Hunting Superfluous Locks with Model Checking

Viet-Anh Nguyen, Wendelin Serwe, Radu Mateescu, Eric Jenn

Hunting Superfluous Locks with Model Checking

Viet-Anh Nguyen[1], Wendelin Serwe[2], Radu Mateescu[2], and Eric Jenn[1]

[1] IRT Saint Exupéry, Toulouse, France
[2] Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP*, LIG, 38000 Grenoble, France

Abstract. Parallelization of existing sequential programs to increase their performance and exploit recent multi and many-core architectures is a challenging but inevitable effort. One increasingly popular parallelization approach is based on OpenMP, which enables the designer to annotate a sequential program with constructs specifying the parallel execution of code blocks. These constructs are then interpreted by the OpenMP compiler and runtime, which assigns blocks to threads running on a parallel architecture. Although this scheme is very flexible and not (very) intrusive, it does not prevent the occurrence of synchronization errors (e.g., deadlocks) or data races on shared variables. In this paper, we propose an iterative method to assist the OpenMP parallelization by using formal methods and verification. In each iteration, potential data races are identified by applying to the OpenMP program a lockset analysis, which computes the set of shared variables that potentially need to be protected by locks. To avoid the insertion of superfluous locks, an abstract, action-based formal model of the OpenMP program is extracted and analyzed using the ACTL on-the-fly model checker of the CADP formal verification toolbox. We describe the method, compare it with existing work, and illustrate its practical use.

1 Introduction

Nowadays, to take full advantage of modern hardware architectures (multi-core and many-core processors, Systems-on-Chip, etc.), it is necessary to parallelize applications, even in constrained environments, such as avionics. Designing correct parallel programs on shared-memory architectures is a difficult task facing classical synchronization issues, such as the presence of data races (concurrent accesses to shared variables that make the program nondeterministic [20]) or deadlocks, both of which are unacceptable for critical systems. These difficulties occur not only in the design of new parallel programs, but also in the parallelization of existing sequential programs, which have been optimized during years and for which it is too costly to redevelop parallel versions from scratch.

An increasingly popular approach to parallelize sequential code is based on OpenMP [33], which does not require to modify the code but simply annotate it with parallelization constructs expressing a variety of mechanisms (creating parallel regions executed by teams of threads, inserting locks on variables and array

* Institute of Engineering Univ. Grenoble Alpes

elements, introducing synchronizations, etc.). The underlying compiler and execution framework are in charge of implementing these constructs, building and executing the parallel program on a given architecture. Unfortunately, OpenMP is not equipped with a formal semantics suitable for reasoning about OpenMP programs and ensuring the correctness of annotation-based parallelization: [33, Section 1.1] explicitly states that "OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs."

A naive way to eliminate data races in a parallel program is to protect all shared variables by locks that serialize the accesses of parallel threads to these variables. Although safe, this approach may introduce deadlocks, and also increase the overhead, negatively impacting the performance of the program. Even more importantly, the approach may induce a too sequential execution of the program and thus not (fully) exploit the benefits of parallelization. In this paper, we refine this naive lock-based approach and propose an iterative method to prevent data races in safety-critical parallel applications. The method combines a simple lockset analysis [36] to detect all the shared variables potentially unprotected by locks (which may produce false positives about variables that actually do not need to be protected) and a model checking analysis to reduce the number of false positives and consequently avoid introducing superfluous locks.

Lockset analysis is based on the application of a "locking discipline", by considering that a race condition may occur if a shared variable is not protected by an appropriate lock. Lockset based race detectors are easy to implement and never produce false negatives, i.e., they detect all potential data races, which is essential for safety-critical applications. However, these detectors are pessimistic, since data races can be prevented not only by using locks, but also by performing accesses to shared variables sequentially.

We exhibit such sequentiality using model checking, by extracting from the parallel program a formal model capturing (an abstraction of) the concurrency and data dependencies, and detecting the presence of concurrent accesses to shared variables using temporal properties in ACTL (*Action Computation Tree Logic*) [9,8]. The precision of the model has no impact on the soundness of the method (since the lockset analysis has already produced a data race free, albeit not optimal, parallel program), but only on the efficiency of the parallel code (a better model precision will yield a more accurate analysis and hence a more drastic elimination of superfluous locks). We instantiated this method on top of the CADP (Construction and Analysis of Distributed Processes)[3] [14] verification toolbox and illustrated it for the design of data race free OpenMP applications.

Stefania's work on defining action-based temporal logics and various extensions thereof, as well as on designing efficient on-the-fly verification algorithms, was a source of inspiration in this field. The implementation of ACTL used in this paper relies on the translation from ACTL to the modal μ-calculus ($L\mu$)

[3] `http://cadp.inria.fr`

proposed by Stefania and colleagues [10]. For checking ACTL formulas, we used the on-the-fly model checker EVALUATOR [29] of CADP, which handles formulas of MCL, an extension of alternation-free $L\mu$ with data and generalized Büchi automata. Although it relies on different techniques based on the local resolution of Boolean equation systems [27], EVALUATOR is similar in spirit to the on-the-fly model checkers FMC [16] and UMC [5] developed by Stefania and colleagues, which handle formulas of μACTL (ACTL extended with fixed point operators) and UCTL (μACTL extended with state-based and data-aware operators), respectively.

