2 Model-Checking of the Threading Semantics

While the approach presented in the previous section uses a fast thread-modular analysis it is also prone to potential false-positives. An alternative precise ap-

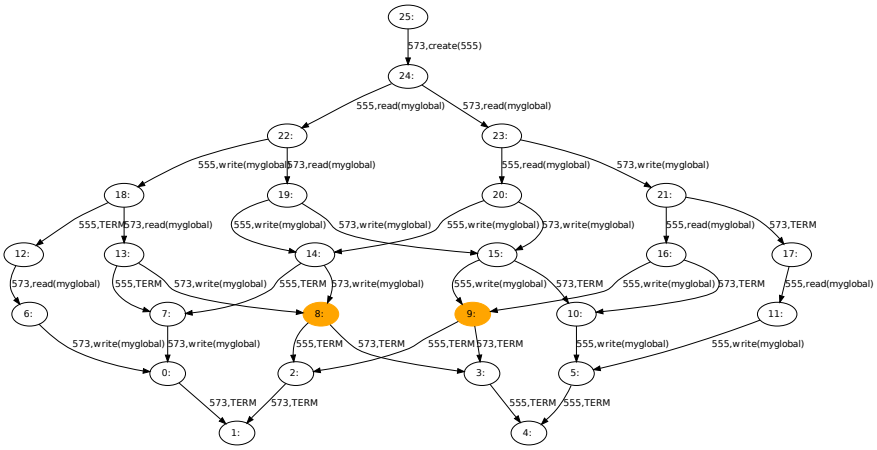

Fig. 5. A labeled transition system generated by applying the threading model. Colored vertices designate states with an imminent data race.

proach is to use the operational semantics from Section 3 and model-check if a low-level data race is possible.

In our interleaving semantics a data race will happen if in a state σ there are two threads θ_1, θ_2 and at least one of the threads is write enabled on a shared variable v, while the other one can read or write to v, i. e.:

$$\mathit{datarace}(\theta) = \mathit{enabled}(\mathtt{write}_v, \theta_1, \sigma) \wedge (\mathit{enabled}(\mathtt{write}_v, \theta_2, \sigma) \vee \mathit{enabled}(\mathtt{read}_v, \theta_2, \sigma))$$

Using this predicate low-level data races can be detected by checking (on-the-fly) whether there is a path such that a data race can be reached, which translates to the CTL formula

$$\mathbf{EF}\ \mathit{datarace}(v)$$

An example of such a transition system is shown in Figure 5. Global configurations are numbered nodes; the transition system considers program actions relating to shared variable *myglobal*, to which there are read and write accesses. We use *TERM* for the successful termination action ε; the number in a label represents the global transition relation $s \xrightarrow{\theta,\mathtt{action}} s'$. Global states 8 and 9 represent locations at which data races happen, because the two preceding actions in both states are unsynchronized write accesses.

There are a few caveats, though: Constructing the labeled transition system potentially results in an exponential blow-up both in the numbers of threads created and the number of thread operations. Moreover, there is a possibly unbounded number of threads, if thread creation happens in an (unbounded) loop.

Algorithm 2: On-the-fly reachability checking using BFS.

```
begin
    Σ_worklist ← {⟨{θ_main ↦ Π(main)}, κ_∅, λl.⊥⟩};
    Σ_visited ← ∅;
    while Σ_worklist ≠ ∅ do
        σ_current ← dequeue(Σ_worklist);
        if σ_current ⊨ φ then
            WARN(σ_current);
            return true
        foreach σ' ∈ {σ' | σ_current --θ,a-->_G σ' ∧ σ' ∉ Σ_visited} do
            enqueue(σ', Σ_worklist);
        enqueue(σ_current, Σ_visited);
    return false
```

In practice, any model-checking would use a k-bound for the maximum numbers of threads that a single thread can create *per local state.*

Finally, our interleaving semantics is a faithful abstraction of the real program by only considering thread-specific concepts and read/writes to shared variables. Mapping a threaded program to this abstraction is a non-trivial task when considering function calls and some subtleties of the POSIX standard. In Section 5 we point out some of the issues involved when analyzing real code.

Implementation Algorithm 2 outlines the implementation used to check whether a configuration satisfying a given predicate $\phi : \Sigma \to \mathbb{B}$ is reachable, and warns if an appropriate configuration satisfying ϕ is found (denoted by $\sigma \vDash \phi$).

A noteworthy characteristic of the algorithm is that the reachable states of the model are not generated *a priori* but during the analysis itself, i. e., on the fly. This happens at the **foreach**-loop where solely the immediate successors of $\sigma_{current}$ are explored.

The use of a breadth-first-search was motivated by the observation how the interleaved semantics influences the model. Different interleavings for termination of concurrently executing threads constitute a large part of the model. A depth-first search would explore all these interleavings, which are not interesting concerning data race detection.

Furthermore, the implementation is able to augment warnings with precise *witness*-information in contrast to the lockset algorithm. The `WARN` procedure is able to issue warnings with line numbers that can precisely locate the problem, hence substantially facilitating tracking down defects.

Soundness and Completeness Under the same assumptions as in section 4.1, the model-checking approach presented here is *sound and complete* up to the fixed thread bound k, i. e., if each program instruction that instantiates a thread

is successfully executed at most k times. Hence, imprecision is introduced whenever thread instantiation is nested within loops that exceed the thread bound during execution. In those cases the analysis is neither sound nor complete. Fortunately, such bugs manifest rarely in practice.

4.3 Combining Both Analyses: The Layered Approach

The lock set algorithm we introduced is designed for performance, at the cost of possible spurious warnings. Since it is sound, each variable for which the analysis yields a non-empty set of distinct locks protecting it, is regarded as safe.

On the other hand, model-checking offers precise results. However, the state-explosion problem often renders (detailed) models of concurrent programs too large for model-checking purposes.

A natural consequence is to use a combined layered approach:

1. The lock set algorithm yields a (global) lock set for each shared variable. Variable with a non-empty lock set are safe.
2. Apply model-checking to the remaining shared variables in isolation. If a data race is reachable, report that data race.

The usage of our two-stage approach for data races can essentially be thought of as a false-positive elimination for the static lock set-algorithm. It is important to note that the lock-set analysis does not worsen the precision of model-checking. It can be formally shown that if a non-empty lock set is found, a data race cannot be detected using the model-checking approach [12].

5 Experiments

The core ideas of our layered approach have been implemented on top of the industrial-strength analysis tool *Goanna* [8]. Goanna analyzes C/C++ code using static analysis and model checking to detect bugs in large scale code. For our purposes we made use of the fact that the tool can readily produce control flow graphs, allows model generation with custom labels based on syntactic abstraction, and supports a summary-based interprocedural analysis.

However, a number of simplifications were made: The maximum thread creation bound was set to 2, a pre-processing heuristics was used to detect the shared variables, and potential aliases as well as dynamic memory allocations were ignored. Moreover, for handling the threading semantics we inlined function calls, which is clearly not scalable, but sufficient for experiments.

All experiments were executed on a Mobile Core2Duo Processor with a clock frequency of 1.83 Ghz and 4 GB of memory running on Ubuntu Linux 9.10. We measured both the complete tool runtime including some internal computation done by Goanna (denoted T_{total}) as well as the wall clocktimes of the multi-threading analyses presented in this paper (denoted T_{MTA}).

Table 1. Evaluation results on benchmark examples.

Test Case	Lock set			Combined		
	Correct	T_{MTA}	T_{total}	Correct	T_{MTA}	T_{total}
`low race.c`	y	0.01 s	0.19 s	y	0.02 s	0.21 s
`low corrected.c`	y	0.01 s	0.18 s	y	0.01 s	0.18 s
`low extended.c`	n	0.01 s	0.20 s	y	0.06 s	0.25 s
`low readwrite.c`	n	0.01 s	0.15 s	y	0.27 s	0.41 s

Table 2. Evaluation results on OpenTFTP.

Analysis	# Races	Correct/Incorrect	T_{MTA}	T_{total}	$\frac{T_{\text{MTA}}}{T_{\text{total}}}$	$\frac{T_{\text{MTA}}}{\#Vars}$	$\frac{T_{\text{MTA}}}{kLOC}$
Lock set	15	4/11 (27 %)	7.58 s	38.63 s	19.6 %	0.47 s	3.03 s
Combined	0	n.a.	131.49 s	153.86 s	85.4 %	8.21 s	51.94 s
Combined*	4	4/0 (100 %)	2176.85 s	2194.36 s	99.2 %	136.05 s	869.69 s

Benchmark Examples Several classical examples were chosen to investigate the the effectiveness of the solution[7]. These include a simple low-level data race on a shared variable (`low_race.c`), its corrected version (`low_corrected.c`), as well as some more advanced examples: `low_extended.c` is free of races on two out of four shared variables, and (the absence of) races depends on the happens-before relation. The program `low_readwrite.c` features two locks, where at least one of them is required to read a shared variable, but both must be acquired when writing the shared variable. This is a typical implementation of a read/write lock where multiple readers are allowed, but only one writer.

We used both the lock set-approach and the layered approach. Table 1 shows the overall results. While runtime is negligible, it is important to note that the simple examples are handled correctly by the lock set algorithm, while the more advanced cases require the model checking step for false-positive elimination as shown in the Combined column.

Case Study OpenTFTP The TFTP server software OpenTFTP was used as a real-world software example. The size of the program is about 2.5 KLOC, and it features high functional complexity coupled with a lot of multi-threading and synchronization-related constructs. Worker threads are generated for incoming requests, and shared resources like sockets and locks are protected using mutexes. Furthermore, structured data types (`structs`) are used, whose impact on the precision can be evaluated. Obviously the analyses had to be considered in an interprocedural setting to obtain meaningful results.

