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

C. Enea and A. Lal (Eds.): CAV 2023, LNCS 13965, pp. 265–287, 2023.
https://doi.org/10.1007/978-3-031-37703-7_13

have no defined behavior, but race-free programs are guaranteed to behave in a sequentially consistent manner [25].

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 Sect. 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 Fortran. A large sub-language of OpenMP has the SC4DRF guarantee.[1] 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 Sect. 3, along with the results of some experiments applying our tool to a suite of C/OpenMP programs. In Sect. 4, we discuss related work and Sect. 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.

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

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 $\sigma' = \mathsf{next}(\sigma)$, 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) $1 \le i < j \le |\alpha|$, (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 $[\alpha]$

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

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. $\forall i \in \mathsf{TID}$, $\mathbf{a}'_i = \begin{cases} \mathbf{a}_i \cup \{t\} & \text{if } t \text{ is an nsync statement in thread } i \\ \emptyset & \text{if } t = \mathsf{exit}_0 \text{ or } i = \mathsf{tid}(t) \wedge \sigma_i \in R_i \\ & \text{otherwise} \end{cases}$
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 $e = v \xrightarrow{t} v' \in E$, $i = \mathsf{tid}(t)$, σ the local state of thread i at v, and σ' the local state of thread i at v'. We say *e detects a race* if either (i) $\sigma \in R_i \setminus \mathsf{Acquire}_i$ and thread i detects a race in v, (ii) $\sigma' \in \mathsf{Acquire}_i$ and thread i detects a race in v', or (ii) $t = \mathsf{exit}_0$ and any thread 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, or there is some $v = \langle s, \mathbf{a} \rangle \in V$ that is reachable from u, and $i \in \mathsf{TID}$, such that $\mathsf{enabled}(s) = \emptyset$ and thread i detects a race in v. □

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 from a thread at an $R_i \setminus \mathsf{Acquire}_i$ state is executed, 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. When an edge in thread i terminates at an $\mathsf{Acquire}_i$ state, a similar race check takes place. When an exit_0 occurs, or a node with no outgoing edges is reached, $\mathbf{a}_i$ and $\mathbf{a}_j$ are compared for all $i, j \in \mathsf{TID}$ with $i \neq j$. 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 $u_0 = \langle s_0, \emptyset^{TID} \rangle \in V$. 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 Sect. 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 Sect. 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 Sect. 3.1, we summarize the basics of OpenMP. In Sect. 3.2, we provide the necessary background on CIVL-C and the primitives used in the transformation. In Sect. 3.3, we describe the transformation itself. In Sect. 3.4, we report the results of experiments using this tool.

All software and other artifacts necessary to reproduce the experiments, as well as the full results, are included in a VirtualBox virtual machine available at https://doi.org/10.5281/zenodo.7978348.

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 Fig. 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.

```
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

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.

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 Sect. 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

```
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 check_and_clear_all() {
8    for (int i=0; i<nthreads; i++)
9      for (int j=i+1; j<nthreads; j++) check_conflict(i, j);
10   for (int i=0; i<nthreads; i++) reads[i] = writes[i] = $mem_empty();
11 }
12 void run(int tid) {
13   void pop() { reads[tid]=$read_set_pop(); writes[tid]=$write_set_pop(); }
14   void push() { $read_set_push(); $write_set_push(); }
15   void check() {
16     for (int i=0; i<nthreads; i++) { if (i==tid) continue; check_conflict(tid, i); }
17   }
18   // local variable declarations
19   $local_start(); push(); S pop(); $local_end();
20 }
21 for (int i=0; i<nthreads; i++) reads[i] = writes[i] = $mem_empty();
22 $parfor (int tid:0..nthreads-1) run(tid);
23 check_and_clear_all();
```

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

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 Fig. 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 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 `check_and_clear_all` checks that no conflict exists between any two threads and clears the 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 Sect. 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. A release statement becomes

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

It is similar to the acquire case, except that the check occurs upon leaving the release location, i.e., after the yield. 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(); $local_end(); $barrier_call(); if (tid==0) check_and_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) checks for conflicts and clears all the shared mem-sets. A second barrier call is used to prevent other threads from racing ahead before this check and clear is complete. This protocol mimics the events that take place atomically with an exit_0 transition in Sect. 2.

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.[2]

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 Sect. 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,

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

```
// 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;
}
```

```
// 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];
```

```
// 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).

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 min 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 (Fig. 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 (Fig. 3, left). The semantics for reduction 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;

```
// 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;
 }
}
```

```
// 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).

- 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 Fig. 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 (Fig. 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 27 min. Individual times ranged from 1 to 150 seconds: 66 took less than 5s, 80 took less than 30s, and 82 took less than 1 min. (All CIVL runs used an M1 MacBook Pro with 16GB memory.)

Total CIVL runtime on the 20 extra tests was 210s. LLOV analyzes all 88 DRB problems in less than 15 s (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 subsequently 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., et al.: 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 Trans. Electronic Comput. EC **15**(5), 757–763 (1966). https://doi.org/10.1109/PGEC.1966.264565
5. Betts, A., et al.: The design and implementation of a verification technique for GPU kernels. ACM Trans. Program. Lang. Syst. **37**(3) (2015). https://doi.org/10.1145/2743017

6. Blom, S., Darabi, S., Huisman, M., Safari, M.: Correct program parallelisations. Int. J. Softw. Tools Technol. Trans. **23**(5), 741–763 (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, HotPar 2011, pp. 1–6. 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 (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 Trans. Archit. Code Optimiz. (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.) IWOMP 2020. LNCS, vol. 12295, pp. 149–164. Springer, 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.) LCPC 2016. LNCS, vol. 10136, pp. 106–120. Springer, 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 Comput. Sci. Eng. **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, 2 edn. MIT press, Cambridge, MA, USA (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. (ed.): Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (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., Adeli, H., Kim, H., Kang, H., Kim, K.J., Kiumi, A., Kang, B.-H. (eds.) ASEA 2011. CCIS, vol. 257, pp. 424–436. Springer, 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 (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 Trans. Comput. **C-28**(9), 690–691 (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 (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 1991: 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 Trans. Program. Lang. Syst. **33**, 3:1–3:55 (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.) IWOMP 2017. LNCS, vol. 10468, pp. 249–265. Springer, 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: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12224, pp. 376–400. Springer, Cham (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, pp. 62–71. WBIA 2009. Association for Computing Machinery, New York (2009). https://doi.org/10.1145/1791194.1791203
31. Siegel, S.F., et al.: 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