Also, Stefania's contributions on ACTL characteristic formulas [12] and the adequacy of action-based logics with bisimulations [11] paved the way towards an $L\mu$ fragment adequate with divergence-sensitive branching bisimulation [30]. Recently, this $L\mu$ fragment, which subsumes μACTL\X (μACTL without the next time operator) was extended with strong modalities and equipped with an improved compositional verification technique [24] applicable to the ACTL formulas we use for detecting concurrent accesses to shared variables.

The paper is organized as follows. Section 2 gives a brief overview of OpenMP and data races. Section 3 presents our parallelization workflow and its practical implementation using CADP. Section 4 reviews existing work on data race prevention. Finally, Section 5 gives some concluding remarks and directions for future work.

2 OpenMP

OpenMP [33][4] is an API (Application Programming Interface) for developing portable parallel programs using a shared memory communication paradigm in the C, C++, and Fortran programming languages. The OpenMP API consists of directives to extend the base languages with portable parallel programming constructs (to be implemented by an OpenMP-compliant compiler) and functions and environment variables (to be implemented by a corresponding runtime library). In the C/C++ languages, OpenMP directives are pragmas of the form of `#pragma omp` .... Directives and calls to library routines are grouped under the generic designation of *constructs*. OpenMP supports both parallel and sequential execution, the latter being achieved by simply ignoring the OpenMP constructs.[5]

The execution model of OpenMP follows a fork-join discipline. Initially, an OpenMP program begins with a single *thread* of execution. Whenever a thread encounters an OpenMP `parallel` construct, the thread creates a *team* of threads (containing at least itself) and becomes the *master thread* of the team. The code executed by each of these threads depends on the code inside the `parallel` construct. These threads then execute independently and synchronize (using an

[4] `http://www.openmp.org`

[5] However, OpenMP does neither require nor guarantee that parallel and sequential executions produce the same results; also, executing the same program with a different number of threads may yield different results [33, Section 1.3].

implicit barrier) on termination; only the master thread continues afterwards. OpenMP supports nested parallelism: each thread of a team can itself create a new team, when it encounters a `parallel` construct. Hence, several teams may exist simultaneously.

Initially designed for the parallelization of regular loops, OpenMP supports since version 3.0 also the notion of *task*. In OpenMP, a task is a pair of a piece of code together with a specific piece of data. Tasks can be generated explicitly or implicitly, and are to be executed by the threads. As an example, the `for` construct implicitly creates a task for each iteration of the associated `for`-loop. The `single` construct implicitly creates a task for its associated code, so that this code is executed exactly once; other threads encountering the construct wait for its termination (using an implicit barrier). A task can be suspended (and resumed later) only on so-called *task scheduling points*, e.g., when creating new task(s) or when waiting on a barrier. OpenMP provides constructs to express further constraints on task creation and execution, such as the fact that a task can only be executed if another task has completed.

All OpenMP threads have access to the same shared memory. Each thread may read or write any shared variables, and there is no constraint as to when those operations are allowed to occur. The memory model of OpenMP is relaxed-consistency: each thread has a *temporary view* of the shared memory; this temporary view is not required to be consistent with the memory at all times. Each thread has also access to a local, private memory, to which other threads have no access.

OpenMP provides synchronization constructs (locks, barriers, etc.), variable attributes defining data sharing (private, shared, etc.), a flush operation (enforcing the consistency of the temporary views with the shared memory), and data-dependencies between tasks (enforcing the execution of a task computing some value before the execution of a task using this value).

In OpenMP, parallelization is user-directed, i.e., the programmer explicitly uses the OpenMP constructs to specify how to parallelize the execution of the program. Hence, OpenMP relies on the programmer to ensure the correctness of the program [33, Section 1.1], e.g., to ensure the absence of memory management errors, such as data races. However, this is a heavy responsibility for the programmer, given the inherent complexity of parallelization, the rich set of constructs, and the fact the creation and ordering of tasks might depend on information only available at runtime. Consequently, any assistance to the programmer is more than welcome.

A first approach to assist programmers is to make a list of common pitfalls and to derive a set of recommended best practices and coding guidelines [38]. These common mistakes can be classified into two categories: errors (leading to an incorrect behavior) and performance issues (leading to inefficient programs). For instance, using the clause `default (none)` implies that data-sharing attributes of all variables in all parallel regions of the OpenMP program must be explicitly specified. Although following carefully chosen coding guidelines may

ensure correctness, it might be difficult to apply these guidelines to an existing sequential code and obtain a parallel version with acceptable performance.

A second approach is the development of analysis tools that detect errors. However, due to the expressive power of the OpenMP constructs and in particular the fact that the parallel execution might depend on data values, developing such analysis tools is extremely challenging. As for the first approach, limiting the scope of the tools to a subset of OpenMP constructs is not always acceptable in an industrial context (the constructs have been included for good reasons). Similarly, applying coarse data abstractions increases the rate of false positives/negatives and reduces the practical usefulness of the analysis.

In the following section, we present a method to assist the programmer to ensure the absence of data races. To illustrate this method, we use the following simple example, which computes the sum of the squares of the elements of an array `a`, counting the last element (`a[4]`) twice. Figure 1 shows the OpenMP code.[6] The construct `#pragma omp parallel` (line 5) creates a team of threads to execute the block from lines 6–15 in parallel; the number of threads in the team is determined at runtime, depending on the available hardware. The construct `#pragma omp for schedule (static, 1)` (line 7) indicates that the body of the `for`-loop should be statically splitted in as many tasks as there are iterations (i.e., 5). Obviously, there is a data race on the update of variable `sum` in the `for`-loop (line 11). However, there is no data race for the accesses to the array `a` and variable `sum` between the body of the `for`-loop and assignment at line 14, because the assignment is executed after the termination of all iterations of the `for`-loop (i.e., there is an implicit barrier at the end of the `#pragma omp for` construct), and the assignment is executed by a single thread (this is ensured by the `single` construct at line 13).

3 Parallelization Workflow

Figure 2 depicts the suggested workflow for the parallelization of a sequential application, guaranteeing the absence of data races by enforcing a locking discipline and avoiding superfluous locks by means of model checking. This iterative flow comprises several activities in each iteration:

1. Lockset analysis is used to build the variable-set *SUV* (*Set of Unprotected Variables*), which might present the risk of a data race. It is sufficient to use the simplest version of lockset analysis presented in [36], which raises an alarm if any access to a shared variable is not protected by a lock.
2. A formal model expressed in the LNT language [15] is built to capture all the possible control flows of OpenMP threads and their synchronizations. The process of building the LNT model is presented in the next section.
3. The verification tools provided by CADP are used to identify the "sequentiality constraints" that prevent some data race conditions to occur. The set of unprotected variables is updated accordingly.

[6] The meaning of "work unit" (in the comments) can be found in Section 3.2.

```
1  int a[5] = {2, 3, 4, 5, 6};
2  int main()
3  {
4      int i, sum = 0; // work unit WU0
5      #pragma omp parallel
6      {
7          #pragma omp for schedule (static, 1)
8          for (i = 0; i < 5; i++)
9          { // work units WU1 to WU5
10           a[i] = a[i] * a[i];
11           sum += a[i];
12         }
13         #pragma omp single
14         sum += a[4]; // work unit WU6
15     }
16     return 0;
17 }
```

Fig. 1. Minimalist OpenMP example

4. The refined list of unprotected variables is given to the programmer, who can add new locks in the program to protect them.

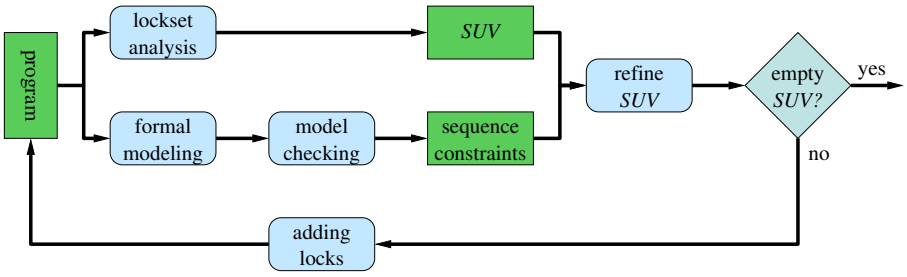

Fig. 2. Parallelization workflow

Considering that the programmers may make a mistake when adding locks to the program, the verification process is repeated until the list of unprotected variables is empty. At the end, we ensure that the program is free from data races. These steps are detailed in the following.

3.1 Lockset Algorithm

Lockset analysis [36] is a technique for dynamic detection of possible data races. The technique has been successfully implemented in the Eraser tool. Rather

than checking the absence of data races, lockset analysis checks whether a program adheres to a *locking discipline*, which requires that each access to a shared variable is protected by at least one lock. Notice that the respect of this locking discipline guarantees absence of data races, because all accesses to a shared variable are mutually exclusive.

The original definition of the simplest possible version of the lockset algorithm [36, Section 2] is the following. "For each shared variable v, the algorithm maintains the set $C(v)$ of candidate locks for v. This set contains those locks that have protected v for the computation so far. That is, a lock l is in $C(v)$ if, in the computation up to that point, every thread that has accessed v was holding l at the moment of the access. When a new variable v is initialized, its candidate set $C(v)$ is considered to hold all possible locks. When the variable is accessed, the algorithm updates $C(v)$ with the intersection of $C(v)$ and the set of locks held by the current thread. This process, called lockset refinement, ensures that any lock that consistently protects v is contained in $C(v)$. If some lock l consistently protects v, it will remain in $C(v)$ as $C(v)$ is refined. If $C(v)$ becomes empty this indicates that there is no lock that consistently protects v."

Under the hypothesis that the set of locks held by a thread for each point of the program is deterministic, a single run of the lockset algorithm is sufficient. Otherwise, for instance if the operations on locks are data-dependent or vary in different branches of conditional statements, several runs might be necessary.

The basic lockset algorithm can be refined [36] to reduce the number of false alarms, for instance to take into account that read accesses need not to be protected if there is no concurrent write access. For the purpose of this paper, the basic algorithm is sufficient.

3.2 OpenMP to LNT

In order to build an LNT model that captures the control flows of threads and their synchronizations, the OpenMP program is broken into *work units* (or blocks). A work unit is defined as a part of a task, containing neither conditional branches synchronizations, nor task scheduling points. Thus, the execution of a work unit is never interrupted by the runtime scheduler. A work unit graph may contain two types of nodes: *basic nodes* represent work units of the program, and *synchronization nodes* represent synchronizations between threads enforced by OpenMP constructs (i.e., `#pragma omp critical`, `omp_set_lock()`, `omp_unset_lock()`, ...).

An edge between a pair of nodes represents the execution order. For a pair of basic nodes, this edge reflects the order of the corresponding work units, which is imposed by the control flow. For a pair of a basic node and a synchronization node, the edge reflects that the work unit denoted by the basic node starts (respectively, ends) with a synchronization construct corresponding to the synchronization node. The work unit graph can be obtained by static analysis of the code, akin to the construction of a control flow graph in an optimizing compiler.

Figure 3 represents a work unit graph of the program given in Figure 1. This work unit graph contains no synchronization nodes, but the one shown later

in Figure 5 contains basic nodes (depicted as circles) and synchronization nodes (lock and unlock nodes, depicted as rectangles with rounded corners). Inspection of the OpenMP source code (Figure 1) yields the following information about variables accessed by the various work units. All work units access (read/write) the variable `sum`. All work units but WU0 access (read/write) the array `a`; however, they access separate elements—the only exception being `a[4]`, which is read and written by WU5 and read by WU6. Thus, there might be a data race for variables `a[4]` and `sum`.

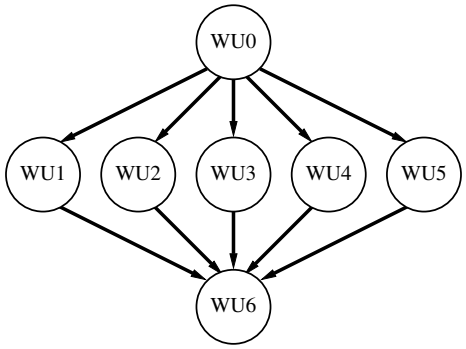

Fig. 3. Work unit graph for the program of Figure 1

To analyze the work unit graph using model checking, we first transform it into an LNT model by applying the following rules:

- Basic nodes are modeled as LNT processes
- Lock/unlock nodes are modeled as synchronizations on gates `ACQUIRE`/`RELEASE` representing acquire/release actions on the lock; two further actions `INIT` and `DESTROY` denote the creation and deletion of the lock
- Barrier nodes are modeled as multiway rendezvous on dedicated LNT synchronization gates
- Edges are modeled as LNT sequential composition
- Branch conditions are modeled as nondeterministic choice using the `select` operator

For example, Table 1 shows the LNT code for the work unit graph of Figure 3.

3.3 Sequentiality Detection

A data race may occur on a shared variable x if at least two work units WUi and WUj accessing x can execute concurrently at some moment. If the two work units always execute in a deterministic order, they cannot cause a data race on x, meaning that it is not necessary to protect x by a lock.

Table 1. LNT code for work unit graph of Figure 3

```
module OMP is
   process MAIN [WU0, WU1, WU2, WU3, WU4, WU5, WU6: none] is
      WU0;
      par
         WU1
      || WU2
      || WU3
      || WU4
      || WU5
      end par;
      WU6
   end process
end module
```

To detect the sequential execution of two work units, we exploit the work unit graph of the OpenMP program. The LNT model of this graph represents all the possible interleavings of work units (encoded as basic nodes in the graph, and simply as gate names in the LNT model) permitted by the OpenMP parallelization constructs (encoded as synchronization nodes in the graph and as gate names or implicit synchronizations in the LNT model). The behavior of the LNT model is represented by an LTS, in which every state corresponds to a global state of the work unit graph (i.e., an abstract state of the OpenMP program), each action denotes the execution of a basic node or a synchronization node, and each transition indicates that the program can move from one state to another by performing a certain action.

In terms of this LTS, the sequential execution of two basic nodes corresponding to work units WUi and WUj can be ensured by checking that, in every state, it is not possible to execute both basic nodes immediately. This property can be expressed in ACTL [8] as follows:

$$\neg \mathsf{EF}_{\mathsf{true}}(\mathsf{EX}_{\mathrm{WU}i}\mathsf{true} \wedge \mathsf{EX}_{\mathrm{WU}j}\mathsf{true})$$

The formula expresses the absence of a transition sequence leading (from the initial state of the LTS) to a state having an outgoing transition labeled by WUi and an outgoing transition labeled by WUj.

For the LNT model of work unit graph shown on Table 1, the above formula holds for all pairs of work units (WU0, WUi) and (WUi, WU6) with $1 \leq i \leq 5$, and fails for all pairs of work units (WUi, WUj) with $1 \leq i, j \leq 5$ and $i \neq j$. This reflects the structure of the work unit graph (WU0 is executed before WU1, ..., WU5, which are executed before WU6) and indicates the possibility of data races on the shared variables accessed by work units WU1, ..., WU5, which therefore must be protected by locks.

3.4 Inserting Locks

To protect the access to `sum` in the `for`-loop, the programmer declares its body as `critical`. The resulting code is shown on Figure 4, the corresponding work unit graph on Figure 5 (the principal difference with Figure 3 being the addition of Lock/Unlock nodes), and the corresponding LNT code in Table 2. The principal difference between Tables 1 and 2 is that the execution of the work units WU1, WU2, WU3, WU4, and WU5 is protected by a lock (represented by process `LOCK`), which has to be acquired before the execution of the work unit, and released afterwards: these steps are grouped into to process `PROTECTED_WU` (used similarly to a procedure). Process `LOCK` executes as an additional process. It has a local variable `FREE` indicating the status of the lock, and ensures that the lock can only acquired when it is free and that only the process holding the lock can release it. Gates `INIT` (respectively, `DESTROY`) are used to start (respectively, terminate) the execution of the lock.

Rerunning lockset analysis on the modified program, the accesses to `a` and `sum` in work unit 6 are still not protected by a lock, but model checking shows sequentiality of work units 1 to 5 with work unit 6, and thus the absence of a data race without the need of adding any further lock.

4 Related Work

Much effort has been spent on detecting data races in parallel programs. These efforts can be classified into dynamic and static approaches [4].

Dynamic techniques rely on observations of the running program. Such techniques have been implemented in several tools for race detection in OpenMP programs. Happens-before analysis monitors accesses to shared variables. If an access to a shared variable is logically concurrent with any previous conflicting access, the tool will raise an alarm; a pair of concurrent accesses to the same variable is conflicting if and only if at least one of them is a write. This technique leads to no false positives (i.e., each detected issue is indeed a data race), but might produce false negatives, because its precision depends on the definition of logically concurrent accesses, which is often expensive to compute. The happens-before technique has been implemented for instance in RaceStand [23].

Closer related to our suggested method are techniques based on lockset analysis, which we also use in our flow. The lockset analysis [36] (see also Section 3.1) aims at enforcing a locking discipline, rather than checking for absence of data races in general. Tools based on lockset analysis raise an alarm when some access to a shared variable is not protected by an appropriate lock. Lockset based race detectors are safe (i.e., they do not produce false negatives), but too pessimistic because locking is not the only way to provide safe synchronization. Thus, the Eraser tool [36] implements various improvements of the simple lockset algorithm to reduce the number of false positives, taking into account frequent programming patterns. Our workflow has a similar goal, but rather than trying to improve the algorithm, we use model checking to filter the results. This has

```
int a[5] = {2, 3, 4, 5, 6};
int main()
{
   int i, sum = 0; // work unit WU0
   #pragma omp parallel
   {
      #pragma omp for schedule (static, 1)
      for (i = 0; i < 5; i++)
      { // work units WU1 to WU5
         #pragma omp critical
         {
            a[i] = a[i] * a[i];
            sum += a[i];
         }
      }
      #pragma omp single
      sum += a[4]; // work unit WU6
   }
   return 0;
}
```

Fig. 4. Corrected OpenMP code for Figure 1
(added `#pragma omp critical` inside the `for`-loop)

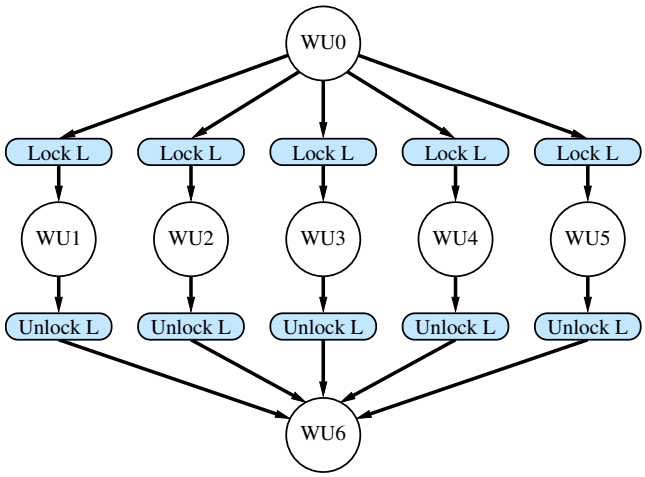

Fig. 5. Work unit graph for the program of Figure 4

Table 2. LNT code for work unit graph of Figure 5

```
module OMP2 is
   channel LOCK_CHANNEL is (Nat) end channel
   process LOCK [INIT, DESTROY: none,
                 ACQUIRE, RELEASE: LOCK_CHANNEL] is
      var FREE: Bool, TID: Nat in
         INIT;
         FREE := true; -- initially the lock is free
         TID := 1;     -- to make LNT2LOTOS happy
         loop L in
            select
               only if FREE then ACQUIRE (?TID) end if
            []
               only if not (FREE) then RELEASE (TID) end if
            []
               DESTROY; break L
            end select;
            FREE := not (FREE)
         end loop
      end var
   end process
   process PROTECTED_WU [WU: none,
                         ACQUIRE, RELEASE: LOCK_CHANNEL]
                        (id: Nat) is
      ACQUIRE (id); -- acquire the lock
      WU;           -- work
      RELEASE (id)  -- release the lock
   end process
   process MAIN [WU0, WU1, WU2, WU3, WU4, WU5, WU6,
                 INIT, DESTROY: none,
                 ACQUIRE, RELEASE: LOCK_CHANNEL] is
      par INIT, ACQUIRE, RELEASE, DESTROY in
         WU0;
         INIT;
         par
            PROTECTED_WU [WU1, ACQUIRE, RELEASE] (1)
         || PROTECTED_WU [WU2, ACQUIRE, RELEASE] (2)
         || PROTECTED_WU [WU3, ACQUIRE, RELEASE] (3)
         || PROTECTED_WU [WU4, ACQUIRE, RELEASE] (4)
         || PROTECTED_WU [WU5, ACQUIRE, RELEASE] (5)
         end par;
         DESTROY;
         WU6
      ||
         LOCK [INIT, DESTROY, ACQUIRE, RELEASE]
      end par
   end process
end module
```

the advantage of taking into account dependencies, without any prior knowledge about the kind of dependency.

To improve precision, some tools combine several approaches. Adaptive Dynamic Analysis Tool (ADAT) [22] selects, for a given program, suitable race detection techniques, based on their scalability and their efficiency in term of thread labeling, access filtering and access detecting. Intel® Thread Checker [35] emulates the sequential version of the application and uses it to derive the happens-before relation. The tool checks the data dependency of accesses to shared variables (whenever an OpenMP directive is detected) by using sequentially traced information, and reports the accesses as races if their dependency satisfies an anti (write-after-read), flow (read-after-write), and output data dependency (write-after-write). Oracle® Developer Studio Thread Analyzer [34] is based on similar techniques and yields comparable results as Thread Checker [18].

Static analysis tools do not require to run the program. To reduce the complexity of the analysis, some approaches limit their scope to subsets of OpenMP. For instance, the race avoidance tool [37] is limited to OpenMP programs using only the `#pragma omp parallel for` construct. ompVerify [3] detects data races using a polyhedral model (used to describe execution order of statement instance, and the relation of statement instances to the memory cells where they are read or write). The tool covers a class of program fragments called Affine Control Loops.

OpenMP Analysis ToolKit (OAT) [26] uses an SMT (Satisfiability Modulo Theories) solver based symbolic analysis to detect data races. In the tool, every parallel code region of an OpenMP program is encoded into a first-order logic formula, which is then solved by the SMT solver. The solution reported by the SMT solver is interpreted to point out errors and generate a feasible execution trace that reveals the errors. Nonconcurrency analysis [25] statically detects whether two statements in an OpenMP program will not be executed concurrently by different threads in a team. The RacerD tool [6,17] of the Infer static analyser[7] for concurrent Java code aims at easily usable and understandable bug reports. Thus RacerD favors the absence of false positives over the guarantee of the absence of data races.

Model checking is another static analysis technique, based on the analysis of the reachable state space derived from a *model* of the program. In the context of parallel programs, this state space is in general huge, because it has to take into account every possible execution scenario. Thus, usually it is necessary to apply property-preserving abstractions to reduce the state space to a tractable size, but still preserve the control-flow and operations on shared variables. The more concrete the model, the more precise the results reported by the model checker, but the more resources are required. Hence, the principal challenge of model checking based approaches is to find the right abstraction level, with just the right balance between precision and analysis complexity. This challenge can be somehow circumvented by not using the model-checker for the verification of the core property, but to supplement another analysis technique, for instance

[7] `https://fbinfer.com/`

to construct a more precise happens-before relation [32] or to refine the set of variables that need a lock as in our approach.

In order to further enhance the accuracy and the scalability of data race analysis, some approaches employ static analysis to provide guided information for the dynamic race detectors. For example, ARCHER [2] first identifies data race-free code regions (i.e., which do not contain data dependencies) with a static analysis, and then instruments only the remaining, potentially unsafe regions, for data race detection. Another approach is the combination of a thread labeling scheme (to maintain the logical concurrency of thread segments) with the happens-before technique (to analyze the happens-before relations to detect conflicting accesses to every shared memory location) [19,21]. The ThreadSafe [1] tool (for Java code) applies the principles of the lockset algorithm in the setting of a static analysis: locksets are computed for abstract summaries of methods. A generic and formal OpenMP epoch model, which describes memory events of all OpenMP threads that occur between two synchronization events, has been used as basis for determining the happens-before relations, which are then applied for detecting data races [7].

5 Conclusion

We proposed an iterative method to ensure the absence of data races in parallel programs using a combination of lockset analysis (to identify unprotected shared variables) and ACTL model checking (to detect superfluous locks). Although simple, our method is modular, separating the concerns of parallelization and verification on a formal model of the program. This enables to balance the precision of the model with the quality of the resulting data race free parallel program: a more detailed model will increase accuracy in detecting superfluous locks, but also require more computing resources to analyze it by model checking. We illustrated the method on the parallelization of programs using OpenMP, by proposing an intermediate representation of the concurrent blocks and their synchronizations as a work unit graph, which is transformed into an LNT model.

The proposed method could be applied for parallel programs in other languages as well, provided that a suitable translation to LNT is available (e.g., AADL2LNT [31]). Once an abstract LNT model of the parallel program is available, it can be used not only for checking sequentiality constraints and deadlocks as in our analysis flow, but also other properties, both qualitative (safety, liveness, fairness) and quantitative ones [28]. The method can be further refined by tackling an industrial case-study involving parallelization using OpenMP. For model checking large work unit graphs, the compositional verification techniques for ACTL provided by CADP [30,13] (and notably the recent combined bisimulation technique [24], suitable for the ACTL sequentiality detection formulas given in Section 3.3), can be experimented to find appropriate composition strategies. Also, alternative ways of translating an OpenMP application into an LNT program can be investigated, with various degrees of abstraction.

Acknowledgements. This work has been supported by the CAPHCA (*Critical Applications on Predictable High-Performance Computing Architectures*) project funded by the PIA programme of the French government.

References

1. Atkey, R., Sannella, D.: ThreadSafe: Static Analysis for Java Concurrency. Electronic Communications of the EASST **72** (2015). https://doi.org/10.14279/tuj.eceasst.72.1025
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 (May 2016). https://doi.org/10.1109/IPDPS.2016.68
3. Basupalli, V., Yuki, T., Rajopadhye, S., Morvan, A., Derrien, S., Quinton, P., Wonacott, D.: ompVerify: Polyhedral Analysis for the OpenMP Programmer. In: Chapman, B.M., Gropp, W.D., Kumaran, K., Müller, M.S. (eds.) OpenMP in the Petascale Era, vol. 6665, pp. 37–53. Springer (2011). https://doi.org/10.1007/978-3-642-21487-5_4, https://hal.inria.fr/hal-00752626
4. Beckman, N.E.: A Survey of Methods for Preventing Race Conditions (2006)
5. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A State/Event-based Model-checking Approach for the Analysis of Abstract System Properties. Sci. Comput. Program. **76**(2), 119–135 (2011). https://doi.org/10.1016/j.scico.2010.07.002
6. Blackshear, S., Gorogiannis, N., O'Hearn, P.W., Sergey, I.: RacerD: Compositional Static Race Detection. Proc. ACM Program. Lang. **2**(OOPSLA), 144:1–144:28 (Oct 2018). https://doi.org/10.1145/3276514, http://doi.acm.org/10.1145/3276514
7. Cramer, T., Schwitanski, S., Münchhalfen, F., Terboven, C., Müller, M.S.: An OpenMP Epoch Model for Correctness Checking. In: 2016 45th International Conference on Parallel Processing Workshops (ICPPW). pp. 299–308 (Aug 2016). https://doi.org/10.1109/ICPPW.2016.51
8. De Nicola, R., Fantechi, A., Gnesi, S., Ristori, G.: An Action-Based Framework for Verifying Logical and Behavioural Properties of Concurrent Systems. Computer Networks and ISDN Systems **25**(7), 761–778 (Feb 1993). https://doi.org/10.1016/0169-7552(93)90047-8
9. De Nicola, R., Vaandrager, F.: Action versus State Based Logics for Transition Systems. In: Guessarian, I. (ed.) Semantics of Systems of Concurrent Processes: Proceedings of the LITP Spring School on Theoretical Computer Science, La Roche Posay, France. Lecture Notes in Computer Science, vol. 469, pp. 407–419. Springer (Apr 1990)
10. Fantechi, A., Gnesi, S., Ristori, G.: From ACTL to Mu-Calculus. In: Proceedings of the ERCIM Workshop on Theory and Practice in Verification (Pisa, Italy). pp. 3–10 (Jan 1992)
11. Fantechi, A., Gnesi, S., Ristori, G.: Model Checking for Action-Based Logics. Formal Methods in System Design **4**(2), 187–203 (1994). https://doi.org/10.1007/BF01384084
12. Fantechi, A., Gnesi, S., Ristori, G.: Modelling Transition Systems within an Action Based Logic. Technical report, IEI-CNR, Pisa (1996)

13. Garavel, H., Lang, F., Mateescu, R.: Compositional Verification of Asynchronous Concurrent Systems Using CADP. Acta Informatica **52**(4), 337–392 (Apr 2015)
14. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. Springer International Journal on Software Tools for Technology Transfer (STTT) **15**(2), 89–107 (Apr 2013)
15. Garavel, H., Lang, F., Serwe, W.: From LOTOS to LNT. In: Katoen, J.P., Langerak, R., Rensink, A. (eds.) ModelEd, TestEd, TrustEd – Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday. Lecture Notes in Computer Science, vol. 10500, pp. 3–26. Springer (Oct 2017)
16. Gnesi, S., Mazzanti, F.: On the Fly Verification of Network of Automata. In: Arabnia, H.R. (ed.) Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications PDPTA'1999 (Las Vegas, Nevada, USA). pp. 1040–1046. CSREA Press (June–July 1999)
17. Gorogiannis, N., O'Hearn, P.W., Sergey, I.: A True Positives Theorem for a Static Race Detector. Proc. ACM Program. Lang. **3**(POPL), 57:1–57:29 (Jan 2019). https://doi.org/10.1145/3290370
18. Ha, O.K., Kim, Y.J., Kang, M.H., Jun, Y.K.: Empirical Comparison of Race Detection Tools for OpenMP Programs. In: Ślezak, D., Kim, T.h., Yau, S.S., Gervasi, O., Kang, B.H. (eds.) Grid and Distributed Computing. pp. 108–116. Communications in Computer and Information Science, Springer (2009)
19. 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, ACM (2012). https://doi.org/10.1145/2338967.2336808
20. Henzinger, T.A., Jhala, R., Majumdar, R.: Race checking by context inference. In: Pugh, W., Chambers, C. (eds.) Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI'2004 (Washington, DC, USA). pp. 1–13. ACM (June 2004). https://doi.org/10.1145/996841.996844
21. Kang, M.H., Ha, O.K., Jun, S.W., Jun, Y.K.: A Tool for Detecting First Races in OpenMP Programs. In: Malyshkin, V. (ed.) Parallel Computing Technologies. pp. 299–303. Springer (2009)
22. Kim, Y., Song, S., Jun, Y.: ADAT: An Adaptable Dynamic Analysis Tool for Race Detection in OpenMP Programs. In: 2011 IEEE Ninth International Symposium on Parallel and Distributed Processing with Applications. pp. 304–310 (May 2011). https://doi.org/10.1109/ISPA.2011.49
23. Kim, Y.J., Park, M.Y., Park, S.H., Jun, Y.K.: A Practical Tool for Detecting Races in OpenMP Programs. In: Malyshkin, V. (ed.) Parallel Computing Technologies. pp. 321–330. Springer (2005)
24. Lang, F., Mateescu, R., Mazzanti, F.: Compositional Verification of Concurrent Systems by Combining Bisimulations. In: Proceedings of the 3rd World Congress on Formal Methods FM'2019 (Porto, Portugal). Lecture Notes in Computer Science, Springer (Oct 2019), to appear
25. Lin, Y.: Static Nonconcurrency Analysis of OpenMP Programs. In: Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming. pp. 36–50. IWOMP'05/IWOMP'06, Springer (2008), http://dl.acm.org/citation.cfm?id=1892830.1892835
26. Ma, H., Diersen, S.R., Wang, L., Liao, C., Quinlan, D., Yang, Z.: Symbolic Analysis of Concurrency Errors in OpenMP Programs. In: 2013 42nd International Conference on Parallel Processing. pp. 510–516 (Oct 2013). https://doi.org/10.1109/ICPP.2013.63

27. Mateescu, R.: CAESAR_SOLVE: A Generic Library for On-the-Fly Resolution of Alternation-Free Boolean Equation Systems. Springer International Journal on Software Tools for Technology Transfer (STTT) **8**(1), 37–56 (Feb 2006), full version available as INRIA Research Report RR-5948, July 2006
28. Mateescu, R., Serwe, W.: Model Checking and Performance Evaluation with CADP Illustrated on Shared-Memory Mutual Exclusion Protocols. Science of Computer Programming **78**(7), 843–861 (Jul 2013)
29. Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) Proceedings of the 15th International Symposium on Formal Methods (FM'08), Turku, Finland. Lecture Notes in Computer Science, vol. 5014, pp. 148–164. Springer (May 2008)
30. Mateescu, R., Wijs, A.: Property-Dependent Reductions Adequate with Divergence-Sensitive Branching Bisimilarity. Sci. Comput. Program. **96**(3), 354–376 (2014)
31. Mkaouar, H., Zalila, B., Hugues, J., Jmaiel, M.: From AADL Model to LNT Specification. In: de la Puente, J.A., Vardanega, T. (eds.) Proceedings of the 20th ADA-Europe International Conference on Reliable Software Technologies (Ada-Europe'15), Madrid, Spain. Lecture Notes in Computer Science, vol. 9111, pp. 146–161. Springer (2015)
32. Nakade, R., Mercer, E., Aldous, P., McCarthy, J.: Model-Checking Task Parallel Programs for Data-Race. In: Dutle, A., Muñoz, C., Narkawicz, A. (eds.) NASA Formal Methods. pp. 367–382. Springer International Publishing (2018)
33. OpenMP Architecture Review Board: OpenMP Application Programming Interface (Nov 2018), https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf
34. Oracle Studio 12.6: Thread Analyzer User's Guide (Jun 2017), https://docs.oracle.com/cd/E77782_01/html/E77800/index.html
35. Petersen, P., Shah, S.: OpenMP Support in the Intel® Thread Checker. In: Voss, M.J. (ed.) OpenMP Shared Memory Parallel Programming. pp. 1–12. Springer (2003)
36. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multithreaded Programs. ACM Trans. Comput. Syst. **15**(4), 391–411 (Nov 1997). https://doi.org/10.1145/265924.265927
37. Shah, D.: Analysis of an OpenMP Program for Race Detection. Master's thesis, San Jose State University (2009)
38. Süß, M., Leopold, C.: Common Mistakes in OpenMP and How to Avoid Them: A Collection of Best Practices. In: Proceedings of the 2005 and 2006 International Conference on OpenMP Shared Memory Parallel Programming. pp. 312–323. IWOMP'05/IWOMP'06, Springer (2008), http://dl.acm.org/citation.cfm?id=1892830.1892863