Out of 23 globally defined variables, 16 were identified as potentially shared and written to by at least on concurrent thread. Two distinct threads were identified, one being the main thread while the other is the `processRequest` worker-thread which is started for each incoming request; hence, thread creation

[7] `http://www4.in.tum.de/~mund/races.tar.bz`

is nested inside a loop. The initial run reported a multi-threaded control-flow graph with 2,339 distinct control-states and 3,766 transitions, and data races on 15 out of 16 shared variables.[8] Inspection revealed that the software was not programmed with respect to the POSIX standard, but with respect to some hidden assumptions on Linux, exploiting the fact that concurrently executing threads can release any lock held by any thread. We adjusted our model for this; the modified approach is denoted *Combined**, yielding precise results. Table 2 shows the results for the data race analysis using the lock set algorithm, the combined approach based on the POSIX standard, and the modified model based on the specific implementation on Linux exploited by the software.

6 Conclusion and Future Work

We propose a static implementation of the Eraser lock-set algorithm to detect possible data races in software. This analysis is sound but may result in false warnings. We add a second analysis step that model checks if potential data races detected by the lock-set analysis, can ever occur during program execution. Our two-step analysis takes into account the semantics of the Pthreads library, and is precise at the cost of a higher analysis overhead.

In future work, the performance of the second step could be improved further: As we consider only reachability properties, we could apply a strong partial-order reduction by abstracting from all concrete sequences of actions and considering only all possible global states. Moreover, instead of inlining procedure some enriched summary information should be sufficient.

Other future work includes the analysis of other concurrency properties, such as deadlocks or high-level data races. Finally, the layered approach presented in this work may be applicable to other types of analysis. For properties where fast over-approximations exist, it may be possible to balance speed and precision in a similar way.

Acknowledgments. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and the Digital Economy and the Australian Research Council through the ICT Centre of Excellence program

References

1. C. Artho and A. Biere. Applying static analysis to large-scale, multithreaded Java programs. In *Proc. 13th ASWEC*, pages 68–75, Canberra, Australia, 2001. IEEE Computer Society Press.

[8] A multi-threaded control-flow graph embeds subgraphs of child threads into calls to `pthread_create`. The states and transitions thus correspond to local states; the number of global states is exponential in the number of local states and threads.

2. C. Artho, A. Biere, and K. Havelund. Using block-local atomicity to detect stale-value concurrency errors. In *Proc. 2nd Int. Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, volume 3299 of *LNCS*, pages 150–164, Taipei, Taiwan, 2004. Springer.
3. C. Artho, K. Havelund, and A. Biere. High-level data races. *Journal on Software Testing, Verification & Reliability (STVR)*, 13(4):220–227, 2003.
4. Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. *SIGPLAN Not.*, 36(5):203–213, May 2001.
5. E. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: SAT-based predicate abstraction for ANSI-C. In *Proc. TACAS 2005*, volume 3440 of *LNCS*, pages 570–574. Springer, 2005.
6. Ricardo Dias, V. Pessanha, and J. M. S. Loureno. Precise detection of atomicity violations. In *Proc. Haifa Verification Conf. (HVC 2012)*, Lecture Notes in Computer Science. Springer-Verlag, 11 2012.
7. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: a race and transaction-aware Java runtime. *SIGPLAN Not.*, 42(6):245–255, June 2007.
8. A. Fehnker, R. Huuck, P. Jayet, M. Lussenburg, and F. Rauch. Model checking software at compile time. In *Proc. TASE 2007*. IEEE Computer Society, 2007.
9. Cormac Flanagan, Stephen N. Freund, Marina Lifshin, and Shaz Qadeer. Types for atomicity: Static checking and inference for java. *ACM Trans. Program. Lang. Syst.*, 30(4):20:1–20:53, August 2008.
10. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(3):299, 1996.
11. J. M. S. Loureno, D. Sousa, B. C. Teixeira, and Ricardo Dias. Detecting concurrency anomalies in transactional memory programs. *Computer Science and Information Systems*, 8(2), 04 2011.
12. Jakob Mund. *Finding Common Defects in Multi-Threaded Programs at Compile Time*. PhD thesis, University of Augsburg, 2010.
13. M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. In *Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 308–319. ACM, 2006.
14. B. Nichols, D. Buttlar, and J. Farrell. *Pthreads Programming*. O'Reilly, 1996.
15. P. Pratikakis, J.S. Foster, and M. Hicks. LOCKSMITH: context-sensitive correlation analysis for race detection. *ACM SIGPLAN Notices*, 41(6):320–331, 2006.
16. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.
17. Edith Schonberg. On-the-fly detection of access anomalies. In *In Proc. SIGPLAN 1989 Conf. on Programming Language Design and Implementation (PLDI 1989)*, PLDI '89, pages 285–297, New York, NY, USA, 1989. ACM.
18. H. Seidl and V. Vojdani. Region analysis for race detection. *Static Analysis*, pages 171–187, 2010.
19. W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232, 2003.
20. C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *Proc. ATVA 2008*, volume 5311 of *LNCS*, pages 126–140. Springer, 2008.