Model Checking Race-freedom When "Sequential Consistency for Data-race-free Programs" is Guaranteed*

Wenhao Wu[1]✉, Jan Hückelheim[2], Paul D. Hovland[2], Ziqing Luo[1], and Stephen F. Siegel[1]

[1] University of Delaware, Newark DE 19716, USA
{wuwenhao,ziqing,siegel}@udel.edu
[2] Argonne National Laboratory, Lemont IL 60439, USA
{jhueckelheim,hovland}@anl.gov

Abstract. Many parallel programming models guarantee that if all sequentially consistent (SC) executions of a program are free of data races, then all executions of the program will appear to be sequentially consistent. This greatly simplifies reasoning about the program, but leaves open the question of how to verify that all SC executions are race-free. In this paper, we show that with a few simple modifications, model checking can be an effective tool for verifying race-freedom. We explore this technique on a suite of C programs parallelized with OpenMP.

Keywords: data race · model checking · OpenMP

1 Introduction

Every multithreaded programming language requires a memory model to specify the values a thread may obtain when reading a variable. The simplest such model is *sequential consistency* [22]. In this model, an execution is an interleaved sequence of the execution steps from each thread. The value read at any point is the last value that was written to the variable in this sequence.

There is no known efficient way to implement a full sequentially consistent model. One reason for this is that many standard compiler optimizations are invalid under this model. Because of this, most multithreaded programming languages (including language extensions) impose a requirement that programs do not have *data races*. A data race occurs when two threads access the same variable without appropriate synchronization, and at least one access is a write. (The notion of appropriate synchronization depends on the specific language.) For data race-free programs, most standard compiler optimizations remain valid. The Pthreads library is a typical example, in that programs with data races have no defined behavior, but race-free programs are guaranteed to behave in a sequentially consistent manner [25].

* Version 2 of 20 July 2023

Modern languages use more complex "relaxed" memory models. In this model, an execution is not a single sequence, but a set of events together with various relations on those events. These relations—e.g., *sequenced before*, *modification order*, *synchronizes with*, *dependency-ordered before*, *happens before* [21]—must satisfy a set of complex constraints spelled out in the language specification. The complexity of these models is such that only the most sophisticated users can be expected to understand and apply them correctly. Fortunately, these models usually provide an escape, in the form of a substantial and useful language subset which is guaranteed to behave sequentially consistently, as long as the program is race-free. Examples include Java [23], C and C++ since their 2011 versions (see [8] and [21, §5.1.2.4 Note 19]), and OpenMP [26, §1.4.6].

The "guarantee" mentioned above actually consists of two parts: (1) all executions of data race-free programs in the language subset are sequentially consistent, and (2) if a program in the language subset has a data race, then it has a sequentially consistent execution with a data race [8]. Putting these together, we have, for any program P in the language subset:

> (SC4DRF) *If all sequentially consistent executions of P are data race-free, then all executions of P are sequentially consistent.*

The consequence of this is that the programmer need only understand sequentially consistent semantics, both when trying to ensure P is race-free, and when reasoning about other aspects of the correctness of P. This approach provides an effective compromise between usability and efficient implementation.

Still, it is the programmer's responsibility to ensure that all sequentially consistent executions of the program are race-free. Unfortunately, this problem is undecidable [4], so no completely algorithmic solution exists. As a practical matter, detecting and eliminating races is considered one of the most challenging aspects of parallel program development. One source of difficulty is that compilers may "miscompile" racy programs, i.e., translate them in unintuitive, non-semantics-preserving ways [7]. After all, if the source program has a race, the language standard imposes no constraints, so any output from the compiler is technically correct.

Researchers have explored various techniques for race checking. Dynamic analysis tools (e.g., [18]) have experienced the most uptake. These techniques can analyze a single execution precisely, and report whether a race occurred, and sometimes can draw conclusions about closely related executions. But the behavior of many concurrent programs depends on the program input, or on specific thread interleavings, and dynamic techniques cannot explore all possible behaviors. Moreover, dynamic techniques necessarily analyze the behavior of the executable code that results from compilation. As explained above, racy programs may be miscompiled, even possibly removing the race, in which case a dynamic analysis is of limited use.

Approaches based on static analysis, in contrast, have the potential to verify race-freedom. This is extremely challenging, though some promising research prototypes have been developed (e.g., [10]). The most significant limitation is imprecision: a tool may report that race-free code has a possible race— a "false

alarm". Some static approaches are also not sound, i.e., they may fail to detect a race in a racy program; like dynamic tools, these approaches are used more as bug hunters than verifiers.

Finite-state model checking [15] offers an interesting compromise. This approach requires a finite-state model of the program, which is usually achieved by placing small bounds on the number of threads, the size of inputs, or other program parameters. The reachable states of the model can be explored through explicit enumeration or other means. This can be used to implement a sound and precise race analysis of the model. If a race is found, detailed information can be produced, such as a program trace highlighting the two conflicting memory accesses. Of course, if the analysis concludes the model is race-free, it is still possible that a race exists for larger parameter values. In this case, one can increase those values and re-run the analysis until time or computational resources are exhausted. If one accepts the "small scope hypothesis"—the claim that most defects manifest in small configurations of a system—then model checking can at least provide strong evidence for the absence of data races. In any case, the results provide specific information on the scope that is guaranteed to be race-free, which can be used to guide testing or further analysis.

The main limitation of model checking is state explosion, and one of the most effective techniques for limiting state explosion is *partial order reduction* (POR) [17]. A typical POR technique is based on the following observation: from a state s at which a thread t is at a "local" statement—i.e., one which commutes with all statements from other threads—then it is often not necessary to explore all enabled transitions from s; instead, the search can explore only the enabled transitions from t. Usually local statements are those that access only thread-local variables. But if the program is known to be race-free, shared variable accesses can also be considered "local" for POR. This is the essential observation at the heart of recent work on POR in the verification of Pthreads programs [29].

In this paper, we explore a new model checking technique that can be used to verify race-freedom, as well as other correctness properties, for programs in which threads synchronize through locks and barriers. The approach requires two simple modifications to the standard state reachability algorithm. First, each thread maintains a history of the memory locations accessed since its last synchronization operation. These sets are examined for races and emptied at specific synchronization points. Second, a novel POR is used in which only lock (release and acquire) operations are considered non-local. In Section 2, we present a precise mathematical formulation of the technique and a theorem that it has the claimed properties, including that it is sound and precise for verification of race-freedom of finite-state models.

Using the CIVL symbolic execution and model checking platform [31], we have implemented a prototype tool, based on the new technique, for verifying race-freedom in C/OpenMP programs. OpenMP is an increasingly popular directive-based language for writing multithreaded programs in C, C++, or For-

tran. A large sub-language of OpenMP has the SC4DRF guarantee.[3] While the theoretical model deals with locks and barriers, it can be applied to many OpenMP constructs that can be modeled using those primitives, such as atomic operations and critical sections. This is explained in Section 3, along with the results of some experiments applying our tool to a suite of C/OpenMP programs. In Section 4, we discuss related work and Section 5 concludes.

2 Theory

We begin with a simple mathematical model of a multithreaded program that uses locks and barriers for synchronization.

Definition 1. Let TID be a finite set of positive integers. A *multithreaded program with thread ID set* TID comprises

1. a set Lock of *locks*
2. a set Shared of *shared states*
3. for each $i \in \mathsf{TID}$:
   (a) a set Local_i, the *local states of thread* i, which is the union of five disjoint subsets, $\mathsf{Acquire}_i$, $\mathsf{Release}_i$, $\mathsf{Barrier}_i$, Nsync_i, and Term_i
   (b) a set Stmt_i of *statements*, which includes the *lock statements* $\mathsf{acquire}_i(l)$ and $\mathsf{release}_i(l)$ (for $l \in \mathsf{Lock}$), and the *barrier-exit* statement exit_i; all others statements are known as *nsync (non-synchronization) statements*
   (c) for each $\sigma \in \mathsf{Acquire}_i \cup \mathsf{Release}_i \cup \mathsf{Barrier}_i$, a local state $\mathsf{next}(\sigma) \in \mathsf{Local}_i$
   (d) for each $\sigma \in \mathsf{Acquire}_i \cup \mathsf{Release}_i$, a lock $\mathsf{lock}(\sigma) \in \mathsf{Lock}$
   (e) for each $\sigma \in \mathsf{Nsync}_i$, a nonempty set $\mathsf{stmts}(\sigma) \subseteq \mathsf{Stmt}_i$ of nsync statements and function

$$\mathsf{update}(\sigma) \colon \mathsf{stmts}(\sigma) \times \mathsf{Shared} \to \mathsf{Local}_i \times \mathsf{Shared}.$$

All of the sets Local_i and Stmt_i ($i \in \mathsf{TID}$) are pairwise disjoint. □

Each thread has a unique thread ID number, an element of TID. A local state for thread i encodes the values of all thread-local variables, including the program counter. A shared state encodes the values of all shared variables. (Locks are not considered shared variables.) A thread at an *acquire* state σ is attempting to acquire the lock $\mathsf{lock}(\sigma)$. At a *release* state, the thread is about to release a lock. At a *barrier* state, a thread is waiting inside a barrier. After executing one of the three operations, each thread moves to a unique next local state. A thread that reaches a *terminal* state has terminated. From an *nsync* state, any positive number of statements are enabled, and each of these statements may read and update the local state of the thread and/or the shared state.

For $i \in \mathsf{TID}$, the *local graph* of thread i is the directed graph with nodes Local_i and an edge $\sigma \to \sigma'$ if either (i) $\sigma \in \mathsf{Acquire}_i \cup \mathsf{Release}_i \cup \mathsf{Barrier}_i$ and

[3] Any OpenMP program that does not use non-sequentially consistent atomic directives, `omp_test_lock`, or `omp_test_nest_lock` [26, §1.4.6]

$\sigma' = \mathsf{next}(\sigma)$, or (ii) $\sigma \in \mathsf{Nsync}_i$ and there is some $\zeta' \in \mathsf{Shared}$ such that (σ', ζ') is in the image of $\mathsf{update}(\sigma)$.

Fix a multithreaded program P and let

$$\mathsf{LockState} = (\mathsf{Lock} \to \{0\} \cup \mathsf{TID})$$
$$\mathsf{State} = \left(\prod_{i \in \mathsf{TID}} \mathsf{Local}_i\right) \times \mathsf{Shared} \times \mathsf{LockState} \times 2^{\mathsf{TID}}.$$

A *lock state* specifies the owner of each lock. The owner is a thread ID, or 0 if the lock is free. The elements of State are the (global) *states* of P. A state specifies a local state for each thread, a shared state, a lock state, and the set of threads that are currently blocked at a barrier.

Let $i \in \mathsf{TID}$ and $L_i = \mathsf{Local}_i \times \mathsf{Shared} \times \mathsf{LockState} \times 2^{\mathsf{TID}}$. Define

$$\mathsf{enabled}_i \colon L_i \to 2^{\mathsf{Stmt}_i}$$
$$\lambda \mapsto \begin{cases} \{\mathsf{acquire}_i(l)\} & \text{if } \sigma \in \mathsf{Acquire}_i \wedge l = \mathsf{lock}(\sigma) \wedge \theta(l) = 0 \\ \{\mathsf{release}_i(l)\} & \text{if } \sigma \in \mathsf{Release}_i \wedge l = \mathsf{lock}(\sigma) \wedge \theta(l) = i \\ \{\mathsf{exit}_i\} & \text{if } \sigma \in \mathsf{Barrier}_i \wedge i \notin w \\ \mathsf{stmts}(\sigma) & \text{if } \sigma \in \mathsf{Nsync}_i \\ \emptyset & \text{otherwise.} \end{cases}$$

where $\lambda = (\sigma, \zeta, \theta, w) \in L_i$. This function returns the set of statements that are enabled in thread i at a given state. This function does not depend on the local states of threads other than i, which is why those are excluded from L_i. An acquire statement is enabled if the lock is free; a release is enabled if the calling thread owns the lock. A barrier exit is enabled if the thread is not currently in the barrier blocked set.

Execution of an enabled statement in thread i updates the state as follows:

$$\mathsf{execute}_i \colon \{(\lambda, t) \in L_i \times \mathsf{Stmt}_i \mid t \in \mathsf{enabled}_i(\lambda)\} \to L_i$$
$$(\lambda, t) \mapsto \begin{cases} (\sigma', \zeta, \theta[l \mapsto i], w') & \text{if } \sigma \in \mathsf{Acquire}_i \wedge t = \mathsf{acquire}_i(l) \wedge \sigma' = \mathsf{next}(\sigma) \\ (\sigma', \zeta, \theta[l \mapsto 0], w') & \text{if } \sigma \in \mathsf{Release}_i \wedge t = \mathsf{release}_i(l) \wedge \sigma' = \mathsf{next}(\sigma) \\ (\sigma', \zeta, \theta, w') & \text{if } \sigma \in \mathsf{Barrier}_i \wedge t = \mathsf{exit}_i \wedge \sigma' = \mathsf{next}(\sigma) \\ (\sigma', \zeta', \theta, w') & \text{if } \sigma \in \mathsf{Nsync}_i \wedge t \in \mathsf{stmts}(\sigma) \wedge \\ & \mathsf{update}(\sigma)(t, \zeta) = (\sigma', \zeta') \end{cases}$$

where $\lambda = (\sigma, \zeta, \theta, w)$ and in each case above

$$w' = \begin{cases} w \cup \{i\} & \text{if } \sigma' \in \mathsf{Barrier}_i \wedge w \cup \{i\} \neq \mathsf{TID} \\ \emptyset & \text{if } \sigma' \in \mathsf{Barrier}_i \wedge w \cup \{i\} = \mathsf{TID} \\ w & \text{otherwise.} \end{cases}$$

Note a thread arriving at a barrier will have its ID added to the barrier blocked set, unless it is the last thread to arrive, in which case all threads are released from the barrier.

At a given state, the set of enabled statements is the union over all threads of the enabled statements in that thread. Execution of a statement updates the state as above, leaving the local states of other threads untouched:

$$\begin{gathered}\mathsf{enabled}\colon \mathsf{State} \to 2^{\mathsf{Stmt}} \\ s \mapsto \bigcup_{j \in \mathsf{TID}} \mathsf{enabled}_j(\xi_j, \zeta, \theta, w) \\ \mathsf{execute}\colon \{(s,t) \in \mathsf{State} \times \mathsf{Stmt} \mid t \in \mathsf{enabled}(s)\} \to \mathsf{State} \\ (s,t) \mapsto \langle \xi[i \mapsto \sigma], \zeta', \theta', w' \rangle,\end{gathered}$$

where $s = \langle \xi, \zeta, \theta, w \rangle \in \mathsf{State}$, $t \in \mathsf{enabled}(s)$, $i = \mathsf{tid}(t)$, and $\mathsf{execute}_i(\xi_i, \zeta, \theta, w, t) = \langle \sigma, \zeta', \theta', w' \rangle$.

Definition 2. A *transition* is a triple $s \xrightarrow{t} s'$, where $s \in \mathsf{State}$, $t \in \mathsf{enabled}(s)$, and $s' = \mathsf{execute}(s,t)$. An *execution* α of P is a (finite or infinite) chain of transitions $s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots$. The *length* of α, denoted $|\alpha|$, is the number of transitions in α. □

Note that an execution is completely determined by its initial state s_0 and its statement sequence $t_1 t_2 \cdots$.

