Multi-Threaded Software Model Checking via Parallel Trace Abstraction Refinement

Max Barth
Department of Computer Science
LMU Munich
Munich, Germany
Max.Barth@lmu.de

Marie-Christine Jakobs
Department of Computer Science
LMU Munich
Munich, Germany
M.Jakobs@lmu.de



***Abstract*—Automatic software verification is a valuable means for software quality assurance. However, automatic verification and in particular software model checking can be time-consuming, which hinders its practical applicability e.g., the use in continuous integration. One solution to address the issue is to reduce the response time of the verification procedure by leveraging today's multi-core CPUs.**

In this paper, we propose a solution to parallelize trace abstraction, an abstraction-based approach to software model checking. The underlying idea of our approach is to parallelize the abstraction refinement. More concretely, our approach analyzes different traces (syntactic program paths) that may violate the safety property in parallel. We realize our parallelized version of trace abstraction in the verification tool ULTIMATE AUTOMIZER and perform a thorough evaluation. Our evaluation shows that our parallelization is more effective than sequential trace abstraction and provides results significantly faster on many time-consuming tasks. Also, our approach is more effective than DSS, a recent parallel approach to abstraction-based software model checking.

I. INTRODUCTION

Software failures may have severe consequences. In particular for safety-critical systems, it is therefore important to not only detect software failures but also prove their absence. While testing may reveal failures, it typically fails to prove their absence. In contrast, formal software verification techniques support both, the detection of specification violations (i.e., bugs) and the proof that a program is correct wrt. its specification. However, formal verification needs to become less time-consuming to fit into the software development process.

One solution is to leverage today's multi-core CPUs and reduce the response time via parallelization. Parallel portfolio approaches, e.g., [1], [2], [3], [4], which run different software verifiers in parallel, facilitate easy parallelization. Unfortunately, they also waste a lot of resources because they only use the result of the first verifier that finishes successfully. Other parallelization approaches, e.g. [5], [6], [7], split the verification task (program plus property) into several subtasks and analyze them in parallel. Still, finding a good splitting is difficult. Also, many parallel verification approaches, e.g., [8], [9], [10], [11], [12], parallelize the computation of the (abstract) state space, which requires complicated synchronization. In contrast, we follow an orthogonal approach and aim to achieve the reduction in response time by parallelizing the abstraction refinement loop performed in counterexample-guided abstraction refinement (CEGAR) [13], which to the best of our knowledge has not been tried before in the verification of sequential programs. Concretely, we parallelize the refinement loop of trace abstraction [14], [15], an automata-theoretic approach to software model checking that is based on CEGAR and employs interpolation during refinement.

```
0: if x>0
1:  x=-x;
2: else
3:  while x>-10
4:    x=x-1;
5: assert x!=0;
```

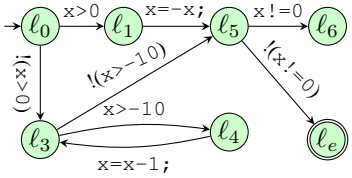


Fig. 1. Code (left) and program automaton (right) of our example `NotZero`

Trace abstraction [14], [15] is an automata-based software model checking approach for safety properties. It maintains an automaton (the abstraction), which describes those syntactical program paths that lead to a property violation and whose feasibility still needs to be checked. During verification, trace abstraction alternates between (a) selecting a path from the current abstraction, (b) checking the feasibility of the selected path, and when proven infeasible (c) determining an automaton that describes the infeasible path and paths with similar reasons of infeasibility before (d) refining the abstraction, i.e., removing the determined paths from the abstraction.

We observe that steps (b) and (c) can be done independently when given the selected path and the program. Hence, our parallelization uses several worker instances for steps (b) and (c), which all consider different paths. In addition, a single coordinator is responsible for steps (a) and (d) as well as the communication with the workers.

We demonstrate our parallelization for two worker threads on the program[1] shown in Fig. 1. First, the coordinator selects the two paths π_1 =`x>0,x=-x;,(x!=0)` and π_2 =`!(x>0),!(x>-10),!(x!=0)`, respectively following the if and the else branch. Note that both paths are semantically infeasible. The coordinator assign paths π_1

[1]For more details on the program automaton, we refer to the next section.

https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_58

This article is licensed under a Creative Commons Attribution 4.0 International License

and π_2 to separate workers. Assume that the coordinator first gets the result for path π_1. Thus, it excludes π_1 from its current abstraction. For demonstration purposes, we assume that after the abstraction refinement the result for path π_2 is not yet available. Hence, the coordinator selects a third path $\pi_3 =$ `!(x>0),x>-10,x=x-1;,!(x>-10),!(x!=0)` and assigns it to the free worker. Next, we assume that the coordinator receives the result for π_2 and gets to know that it can exclude all paths following the else branch. After removing the paths from the else branch from the abstraction, the coordinator detects that there are no further paths to check and reports the program is proven safe. While this example wastes CPU time on processing path π_3, the response time of the verification should still be shorter because it processes paths π_1 and π_2 in parallel.

At first glance, realizing the demonstrated parallelization may appear effortless. However, the intricacies lie in the details. For an effective parallelization of the refinement loop, we should analyze dissimilar paths in parallel, which allow us to refine different parts of the abstraction. To this end, we introduce a new search algorithm that allows our parallelization to select dissimilar paths. Also, integrating the parallelization into the sequential model checker ULTIMATE AUTOMIZER is non-trivial, as the lack of related approaches suggests. One reason is that we cannot safely access the same SMT solver context from different threads, nor can we combine formulas from different SMT solvers contexts. Thus, the integration requires deep copies of the solver contexts and extensive transfers of formulas between them.

We implement the demonstrated idea of parallel trace abstraction refinement in the tool ULTIMATE AUTOMIZER and thoroughly evaluate it on verification tasks of the SVBenchmark. Our evaluation shows that parallelization can be beneficial wrt. effectiveness and response time. In addition, we submitted one particular configuration of our parallel trace abstraction [16], which uses two worker threads, for independent evaluation to SV-COMP 2026[2] [17] in the meta categories `C.ReachSafety` and `C.NoOverflows`. Despite parallel trace abstraction being more beneficial with a wall-time limit and SV-COMP using a CPU-time limit, our submission wins the category `C.NoOverflows` and solves more tasks than ULTIMATE AUTOMIZER's sequential submission in both meta-categories.

In summary, our paper makes the following contributions.

- We conceptually develop a parallel version of the trace abstraction algorithm (Sec. III), an abstraction-based algorithm for software model checking targeting safety properties, and integrate it into ULTIMATE AUTOMIZER.
- We develop a search algorithm (Sec. III-C) to find diverse traces in the trace abstraction.
- We perform a rigorous experimental evaluation of parallel trace abstraction (Sec. IV) considering 14 560 C verification tasks of the SVBenchmark and four different numbers of workers. Also, we compare against a state-of-the-art parallelization approach.

[2]https://sv-comp.sosy-lab.org/2026/results/results-verified/

II. SEQUENTIAL TRACE ABSTRACTION

In this paper, we aim to parallelize trace abstraction [14], [15], a software verification technique to analyze whether a program adheres to a given safety property. Trace abstraction, which is also known as automata-based software model checking, maintains and iteratively refines (i.e., reduces the size of) a set of traces, a subset of the syntactical program paths that may cause a property violation. During verification, the set of traces forms the current (trace) abstraction of the program and is represented via an automaton. In the following, we formally introduce the notion of traces, define the initial trace abstraction, and explain the sequential verification procedure that iteratively refines trace abstractions.

A. Traces and Initial Trace Abstraction

Our goal is to use traces to represent (syntactic) program paths. To this end, a trace describes the order in which a (syntactic) program path performs its operations. The set Σ describes the set of all possible program operations. Hence, *traces* are sequences $\pi \in \Sigma^*$ of program operations. We order these traces based on their prefix relation. More concretely, we write $\pi' \preceq \pi$ if there exists $\pi'' \in \Sigma^*$ such that $\pi = \pi'\pi''$.

During verification, we consider sets $\mathcal{S}_\pi$ of traces that represent syntactic program paths that will violate the safety property of interest if they are semantically feasible. For the efficient representation of a set $\mathcal{S}_\pi$ of traces, we use a finite automaton A with alphabet Σ. The automaton's language is exactly the set of traces, i.e., $\mathcal{L}(A) = \mathcal{S}_\pi$. The verification starts with the set $\mathcal{S}_\mathcal{I}$ of all traces that model a syntactic program path that potentially violates the safety property of interest. Since one may encode safety properties as the unreachability of certain program locations (named error locations) [18], we restrict ourselves to safety properties encoded by error locations. Thus, the set $\mathcal{S}_\mathcal{I}$ contains all traces that model a syntactic program path that may end in an error location.

To describe the set $\mathcal{S}_\mathcal{I}$, we need to model the program as an automaton that accepts the traces from the set $\mathcal{S}_\mathcal{I}$. To this end, we use a *(program) automaton* $A_\mathcal{P} = (L, \delta_\mathcal{P}, \ell_0, F_\mathcal{P})$ [14]. Its set L of states represents the program locations, which are closely related to the program counter values. The initial location $\ell_0 \in L$ relates to the program counter value at the start of the program. Furthermore, the transition relation $\delta_\mathcal{P} \subseteq L \times \Sigma \times L$ models the program's control-flow. Transitions $(\ell, \mathtt{op}, \ell') \in \delta_\mathcal{P}$ describe which operations `op` may be executed at a given program location ℓ and where to proceed (ℓ') after the execution of operation `op`. Finally, the set $F_\mathcal{P} \subseteq L$ contains the accepting states, which correspond to the program's error locations and determine the safety property. Hence, a program automaton accepts exactly the traces in $\mathcal{S}_\mathcal{I}$.

Figure 1 shows the source code (left) and the resulting program automaton (right) of our example program `NotZero`. Except for the error location ℓ_e, the index of a location

Algorithm 1 Sequential trace abstraction procedure [14]

Input: program automaton $A_{\mathcal{P}}$

1: $A = A_{\mathcal{P}}$;
2: **while** True **do**
3: **if** $\mathcal{L}(A) == \emptyset$ **then** return SAFE;
4: select $\pi \in \mathcal{L}(A)$
5: (result, interpolants) = checkSAT(φ_π);
6: **if** result == SAT **then** return UNSAFE;
7: A_π = generateAutomaton(π, interpolants, A);
8: $A = A \setminus A_\pi$

in the program automaton refers to the corresponding line number in the source code of the program. Furthermore, the automaton contains one edge per assignment and two assume statement edges per if, while, or assert statement, namely one for each evaluation of their condition. To let the violation of the assertion result in a property violation, the false evaluation of the assertion's condition leads to the error location ℓ_e.

B. Iterative Trace Abstraction Refinement

Next, we describe the iterative verification procedure, namely the sequential trace abstraction algorithm. The verification procedure employs counterexample-guided abstraction refinement (CEGAR) [13] and is shown in Alg. 1.

The algorithm gets a program automaton, which accepts the traces that represent syntactic program paths that lead to a property violation (i.e., an error location). During verification, the algorithm maintains automaton A to represent the current trace abstraction, which contains all traces that model a syntactic program path to an error location for which the semantic feasibility has not yet been checked. Line 1 initializes the trace abstraction with the program automaton $A_{\mathcal{P}}$. Lines 2–8 perform the iterative refinement of the trace abstraction. The verification will end with result SAFE (i.e., property proven) in line 3 if the language of the automaton A is empty, i.e., there does not exist a trace that models a feasible program path that ends in an error location. If the language is non-empty, there still exist syntactic program paths to error locations for which the semantic feasibility has not yet been checked. Hence, line 4 uses a breadth-first search to select a trace $\pi \in \mathcal{L}(A)$ representing one such path. Thereafter, line 5 checks the semantic feasibility of that path. To this end, it first constructs an SSA-based formula encoding φ_π of π and subsequently checks φ_π's satisfiability with an SMT solver. For example, for trace `!(x>0),!(x>-10),!(x!=0)` we check formula $\neg(x_0 > 0) \wedge \neg(x_0 > -10) \wedge \neg(x_0 \neq 0)$ and for trace `x>0,x=-x;,!(x!=0)` we check formula $x_0 > 0 \wedge x_1 = -x_0 \wedge \neg(x_1 \neq 0)$. If the result of the satisfiability check is SAT, we proved that the syntactic program path represented by trace π is semantically feasible. Since all program paths represented by a trace $\pi \in \mathcal{L}(A)$ end in an error location, we found a feasible counterexample and line 6 returns UNSAFE (i.e., property violation detected). Otherwise, we refine the trace abstraction in lines 7 and 8. First, we construct an *interpolant automaton* A_π [14] that accepts π and other traces with similar reasons of semantic infeasibility. Note that the construction of A_π uses abstraction A to focus on traces that are still relevant and, thus, to make the construction more efficient. Figure 2 shows a possibly interpolant automaton for trace `!(x>0),!(x>-10),!(x!=0)`. The automaton encodes the trace and extends it with an additional loop. Also, the automaton records the infeasibility argument for its accepted traces by annotating its states with assertions. Typically, we use interpolation to derive the assertions from the unsatisfiability proof of the trace's formula encoding. Given the interpolant automaton, which by construction only accepts traces that are semantically infeasible, line 8 refines our current trace abstraction. More concretely, it removes the traces accepted by the interpolant automaton from the current trace abstraction by computing the difference of our current abstraction (automaton A) and the interpolant automaton A_π.

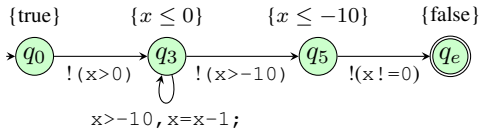


Fig. 2. An interpolant automaton for trace `!(x>0),!(x>-10),!(x!=0)`

III. Towards Parallel Trace Abstraction

Next, we describe how we parallelize the iterative trace abstraction procedure, which we described in the previous section. The idea is to parallelize the loop performing the iterative refinement of the trace abstraction.

A. