A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker

Dirk Beyer
LMU Munich, Germany

Karlheinz Friedberger
University of Passau, Germany

Verifying multi-threaded programs is becoming more and more important, because of the strong trend to increase the number of processing units per CPU socket. We introduce a new configurable program analysis for verifying multi-threaded programs with a bounded number of threads. We present a simple and yet efficient implementation as component of the existing program-verification framework CPACHECKER. While CPACHECKER is already competitive on a large benchmark set of sequential verification tasks, our extension enhances the overall applicability of the framework. Our implementation of handling multiple threads is orthogonal to the abstract domain of the data-flow analysis, and thus, can be combined with several existing analyses in CPACHECKER, like value analysis, interval analysis, and BDD analysis. The new analysis is modular and can be used, for example, to verify reachability properties as well as to detect deadlocks in the program. This paper includes an evaluation of the benefit of some optimization steps (e.g., changing the iteration order of the reachability algorithm or applying partial-order reduction) as well as the comparison with other state-of-the-art tools for verifying multi-threaded programs.

1 Introduction

Program verification has successfully been applied to programs to find errors in applications. There exist many approaches to verify single-threaded programs (cf. SV-COMP for an overview [1]), and several of them are already implemented in the open-source program-verification framework CPACHECKER [4, 10]. For multi-threaded programs a new dimension of complexity has to be taken into account: the verification tool has to efficiently analyze all possible thread interleavings. CPACHECKER did not support the analysis of multi-threaded programs for a long time. Our work focuses on a new, simple configurable program analysis that reuses several existing components of the framework. The approach is sound and can be combined with several steps of optimization to achieve an efficient analysis for multi-threaded programs.

Our analysis is based on a standard state-space exploration using a given control-flow automaton that represents the program. For a program state with several active threads, we compute the succeeding program state for each of those threads, i.e. basically we compute every possible interleaving of the threads. The approach is orthogonal to other data-flow based analyses in CPACHECKER, thus it can be combined with algorithms like CEGAR [7] and analyze an potentially infinite state space.

1.0.1 Related Work

A prototypical version of our analysis was already applied for the category of concurrent programs during the SV-COMP'16 [1]. Due to some unsupported features and missing parts of the optimization that where implemented later, the score in this category was low at that time. The experimental results that we report show that the current version of the implementation performs much better.

Just like several other tools [15, 8, 9], we explore possible interleavings of different thread executions and our optimization methods include partial order reduction [12]. In contrast to verification techniques

J. Bouda et al. (Eds.): 11th Doctoral Workshop
on Mathematical and Engineering Methods
in Computer Science (MEMICS 2016)
EPTCS 233, 2016, pp. 61–71, doi:10.4204/EPTCS.233.6

for multi-threaded programs like constraint-based representation [13] that limits the domain to Horn clauses and predicate abstraction or sequentialization [11, 16] that transforms the program on source-code level before starting the analysis, our approach computes the interleaving of threads on-the-fly and is independent from the applied analysis. This makes it possible to integrate our approach easily with data-flow analyses of different abstract domains, such as value analysis [5] and BDD analysis [6].

2 Analysis of Multi-Threaded Programs in CPACHECKER

The following section provides an overview of some basic concepts and definitions used for our approach. We describe the program representation and the details of our configurable program analysis.

2.1 Program Representation

A program is represented by a *control-flow automaton* (CFA) $A = (L, l_0, G)$, which consists of a set L of program locations (modeling the program counter), a set $G \subseteq L \times Ops \times L$ (modeling the control flow with assignment and assumption operations from Ops), and an initial program location l_0 (entry point of the main function).

Let V be the set of variables in the program. The *concrete data state for a program location* assigns a value to each variable from the set V; the set C contains all concrete data states. For every edge $g \in G$, the transition relation is defined by $\xrightarrow{g} \subseteq C \times \{g\} \times C$. The union of all edges defines the complete transfer relation $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. If there exists a chain of concrete data states $\langle c_0, c_1, ..., c_n \rangle$ with $\forall c_i$: there exists a program location l_i for which c_i is a concrete data state and $\forall i : 1 \leq i \leq n \Rightarrow \forall i : 1 \leq i \leq n \Rightarrow \exists g : c_{i-1} \xrightarrow{g} c_i \wedge (l_{i-1}, g, l_i) \in G$, then the state c_n is *reachable* from c_0 for l_0.

Our analysis is a reachability analysis and unrolls the program lazily [14] into an *abstract reachability graph* (ARG) [2]. The ARG is a directed acyclic graph that consists of abstract states (representing the abstract program state, e.g., including program location and variable assignments) and edges modeling the transfer relation that leads from one abstract state to the next one.

2.2 ThreadingCPA

CPACHECKER is based on the concept of *configurable program analysis* (CPA) [3]. Thus, different aspects of a program are analyzed by different components (denoted as CPAs). A default analysis in CPACHECKER [4] uses the LocationCPA to track the program location (program counter) and the CallstackCPA to track call stacks (function calls and their corresponding return location in the CFA). Thus, for the analysis of sequential programs, each abstract state that is reached during an analysis consists of exactly one program location and one call stack.

For the analysis of multi-threaded programs we have developed a new ThreadingCPA that replaces both the LocationCPA and the CallstackCPA and explores the state space of a multi-threaded program on-the-fly. The benefit of the ThreadingCPA is that it is able to track several program locations (one per thread) together with their call stacks (also one per thread). For simplicity of the definition we ignore the handling of call stacks in the next section. The reader can simply assume that for each program location there is also a call stack. The ThreadingCPA has to handle multiple call stacks (one per thread), whereas the CallstackCPA only handles a single call stack.

The definition of the ThreadingCPA $\mathbb{T} = (D_{\mathbb{T}}, \rightsquigarrow_{\mathbb{T}}, merge_{\mathbb{T}}, stop_{\mathbb{T}})$ follows the structure of a configurable program analysis:

Domain: The abstract domain $D_{\mathscr{T}} = (C, \mathscr{T}, [[\cdot]])$ is a triple of the set C of concrete states, the flat semi-lattice $\mathscr{T} = (T, \sqsubseteq, \sqcup, \top)$, and the concretization function $[[\cdot]] : \mathscr{T} \to 2^C$. Let I be the set of all possible thread identifiers, e.g., a set of names used to identify threads in the program. The type of abstract states $T : I \rightharpoonup \mathscr{L}$ consists of all assignments of thread identifiers $t \in I$ to program locations $l \in \mathscr{L} = L \cup \{\top_L\}$. The special program location $\top_L$ represents an unknown program location. The top element $\top \in T$, with $\top(t) = \top_L$ for all $t \in I$, is the abstract state that holds no specific program location for any thread identifier. Each abstract threading state $s \in T$ is represented by the assignments $\{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, ...\}$ of thread identifiers to their current program location. The partial order $\sqsubseteq$ induces a semi-lattice for the abstract states. The join operator $\sqcup$ yields the least upper bound of given abstract states. The top element $\top$ of the semi-lattice is defined as $\top = \sqcup T$.

Merge: The ThreadingCPA uses the merge operator $merge_{sep}$, which does not combine different elements.

Stop: The ThreadingCPA uses the termination operator $stop_{sep}$, which defines coverage only in case of equal abstract states.

Transfer: The transfer relation $\rightsquigarrow_{\mathbb{T}}$ determines the syntactic successor for the current state and is based on the transfer relation of the LocationCPA. The implementation is simple: The transfer relation returns all possible successors for all threads that are active in an abstract state, i.e., it applies the transfer relation of the LocationCPA for each active thread. Additionally, thread-management-related operations are included, such that creating or joining threads (when calling *pthread_create* or *pthread_join*) is defined. It is in theory sufficient to only handle these two function calls, because other thread-related function calls do not change the number of threads or the progress of the state-space exploration. The transfer relation $\rightsquigarrow_{\mathbb{T}}$ has the transfer $s \overset{g}{\rightsquigarrow} s'$ for two abstract states $s = \{t_1 \mapsto l^{t_1}, t_2 \mapsto l^{t_2}, ..., t_N \mapsto l^{t_N}\}$ and $s' = \{t_1 \mapsto l'^{t_1}, t_2 \mapsto l'^{t_2}, ..., t_N \mapsto l'^{t_M}\}$ and $g = (l^{t_i}, op, l'^{t_i})$ if

1. the operation op matches the *pthread_create* statement for t_i that is in program location l^{t_i} and creates a new thread t_{new} starting from a CFA node $l_0^{t_{new}} \in L$:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \cup \{t_{new} \mapsto l_0^{t_{new}}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., an existing thread t_i matches the program location l^{t_i} and moves along the edge g towards program location l', and the initial program location $l_0^{t_{new}}$ of the new thread t_{new} is added to the current abstract state.
2. the operation op matches the *pthread_join* statement for t_i that is in program location l^{t_i} and waits for a thread t_{exit} to exit, t_{exit} exits at program location $l_E^{t_{exit}}$, and $t_{exit} \mapsto l_E^{t_{exit}} \in s$:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \setminus \{t_{exit} \mapsto l_E^{t_{exit}}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., an existing thread t_i matches the program location l^{t_i} and moves along the edge g towards program location l'^{t_i}, and the program location $l_E^{t_{exit}}$ of the thread t_{exit} is removed from the current abstract state, if the thread t_{exit} has already been at this program location.
3. otherwise, if the operation op is not related to thread management:
$$s' = s \setminus \{t_i \mapsto l^{t_i}\} \cup \{t_i \mapsto l'^{t_i}\}$$
i.e., thread t_i matches the program location l^{t_i} and moves along the edge towards l'^{t_i}.

For a basic analysis for multi-threaded programs the handling of the operations *pthread_create* and *pthread_join* is sufficient. Additional thread management like mutex locks (details in Section 4.3) can be applied on top of this transfer relation. We assume C statements as atomic statements, i.e., interleaving of threads is considered to happen on statement level (matching the encoding of the program as CFA). This might be insufficient for real-world programs, but is good enough for several examples and in theory the CFA could be inflated with read and write operations for memory registers.

```
1  pthread_t id1, id2;
2  int i=1, j=1;
3
4  void main() {
5    pthread_create(&id1, 0, t1, 0);
6    pthread_create(&id2, 0, t2, 0);
7
8    pthread_join(id1, 0);
9    pthread_join(id2, 0);
10
11   assert(j <= 8);
12 }
13
14 void t1() {
15   i+=j;
16   i+=j;
17 }
18
19 void t2() {
20   j+=i;
21   j+=i;
22 }
```

Figure 1: Program with concurrent threads

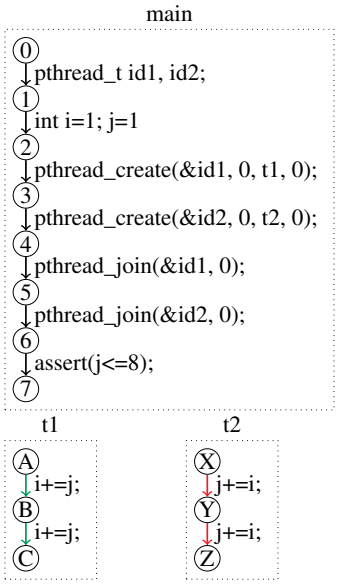

Figure 2: CFA for the functions of the program

2.3 Example

The following example applies our new ThreadingCPA to a given program. In contrast to the simplified illustration below, a real-world analysis would combine the ThreadingCPA with another analysis, e.g., to track assignments, such as value analysis or BDD analysis.

The example program (cf. Fig. 1 for the source code) creates two additional threads that change the value of global variables. Afterwards, the main method checks the assignment of a global variable. In this example, the property holds. The program's functions are represented as CFAs in Figure 2. The ThreadingCPA produces the ARG in Fig. 3, where each abstract state is labeled with the indices of the program locations of all active threads.

The analysis starts at entry location l_0 of the main function and analyzes all possible interleavings. After reaching the statement *pthread_create*, an additional program location is tracked for the newly created thread, e.g., when reaching program location l_3 in the main function, the abstract state is enriched with the initial program location l_A of the newly created thread.

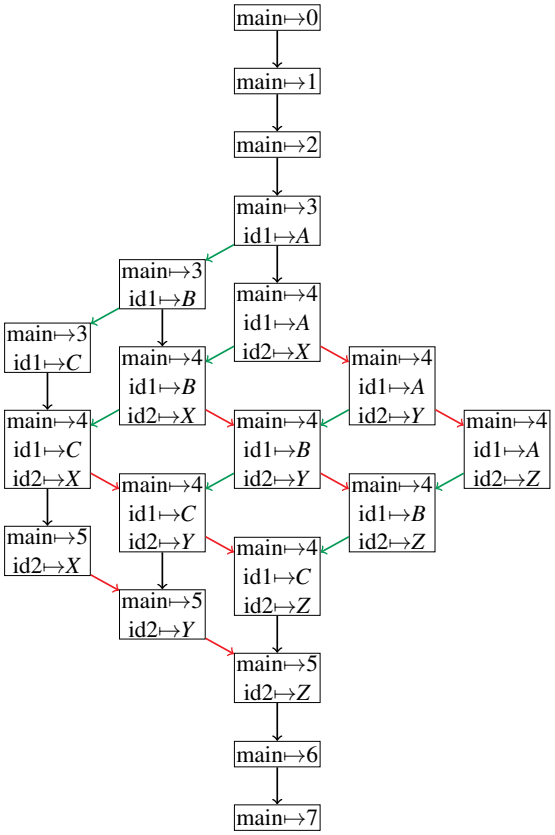

Figure 3: ARG of the interleaved threads of the program

As the ThreadingCPA merges its abstract states when reaching the same program locations via different execution paths, the diamond-like structure in the ARG is the result of interleaved thread-execution of two (or more) threads. When exploring the statement *pthread_join*, the program-exit location of the exiting thread is removed from the abstract state. This is visible in Fig. 3 for each abstract state with an outgoing edge leading from program location l_4 towards program location l_5, because the program-exit location l_C of the joining thread t_1 (identified by id 1) is removed from the abstract state.

3 Optimization

The simple definition of the ThreadingCPA allows (and needs) a wide range of optimization to gain competitive efficiency. In the following, we define some approaches and show how fluently they match existing concepts in CPACHECKER.

3.1 Partitioning of Reached Abstract States

The reachability algorithm [3] has two important operators *merge* and $stop$ that are defined as operations on sets of reached abstract states. These operations can merge abstract states and combine their information into a new abstract state or detect coverage, i.e., an abstract state is implied by another one and thus the exploration can stop at that point. In each iteration of the reachability algorithm, these operators are by default applied to *all* combinations of new explored abstract states and previously reached abstract states. However, applying such an operator to *all* previously reached abstract states is inefficient, because most of the abstract states are irrelevant for a concrete application of these operators. For example, comparing abstract states from different program locations is useless, because there will not be any important relation between them.

Partitioning the set of abstract states makes it possible to perform both operations much more efficiently, as only a (small) subset of the previously reached abstract states has to be considered in the computation. This basic optimization is also applied for verifying single-threaded programs. Each partition is identified by a constant key that is based on the program location of the abstract state, as only states from equal program locations are considered for merging or coverage. We extended the existing partitioning of abstract states, such that it uses the tuple of program locations for all threads in an abstract state. This new partitioning can also be combined with partitionings provided by other CPAs.

3.2 Waitlist Order

For finding property violations it is often sufficient to only analyze interleavings with a low number of thread interleavings. As the exploration algorithm in CPACHECKER analyzes the reachable state space state by state, there exists the possibility to *prioritize abstract states* during the exploration: The abstract states waiting to be analyzed are simply sorted by some criteria. This optimization is a heuristic depending on the internal structure of the analyzed program and the executed analysis. For a bug-free program this heuristic does not bring any benefit. however an existing error path in a faulty program might be found sooner.

The most-often used orderings of abstract states cause the state-space exploration to perform either depth-first search (DFS) or breadth-first search (BFS), i.e., the list of waiting abstract states is ordered in the same manner as abstract states are explored (BFS) or reverse (DFS). For multi-threaded programs, we added a new ordering of this list based on the number of active threads, such that states with fewer active threads are considered first. The new ordering can also be combined with existing orderings, i.e., the

first criteria for ordering is based on the number of active threads, the second criteria uses the exploration order.

3.3 Partial-Order Reduction

With multi-threaded programs, the most common form of optimization is *partial-order reduction* (POR) [12, 19, 18]. POR aims to avoid unnecessary interleavings of threads and improves the performance of the analysis by reducing the explored state space. However, its application depends on the property to be verified, because all necessary program paths must remain reachable.

In our case (reachability analysis), we started with a simple separation of program operations (modeled as CFA edges) into *thread-local* and *global* operations. We conservatively apply a static analysis for all program variables and memory accesses, on whether they are declared and used in *global* scope or only *locally* in the context of a thread. Because CPACHECKER uses several dummy operations (e.g., for temporary variables or function returns), a majority of CFA edges is marked as *thread-local*.

If a statement is *thread-local* for a thread, we do not simulate any interleaving after analyzing this operation, but the analysis executes the current thread further, until a global operation (in the same thread) is reached. This behavior is *sound*, because no interaction between threads is possible, due to the definition of *thread-local* operations. Thus, we only need to synchronize all available threads after the next *global* transition.

Our approach can analyze program with loops as well, because we execute both paths, i.e., the loop and the concurrent thread, and none of them disables the other path. Thus, any possible interaction between CFA edges of the loop and other threads is considered. Our approach handles loops implicitly, thus we do not have to actively check for loops, but simply apply the reachability algorithm combined with the described POR technique.

4 Extensions

During our work on the analysis of multi-threaded programs, we explored some assumptions in CPACHECKER that need to be considered when integrating such a basic analysis as the ThreadingCPA. We also noticed several features that can also be specified or implemented for the analysis of multi-threaded programs. In the following, we describe the extensions that we have developed in order to use the full potential of the framework.

4.1 Cloning for CFAs

CPACHECKER has a modular structure, such that many components can be combined without knowing (and depending on) details about each other. As the analysis of multi-threaded programs should fit into this design, we decided not to modify each analysis that should be combined with our new approach, but use an approach that allows us to re-use as much existing code as possible.

The basic problem with the existing components of CPACHECKER is that many of them rely on knowing only their current function scope, and solely identify a variable by its name combined with the name of the function scope it was declared in. For example, many analyses (including value analysis and BDD analysis) use the identifier $f::x$ for a variable x declared in function f. This identifier is used in the internal data structures whenever the variable is used during the program analysis. In a multi-threaded program, the same function f might be called in different threads, such that $f::x$ is not unique for one

variable any more at a certain point in the program's execution. The existing analyses do not know about two variables with the same identifier and would, e.g., assign a wrong value to one of them.

Our solution is simple: We use different function names for each thread by *cloning* the function and inserting the corresponding indexed function name. For a function f we create a clone f' by copying the corresponding CFA nodes from L and edges from G, while renaming all appearances of the function's identifier in the clone. Cloning functions causes all function-local variables to be unique for different threads in the later applied analysis, e.g., the identifier $f::x$ is distinct from $f'::x$. An analysis using the identifier does not even have to know whether the function is cloned and can simply assume uniqueness of identifiers for all variables.

4.2 Deadlock Detection

A deadlock [17] is defined as an abstract state where two (or more) competing actions wait for each other to finish, and thus neither ever does. CPACHECKER allows the user to define the goal of an analysis by giving a specification in form of an automaton. Detecting deadlocks in the program can be done by giving an observer automaton that monitors the abstract states of the ThreadingCPA and reports deadlocks. This approach is independent of any further analysis and can be combined with, e.g., value analysis or BDD analysis.

4.3 Mutex Locks

Mutex locks are commonly used to synchronize threads, e.g., to manage access to shared memory. In our implementation, mutex locks are stored as part of the abstract state of the ThreadingCPA. If a mutex lock is requested along a CFA edge, but not available in the preceding abstract state, the transfer relation does not yield a successive abstract state for the CFA edge.

Additionally, we use mutex locks for more use cases: We simulate atomic sequences of statements and some aspects of partial order reduction as mutex locks in the ThreadingCPA. Entering an atomic sequence requires an *atomic mutex lock*, which is released after leaving the atomic sequence. Consecutive CFA edges containing only thread-local operations (see Section 3.3) are modeled and analyzed as atomic sequence.

5 Evaluation

In this section we evaluate different configurations of the ThreadingCPA and compare it with other state-of-the-art tools. The evaluation is performed on machines with a 2.6 GHz Intel Xeon E5-2650 v2 CPU running Ubuntu 16.04 (Linux 4.4.0). Each single verification run is limited to 15 min of run time and 15 GB of memory. The 1016 benchmark tasks are taken from the category of multi-threaded programs at SV-COMP'16 [1]. The tasks are C programs, where reaching a specific function call is considered as property violation. We use CPACHECKER [2] 1.6.1 in revision 23011.

[1] https://github.com/sosy-lab/sv-benchmarks/releases/tag/svcomp16
[2] https://cpachecker.sosy-lab.org/

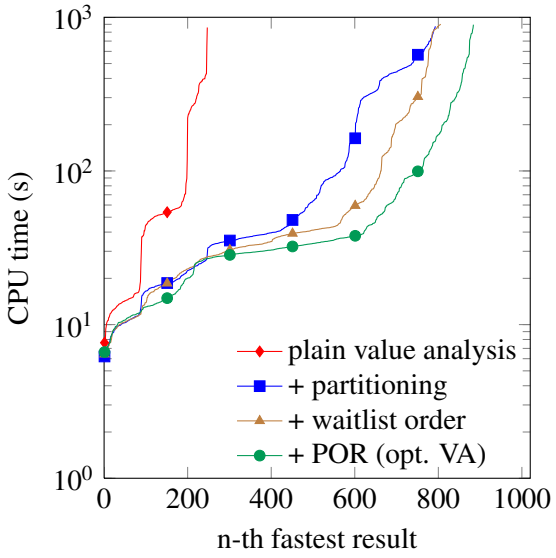

Figure 4: Quantile plot for different configurations of the value analysis, corresponding to step-wise applied optimization steps

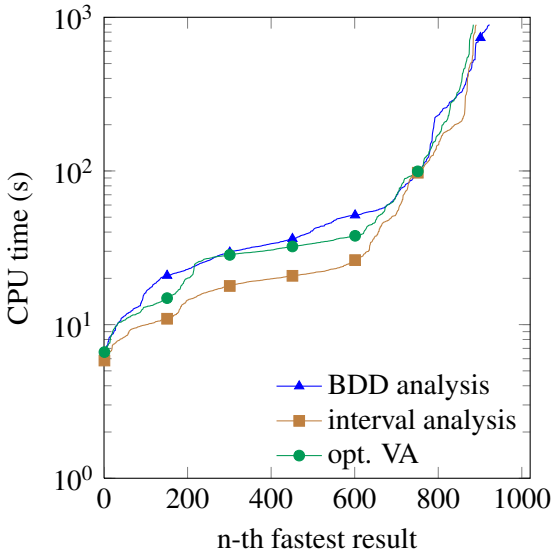

Figure 5: Quantile plot for different abstract domains using the ThreadingCPA within CPACHECKER

5.1 Optimization Steps

First, we show the effect of applying each optimization step from Section 3 successively, i.e., on top of the previous optimization. Starting with a plain (non-optimized) configuration of the ThreadingCPA combined with the value analysis, we step-wise apply optimization in form of

- reached-set partitioning (see Section 3.1) based on the abstract states,
- waitlist ordering (see Section 3.2) based on the number of threads, and
- POR (see Section 3.3) based on *local-scope* and *global* statements.

The optimization steps are independent of the value analysis and can also be applied to any other analysis like BDD analysis and interval analysis, where the same benefit will be visible. Figure 4 shows a quantile plot containing the run time of correctly solved verification tasks. The evaluation shows that the verification process benefits from each of the optimization steps. For small tasks that can be verified within a few second, e.g., because of only a few thread interleavings in the program, the benefit of optimization is small. For tasks that need more run time the benefit becomes visible.

We noticed that the heuristic of ordering the waiting abstract states is beneficial in two ways: first, some property violations are found earlier (some property violations need only a small number of interleavings); second, some unsupported operations (like assigning several thread instances to the same thread identifier) are reached earlier and the analysis can abort immediately without wasting time.

Compared to the plain value analysis, partitioning the reached set improves the performance and reduces the run time of the analysis by more than an order of magnitude. Additionally changing the waitlist order improves run time in several cases, mostly for tasks with a property violation. However, in our benchmark this optimization step does not lead to more correctly solved tasks. POR causes a lower number of explored abstract states, and thus the performance increases.

5.2 Abstract Domains

Second, we combine the ThreadingCPA with different analyses, such as value analysis, interval analysis, and BDD analysis, which are already implemented in the CPACHECKER framework and are normally used for the analysis of single-threaded programs. We only evaluate the optimized version of each combination. The analyses could also be combined with CEGAR [7], however the current benchmark does not benefit from it, and thus we just execute a reachability algorithm to verify the specification. We show that we can verify the majority of benchmark programs and discuss strengths and weaknesses of the analyses. As all compared analyses use the same framework (parser, algorithm, ...), we expect our evaluation to be fair for all implemented approaches and allow a precise comparison. Figure 5 shows the quantile plot of correct results for the combinations of the ThreadingCPA with other analyses.

The BDD analysis is optimized for BFS in the reachability algorithm, whereas value analysis and interval analysis use DFS as basic order for the list of waiting abstract states during the exploration algorithm (see Section 3.2). Thus, the state-space exploration traverses program locations and thread interleavings in another order and finds the corresponding abstract states in a different order, too. Depending on the verification task, this can result in an in- or decreased performance compared to the value analysis.

5.3 Other Tools

Third, we compare the (optimized) value analysis with two other state-of-the-art verification tools, namely CBMC [3] and VVT [4]. Both tools are executed as in the SV-COMP'16 and are chosen, because they do not apply special approaches like sequentialisation, but rely on a similar state-space exploration technique as our approach in CPACHECKER. Figure 6 shows the quantile plot of correct results for CBMC, VVT, and CPACHECKER (using the optimized value analysis). The ThreadingCPA (combined with value analysis) is competitive with the other tools. The plot for CPACHECKER matches the trend of the other tools with only some differences. At the left side of the plot the initial start-up time of a few seconds for CPACHECKER is visible, whereas other tools already solve some of the given instance within this time. Due to the missing support for pointer aliasing and array computations in the value analysis as well as due to our simple kind of POR, CPACHECKER can not solve as many verification tasks as other tools within the time limit.

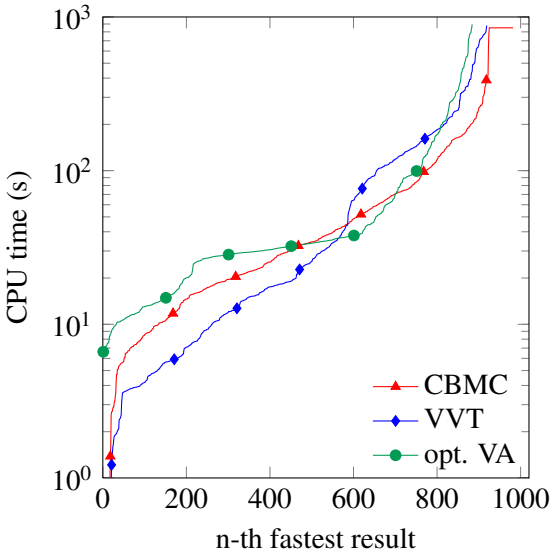

Figure 6: Quantile plot for comparison of other verifiers with support for multi-threaded programs

[3] http://www.cprover.org/cbmc/
[4] https://vvt.forsyte.at/

6 Conclusion

This paper presents a basic approach to support the analysis of multi-threaded programs in CPACHECKER. We formally defined a new ThreadingCPA in the framework and demonstrated that several core components can be reused. Re-using existing analyses is possible without any further overhead. Due to our simple approach, there are a few limitations that have to be considered when verifying multi-threaded programs with CPACHECKER. Our approach for partial order reduction is simple and can be extended with more advanced techniques to further reduce the number of explored abstract states. The maximum number of threads is bounded, because of possible conflicts in function names. To avoid naming conflicts, we clone each function's CFA several times before starting the analysis. The number of clones cannot be changed afterwards. If we run out of clones during the analysis and would need more due to a naming conflict, we abort the analysis and report an insufficient number of threads.

As the ThreadingCPA identifies each thread only by the variable it is assigned to, we currently can not analyze more complex thread management such as pointer aliasing for the thread identifier or more complex locking mechanisms. Our framework already contains a mechanism for exchanging information between abstract states on a state-level during the analysis. The analysis of multi-threaded programs could be extended to exchange information about thread management with another analysis capable of such data, such that we could analyze more difficult thread management with the ThreadingCPA.

Possible ideas for optimization have been implemented and evaluated. The evaluation shows that the results of different analyses based on the ThreadingCPA are competitive with other state-of-the-art tools.

References

[1] D. Beyer (2016): *Reliable and Reproducible Competition Results with* BENCHEXEC *and Witnesses*. In: *Proc. TACAS*, Springer, pp. 887–904, doi:10.1007/978-3-662-49674-9_55. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2016-TACAS.Reliable_and_Reproducible_Competition_Results_with_BenchExec_and_Witnesses.pdf`.

[2] D. Beyer, T. A. Henzinger, R. Jhala & R. Majumdar (2007): *The Software Model Checker* BLAST. *Int. J. Softw. Tools Technol. Transfer* 9(5-6), pp. 505–525, doi:10.1007/s10009-007-0044-z. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2007-STTT.The_Software_Model_Checker_BLAST.pdf`.

[3] D. Beyer, T. A. Henzinger & G. Théoduloz (2007): *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis*. In: *Proc. CAV*, LNCS 4590, Springer, pp. 504–518, doi:10.1007/978-3-540-73368-3_51. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2007-CAV.Configurable_Software_Verification.pdf`.

[4] D. Beyer & M. E. Keremoglu (2011): CPACHECKER*: A Tool for Configurable Software Verification*. In: *Proc. CAV*, LNCS 6806, Springer, pp. 184–190, doi:10.1007/978-3-642-22110-1_16. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2011-CAV.CPAchecker_A_Tool_for_Configurable_Software_Verification.pdf`.

[5] D. Beyer & S. Löwe (2013): *Explicit-State Software Model Checking Based on CEGAR and Interpolation*. In: *Proc. FASE*, LNCS 7793, Springer, pp. 146–162, doi:10.1007/978-3-642-37057-1_-11. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2013-FASE.Explicit-State_Software_Model_Checking_Based_on_CEGAR_and_Interpolation.pdf`.

[6] D. Beyer & A. Stahlbauer (2013): *BDD-Based Software Model Checking with* CPACHECKER. In: *Proc. MEMICS*, LNCS 7721, Springer, pp. 1–11, doi:10.1007/978-3-642-36046-6_1. Available at `http://www.sosy-lab.org/~dbeyer/Publications/2013-MEMICS.BDD-Based_Software_`

Model_Checking_with_CPAchecker.pdf.

[7] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu & H. Veith (2003): *Counterexample-guided abstraction refinement for symbolic model checking*. *J. ACM* 50(5), pp. 752–794, doi:10.1145/876638.876643.

[8] E. M. Clarke, D. Kröning, N. Sharygina & K. Yorav (2005): SATABS*: SAT-Based Predicate Abstraction for ANSI-C*. In: *Proc. TACAS*, LNCS 3440, Springer, pp. 570–574, doi:10.1007/978-3-540-31980-1_40.

[9] L. Cordeiro & B. Fischer (2011): *Verifying multi-threaded software using smt-based context-bounded model checking*. In: *Proc. ICSE*, ACM, pp. 331–340, doi:10.1145/1985793.1985839.

[10] Matthias Dangl, Stefan Löwe & Philipp Wendler (2015): CPACHECKER *with Support for Recursive Programs and Floating-Point Arithmetic*. In: *Proc. TACAS*, LNCS 9035, Springer, pp. 423–425, doi:10.1007/978-3-662-46681-0_34.

[11] B. Fischer, O. Inverso & G. Parlato (2013): *CSeq: A Sequentialization Tool for C (Competition Contribution)*. In: *Proc. TACAS*, LNCS 7795, Springer, pp. 616–618, doi:10.1007/978-3-642-36742-7_46.

[12] Patrice Godefroid (1996): *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*. LNCS 1032, Springer, doi:10.1007/3-540-60761-7.

[13] A. Gupta, C. Popeea & A. Rybalchenko (2011): *Threader: A Constraint-Based Verifier for Multi-threaded Programs*. In: *CAV*, LNCS 6806, Springer, pp. 412–417, doi:10.1007/978-3-642-22110-1_32.

[14] T. A. Henzinger, R. Jhala, R. Majumdar & G. Sutre (2002): *Lazy abstraction*. In: *Proc. POPL*, ACM, pp. 58–70, doi:10.1145/503272.503279.

[15] G. J. Holzmann (1997): *The* SPIN *Model Checker*. *IEEE Trans. Softw. Eng.* 23(5), pp. 279–295, doi:10.1109/32.588521.

[16] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2015): *Lazy-CSeq: A Context-Bounded Model Checking Tool for Multi-threaded C-Programs*. In: *Proc. ASE*, ACM, pp. 807–812, doi:10.1109/ASE.2015.108.

[17] Sreekaanth S. Isloor & T. Anthony Marsland (1980): *The Deadlock Problem: An Overview*. *IEEE Computer* 13(9), pp. 58–78, doi:10.1109/MC.1980.1653786.

[18] Doron A. Peled (1993): *All from One, One for All: on Model Checking Using Representatives*. In: *Proc. CAV*, LNCS 697, Springer, pp. 409–423, doi:10.1007/3-540-56922-7_34.

[19] Antti Valmari (1989): *Stubborn sets for reduced state space generation*. In: *Proc. Petri Nets*, LNCS 483, Springer, pp. 491–515, doi:10.1007/3-540-53863-1_36.