Having specified the semantics of the computational model, we now turn to the concept of the *data race*. The traditional definition requires the notion of "conflicting" accesses: two accesses to the same memory location conflict when at least one is a write. The following abstracts this notion:

Definition 3. A symmetric binary relation $\mathsf{conflict}$ on Stmt is a *conflict relation* for P if the following hold for all $t_1, t_2 \in \mathsf{Stmt}$:

1. if $(t_1, t_2) \in \mathsf{conflict}$ then t_1 and t_2 are nsync statements from different threads
2. if t_1 and t_2 are nsync statements from different threads and $(t_1, t_2) \notin \mathsf{conflict}$, then for all $s \in \mathsf{State}$, if $t_1, t_2 \in \mathsf{enabled}(s)$ then

$$\mathsf{execute}(\mathsf{execute}(s, t_1), t_2) = \mathsf{execute}(\mathsf{execute}(s, t_2), t_1).$$ □

Fix a conflict relation for P for the remainder of this section.

The next ingredient in the definition of *data race* is the *happens-before* relation. This is a relation on the set of *events* generated by an execution. An event is an element of $\mathsf{Event} = \mathsf{Stmt} \times \mathbb{N}$.

Definition 4. Let $\alpha = (s_0 \xrightarrow{t_1} s_1 \xrightarrow{t_2} \cdots)$ be an execution. The *trace of* α is the sequence of events $\mathsf{tr}(\alpha) = \langle t_1, n_1 \rangle \langle t_2, n_2 \rangle \cdots$, of length $|\alpha|$, where n_i is the number of $j \in [1, i]$ for which $\mathsf{tid}(t_j) = \mathsf{tid}(t_i)$. We write $[\alpha]$ for the set of events occurring in $\mathsf{tr}(\alpha)$. □

A trace labels the statements executed by a thread with consecutive integers starting from 1. Note the cardinality of $[\alpha]$ is $|\alpha|$, as no two events in $\mathsf{tr}(\alpha)$ are equal. Also, $[\alpha]$ is invariant under transposition of two adjacent commuting transitions from different threads.

Given an execution α, the *happens-before relation of* α, denoted $\mathsf{HB}(\alpha)$, is a binary relation on $[\alpha]$. It is the transitive closure of the union of three relations:

1. the intra-thread order relation

$$\{(\langle t_1, n_1\rangle, \langle t_2, n_2\rangle) \in [\alpha] \times [\alpha] \mid \mathsf{tid}(t_1) = \mathsf{tid}(t_2) \wedge n_1 < n_2\}.$$

2. the release-acquire relation. Say $\mathsf{tr}(\alpha) = e_1 e_2 \ldots$ and $e_i = \langle t_i, n_i\rangle$. Then (e_i, e_j) is in the release-acquire relation if there is some $l \in \mathsf{Lock}$ such that all of the following hold: (i) $1 \leq i < j \leq |\alpha|$, (ii) t_i is a release statement on l, (iii) t_j is an acquire statement on l, and (iv) whenever $i < k < j$, t_k is not an acquire statement on l.
3. the barrier relation. For any $e = \langle t, n\rangle \in [\alpha]$, let $i = \mathsf{tid}(t)$ and define

$$\mathsf{epoch}(e) = |\{e' \in [\alpha] \mid e' = \langle \mathsf{exit}_i, j\rangle \text{ for some } j \in [1, n]\}|,$$

the number of barrier exit events in thread i preceding or including e. The barrier relation is

$$\{(e, e') \in [\alpha] \times [\alpha] \mid \mathsf{epoch}(e) < \mathsf{epoch}(e')\}.$$

Two events "race" when they conflict but are not ordered by happens-before:

Definition 5. Let α be an execution and $e, e' \in [\alpha]$. Say $e = \langle t, n\rangle$ and $e' = \langle t', n'\rangle$. We say e and e' *race in* α if $(t, t') \in \mathsf{conflict}$ and neither (e, e') nor (e', e) is in $\mathsf{HB}(\alpha)$. The *data race relation of* α is the symmetric binary relation on $[\alpha]$

$$\mathsf{DR}(\alpha) = \{(e, e') \in [\alpha] \times [\alpha] \mid e \text{ and } e' \text{ race in } \alpha\}.$$ □

Now we turn to the problem of detecting data races. Our approach is to explore a modified state space. The usual state space is a directed graph with node set State and transitions for edges. We make two modifications. First, we add some "history" to the state. Specifically, each thread records the nsync statements it has executed since its last lock event or barrier exit. This set is checked against those of other threads for conflicts, just before it is emptied after its next lock event or barrier exit. The second change is a reduction: any state that has an enabled statement that is not a lock statement will have outgoing edges from only one thread in the modified graph.

A well-known technical challenge with partial order reduction concerns cycles in the reduced state space. We deal with this challenge by assuming that P comes with some additional information. Specifically, for each i, we are given a set R_i, with $\mathsf{Release}_i \cup \mathsf{Acquire}_i \subseteq R_i \subseteq \mathsf{Local}_i$, satisfying: any cycle in the local graph of thread i has at least one node in R_i. In general, the smaller R_i, the more effective the reduction. In many application domains, there are no cycles in the local graphs, so one can take $R_i = \mathsf{Release}_i \cup \mathsf{Acquire}_i$. For example, standard *for* loops in C, in which the loop variable is incremented by a fixed amount at each iteration, do not introduce cycles, because the loop variable will take on a new value at each iteration. For *while* loops, one may choose one node from the loop body to be in R_i. *Goto* statements may also introduce cycles and could require additions to R_i.

Definition 6. The *race-detecting state graph* for P is the pair $G = (V, E)$, where

$$V = \mathsf{State} \times \Big(\prod_{i \in \mathsf{TID}} 2^{\mathsf{Stmt}_i} \Big)$$

and $E \subseteq V \times \mathsf{Stmt} \times V$ consists of all $(\langle s, \mathbf{a} \rangle, t, \langle s', \mathbf{a}' \rangle)$ such that, letting σ_i be the local state of thread i in s,

1. $s \xrightarrow{t} s'$ is a transition in P
2. $\forall i \in \mathsf{TID}$, $\mathbf{a}'_i = \begin{cases} \mathbf{a}_i \cup \{t\} & \text{if } t \text{ is an nsync statement in thread } i \\ \emptyset & \text{if } t = \mathsf{exit}_0 \text{ or } i = \mathsf{tid}(t) \wedge \sigma_i \in R_i \\ \mathbf{a}_i & \text{otherwise} \end{cases}$
3. if there is some $i \in \mathsf{TID}$ such that $\sigma_i \notin R_i$ and thread i has an enabled statement at s, then $\mathsf{tid}(t)$ is the minimal such i. □

The race-detecting state graph may be thought of as a directed graph in which the nodes are V and edges are labeled by statements. Note that at a state where all threads are in the barrier, exit_0 is the only enabled statement in the race-detecting state graph, and its execution results in emptying all the $\mathbf{a}_i$. A lock event in thread i results in emptying $\mathbf{a}_i$ only.

Definition 7. Let P be a multithreaded program and $G = (V, E)$ the race-detecting state graph for P.

1. Let $u = \langle s, \mathbf{a} \rangle \in V$ and $i \in \mathsf{TID}$. We say *thread i detects a race in u* if there exist $j \in \mathsf{TID} \setminus \{i\}$, $t_1 \in \mathbf{a}_i$, and $t_2 \in \mathbf{a}_j$ such that $(t_1, t_2) \in \mathsf{conflict}$.
2. Let $e = v \xrightarrow{t} v' \in E$, $i = \mathsf{tid}(t)$, and σ' the local state of thread i at v'. We say e *detects a race* if $\sigma' \in R_i \cup \mathsf{Barrier}_i \cup \mathsf{Term}_i$ and thread i detects a race in v'.
3. We say G *detects a race from u* if E contains an edge that is reachable from u and detects a race. □

Definition 7 suggests a method for detecting data races in a multithreaded program. The nodes and edges of the race-detecting state graph reachable from an initial node are explored. (The order in which they are explored is irrelevant.) When an edge in thread i brings thread i to an R_i, barrier, or terminal state, the elements of $\mathbf{a}_i$ are compared with those in $\mathbf{a}_j$ for all $j \in \mathsf{TID} \setminus \{i\}$ to see if a conflict exists, and if so, a data race is reported. This approach is sound and precise in the following sense:

Theorem 1. *Let P be a multithreaded program, and $G = (V, E)$ the race-detecting state graph for P. Let $s_0 \in \mathsf{State}$ and let $u_0 = \langle s_0, \emptyset^{\mathsf{TID}} \rangle \in V$. Assume the set of nodes reachable from u_0 is finite. Then*

1. *P has an execution from s_0 with a data race if, and only if, G detects a race from u_0.*
2. *If there is a data race-free execution of P from s_0 to some state s_f with $\mathsf{enabled}(s_f) = \emptyset$ then there is a path in G from u_0 to a node with state component s_f.*

A proof of Theorem 1 is given in https://arxiv.org/abs/2305.18198.

Example 1. Consider the 2-threaded program represented in pseudocode:

$$t_1\colon\ \mathsf{acquire}(l_1);\ \texttt{x=1};\ \mathsf{release}(l_1);$$
$$t_2\colon\ \mathsf{acquire}(l_2);\ \texttt{x=2};\ \mathsf{release}(l_2);$$

where l_1 and l_2 are distinct locks. Let $R_i = \mathsf{Release}_i \cup \mathsf{Acquire}_i$ $(i = 1, 2)$. One path in the race-detecting state graph G executes as follows:

$$\mathsf{acquire}(l_1);\ \texttt{x=1};\ \mathsf{release}(l_1);\ \mathsf{acquire}(l_2);\ \texttt{x=2};\ \mathsf{release}(l_2);.$$

A data race occurs on this path since the two assignments conflict but are not ordered by happens-before. The race is not detected, since at each lock operation, the statement set in the other thread is empty. However, there is another path

$$\mathsf{acquire}(l_1);\ \texttt{x=1};\ \mathsf{acquire}(l_2);\ \texttt{x=2};\ \mathsf{release}(l_1);$$

in G, and on this path the race is detected at the release.

3 Implementation and Evaluation

We implemented a verification tool for C/OpenMP programs using the CIVL symbolic execution and model checking framework. This tool can be used to verify absence of data races within bounds on certain program parameters, such as input sizes and the number of threads. (Bounds are necessary so that the number of states is finite.) The tool accepts a C/OpenMP program and transforms it into CIVL-C, the intermediate verification language of CIVL. The CIVL-C program has a state space similar to the race-detecting state graph described in Section 2. The standard CIVL verifier, which uses model checking and symbolic execution techniques, is applied to the transformed code and reports whether the given program has a data race, and, if so, provides precise information on the variable involved in the race and an execution leading to the race.

The approach is based on the theory of Section 2, but differs in some implementation details. For example, in the theoretical approach, a thread records the set of non-synchronization statements executed since the thread's last synchronization operation. This data is used only to determine whether a conflict took place between two threads. Any type of data that can answer this question would work equally well. In our implementation, each thread instead records the set of memory locations read, and the set of memory locations modified, since the last synchronization. A conflict occurs if the read or write set of one thread intersects the write set of another read. As CIVL-C provides robust support for tracking memory accesses, this approach is relatively straightforward to implement by a program transformation.

In Section 3.1, we summarize the basics of OpenMP. In Section 3.2, we provide the necessary background on CIVL-C and the primitives used in the transformation. In Section 3.3, we describe the transformation itself. In Section 3.4, we report the results of experiments using this tool.

The experiments were run using CIVL revision 5815 (http://civl.dev). All artifacts necessary to reproduce the experiments, as well as the full results, are available at https://github.com/verified-software-lab/sc4drf.git.

3.1 Background on OpenMP

OpenMP is a pragma-based language for parallelizing programs written in C, C++ and Fortran [13]. OpenMP was originally designed and is still most commonly used for shared-memory parallelization on CPUs, although the language is evolving and supports an increasing number of parallelization styles and hardware targets. We introduce here the OpenMP features that are currently supported by our implementation in CIVL. An example that uses many of these features is shown in Figure 1.

The `parallel` construct declares the following structured block as a *parallel region*, which will be executed by all threads concurrently. Within such a parallel region, programmers can use *worksharing* constructs that cause certain parts of the code to be executed only by a subset of threads. Perhaps most importantly, the *loop worksharing construct* can be used inside a parallel region to declare a `omp for` loop whose iterations are mapped to different threads. The mapping of iterations to threads can be controlled through the `schedule` clause, which can take values including `static`, `dynamic`, `guided` along with an integer that defines the *chunk size*. If no schedule is explicitly specified, the OpenMP run time is allowed to use an arbitrary mapping. Furthermore, a structured block within a worksharing loop may be declared as `ordered`, which will cause this block to be executed sequentially in order of the iterations of the worksharing loop. Worksharing for non-iterative workloads is supported through the `sections` construct, which allows the programmer to define a number of different structured blocks of code that will be executed in parallel by different threads.

Programmers may use pragmas and clauses for `barrier`s, `atomic` updates, and locks. OpenMP supports named `critical` sections, allowing no more than one thread at a time to enter a critical section with that name, and unnamed critical sections that are associated with the same global mutex. OpenMP also offers `master` and `single` constructs that are executed only by the *master thread* or one arbitrary thread.

Variables are shared by all threads by default. Programmers may change the default, as well as the scope of individual variables, for each parallel region using the following clauses: `private` causes each thread to have its own variable instance, which is uninitialized at the start of the parallel region and separate from the original variable that is visible outside the parallel region. The `firstprivate` scope declares a private variable that is initialized with the value of the original variable, whereas the `lastprivate` scope declares a private variable that is uninitialized, but whose final value is that of the logically last worksharing loop iteration or lexically last section. The `reduction` clause initializes each instance to the neutral element, for example 0 for `reduction(+)`. Instances are combined into the original variable in an implementation-defined order.

```
1  #pragma omp parallel shared(b) private(i) shared(u,v)
2  { // parallel region: all threads will execute this
3    #pragma omp sections         // sections worksharing construct
4    {
5      #pragma omp section        // one thread will do this...
6      { b = 0; v = 0; }
7      #pragma omp section        // while another thread does this...
8      u = rand();
9    }
10   // loop worksharing construct partitions iterations by schedule.  Each thread has a
11   // private copy of b; these are added back to original shared b at end of loop...
12   #pragma omp for reduction(+:b) schedule(dynamic,1)
13   for (i=0; i<10; i++) {
14     b = b + i;
15     #pragma omp atomic seq_cst // atomic update to v
16     v+=i;
17     #pragma omp critical (collatz) // one thread at a time enters critical section
18     u = (u%2==0) ? u/2 : 3*u+1;
19   }
20 }
```

Fig. 1. OpenMP Example

CIVL can model OpenMP types and routines to query and control the number of threads (`omp_set_num_threads`, `omp_get_num_threads`), get the current thread ID (`omp_get_thread_num`), interact with locks (`omp_init_lock`, `omp_destroy_lock`, `omp_set_lock`, `omp_unset_lock`, and obtain the current wall clock time (`omp_get_wtime`).

3.2 Background on CIVL-C

The CIVL framework includes a front-end for preprocessing, parsing, and building an AST for a C program. It also provides an API for transforming the AST. We used this API to build a tool which consumes a C/OpenMP program and produces a CIVL-C "model" of the program. The CIVL-C language includes most of sequential C, including functions, recursion, pointers, structs, and dynamically allocated memory. It adds nested function definitions and primitives for concurrency and verification.

In CIVL-C, a thread is created by *spawning* a function: `$spawn f(...);`. There is no special syntax for shared or thread-local variables; any variable that is in scope for two threads is shared. CIVL-C uses an interleaving model of concurrency similar to the formal model of Section 2. Simple statements, such as assignments, execute in one atomic step.

Threads can synchronize using *guarded commands*, which have the form `$when (`e`)` S. The first atomic substatement of S is guaranteed to execute only from a state in which e evaluates to *true*. For example, assume thread IDs are numbered from 0, and a lock value of -1 indicates the lock is free. The *acquire* lock operation may be implemented as `$when (l<0) l=tid;`, where `l` is an integer shared variable and `tid` is the thread ID. A *release* is simply `l=-1;`.

A convenient way to spawn a set of threads is `$parfor (int` i`:`d`)` S. This spawns one thread for each element of the 1d-domain d; each thread executes S with i bound to one element of the domain. A 1d-domain is just a set of integers;

e.g., if a and b are integer expressions, the domain expression $a..b$ represents the set $\{a, a+1, \ldots, b\}$. The thread that invokes the `$parfor` is blocked until all of the spawned threads terminate, at which point the spawned threads are destroyed and the original thread proceeds.

CIVL-C provides primitives to constrain the interleaving semantics of a program. The program state has a single atomic lock, initially free. At any state, if there is a thread t that owns the atomic lock, only t is enabled. When the atomic lock is free, if there is some thread at a `$local_start` statement, and the first statement following `$local_start` is enabled, then among such threads, the thread with lowest ID is the only enabled thread; that thread executes `$local_start` and obtains the lock. When t invokes `$local_end`, t relinquishes the atomic lock. Intuitively, this specifies a block of code to be executed atomically by one thread, and also declares that the block should be treated as a local statement, in the sense that it is not necessary to explore all interleavings from the state where the local is enabled.

Local blocks can also be broken up at specified points using function `$yield`. If t owns the atomic lock and calls `$yield`, then t relinquishes the lock and does not immediately return from the call. When the atomic lock is free, there is no thread at a `$local_start`, a thread t is in a `$yield`, and the first statement following the `$yield` is enabled, then t may return from the `$yield` call and re-obtain the atomic lock. This mechanism can be used to implement the race-detecting state graph: thread i begins with `$local_start`, yields at each R_i node, and ends with `$local_end`.

CIVL's standard library provides a number of additional primitives. For example, the concurrency library provides a barrier implementation through a type `$barrier`, and functions to initialize, destroy, and invoke the barrier.

The *mem* library provides primitives for tracking the sets of memory locations (a variable, an element of an array, field of a struct, etc.) read or modified through a region of code. The type `$mem` is an abstraction representing a set of memory locations, or *mem-set*. The state of a CIVL-C thread includes a stack of mem-sets for writes and a stack for reads. Both stacks are initially empty. The function `$write_set_push` pushes a new empty mem-set onto the write stack. At any point when a memory location is modified, the location is added to the top entry on the write stack. Function `$write_set_pop` pops the write stack, returning the top mem-set. The corresponding functions for the read stack are `$read_set_push` and `$read_set_pop`. The library also provides various operations on mem-sets, such as `$mem_disjoint`, which consumes two mem-sets and returns *true* if the intersection of the two mem-sets is empty.

3.3 Transformation for Data Race Detection

The basic structure for the transformation of a parallel construct is shown in Figure 2. The user specifies on the command line the default number of threads to use in a parallel region. After this, two shared arrays are allocated, one to record the read set for each thread, and the other the write set. Rather than updating these arrays immediately with each read and write event, a thread

```
1  int nthreads = ...;
2  $mem reads[nthreads], writes[nthreads];
3  void check_conflict(int i, int j) {
4    $assert($mem_disjoint(reads[i], writes[j]) && $mem_disjoint(writes[i], reads[j]) &&
5            $mem_disjoint(writes[i], writes[j]));
6  }
7  void clear_all() {
8    for (int i=0; i<nthreads; i++) reads[i] = writes[i] = $mem_empty();
9  }
10 void run(int tid) {
11   void pop() { reads[tid]=$read_set_pop(); writes[tid]=$write_set_pop(); }
12   void push() { $read_set_push(); $write_set_push(); }
13   void check() {
14     for (int i=0; i<nthreads; i++) { if (i==tid) continue; check_conflict(tid, i); }
15   }
16   // local variable declarations
17   $local_start(); push(); S pop(); $local_end();
18 }
19 for (int i=0; i<nthreads; i++) reads[i] = writes[i] = $mem_empty();
20 $parfor (int tid:0..nthreads-1) run(tid);
21 check_and_clear_all();
```

Fig. 2. Translation of `#pragma omp parallel` S

updates them only at specific points, in such a way that the shared sets are current whenever a data race check is performed.

The auxiliary function `check_conflict` asserts no read-write or write-write conflict exists between threads i and j. Function `clear_all` clears all shared mem-sets.

Each thread executes function `run`. A local copy of each private variable is declared (and, for `firstprivate` variables, initialized) here. The body of this function is enclosed in a local region. The thread begins by pushing new entries onto its read and write stacks. As explained in Section 3.2, this turns on memory access tracking. The body S is transformed in several ways. First, references to the private variable are replaced by references to the local copy. Other OpenMP constructs are translated as follows.

Lock operations. Several OpenMP operations are modeled using locks. The `omp_set_lock` and `omp_unset_lock` functions are the obvious examples, but we also use locks to model the behavior of atomic and critical section constructs. In any case, a lock acquire operation is translated to

```
pop(); check(); $yield(); acquire(l); push();
```

The thread first pops its stacks, updating its shared mem-sets. At this point, the shared structures are up-to-date, and the thread uses them to check for conflicts with other threads. This conforms with Definition 7(2), that a race check occur upon arrival at an acquire location. It then yields to other threads as it attempts to acquire lock l. Once acquired, it pushes new empty entries onto its stack and resumes tracking. Similarly, a release statement becomes

```
pop(); check(); $yield(); release(l); push();
```

A similar sequence is inserted in any loop (e.g., a *while* loop or a *for* loop not in standard form) that may create a cycle in the local space, only without the release statement.

Barriers. An explicit or implicit barrier in S becomes

```
pop(); check(); $local_end(); $barrier_call();
if (tid==0) clear_all();
$barrier_call(); $local_start(); push();.
```

The CIVL-C `$barrier_call` function must be invoked outside of a local region, as it may block. Once all threads are in the barrier, a single thread (0) clears all the shared mem-sets. A second barrier call is used to prevent other threads from racing ahead before this clear completes.

Atomic and critical sections. An OpenMP atomic construct is modeled by introducing a global "atomic lock" which is acquired before executing the atomic statement and then released. The acquire and release actions are then transformed as described above. Similarly, a lock is introduced for each critical section name (and the anonymous critical section); this lock is acquired before entering a critical section with that name and released when departing.

Worksharing constructs. Upon arriving at a `for` construct, a thread invokes a function that returns the set of iterations for which the thread is responsible. The partitioning of the iteration space among the threads is controlled by the construct clauses and various command line options. If the construct specifies the distribution strategy precisely, then the model uses only that distribution. If the construct does not specify the distribution, then the decisions are based on command line options. One option is to explore all possible distributions. In this case, when the first thread arrives, a series of nondeterministic choices is made to construct an arbitrary distribution. The verifier explores all possible choices, and therefore all possible distributions. This enables a complete analysis of the loop's execution space, but at the expense of a combinatorial explosion with the number of threads or iterations. A different command line option allows the user to specify a particular default distribution strategy, such as *cyclic*. These options give the user some control over the completeness-tractability tradeoff. For `sections`, only cyclic distribution is currently supported, and a `single` construct is executed by the first thread to arrive at the construct.

3.4 Evaluation

We applied our verifier to a suite comprised of benchmarks from DataRaceBench (DRB) version 1.3.2 [35] and some examples written by us that use different concurrency patterns. As a basis for comparison, we applied a state-of-the-art static analyzer for OpenMP race detection, LLOV v.0.3 [10], to the same suite.[4]

[4] While there are a number of effective dynamic race detectors, the goal of those tools is to detect races on a particular execution. Our goal is more aligned with that

LLOV v.0.3 implements two static analyses. The first uses polyhedral analysis to identify data races due to loop-carried dependencies within OpenMP parallel loops [9]. It is unable to identify data races involving critical sections, atomic operations, master or single directives, or barriers. The second is a phase interval analysis to identify statements or basic blocks (and consequently memory accesses within those blocks) that may happen in parallel [10]. Phases are separated by explicit or implicit barriers and the minimum and maximum phase in which a statement or basic block may execute define the phase interval. The phase interval analysis errs in favor of reporting accesses as potentially happening in parallel whenever it cannot prove that they do not; consequently, it may produce false alarms.

The DRB suite exercises a wide array of OpenMP language features. Of the 172 benchmarks, 88 use only the language primitives supported by our CIVL OpenMP transformer (see Section 3.1). Some of the main reasons benchmarks were excluded include: use of C++, `simd` and `task` directives, and directives for GPU programming. All 88 programs also use only features supported by LLOV. Of the 88, 47 have data races and 41 are labeled race-free.

We executed CIVL on the 88 programs, with the default number of OpenMP threads for a parallel region bounded by 8 (with a few exceptions, described below). We chose cyclic distribution as the default for OpenMP *for* loops. Many of the programs consume positive integer inputs or have clear hard-coded integer parameters. We manually instrumented 68 of the 88, inserting a few lines of CIVL-C code, protected by a preprocessor macro that is defined only when the program is verified by CIVL. This code allows each parameter to be specified on the CIVL command line, either as a single value or by specifying a range. In a few cases (e.g., DRB055), "magic numbers" such as 500 appear in multiple places, which we replaced with an input parameter controlled by CIVL. These modifications are consistent with the "small scope" approach to verification, which requires some manual effort to properly parameterize the program so that the "scope" can be controlled.

We used the range 1..10 for inputs, again with a few exceptions. In three cases, verification did not complete within 3 minutes and we lowered these bounds as follows: for DRB043, thread bound 8 and input bound 4; for the Jacobi iteration kernel DRB058, thread bound 4 and bound of 5 on both the matrix size and number of iterations; for DRB062, thread bound 4 and input bound 5.

CIVL correctly identified 40 of the 41 data-race-free programs, failing only on DRB139 due to nested parallel regions. It correctly reported a data race for 45 of the 47 programs with data races, missing only DRB014 (Figure 3, middle) and DRB015. In both cases, CIVL reports a bound issue for an access to `b[i][j-1]` when $\texttt{i} > 0$ and $\texttt{j} = 0$, but fails to report a data race, even when bound checking is disabled.

LLOV correctly identified 46 of the 47 programs with data races, failing to report a data race for DRB140 (Figure 3, left). The semantics for reduction

of static analyzers: to cover as many executions as possible, including for different inputs, number of threads, and thread interleavings.

```c
// DRB140 (race)
int a, i;
#pragma omp parallel private(i)
{
 #pragma omp master
 a = 0;
 #pragma omp for reduction(+:a)
 for (i=0; i<10; i++)
  a = a + i;
}
```

```c
// DRB014 (race)
int n=100, m=100;
double b[n][m];
#pragma omp parallel for \
   private(j)
for (i=1;i<n;i++)
 for (j=0;j<m;j++)
  // out of bound access
  b[i][j]=b[i][j-1];
```

```c
// diffusion1 (race)
double *u, *v;
// alloc + init u, v
for (t=0; t<steps; t++) {
 #pragma omp parallel for
 for (i=1; i<n-1; i++) {
  u[i]=v[i]+c*(v[i-1]+v[i]);
 }
 u=v; v=u; // incorrect swap
}
```

Fig. 3. Excerpts from three benchmarks with data races: two from DataRaceBench (left and middle) and erroneous 1d-diffusion (right).

```c
// atomic3 (no race)
int x=0, s=0;
#pragma omp parallel sections \
   shared(x,s) num_threads(2)
{
 #pragma omp section
 {
  x=1;
  #pragma omp atomic write seq_cst
  s=1;
 }
 #pragma omp section
 {
  int done = 0;
  while (!done) {
   #pragma omp atomic read seq_cst
   done = s;
  }
  x=2;
 }
}
```

```c
// bar2 (no race)
// ...create/initialize locks l0, l1;
#pragma omp parallel num_threads(2)
{
  int tid = omp_get_thread_num();
  if (tid == 0) omp_set_lock(&l0);
  else if (tid == 1) omp_set_lock(&l1);
  #pragma omp barrier
  if (tid == 0) x=0;
  if (tid == 0) {
    omp_unset_lock(&l0);
    omp_set_lock(&l1);
  } else if (tid == 1) {
    omp_set_lock(&l0);
    omp_unset_lock(&l1);
  }
  if (tid == 1) x=1;
  #pragma omp barrier
  if (tid == 0) omp_unset_lock(&l1);
  else if (tid == 1) omp_unset_lock(&l0);
}
```

Fig. 4. Code for synchronization using an atomic variable (left) and a 2-thread barrier using locks (right).

specify that the loop behaves as if each thread creates a private copy, initially 0, of the shared variable `a`, and updates this private copy in the loop body. At the end of the loop, the thread adds its local copy onto the original shared variable. These final additions are guaranteed to not race with each other. In CIVL, this is modeled using a lock. However, there is no guarantee that these updates do not race with other code. In this example, thread 0 could be executing the assignment `a=0` while another thread is adding its local result to `a`—a data race. This race issue can be resolved by isolating the reduction loop with barriers.

LLOV correctly identified 38 out of 41 data-race-free programs. It reported false alarms for DRB052 (no support for indirect addressing), DRB054 (failure to propagate array dimensions and loop bounds from a variable assignment), and DRB069 (failure to properly model OpenMP lock behavior).

The DRB suite contains few examples with interesting interleaving dependencies or pointer alias issues. To complement the suite, we wrote 10 additional C/OpenMP programs based on widely-used concurrency patterns (cf. [1]):

- 3 implementations of a synchronization signal sent from one thread to another, using locks or busy-wait loops with critical sections or atomics;
- 3 implementations of a 2-thread barrier, using busy-wait loops or locks;
- 2 implementations of a 1d-diffusion simulation, one in which two copies of the main array are created by two separate malloc calls; one in which they are inside a single malloced object; and
- an instance of a single-producer, single-consumer pattern; and a multiple-producer, multiple-consumer version, both using critical sections.

For each program, we created an erroneous version with a data race, for a total of 20 tests. These codes are included in the experimental archive, and two are excerpted in Figure 4.

CIVL obtains the expected result in all 20. While we wrote these additional examples to verify that CIVL can reason correctly about programs with complex interleaving semantics or alias issues, for completeness we also evaluated them with LLOV. It should be noted, however, that the authors of LLOV warn that it "... does not provide support for the OpenMP constructs for synchronization..." and "... can produce False Positives for programs with explicit synchronizations with barriers and locks." [9] It is therefore unsurprising that the results were somewhat mixed: LLOV produced no output for 6 of our examples (the racy and race-free versions of diffusion2 and the two producer-consumer codes) and produced the correct answer on 7 of the remaning 14. On these problems, LLOV reported a race for both the racy and race-free version, with the exception of diffusion1 (Figure 3, right), where a failure to detect the alias between `u` and `v` leads it to report both versions as race-free.

CIVL's verification time is significantly longer than LLOV's. On the DRB benchmarks, total CIVL time for the 88 tests was 15 minutes, 48 seconds. Individual times ranged from 1 to 156 seconds: 64 took less than 5s, 81 took less than 30s, and 82 took less than 1 minute. (All CIVL runs used an M1 MacBook Pro with 16GB memory.) Total CIVL runtime on the 20 extra tests was 1 minute, 28 seconds. LLOV analyzes all 88 DRB problems in less than 15 seconds (on a standard Linux machine).

4 Related Work

By Theorem 1, if barriers are the only form of synchronization used in a program, only a single interleaving will be explored, and this suffices to verify race-freedom or to find all states at the end of each barrier epoch. This is well known in other contexts, such as GPU kernel verification (cf. [5]).

Prior work involving model checking and data races for unstructured concurrency includes Schemmel et al. [29]. This work describes a technique, using symbolic execution and POR, to detect defects in Pthreads programs. The approach involves intricate algorithms for enumerating configurations of prime event structures, each representing a set of executions. The completeness results deal with the detection of defects under the assumption that the program is

race-free. While the implementation does check for data races, it is not clear that the theoretical results guarantee a race will be found if one exists.

Earlier work of Elmas et al. describes a sound and precise technique for verifying race-freedom in finite-state lock-based programs [16]. It uses a bespoke POR-based model checking algorithm that associates significant and complex information with the state, including, for each shared memory location, a set of locks a thread should hold when accessing that location, and a reference to the node in the depth first search stack from which the last access to that location was performed.

Both of these model checking approaches are considerably more complex than the approach of this paper. We have defined a simple state-transition system and shown that a program has a data race if and only if a state or edge satisfying a certain condition is reachable in that system. Our approach is agnostic to the choice of algorithm used to check reachability. The earlier approaches are also path-precise for race detection, i.e., for each execution path, a race is detected if and only if one exists on that path. As we saw in the example following Theorem 1, our approach is not path-precise, nor does it have to be: to verify race-freedom, it is only necessary to find one race in one execution, if one exists. This partly explains the relative simplicity of our approach.

A common approach for verifying race-freedom is to establish *consistent correlation*: for each shared memory location, there is some lock that is held whenever that location is accessed. LOCKSMITH [27] is a static analysis tool for multithreaded C programs that takes this approach. The approach should never report that a racy program is race-free, but can generate false alarms, since there are race-free programs that are not consistently correlated. False alarms can also arise from imprecise approximations of the set of shared variables, alias analysis, and so on. Nevertheless, the technique appears very effective in practice.

Static analysis-based race-detection tools for OpenMP include OMPRacer [33]. OMPRacer constructs a static graph representation of the happens-before relation of a program and analyzes this graph, together with a novel whole-program pointer analysis and a lockset analysis, to detect races. It may miss violations as a consequence of unsound decisions that aim to improve performance on real applications. The tool is not open source. The authors subsequently released OpenRace [34], designed to be extensible to other parallelism dialects; similar to OMPRacer, OpenRace may miss violations. Prior papers by the authors present details of static methods for race detection, without a tool that implements these methods [32].

PolyOMP [12] is a static tool that uses a polyhedral model adapted for a subset of OpenMP. Like most polyhedral approaches, it works best for affine loops and is precise in such cases. The tool additionally supports may-write access relations for non-affine loops, but may report false alarms in that case. DRACO [36] also uses a polyhedral model and has similar drawbacks.

Hybrid static and dynamic tools include Dynamatic [14], which is based on LLVM. It combines a static tool that finds candidate races, which are subse-

quently confirmed with a dynamic tool. Dynamatic may report false alarms and miss violations.

ARCHER [2] is a tool that statically determines many sequential or provably non-racy code sections and excludes them from dynamic analysis, then uses TSan [30] for dynamic race detection. To avoid false alarms, ARCHER also encodes information about OpenMP barriers that are otherwise not understood by TSan. A follow-up paper discusses the use of the OMPT interface to aid dynamic race detection tools in correctly identifying issues in OpenMP programs [28], as well as SWORD [3], a dynamic tool that can stay within user-defined memory bounds when tracking races, by capturing a summary on disk for later analysis.

ROMP [18] is a dynamic/static tool that instruments executables using the DynInst library to add checks for each memory access and uses the OMPT interface at runtime. It claims to support all of OpenMP except `target` and `simd` constructs, and models "logical" races even if they are not triggered because the conflicting accesses happen to be scheduled on the same thread. Other approaches for dynamic race detection and tricks for memory and run-time efficient race bookkeeping during execution are described in [11, 19, 20, 24].

Deductive verification approaches have also been applied to OpenMP programs. An example is [6], which introduces an intermediate parallel language and a specification language based on permission-based separation logic. C programs that use a subset of OpenMP are manually annotated with "iteration contracts" and then automatically translated into the intermediate form and verified using VerCors and Viper. Successfully verified programs are guaranteed to be race-free. While these approaches require more work from the user, they do not require bounding the number of threads or other parameters.

5 Conclusion

In this paper, we introduced a simple model-checking technique to verify that a program is free from data races. The essential ideas are (1) each thread "remembers" the accesses it performed since its last synchronization operation, (2) a partial order reduction scheme is used that treats all memory accesses as local, and (3) checks for conflicting accesses are performed around synchronizations. We proved our technique is sound and precise for finite-state models, using a simple mathematical model for multithreaded programs with locks and barriers. We implemented our technique in a prototype tool based on the CIVL symbolic execution and model checking platform and applied it to a suite of C/OpenMP programs from DataRaceBench. Although based on completely different techniques, our tool achieved performance comparable to that of the state-of-the-art static analysis tool, LLOV v.0.3.

Limitations of our tool include incomplete coverage of the OpenMP specification (e.g., `target`, `simd`, and `task` directives are not supported); the need for some manual instrumentation; the potential for state explosion necessitating small scopes; and a combinatorial explosion in the mappings of threads to loop iterations, OpenMP sections, or single constructs. In the last case, we have

compromised soundness by selecting one mapping, but in future work we will explore ways to efficiently cover this space. On the other hand, in contrast to LLOV and because of the reliance on model checking and symbolic execution, we were able to verify the presence or absence of data races even for programs using unstructured synchronization with locks, critical sections, and atomics, including barrier algorithms and producer-consumer code.

Acknowledgements. This material is based upon work by the RAPIDS Institute, supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through Advanced Computing (SciDAC) program, under contract DE-AC02-06CH11357 and award DE-SC0021162. Support was also provided by U.S. National Science Foundation awards CCF-1955852 and CCF-2019309.

References

1. Andrews, G.R.: Foundations of Multithreaded, Parallel, and Distributed Programming. Addison-Wesley (2000), https://www.pearson.ch/HigherEducation/Pearson/EAN/9780201357523/Foundations-of-Multithreaded-Parallel-and-Distributed-Programming
2. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Ahn, D.H., Laguna, I., Schulz, M., Lee, G.L., Protze, J., Müller, M.S.: ARCHER: Effectively spotting data races in large OpenMP applications. In: 2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 53–62 (2016). https://doi.org/10.1109/IPDPS.2016.68
3. Atzeni, S., Gopalakrishnan, G., Rakamaric, Z., Laguna, I., Lee, G.L., Ahn, D.H.: SWORD: A bounded memory-overhead detector of OpenMP data races in production runs. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 845–854 (2018). https://doi.org/10.1109/IPDPS.2018.00094
4. Bernstein, A.J.: Analysis of programs for parallel processing. IEEE Transactions on Electronic Computers **EC-15**(5), 757–763 (1966). https://doi.org/10.1109/PGEC.1966.264565
5. Betts, A., Chong, N., Donaldson, A.F., Ketema, J., Qadeer, S., Thomson, P., Wickerson, J.: The design and implementation of a verification technique for GPU kernels. ACM Trans. Program. Lang. Syst. **37**(3) (May 2015). https://doi.org/10.1145/2743017
6. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. Int. J. Softw. Tools Technol. Transf. **23**(5), 741–763 (Oct 2021), https://doi.org/10.1007/s10009-020-00601-z
7. Boehm, H.J.: How to miscompile programs with "benign" data races. In: Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism. pp. 1–6. HotPar'11, USENIX Association, Berkeley, CA, USA (2011), http://dl.acm.org/citation.cfm?id=2001252.2001255
8. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 68–78. PLDI '08, Association for Computing Machinery, New York, NY, USA (2008). https://doi.org/10.1145/1375581.1375591

9. Bora, U., Das, S., Kukreja, P., Joshi, S., Upadrasta, R., Rajopadhye, S.: LLOV: A fast static data-race checker for OpenMP programs. ACM Transactions on Architecture and Code Optimization (TACO) **17**(4), 1–26 (2020). https://doi.org/10.1145/3418597
10. Bora, U., Vaishay, S., Joshi, S., Upadrasta, R.: OpenMP aware MHP analysis for improved static data-race detection. In: 2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC). pp. 1–11 (2021). https://doi.org/10.1109/LLVMHPC54804.2021.00006
11. Boushehrinejadmoradi, N., Yoga, A., Nagarakatte, S.: On-the-fly data race detection with the enhanced OpenMP series-parallel graph. In: Milfeld, K., de Supinski, B.R., Koesterke, L., Klinkenberg, J. (eds.) OpenMP: Portable Multi-Level Parallelism on Modern Systems. pp. 149–164. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-58144-2_10
12. Chatarasi, P., Shirako, J., Kong, M., Sarkar, V.: An extended polyhedral model for SPMD programs and its use in static data race detection. In: Ding, C., Criswell, J., Wu, P. (eds.) Languages and Compilers for Parallel Computing. pp. 106–120. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-52709-3_10
13. Dagum, L., Menon, R.: OpenMP: an industry standard API for shared-memory programming. IEEE computational science and engineering **5**(1), 46–55 (1998). https://doi.org/10.1109/99.660313
14. Davis, M.J.: Dynamatic: An OpenMP Race Detection Tool Combining Static and Dynamic Analysis. Undergraduate research scholars thesis, Texas A&M University (2021), https://oaktrust.library.tamu.edu/handle/1969.1/194411
15. Edmund M. Clarke, J., Grumberg, O., Kroening, D., Peled, D., Veith, H.: Model Checking. MIT press, Cambridge, MA, USA, 2 edn. (2018), https://mitpress.mit.edu/books/model-checking-second-edition
16. Elmas, T., Qadeer, S., Tasiran, S.: Precise race detection and efficient model checking using locksets. Tech. Rep. MSR-TR-2005-118, Microsoft Research (2006), https://www.microsoft.com/en-us/research/publication/precise-race-detection-and-efficient-model-checking-using-locksets/
17. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems — An Approach to the State-Explosion Problem, Lecture Notes in Computer Science, vol. 1032. Springer (1996). https://doi.org/10.1007/3-540-60761-7
18. Gu, Y., Mellor-Crummey, J.: Dynamic data race detection for OpenMP programs. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis (2018). https://doi.org/10.1109/SC.2018.00064
19. Ha, O.K., Jun, Y.K.: Efficient thread labeling for on-the-fly race detection of programs with nested parallelism. In: Kim, T.h., Adeli, H., Kim, H.k., Kang, H.j., Kim, K.J., Kiumi, A., Kang, B.H. (eds.) Software Engineering, Business Continuity, and Education. pp. 424–436. Springer, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-27207-3_47
20. Ha, O.K., Kuh, I.B., Tchamgoue, G.M., Jun, Y.K.: On-the-fly detection of data races in OpenMP programs. In: Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging. pp. 1–10. PADTAD 2012, Association for Computing Machinery, New York, NY, USA (2012). https://doi.org/10.1145/2338967.2336808
21. International Organization for Standardization: ISO/IEC 9899:2018. Information technology — Programming languages — C (2018), https://www.iso.org/standard/74528.html

22. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Transactions on Computers **C-28**(9), 690–691 (Sep 1979). https://doi.org/10.1109/TC.1979.1675439
23. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 378–391. POPL '05, Association for Computing Machinery, New York, NY, USA (2005). https://doi.org/10.1145/1040305.1040336
24. Mellor-Crummey, J.: On-the-fly detection of data races for programs with nested fork-join parallelism. In: Supercomputing'91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing. pp. 24–33. IEEE (1991). https://doi.org/10.1145/125826.125861
25. Open Group: IEEE Std 1003.1: Standard for information technology—Portable Operating System Interface (POSIX(R)) base specifications, issue 7: General concepts: Memory synchronization (2018), https://pubs.opengroup.org/onlinepubs/9699919799/basedefs/V1_chap04.html#tag_04_12
26. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2021), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-2.pdf, version 5.2
27. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Practical static race detection for C. ACM Transactions on Programming Languages and Systems **33**, 3:1–3:55 (January 2011). https://doi.org/10.1145/1889997.1890000
28. Protze, J., Hahnfeld, J., Ahn, D.H., Schulz, M., Müller, M.S.: OpenMP tools interface: Synchronization information for data race detection. In: de Supinski, B.R., Olivier, S.L., Terboven, C., Chapman, B.M., Müller, M.S. (eds.) Scaling OpenMP for Exascale Performance and Portability. pp. 249–265. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-65578-9_17
29. Schemmel, D., Büning, J., Rodríguez, C., Laprell, D., Wehrle, K.: Symbolic partial-order execution for testing multi-threaded programs. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I. pp. 376–400. Springer-Verlag, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-53288-8_18
30. Serebryany, K., Iskhodzhanov, T.: ThreadSanitizer: Data race detection in practice. In: Proceedings of the Workshop on Binary Instrumentation and Applications. p. 62–71. WBIA '09, Association for Computing Machinery, New York, NY, USA (2009). https://doi.org/10.1145/1791194.1791203
31. Siegel, S.F., Zheng, M., Luo, Z., Zirkel, T.K., Marianiello, A.V., Edenhofner, J.G., Dwyer, M.B., Rogers, M.S.: CIVL: The Concurrency Intermediate Verification Language. In: SC15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, New York (Nov 2015). https://doi.org/10.1145/2807591.2807635, article no. 61, pages 1–12
32. Swain, B., Huang, J.: Towards incremental static race detection in OpenMP programs. In: 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness). pp. 33–41. IEEE (2018). https://doi.org/10.1109/Correctness.2018.00009
33. Swain, B., Li, Y., Liu, P., Laguna, I., Georgakoudis, G., Huang, J.: OMPRacer: A scalable and precise static race detector for OpenMP programs. In: SC20: International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–14. IEEE (2020). https://doi.org/10.1109/SC41405.2020.00058
34. Swain, B., Liu, B., Liu, P., Li, Y., Crump, A., Khera, R., Huang, J.: OpenRace: An open source framework for statically detecting data

races. In: 2021 IEEE/ACM 5th International Workshop on Software Correctness for HPC Applications (Correctness). pp. 25–32. IEEE (2021). https://doi.org/10.1109/Correctness54621.2021.00009
35. Verma, G., Shi, Y., Liao, C., Chapman, B., Yan, Y.: Enhancing DataRaceBench for evaluating data race detection tools. In: 2020 IEEE/ACM 4th International Workshop on Software Correctness for HPC Applications (Correctness). pp. 20–30 (2020). https://doi.org/10.1109/Correctness51934.2020.00008
36. Ye, F., Schordan, M., Liao, C., Lin, P.H., Karlin, I., Sarkar, V.: Using polyhedral analysis to verify OpenMP applications are data race free. In: 2018 IEEE/ACM 2nd International Workshop on Software Correctness for HPC Applications (Correctness). pp. 42–50. IEEE (2018). https://doi.org/10.1109/Correctness.2018.00010

A A Reduction Theorem

The goal of this appendix is to prove Theorem 1. To do this, we will first consider programs without barriers, as this simplifies many aspects of the argument. Then we show that Theorem 1 can be easily obtained from the result about programs without barriers.

In this section, we prove a general reduction theorem for programs without barriers. This theorem follows the standard "ample set" approach to partial order reduction. It is shown that if subsets of enabled transitions satisfy specific axioms, then the reduction is sound for detecting data races. The reduction theorem will be a key step in the proof of Theorem 1.

A.1 Statement of the Reduction Theorem

Let P be a program without barriers. This means $\mathsf{Barrier}_i = \emptyset$ for all $i \in \mathsf{TID}$. No exit_i statement occurs in any execution. The wait set component of the state (2^{TID}) has no impact on the $\mathsf{enabled}$ or $\mathsf{execute}$ functions and can be ignored.

Definition 8. A *state graph* of P is a triple $G = (V, E, \mathsf{state})$, where

1. V is a set of *nodes*,
2. $E \subseteq V \times \mathsf{Stmt} \times V$ is a set of *edges*,
3. $\mathsf{state} \colon V \to \mathsf{State}$,
4. if $(u, t, v) \in E$ then $\mathsf{state}(u) \xrightarrow{t} \mathsf{state}(v)$ is a transition,
5. for all $u, v, v' \in V$ and $t \in \mathsf{Stmt}$, $(u, t, v), (u, t, v') \in E \implies v = v'$. □

Fix a state graph $G = (V, E, \mathsf{state})$ of P.
For $u, v \in V$ and $t \in \mathsf{Stmt}$, write $u \xrightarrow{t} v$ for (u, t, v), and define

$$\mathsf{enabled}(u) = \mathsf{enabled}(\mathsf{state}(u))$$
$$\mathsf{ample}_G(u) = \{t \in \mathsf{Stmt} \mid \exists v \in V \,.\, u \xrightarrow{t} v \in E\}.$$

Clearly, $\mathsf{ample}_G(u) \subseteq \mathsf{enabled}(u)$. Also, if $t \in \mathsf{ample}_G(u)$, then there is a unique $v \in V$ such that $u \xrightarrow{t} v \in E$.

Definition 9. A *path in* G is a finite or infinite sequence of nodes and edges $\alpha = (u_0 \xrightarrow{t_1} u_1 \xrightarrow{t_2} \cdots)$ such that $u_0 \xrightarrow{t_1} u_1 \in E$, $u_1 \xrightarrow{t_2} u_2 \in E$, ... The *execution defined by* α is the execution $\overline{\alpha} = (\mathsf{state}(u_0) \xrightarrow{t_1} \mathsf{state}(u_1) \xrightarrow{t_2} \cdots)$. □

The fact that $\overline{\alpha}$ is an execution follows from Definition 8(4).

Definition 10. We say G is *dense* if all of the following hold:

1. for all $u \in V$, if $\mathsf{enabled}(u) \neq \emptyset$ then $\mathsf{ample}_G(u) \neq \emptyset$
2. for all $u \in V$, if $t \in \mathsf{ample}_G(u)$, $t' \in \mathsf{enabled}(u)$, and $\mathsf{tid}(t) = \mathsf{tid}(t')$ then $t' \in \mathsf{ample}_G(u)$

3. for all $u \in V$, if $\mathsf{ample}_G(u) \neq \mathsf{enabled}(u)$ then all statements in $\mathsf{ample}_G(u)$ are nsync statements
4. for any cycle in G (a path of positive length from some node to itself), there is some $v \in V$ on the cycle with $\mathsf{ample}_G(v) = \mathsf{enabled}(v)$. □

We can now state the Reduction Theorem:

Theorem 2. *Let P be a multithreaded program, $G = (V, E, \mathsf{state})$ a dense state graph for P, and $u_0 \in V$. Assume the set of nodes reachable from u_0 is finite.*

1. *If there is an execution from $\mathsf{state}(u_0)$ with a data race then there is a path β in G from u_0 such that $\overline{\beta}$ has a data race.*
2. *If there is an execution from $\mathsf{state}(u_0)$ to a state s_f with $\mathsf{enabled}(s_f) = \emptyset$, then there is a path in G from u_0 to a node u_f with $\mathsf{state}(u_f) = s_f$.*

A.2 Key Lemmas

In this section, we prove some basic lemmas involving commuting transitions and the data race relation. These will be used in the proof of Theorem 2. We continue assuming P has no barriers.

Suppose t is an nsync statement. It follows from the definitions that no statement from another thread can enable or disable t. Moreover, t cannot enable or disable any statement from another thread. This is made precise as follows:

Lemma 1. *Let t_1 be an nsync statement, $t_2 \in \mathsf{Stmt}$, and $s \in \mathsf{State}$. Assume $\mathsf{tid}(t_1) \neq \mathsf{tid}(t_2)$. Then*

1. *if $t_2 \in \mathsf{enabled}(s)$ then $t_1 \in \mathsf{enabled}(s) \iff t_1 \in \mathsf{enabled}(\mathsf{execute}(s, t_2))$*
2. *if $t_1 \in \mathsf{enabled}(s)$ then $t_2 \in \mathsf{enabled}(s) \iff t_2 \in \mathsf{enabled}(\mathsf{execute}(s, t_1))$.*

Proof. Let $p = \mathsf{tid}(t_1)$ and $q = \mathsf{tid}(t_2)$. Say $s = \langle \xi, \zeta, \theta \rangle$.

Proof of (1): Assume $t_2 \in \mathsf{enabled}(s)$. Let $s' = \mathsf{execute}(s, t_2)$. Say $s' = \langle \xi', \zeta', \theta' \rangle$. We have $\xi'_p = \xi_p$, i.e., executing a statement in thread q does not change the local state of thread p. As $\xi_p \in \mathsf{Nsync}$,

$$\mathsf{enabled}_p(\xi_p, \zeta, \theta) = \mathsf{enabled}_p(\xi_p, \zeta', \theta').$$

Hence

$$\mathsf{Stmt}_p \cap \mathsf{enabled}(s) = \mathsf{enabled}_p(\xi_p, \zeta, \theta) = \mathsf{enabled}_p(\xi'_p, \zeta', \theta') = \mathsf{Stmt}_p \cap \mathsf{enabled}(s').$$

It follows that $t_1 \in \mathsf{enabled}(s) \iff t_1 \in \mathsf{enabled}(s')$.

Proof of (2): Assume $t_1 \in \mathsf{enabled}(s)$. Let $s' = \mathsf{execute}(s, t_1)$. As t_1 is an nsync statement, it does not change the state of the locks. Say $s' = \langle \xi', \zeta', \theta \rangle$. We have $\xi'_q = \xi_q$ since executing a statement in thread p does not change the local state of thread q. If t_2 is an acquire or release statement, then $t_2 \in \mathsf{enabled}(s) \iff t_2 \in \mathsf{enabled}(s')$ since these depend only on the local state of q and the state of the locks. If t_2 is an nsync statement, then the desired result follows from part (1), swapping t_1 and t_2. □

Lemma 2. *Let α' be an execution and α a prefix of α'. Then* $\mathsf{DR}(\alpha) \subseteq \mathsf{DR}(\alpha')$.

Proof. Note $\mathsf{tr}(\alpha)$ is a prefix of $\mathsf{tr}(\alpha')$ and $[\alpha] \subseteq [\alpha']$. Suppose $e, f \in [\alpha]$ and $(e, f) \in \mathsf{HB}(\alpha')$. We will show $(e, f) \in \mathsf{HB}(\alpha)$.

Since $\mathsf{HB}(\alpha')$ is a transitive closure, there is a finite sequence of elements of $[\alpha']$

$$e = e_1, e_2, \ldots, e_r = f$$

such that for each i $(1 \leq i < r)$ either (e_i, e_{i+1}) is in the intra-thread order relation or in the release-acquire relation. In either case, e_i occurs before e_{i+1} in the sequence $\mathsf{tr}(\alpha')$. Since $f \in [\alpha]$, that means all $e_i \in [\alpha]$. It follows that $(e, f) \in \mathsf{HB}(\alpha)$.

Suppose e and e' race in α. Then neither happens before the other in α. By the paragraph above, neither happens before the other in α'. Since the statements of e and e' conflict, e and e' race in α'. □

Lemma 3. *Let $\alpha = (s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_n)$ be a finite execution with trace $e_1 \cdots e_n$. Suppose $1 \leq i \leq n-1$, $(e_i, e_{i+1}) \notin \mathsf{HB}(\alpha)$, and $(t_i, t_{i+1}) \notin \mathsf{conflict}$. Then there exists $s' \in \mathsf{State}$ such that*

$$\alpha' = (s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_{i-1}} s_{i-1} \xrightarrow{t_{i+1}} s' \xrightarrow{t_i} s_{i+1} \xrightarrow{t_{i+2}} \cdots \xrightarrow{t_n} s_n)$$

is an execution. Moreover, $[\alpha] = [\alpha']$, $\mathsf{HB}(\alpha) = \mathsf{HB}(\alpha')$, and $\mathsf{DR}(\alpha) = \mathsf{DR}(\alpha')$.

Proof. We have $t_{i+1} \in \mathsf{enabled}(\mathsf{execute}(s_{i-1}, t_i))$. Let $p = \mathsf{tid}(t_i)$ and $q = \mathsf{tid}(t_{i+1})$. Since $(e_i, e_{i+1}) \notin \mathsf{HB}(\alpha)$, $p \neq q$. We claim all of the following hold:

$$t_{i+1} \in \mathsf{enabled}(s_{i-1}) \tag{1}$$

$$t_i \in \mathsf{enabled}(\mathsf{execute}(s_{i-1}, t_{i+1})) \tag{2}$$

$$\mathsf{execute}(\mathsf{execute}(s_{i-1}, t_{i+1}), t_i) = \mathsf{execute}(\mathsf{execute}(s_{i-1}, t_i), t_{i+1}). \tag{3}$$

If the claim holds, take $s' = \mathsf{execute}(s_{i-1}, t_{i+1})$, and the existence of α follows. The proof of the claim is in two cases.

Case 1: t_{i+1} is an nsync statement. Then (1) follows from Lemma 1(1), and (2) follows from Lemma 1(2). If t_i is a lock statement, then (3) follows from the definition of execute, as the two statements modify distinct components of the state. If t_i is an nsync statement, then (3) follows from the assumption that $(e_i, e_{i+1}) \notin \mathsf{conflict}$.

Case 2: t_{i+1} is a lock statement. If t_i is an nsync statement then the claim follows by an argument similar to that of Case 1. So suppose t_i is also a lock statement. We claim that the two lock statements operate on different locks. If both statements are acquires, then they must operate on different locks, as an acquire statement is only enabled when the lock is free. If t_i is an acquire and t_{i+1} a release, then again they must operate on different locks, else $p = q$, as only the thread owning the lock can perform a release operation on that lock. If t_i is a release and t_{i+1} an acquire then again they must operate on different locks, else $(e_i, e_{i+1}) \in \mathsf{HB}(\alpha)$. Finally, if both statements are releases, then they must

operate on different locks, since a release statement is only enabled when the lock is owned by some thread. The claim is now clear, since the two statements are operating on distinct components of the lock state.

We have $[\alpha] = [\alpha']$ since $p \neq q$. We must show $\mathsf{HB}(\alpha) = \mathsf{HB}(\alpha')$. The intra-thread relation is the same in α and α'. The release-acquire relation is also the same since it is not the case that one statement is an acquire statement and the other a release statement on the same lock. Hence the transitive closure of the union of the two relations is identical. Finally, the data race relation depends only on happens-before and $\mathsf{conflict}$, so $\mathsf{DR}(\alpha) = \mathsf{DR}(\alpha')$. □

Lemma 4. *Let* $\alpha = (s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_n)$ *be a finite execution. Suppose* $1 \leq i \leq n-1$*, and* t_i *and* t_{i+1} *are conflicting nsync statements from different threads. Then there exist* $s', s'' \in \mathsf{State}$ *such that*

$$\alpha' = (s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_{i-1}} s_{i-1} \xrightarrow{t_{i+1}} s' \xrightarrow{t_i} s'')$$

is an execution. In addition, α' *contains a data race.*

Proof. Say $\mathsf{tr}(\alpha) = e_1 \cdots e_n$. The existence of s' and s'' follows from Lemma 1. Since t_i and t_{i+1} are in different threads, $\mathsf{tr}(\alpha') = e_1 \cdots e_{i-1} e_{i+1} e_i$. Clearly neither (e_i, e_{i+1}) nor (e_{i+1}, e_i) is in $\mathsf{HB}(\alpha')$. As $(t_i, t_{i+1}) \in \mathsf{conflict}$, $(e_i, e_{i+1}) \in \mathsf{DR}(\alpha')$. □

A.3 Proof of the Reduction Theorem

We now complete the proof of Theorem 2. Let $s_0 = \mathsf{state}(u_0)$ and

$$\alpha = (s_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} s_n)$$

be a finite execution.

We first prove part 1. Suppose α contains a data race. We construct a path in G with a data race. To do this, we show there exists a sequence of pairs $(\beta_i, \gamma_i)_{i \geq 0}$ satisfying all of the following for all $i \geq 0$:

1. β_i is a path in G starting at u_0
2. $|\beta_i| = i$
3. β_i is a prefix of β_{i+1}
4. γ_i is an execution
5. the final state of $\overline{\beta}_i$ is the initial state of γ_i
6. $\overline{\beta}_i \circ \gamma_i$ has a data race
7. $|\gamma_{i+1}| \leq |\gamma_i|$.

In addition we will show there exists $m \geq 0$ such that

$$|\gamma_m| = 0.$$

It follows that β_m is a path in G starting at u_0 and $\overline{\beta}_m$ has a data race.

Let β_0 be the path of length 0 at u_0, and $\gamma_0 = \alpha$.

Suppose $i \geq 0$ and β_i and γ_i have been constructed to satisfy (1)–(7). If $|\gamma_i| = 0$, let $\beta_{i+1} = \beta_i$ and $\gamma_{i+1} = \gamma_i$; clearly (1)–(7) still hold.

So assume $|\gamma_i| > 0$. We construct β_{i+1} and γ_{i+1} as follows.

Let u be the final node of β_i, so $s = \mathsf{state}(u)$ is the initial state of γ_i. Since $|\gamma_i| > 0$, there is at least one enabled statement at s. By Definition 10(1), $\mathsf{ample}_G(u) \neq \emptyset$.

Case 0: $\mathsf{ample}_G(u) = \mathsf{enabled}(u)$. Say $\gamma_i = (s \xrightarrow{t} s') \circ \gamma'$. Since

$$t \in \mathsf{enabled}(s) = \mathsf{enabled}(u) = \mathsf{ample}_G(u),$$

there is a unique $v \in V$ such that $u \xrightarrow{t} v$. Let $\beta_{i+1} = \beta_i \circ (u \xrightarrow{t} v)$ and $\gamma_{i+1} = \gamma'$. Hence

$$\overline{\beta}_{i+1} \circ \gamma_{i+1} = \overline{\beta}_i \circ (s \xrightarrow{t} s') \circ \gamma' = \overline{\beta}_i \circ \gamma_i.$$

It is clear that conditions (1)–(7) hold. Moreover $|\gamma_{i+1}| < |\gamma_i|$. We say that $(\beta_{i+1}, \gamma_{i+1})$ are formed by performing a *shift* on (β_i, γ_i).

Case 1: $\mathsf{ample}_G(u) \neq \mathsf{enabled}(u)$. In this case, $\mathsf{ample}_G(u)$ consists of nsync statements. We explore two sub-cases.

Case 1a: γ_i contains a statement in $\mathsf{ample}_G(u)$. Let t be the first statement in $\mathsf{ample}_G(u)$ in γ_i. Let $p = \mathsf{tid}(t)$. We claim t is the first statement in γ_i from thread p. To see this, let t' be the first statement in γ_i from p. Since p is at an nsync state in s, t' is an nsync statement. By Lemma 1, t' is enabled at s. By Definition 10(2), $t' \in \mathsf{ample}_G(u)$. Hence $t' = t$.

We now transform γ_i by repeatedly transposing t with the statement to its left. Let k be the index of t in γ_i. We construct a sequence of executions $\gamma_{i,j}$ $(0 \leq j \leq k)$. For each j, conditions (4)–(6) will hold with $\gamma_{i,j}$ in place of γ_i, and (7) will hold with $\gamma_{i,j}$ in place of γ_{i+1}. In addition, t will occur in index j of $\gamma_{i,j}$.

We start at k and work down to 0. Let $\gamma_{i,k} = \gamma_i$. Suppose $j \geq 1$ and $\gamma_{i,j}$ has been defined. We will define $\gamma_{i,j-1}$. Let t' be the statement at position $j-1$ in $\gamma_{i,j}$.

If t does not conflict with t', then by Lemma 3, t and t' may be transposed to yield a new execution $\gamma_{i,j-1}$, and $\overline{\beta}_i \circ \gamma_{i,j-1}$ has the same data race relation as $\overline{\beta}_i \circ \gamma_{i,j}$, which has a race.

If t and t' are conflicting nsync statements, by Lemma 4 there is an execution $\gamma_{i,j-1}$ which is the prefix of length $j+1$ of the result of transposing t and t', and $\overline{\beta}_i \circ \gamma_{i,j-1}$ again has a race.

Now $\gamma_{i,0}$ has t in position 0. Let $(\beta_{i+1}, \gamma_{i+1})$ be the result of performing a shift on $(\beta_i, \gamma_{i,0})$. Note $|\gamma_{i+1}| < |\gamma_i|$.

Case 1b: γ_i does not contain a statement in $\mathsf{ample}_G(u)$. In this case we will insert a new statement, appending it to β_i.

Choose any $t \in \mathsf{ample}_G(u)$ and let $p = \mathsf{tid}(t)$. There is no statement in γ_i from thread p. (As argued above, if there were such a statement, then the first statement in γ_i from p would be in $\mathsf{ample}_G(u)$.)

Since t is an nsync statement, it cannot be disabled by statements from other processes. Hence we can extend γ_i to $\tilde{\gamma}_i$ by appending t and one more state. Note $\overline{\beta}_i \circ \tilde{\gamma}_i$ also contains a data race. Now we can apply the technique of Case 1a to $\tilde{\gamma}_i$ to move t to position 0 while maintaining the race, and then perform a shift. In the worst case, $|\gamma_{i+1}| = |\gamma_i|$.

Termination. We have to show that for any $i \geq 0$, there is some $j > i$ such that the construction of (β_j, γ_j) does not involve Case 1b.

So suppose there is some $i \geq 0$ such that for all $j \geq i$, the construction of (β_j, γ_j) involves Case 1b. The paths

$$\beta_i, \beta_{i+1}, \beta_{i+2}, \ldots$$

satisfy (1)–(3). For $j \geq i$, let u_j be the terminal node of β_j.

Since all the u_j are reachable from u_0, and the set of nodes reachable from u_0 is finite, there exist integers j, k such that $i \leq j < k$ and $u_j = u_k$. Hence $u_j, u_{j+1}, \ldots, u_k$ form a cycle in G. By Definition 10(4), there is some l with $j \leq l < k$ and $\mathsf{ample}_G(u_l) = \mathsf{enabled}(u_l)$. But then, the construction of $(\beta_{l+1}, \gamma_{l+1})$ would use Case 0, a contradiction.

This completes the proof of part 1 of Theorem 2.

We now turn to the proof of part 2. Suppose α does not contain a data race, but ends at a state with no enabled statement. The proof is mostly the same as that of part 1. Rather than repeat the proof, we summarize the parts that change.

First, replace invariant 6 ($\overline{\beta}_i \circ \gamma_i$ has a data race) with the following: the final state of execution γ_i is the final state of α. As $\gamma_0 = \alpha$, this holds for $i = 0$.

The second change is that Case 1b of the inductive step never occurs. Recall that in Case 1, u is the final node of path β_i, $s = \mathsf{state}(u)$ is the initial node of γ_i, $\mathsf{ample}_G(u) \neq \emptyset$, and $\mathsf{ample}_G(u) \neq \mathsf{enabled}(u)$. We wish to show that γ_i contains a statement in $\mathsf{ample}_G(u)$. Let t be any element of $\mathsf{ample}_G(u)$ and $p = \mathsf{tid}(t)$. Since thread p is at an nsync state at u, but is at a terminal or acquire state at the end of α (and therefore at the end of γ_i), γ_i must contain a statement from p. Let t' be the first statement from p in γ_i. Then the nsync statement t' is also enabled at u, as statements from other threads cannot enable an nsync statement, and therefore $t' \in \mathsf{ample}_G(u)$. Therefore Case 1a must hold.

The third observation is that in Case 1a, it is never the case that two transitions being transposed conflict, because of the assumption that α is data race-free.

Hence the transformation carried out in the inductive step involves only a sequence of transpositions of adjacent commuting transitions. As such a transposition does not change the final state, the final state of γ_i is invariant. At termination, γ_i is empty and β terminates at a node with state the final state of α.

B The Race Detection Theorem

In this section, we prove Theorem 1. We first prove the theorem under the assumption that P has no barriers.

B.1 Preliminaries

Recall from the discussion before Definition 6 that we assume the program P comes with sets R_i, for each $i \in \mathsf{TID}$. For each i, $\mathsf{Release}_i \cup \mathsf{Acquire}_i \subseteq R_i \subseteq \mathsf{Local}_i$, and any cycle in the local graph of thread i has at least one node in R_i.

Definition 11. For $s \in \mathsf{State}$ and $i \in \mathsf{TID}$, we say thread i is *normal at* s if the local state of thread i in s is not in R_i and thread i has an enabled statement at s. If v is a node in a state graph, thread i is *normal at* v if thread i is normal at $\mathsf{state}(v)$.

As we are assuming P has no barriers, if thread i is normal at s then thread i must be at an nsync state at s.

Lemma 5. *Let P be a multithreaded program without barriers and $G = (V, E)$ the race-detecting state graph for P. Any infinite path in G with only a finite number of nodes has a node at which no thread is normal.*

Proof. Let $\zeta = (u_0 \xrightarrow{t_1} u_1 \xrightarrow{t_2} \cdots)$ be an infinite path in G. For $i \geq 0$, let m_i be the number of threads that are normal at u_i. Let $a \geq 0$ be an integer for which $m_a = \min\{m_k \mid k \geq 0\}$. Suppose $m_a > 0$. We will arrive at a contradiction.

Let i be the minimal ID of a normal thread at u_a. We show by induction that for all $b \geq a$, $m_b = m_a$ and thread i is the minimal ID of a normal thread at u_b. The inductive hypothesis clearly holds for $b = a$.

Suppose $b > a$ and the inductive hypothesis holds for $b-1$. Then $m_{b-1} = m_a$ and i is the minimal ID of a normal thread at u_{b-1}. Moreover, t_b is an nsync statement in thread i, by Definition 6. For $j \in \mathsf{TID} \setminus \{i\}$, thread j is normal at u_{b-1} if and only if thread j is normal at u_b, as nsync statements cannot be enabled or disabled by actions from other threads. Since a was chosen to minimize the m_k, we must have $m_b = m_{b-1} = m_a$ and i is still the minimal thread ID of a thread that is normal at u_b. This proves the inductive step.

Projecting onto the local state of thread i, the suffix of ζ starting from u_a yields an infinite path in the local graph of thread i with a finite number of states, but which never passes through a state in R_i. This path must contain a cycle, contradicting the assumption that every cycle in the local graph has a state in R_i. □

Lemma 6. *Let P be a multithreaded program without barriers and $G = (V, E)$ a race-detecting state graph for P. Define* $\mathsf{state}\colon V \to \mathsf{State}$ *by* $\mathsf{state}(\langle s, \mathbf{a}\rangle) = s$. *Then (V, E, state) is a dense state graph.*

Proof. Let $\langle s, \mathbf{a}\rangle \in V$ and $T = \mathsf{ample}_G(\langle s, \mathbf{a}\rangle)$. There are two cases: In the first case, a thread is normal at s. Then T consists of all enabled statements in thread i, where i is the minimal ID of such a thread. In the second case, there is no normal thread at s. Then T consists of all enabled statements. In either case, if there is an enabled statement at $\langle s, \mathbf{a}\rangle$ then T is nonempty. Hence Definition 10(1) holds.

In the first case, T consists of all enabled statements in one thread. In the second case T consists of all enabled statements in all threads. Hence Definition 10(2) holds.

If T is a proper subset of the enabled statements then the first case holds. In this case, the statements of T come from an nsync state, hence are all nsync statements. So Definition 10(3) holds.

If α is a cyclic path in G, then there is an infinite path in G with a finite number of nodes, formed by repeating α infinitely. By Lemma 5, there is a node u on α at which no thread is normal. According to Definition 6, u is fully enabled, satisfying Definition 10(4). □

One direction of the proof of Theorem 1 is straightforward:

Lemma 7. *If G detects a race from u_0 then P has an execution starting from s_0 with a data race.*

Proof. Let u be the target node of an edge in G that detects a race.

Let t_1 and t_2 be a pair of conflicting statements stored at u, and $i = \mathsf{tid}(t_1)$ and $j = \mathsf{tid}(t_2)$. There is a path ζ from u_0 that terminates at u. At least one edge in ζ is labeled with t_1; let e_1 be the last such edge. When e_1 executes, t_1 is added to $\mathbf{a}_i$ and is not removed by any statement on ζ after that point. Since any release statement in thread i removes all statements from $\mathbf{a}_i$, no release statement in ζ occurs in thread i after t_1. Define e_2 similarly.

In the trace resulting from ζ, the events corresponding to e_1 and e_2 cannot be ordered by happens-before, because there is no release event in thread i after e_i, and no release event in thread j after e_j. Hence the path defines an execution with a data race. □

The other direction is more involved.

B.2 Block Decomposition

We continue with our assumption that P has no barriers. Let $G = (V, E, \mathsf{state})$ be the race-detecting state graph.

Given a finite path $\alpha = (u_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} u_n)$ in G, we define integers N and $i_0, \ldots, i_N$ as follows. Let $i_0 = 0$. Assume $j \geq 0$ and i_j has been defined. If $i_j = n$ then $N = j$. If $i_j < n$, define i_{j+1} by

$$C = \{k \in [i_j + 2, n] \mid \mathsf{tid}(t_k) \neq \mathsf{tid}(t_{i_j+1}) \text{ or } \text{thread } \mathsf{tid}(t_k) \text{ is not normal at } u_{k-1}\}$$

$$i_{j+1} = \begin{cases} \min(C) - 1 & \text{if } C \neq \emptyset \\ n & \text{otherwise.} \end{cases}$$

Definition 12. Let $\alpha = (u_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} u_n)$ be a finite path in G. Define N and $i_0, \ldots, i_N$ as above. The *block length* of α is N. For $1 \leq j \leq N$, let

$$B_j = (u_{i_{j-1}} \xrightarrow{t_{i_{j-1}+1}} \cdots \xrightarrow{t_{i_j}} u_{i_j}).$$

The path B_j is a *block* of α. Define $\mathsf{tid}(B_j) = \mathsf{tid}(t)$, where $t = t_{i_{j-1}+1}$ is the first statement of B_j. B_j is *initial* if thread $\mathsf{tid}(B_j)$ is normal at $u_{i_{j-1}}$. □

Note: a path of length 0 has block length 0. A path of positive length has a positive block length.

Lemma 8. *Let α be a finite path in G, and $B_1, \ldots, B_N$ the blocks of α. All of the following hold:*

1. *every block has length at least one and all statements in the block come from the same thread*
2. *if $N \geq 1$, $\alpha = B_1 \circ \cdots \circ B_N$*
3. *every lock statement in α occurs as the first statement of some block B*
4. *all statements in a block B other than the first statement of B are nsync statements*
5. *for $1 \leq i \leq N-1$: if B_{i+1} is initial then B_i is initial and $\mathsf{tid}(B_i) < \mathsf{tid}(B_{i+1})$*
6. *if B is a non-initial block and u is the initial node of B, then no thread is normal at u and $\mathsf{ample}_G(u) = \mathsf{enabled}(u)$, and*
7. *if B is a non-initial block, u is any node in B, and $i \in \mathsf{TID} \setminus \{\mathsf{tid}(B)\}$ then thread i is not normal at u.*

Proof. The first four follow immediately from the definition of *block*.

(5). Assume B_i is not initial; we will show B_{i+1} is not initial. Let s be the initial state of B_i, and s' the initial state of B_{i+1}. Let $j = \mathsf{tid}(B_i)$. As B_i is not initial, thread j is not normal at s. Since the first statement of B_i is in thread j, Definition 6 implies that no thread is normal at s. Since all statements in B_i are in thread j, all threads k, for $k \neq j$, are in the same state at s' that they were in s, so thread k is not normal at s'. But thread j also cannot be normal at s', else the block B_i would not end at s'. Hence no thread is normal at s'; in particular, B_{i+1} is not initial.

Now assume B_{i+1}—and therefore B_i—are initial. Let t be the last statement of B_i and t' the first statement of B_{i+1}. Let s be the state immediately preceding t and s' the state immediately following t and preceding t'. As B_{i+1} is initial, thread $\mathsf{tid}(t')$ is normal at s'. Thread $\mathsf{tid}(t)$ is normal at s. (If t is the first

transition of B_i, this follows because B_i is initial. If t is not the first transition of B_i, then this follows from the definition of *block*.) Now $\mathsf{tid}(t') \neq \mathsf{tid}(t)$, else the two transitions would be in the same block. As t is an nsync statement, it cannot enable or disable transitions in other threads, so t' is also enabled at s, and thread $\mathsf{tid}(t')$ is also normal at s, as the local state of thread $\mathsf{tid}(t')$ is the same at s or s'. By Definition 6, $\mathsf{tid}(t) < \mathsf{tid}(t')$. This means $\mathsf{tid}(B_i) < \mathsf{tid}(B_{i+1})$.

(6). If some thread were normal at u, then the ample set for u would consist of the enabled statements from one of the normal threads, so the thread of the first transition of B would be normal at u, i.e., B would be initial. Since B is not initial, no thread is normal at u, and therefore u is fully enabled in G.

(7). By (6), at the initial state of B, no thread is normal. All statements in B belong to thread $\mathsf{tid}(B)$, and these statements cannot enable an nsync statement in another thread. All threads other than thread $\mathsf{tid}(B)$ remain at their original local states at all nodes in B. It follows that these threads are not normal at any node in B. □

From Lemma 8, we conclude that the block decomposition of a path α in G has a prefix of initial blocks in which the block tid is strictly increasing. In particular, no two initial blocks have the same tid. This is followed by a sequence of non-initial blocks, each of which starts with a statement in a non-normal thread p, and is followed by some number of nsync statements in p. Each of these nsync statements is executed from a state at which thread p, and only thread p, is normal.

Definition 13. Let α be a finite path in G and B a block of α. We say B is *complete* if thread $\mathsf{tid}(B)$ is not normal at the final node of B. We say α is *complete* if every block of α is complete. □

Note: a path of length 0 has 0 blocks and is vacuously complete.

Lemma 9. *Let α be a finite path in G. Then every block of α other than the last is complete.*

Proof. Let B be a block of α that is not the last block, $i = \mathsf{tid}(B)$, and let $u \xrightarrow{t} v$ be the last edge of B, and $v \xrightarrow{t'} w$ the first edge of the next block B'.

Assume thread i is normal at v; we will arrive at a contradiction. By Definition 6, t' must belong to a thread that is normal at v, so t' is an nsync statement. Let $j = \mathsf{tid}(t')$. We have $i \neq j$, else B' and B would form a single block. Now t' must be enabled at u, since no statement from another thread can enable an nsync statement. Moreover, j is normal at u, since the local state of thread j is the same at u and v. By Definition 6, $i < j$. But since i is normal at v, there is some nsync statement in thread i enabled at $\mathsf{state}(v)$, whence $j < i$, a contradiction. □

Hence, only the last block of α could be incomplete. However, we now show that under reasonable assumptions, α can be extended to a complete path.

Lemma 10. *Let $u_0 \in V$ and assume the set of nodes in G reachable from u_0 is finite. Then every finite path starting from u_0 can be extended to a complete path.*

Proof. Say $\alpha = (u_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} u_n)$. If $n = 0$, α is vacuously complete, so assume $n \geq 1$.

By Lemma 9, every block other than the last is complete. Suppose the last block is not complete. Let $i = \mathsf{tid}(t_n)$. Thread i is normal at u_n. Hence some (nsync) statement t in thread i is enabled at u_n. Moreover, i must be the least ID of a normal thread at u_n, because if there were some other normal thread j at u_n, with $j < i$, then thread j would also be normal at u_{n-1}, contradicting the assumption that α is a path in G. Hence $t \in \mathsf{ample}_G(u_n)$. Append t, and the resulting node, to α, and the result is still a path in G.

Repeat the above as long as the path is not complete. We claim that eventually, the path must become complete. Otherwise, there is an infinite path in G, starting from a node reachable from u_0, in which thread i is normal at every node, contradicting Lemma 5. □

Note: if a finite execution has a data race, then any extension will also have a data race, by Lemma 2.

Definition 14. Let $\alpha = (u_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} u_n)$ be a finite path in G with $\mathsf{tr}(\overline{\alpha}) = e_1 \ldots e_n$. Define $N, i_0, \ldots, i_N$, and $B_1, \ldots, B_N$ as in Definition 12. For $1 \leq j \leq N$, let

$$b_j = e_{i_{j-1}+1} \cdots e_{i_j}.$$

The sequence b_j is the *event sequence* of block B_j; it is a sequence of elements of $[\overline{\alpha}]$. The sequence $b_1 \cdots b_N$ is the *block-event string* of α; it is a sequence of event sequences. □

We will adopt the following notational shorthand. Suppose $b_1, \ldots, b_n$ are event sequences; say

$$b_j = \langle t_{j,1}, n_{j,1} \rangle \langle t_{j,2}, n_{j,2} \rangle \cdots \langle t_{j,m_j}, n_{j,m_j} \rangle.$$

We will write *there is an execution of the form*

$$s_0 \xrightarrow{b_1} s_1 \xrightarrow{b_2} \cdots \xrightarrow{b_n} s_n$$

to mean there is an execution of the form

$$s_0 \xrightarrow{t_{1,1}} s_{0,1} \xrightarrow{t_{1,2}} \cdots \xrightarrow{t_{1,m_1}} s_1 \xrightarrow{t_{2,1}} s_{1,1} \xrightarrow{t_{2,2}} \cdots \xrightarrow{t_{n,m_n}} s_n.$$

A similar notation will be used for paths in place of executions.

Lemma 11. *Let G be the race-detecting state graph of P. Let α be a path in G with initial node v, blocks $B_1, \ldots, B_N$, and block-event string $b_1 \cdots b_N$. Suppose*

1. $1 \leq i < N$,

2. *B_i is not an initial block,*
3. *B_{i+1} is complete,*
4. *the first statement of B_i does not happen before the first statement of B_{i+1}, and*
5. *no statement of B_i conflicts with a statement of B_{i+1}.*

Then $b_1 \cdots b_{i-1} b_{i+1} b_i b_{i+2} \cdots b_N$ is the block-event string of a path α' from v in G with $\mathsf{DR}(\overline{\alpha'}) = \mathsf{DR}(\overline{\alpha})$.

Proof. Let v_j be the node in α just before b_{j+1} $(0 \le j < N)$ and let v_N be the final node of α. Let $s_j = \mathsf{state}(v_j)$. The execution $\overline{\alpha}$ has the form

$$s_0 \xrightarrow{b_1} \cdots \xrightarrow{b_{i-1}} s_{i-1} \xrightarrow{b_i} s_i \xrightarrow{b_{i+1}} s_{i+1} \xrightarrow{b_{i+2}} \cdots \xrightarrow{b_N} s_N.$$

The idea is to transpose b_i and b_{i+1}. Since there are no conflicts, all nsync statements in one block commute with any statement in the other block, by Lemma 3; furthermore these transpositions do not alter the data race relation. The only statements which are possibly not nsync are the first statements of the two blocks, but assumption (4) and Lemma 3 guarantee these commute, again without altering the data race relation. Therefore there is an execution of the form

$$s_0 \xrightarrow{b_1} \cdots \xrightarrow{b_{i-1}} s_{i-1} \xrightarrow{b_{i+1}} s' \xrightarrow{b_i} s_{i+1} \xrightarrow{b_{i+2}} \cdots \xrightarrow{b_N} s_N,$$

with data race relation $\mathsf{DR}(\overline{\alpha})$. We show the execution corresponds to a path in G.

Let $p = \mathsf{tid}(B_i)$ and $q = \mathsf{tid}(B_{i+1})$. By assumption (4), $p \neq q$. Since B_i is not initial, by Lemma 8(6), no thread is normal at s_{i-1}, and $\mathsf{ample}_G(v_{i-1}) = \mathsf{enabled}(v_{i-1})$. Hence the first statement t_1 of b_{i+1} is in $\mathsf{ample}_G(v_{i-1})$. Let $\tilde{s}_1 = \mathsf{execute}(s_{i-1}, t_1)$. Thus there is a node $\tilde{v}_1 \in V$ with $\mathsf{state}(\tilde{v}_1) = \tilde{s}_1$, and an edge $v_{i-1} \xrightarrow{t_1} \tilde{v}_1 \in E$.

Now the local state of thread q is the same at s_{i-1} and s_i. Hence the local state of q at $\tilde{s}_1$ is the same as the local state of q at $\mathsf{execute}(s_i, t_1)$. In particular, if there is a second statement t_2 in B_{i+1}, then t_2 is a sync statement that is also enabled at $\tilde{s}_1$ and q is the only thread that is normal at $\tilde{s}_1$. Let $\tilde{s}_2 = \mathsf{execute}(\tilde{s}_1, t_2)$. Hence there is a node $\tilde{v}_2 \in V$ with $\mathsf{state}(\tilde{v}_2) = \tilde{s}_2$ and an edge $\tilde{v}_1 \xrightarrow{t_2} \tilde{v}_2 \in E$.

Continuing in this way, we see that b_{i+1} defines a path in G from v_{i-1} to some $v' \in V$ with $\mathsf{state}(v') = s'$. Furthermore, q is not normal at s'.

Since no thread other than q has changed its local state in the path from v_i to v' defined by b_{i+1}, at s' no thread is normal. So again $\mathsf{ample}_G(v') = \mathsf{enabled}(v')$ and b_i defines a path in G from v' to a node v'_{i+1} with $\mathsf{state}(v'_{i+1}) = s_{i+1}$.

We now have two paths in G, both starting at v_i. The first executes $b_i b_{i+1}$ and ends at v_{i+1}; the second executes $b_{i+1} b_i$ and ends at v'_{i+1}. We claim $v'_{i+1} = v_{i+1}$. We already know both nodes have the same state component, s_{i+1}; we must show they have the same $\mathbf{a}$-component. But $\mathbf{a}_p$ is determined solely by the sequence of statements in thread p, and $\mathbf{a}_q$ solely by those of thread q. Since $p \neq q$, the $\mathbf{a}$-component is the same after $b_i b_{i+1}$ or $b_{i+1} b_i$.

Hence the string $b_1 \cdots b_{i-1} b_{i+1} b_i b_{i+2} \cdots b_N$ defines a path α' in G from v. We just need to see this string is the block-event string of α'.

Clearly, the paths $v_0 \xrightarrow{b_1} v_1, \ldots, v_{i-2} \xrightarrow{b_{i-1}} v_{i-1}$ are the first $i-1$ blocks of α', as they are the first $i-1$ blocks of α. Consider the remaining paths

$$v_{i-1} \xrightarrow{b_{i+1}} v', \;\; v' \xrightarrow{b_i} v_{i+1}, \;\; v_{i+1} \xrightarrow{b_{i+2}} v_{i+2}, \;\; \cdots, \;\; v_{N-1} \xrightarrow{b_N} v_N.$$

Since B_i is not initial, by Lemma 8(6), no thread is normal at v_{i-1}. We have already seen that no thread is normal at $s' = \mathsf{state}(v')$. By Lemma 8(5), B_j is not initial for $j \geq i$. Hence no thread is normal at $v_{i+1}, \ldots, v_{N-1}$. It follows that these paths are the remaining $N - i + 1$ blocks of α'. □

B.3 Blocks and data races

Let α be a finite path in G. We say two blocks B and B' of α *race* if there is an event e in B and e' in B' such that e and e' race in α. We say an event e in α *happens before* B if e happens before the first event of B. We say B *happens before* B' if the first event in B happens before the first event in B'.

Definition 15. Let α be a finite path in G with blocks $B_1, \ldots, B_N$. Suppose B_N races with a prior block. Let i be the maximum integer in $[1, N-1]$ such that B_i races with B_N. Let j be the maximum integer in $[1, N]$ such that $\mathsf{tid}(B_j) = \mathsf{tid}(B_i)$. (Note $i \leq j < N$.) The *race distance* of α is $N - j$. □

Hence the race distance is the number of blocks that occur after the last block from the last thread that races with B_N. Note the race distance is at least 1.

Lemma 12. *Let $u_0 \in V$. Assume the set of nodes reachable from u_0 in G is finite. If G has a path starting from u_0 with a data race, then there is a finite complete path β in G starting from u_0, with blocks $B_1, \ldots B_N$ satisfying the following: there is some $i \in [1, N-1]$ such that*

- *B_i and B_N race, and*
- *for all $j \in [i+1, N]$, $\mathsf{tid}(B_j) \neq \mathsf{tid}(B_i)$.*

Proof. If G has a path starting from u_0 with a data race, then truncate the path after the second event involved in the race, and the resulting finite path also has a data race. So we may assume that G has a finite path with a data race.

By Lemma 10, any finite path can be extended to a complete path, and if the original contained a data race, so will the extension, by Lemma 2. Hence we may assume G contains a complete path starting from u_0 with a race.

Let N be the minimal block length of any complete path starting from u_0 with a data race. All complete paths of block length N containing data races must have the last block racing with a previous block, else there would be a complete path of smaller block length with a data race. Among all such paths, let α be one with minimal race distance. Let $B_1, \ldots, B_N$ be the blocks of α.

Hence there is an event in α occurring before B_N that races with some event in B_N. Let t_1 be the last such event. Say t_1 occurs in block B_i. Let $p = \mathsf{tid}(B_i)$. We have

1. There is no racy path from u_0 with block length less than N.
2. There is no racy path from u_0 of block length N with race distance less than that of α.
3. Any two events occurring before B_N that have conflicting transitions are ordered by happens-before. (Else there would be a racy path with block length strictly less than N.)
4. No event in $B_{i+1}, \ldots, B_{N-1}$ races with any event in B_N. (As t_1 is the last event to race with one in B_N.)

Suppose a block from thread p occurs after B_i. We will arrive at a contradiction.

Let B_j be the last block from thread p. By assumption, $i < j < N$. According to Definition 15, the race distance of α is $N - j$. Note B_j is not initial, since there is a previous block from the same thread, and if both were initial it would contradict Lemma 8(5). Let t_2 be the first event of B_j.

Is there a block B between B_j and B_N such that t_2 does not happen before B? There are two cases, both of which lead to the desired contradiction.

Case 1: there is some $k \in [j+1, N-1]$ such that t_2 does not happen before B_k. Choose the minimal such k. Then for $j < l < k$, t_2 happens before B_l, but t_2 does not happen before B_k. The block-event string of α has the form

$$b_1 \cdots b_i \cdots b_j \cdots b_k \cdots b_N.$$

Suppose $j \leq l < k$. Then:

- B_l does not happen before B_k. (If $j = l$, then B_l does not happen before B_k, as t_2 does not happen before B_k. If $j < l$ then B_l does not happen before B_k, else t_2 happens before B_l happens before B_k.)
- B_k does not happen before B_l. (Since B_k occurs after B_l in α.)
- Hence B_k and B_l are not ordered by happens-before. By (3), no statement from B_l conflicts with any statement in B_k.

By Lemma 11, we may repeatedly transpose b_k with the block to its left, until b_k occurs just before b_j, i.e., there exists a path in G from u_0 with block-event string

$$b_1 \cdots b_i \cdots b_{j-1} b_k b_j \cdots b_{k-1} b_{k+1} \cdots b_N.$$

This path has a data race, has block length N, but has race distance $N - j - 1$, one less than that of α, contradicting (2).

Case 2: for all $k \in [j+1, N-1]$, t_2 happens before B_k.

We claim: for $j \leq k < N$, B_k does not happen before B_N. (Proof: we know B_j does not happen before B_N, else t_1 happens before B_j happens before B_N, contradicting the assumption that t_1 races with B_N. If $j < k < N$, B_k does not happen before B_N, else t_2 happens before B_k happens before B_N, i.e., B_j happens before B_N.)

It follows from (4) that for $j \leq k < N$, B_k contains no statement that conflicts with one in B_N.

As $j \leq N-1 < N$, B_{N-1} does not conflict with B_N and B_{N-1} does not happen before B_N. By Lemma 11, there is a path in G from u_0 with block-event string

$$b_1 \cdots b_i \cdots b_j \cdots b_{N-2} b_N b_{N-1}$$

and with a data race occurring between an event in b_i and one in b_N. Truncate the path just after b_N, and the resulting path has block length $N-1$ and contains a data race, contradicting (1) (the minimality of N). □

B.4 Proof of Theorem 1 for Programs Without Barriers

We can now complete the proof of Theorem 1 in the case where P has no barriers. By Lemma 6, G is dense. Part 2 of Theorem 1 then follows from part 2 of Theorem 2.

We now turn to the proof of part 1 of Theorem 1. As Lemma 7 proves one direction, we must prove the other direction. So suppose there is an execution from s_0 with a data race. We must show there is a path in G from u_0 which detects a race.

By Theorem 2(1), there is a path in G from u_0 with a data race. By Lemma 12, there is a complete path α in G from u_0, with blocks $B_1, \ldots, B_N$, and $i \in [1, N-1]$, such that B_i races with B_N and B_i is the last block from thread $p = \mathsf{tid}(B_i)$ in α. Let t_1 be the event in B_i and t_2 the conflicting event in B_N. At the end of B_i, t_1 is in $\mathbf{a}_p$. As thread p does not execute again, t_1 remains in $\mathbf{a}_p$ at the final node v of α. Moreover, t_2 is in $\mathbf{a}_q$ at v, where $q = \mathsf{tid}(B_N)$. The last transition of B_N brings thread q to an R_q or terminal state, and hence detects a data race involving t_1 and t_2.

B.5 Proof of Theorem 1: General Case

The general case of the theorem can be reduced to the case of a program with no barriers in a simple way. The key observation is that if a race occurs in an execution of a program with barriers, then the two conflicting events must occur in the same barrier epoch. This is because any two events in different epochs are ordered by happens-before, and thus cannot form a data race.

The reduction to the barrier-free case requires a simple program transformation. Given a program P (that may contain barriers) let P' be the program that is the same in every respect as P, except that every barrier state in P is made a terminal state in P'.

Suppose ζ is an execution of P, $i \geq 0$, and there is at least one transition in barrier epoch i in ζ. Define an execution ζ' of P' that extracts epoch i of ζ as follows. The initial state s'_0 of ζ' is defined as follows: if $i = 0$, s'_0 is the initial state of ζ. Otherwise, the lock state of s'_0 is the lock state in the state of ζ when all threads are in the i-th barrier. The thread state of thread p in s'_0 is the thread state of p just after the i-th exit_p transition in ζ; if thread p does not execute i exit_p transitions in ζ (i.e., p never enter epoch i), then the thread state of p in s'_0 is the final thread state of p in ζ. The transitions in ζ'

are precisely the transitions in epoch i of ζ, excluding the initial sequence of barrier-exit transitions in epoch i. Clearly, if ζ has a data race that occurs in epoch i, then ζ' has a data race.

Let G be the race-detecting state graph of P, and G' the race-detecting state graph of P'. Suppose now that α is a path in G with at least one transition in epoch i. Let $\zeta = \overline{\alpha}$. There is a corresponding path α' in G', specified as follows: the initial node of α' is $\langle s'_0, \emptyset^{\mathsf{TID}} \rangle$ and $\overline{\alpha'} = \zeta'$.

Now suppose P has an execution ζ with a data race. Let s_0 be the initial state of ζ. Say the first data race in ζ occurs in epoch $m \geq 0$. We will construct a path α in G that starts at $v_0 = \langle s_0, \emptyset^{\mathsf{TID}} \rangle$ and detects a race.

Suppose $0 \leq i < m$. Consider the execution ζ'_i of P' obtained by extracting epoch i from ζ. Let s' be the initial state of ζ'_i and s'' the final state. Since i is not the last barrier epoch of ζ, every thread must be terminal at s''. There is no data race in ζ'_i, because the first race in ζ occurs in epoch m. By Theorem 2(2), there is a path α'_i in G' from $\langle s', \emptyset^{\mathsf{TID}} \rangle$ to a node v'' in G' with state component s''.

The paths $\alpha'_0, \ldots, \alpha'_{m-1}$ can be "stitched together" to form a path β in G as follows: for $1 \leq i < m$ and $j \in \mathsf{TID}$, insert statement exit_j just before the first transition from thread j in α'_i. The path β terminates at a node whose state component is the state of ζ at the beginning of epoch m.

Now consider the execution ζ'_m of P' obtained by extracting epoch m of ζ. This execution has a data race. We may apply Theorem 1 to ζ'_m, since P' has no barriers. Thus there exists a path in P' from a node whose initial state component is the initial state of ζ'_m, and which detects a race. Stitch this path onto β to yield a path in G which detects a race. This proves part 1.

The proof of part 2 is almost exactly the same. Given an execution ending at a final state, again break it up into epochs and apply the barrier-free version of the theorem to the P'_i. Stitch the resulting paths together to yield a path in G ending at a node with state component the final state.

C Full Results

This section shows the expected result (data race or no data race), and the results reported by CIVL and LLOV for all test cases. It also shows runtimes for CIVL on an M1 MacBook Pro with 16GB memory. The additional test cases are shown in Table 1, and the DataRaceBench test cases are shown in Tables 2 and 3. See Section 3.4 for a description of the modifications (including imposition of bounds on inputs and thread counts) made to the DataRaceBench programs for CIVL.

Filename	CIVL time	Expec. Result	CIVL Result	LLOV Result
sync1_no.c	1.09	N	N	P
sync1_yes.c	1.31	P	P	P
critsec3_no.c	1.00	N	N	P
critsec3_yes.c	1.00	P	P	P
atomic3_no.c	1.02	N	N	P
atomic3_yes.c	1.04	P	P	P
bar1_no.c	7.19	N	N	P
bar1_yes.c	1.24	P	P	P
bar2_no.c	1.17	N	N	P
bar2_yes.c	1.17	P	P	P
bar3_no.c	3.56	N	N	P
bar3_yes.c	1.43	P	P	P
diffusion1_no.c	3.12	N	N	N
diffusion1_yes.c	1.38	P	P	N
diffusion2_no.c	22.19	N	N	-
diffusion2_yes.c	5.05	P	P	-
critsec2_no.c	2.02	N	N	-
critsec2_yes.c	1.36	P	P	-
prodcons_no.c	22.60	N	N	-
prodcons_yes.c	1.33	P	P	-

Table 1. Results of additional test cases. CIVL runtime in seconds; Expected Result: P (positive) = data race detected, N (negative) = no race detected; CIVL Result; LLOV Result

Filename	**CIVL time**	**Expec. Result**	**CIVL Result**	**LLOV Result**
DRB001-antidep1-orig-yes.c	1.44	P	P	P
DRB002-antidep1-var-yes.c	1.53	P	P	P
DRB003-antidep2-orig-yes.c	1.51	P	P	P
DRB004-antidep2-var-yes.c	1.69	P	P	P
DRB005-indirectaccess1-orig-yes.c	2.19	P	P	P
DRB006-indirectaccess2-orig-yes.c	2.35	P	P	P
DRB007-indirectaccess3-orig-yes.c	2.34	P	P	P
DRB008-indirectaccess4-orig-yes.c	2.11	P	P	P
DRB009-lastprivatemissing-orig-yes.c	1.31	P	P	P
DRB010-lastprivatemissing-var-yes.c	1.60	P	P	P
DRB011-minusminus-orig-yes.c	1.57	P	P	P
DRB012-minusminus-var-yes.c	1.57	P	P	P
DRB013-nowait-orig-yes.c	1.39	P	P	P
DRB014-outofbounds-orig-yes.c	1.15	P	N	P
DRB015-outofbounds-var-yes.c	1.44	P	N	P
DRB016-outputdep-orig-yes.c	1.30	P	P	P
DRB017-outputdep-var-yes.c	1.66	P	P	P
DRB018-plusplus-orig-yes.c	1.55	P	P	P
DRB019-plusplus-var-yes.c	1.84	P	P	P
DRB020-privatemissing-var-yes.c	1.69	P	P	P
DRB021-reductionmissing-orig-yes.c	1.46	P	P	P
DRB022-reductionmissing-var-yes.c	1.65	P	P	P
DRB023-sections1-orig-yes.c	1.28	P	P	P
DRB028-privatemissing-orig-yes.c	1.47	P	P	P
DRB029-truedep1-orig-yes.c	1.51	P	P	P
DRB030-truedep1-var-yes.c	1.59	P	P	P
DRB031-truedepfirstdimension-orig-yes.c	1.69	P	P	P
DRB032-truedepfirstdimension-var-yes.c	2.15	P	P	P
DRB033-truedeplinear-orig-yes.c	1.49	P	P	P
DRB034-truedeplinear-var-yes.c	1.58	P	P	P
DRB035-truedepscalar-orig-yes.c	1.45	P	P	P
DRB036-truedepscalar-var-yes.c	1.60	P	P	P
DRB037-truedepseconddimension-orig-yes.c	15.89	P	P	P
DRB038-truedepseconddimension-var-yes.c	2.05	P	P	P
DRB039-truedepsingleelement-orig-yes.c	1.49	P	P	P
DRB040-truedepsingleelement-var-yes.c	1.58	P	P	P
DRB041-3mm-parallel-no.c	85.10	N	N	N
DRB043-adi-parallel-no.c	132.75	N	N	N
DRB045-doall1-orig-no.c	3.88	N	N	N
DRB046-doall2-orig-no.c	5.66	N	N	N
DRB047-doallchar-orig-no.c	2.17	N	N	N
DRB048-firstprivate-orig-no.c	4.09	N	N	N

Table 2. DataRaceBench results, part 1.

Filename	CIVL time	Expec. Result	CIVL Result	LLOV Result
DRB050-functionparameter-orig-no.c	3.97	N	N	N
DRB051-getthreadnum-orig-no.c	1.78	N	N	N
DRB052-indirectaccesssharebase-orig-no.c	2.83	N	N	P
DRB053-inneronly1-orig-no.c	4.20	N	N	N
DRB054-inneronly2-orig-no.c	12.78	N	N	P
DRB055-jacobi2d-parallel-no.c	71.78	N	N	N
DRB057-jacobiinitialize-orig-no.c	14.50	N	N	N
DRB058-jacobikernel-orig-no.c	6.31	N	N	N
DRB059-lastprivate-orig-no.c	4.41	N	N	N
DRB060-matrixmultiply-orig-no.c	13.98	N	N	N
DRB061-matrixvector1-orig-no.c	5.75	N	N	N
DRB062-matrixvector2-orig-no.c	156.76	N	N	N
DRB063-outeronly1-orig-no.c	5.36	N	N	N
DRB064-outeronly2-orig-no.c	1.92	N	N	N
DRB065-pireduction-orig-no.c	67.06	N	N	N
DRB066-pointernoaliasing-orig-no.c	4.45	N	N	N
DRB067-restrictpointer1-orig-no.c	4.45	N	N	N
DRB068-restrictpointer2-orig-no.c	5.15	N	N	N
DRB069-sectionslock1-orig-no.c	1.99	N	N	P
DRB073-doall2-orig-yes.c	1.46	P	P	P
DRB074-flush-orig-yes.c	1.58	P	P	P
DRB075-getthreadnum-orig-yes.c	1.18	P	p	P
DRB076-flush-orig-no.c	35.00	N	N	N
DRB077-single-orig-no.c	1.62	N	N	N
DRB088-dynamic-storage-orig-yes.c	1.27	P	P	P
DRB089-dynamic-storage2-orig-yes.c	1.22	P	P	P
DRB090-static-local-orig-yes.c	2.09	P	P	P
DRB093-doall2-collapse-orig-no.c	6.92	N	N	N
DRB103-master-orig-no.c	1.50	N	N	N
DRB104-nowait-barrier-orig-no.c	5.26	N	N	N
DRB108-atomic-orig-no.c	8.66	N	N	N
DRB109-orderedmissing-orig-yes.c	1.38	P	P	P
DRB110-ordered-orig-no.c	92.89	N	N	N
DRB111-linearmissing-orig-yes.c	1.34	P	P	P
DRB113-default-orig-no.c	12.01	N	N	N
DRB120-barrier-orig-no.c	2.09	N	N	N
DRB121-reduction-orig-no.c	23.44	N	N	N
DRB124-master-orig-yes.c	1.17	P	P	P
DRB125-single-orig-no.c	1.82	N	N	N
DRB126-firstprivatesections-orig-no.c	1.06	N	N	N
DRB139-worksharingcritical-orig-no.c	1.14	N	P	N
DRB140-reduction-barrier-orig-yes.c	1.36	P	P	N
DRB141-reduction-barrier-orig-no.c	9.89	N	N	N
DRB169-missingsyncwrite-orig-yes.c	2.80	P	P	P
DRB170-nestedloops-orig-no.c	7.28	N	N	N
DRB172-critical2-orig-no.c	9.09	N	N	N

Table 3. DataRaceBench results, part 2.

D Change Log

Version 2: 20 July 2023

This version corrects an error in the previous version concerning Definition 7. The old version called for a race check when a thread arrives at an acquire state or departs from a state in $R_i \setminus \mathsf{Acquire}_i$; when all threads are in the barrier; and at a state with no enabled transition. This does not suffice for the correctness of Theorem 1. A counterexample with two threads is

$$t_1\colon \texttt{x=1;}\ \ \mathsf{acquire}(l)\texttt{;}$$
$$t_2\colon \texttt{x=2;}.$$

The one execution in the race-detecting state graph proceeds

$$x = 1\ (t_1 : \mathit{check});\ \ x = 2;\ \ \mathsf{acquire}(l)\ (t1 : \mathit{clear});\ \ (\mathit{check\text{-}all}).$$

This does not detect the race at the *check-all* because t_1 cleared in the previous step. The error in the proof of Theorem 1 occurs in Appendix B.4, where it is assumed that the path β is not empty.

This version changes Definition 7 so that a race check occurs whenever a thread arrives at a state in R_i, a barrier state, or a terminal state. The check for races once all threads are in the barrier is then redundant and has been removed. The implementation and its description in Section 3.3 have been updated accordingly. The proof of Appendix B.4 has been corrected and is simpler. Several other minor improvements were made to CIVL, and the experiments were rerun. The results are the same, except for the times, which have been updated. An updated link is supplied for the experimental artifacts.

Various other minor changes and clarifications were made.

Version 1: 20 May 2023

Original submission.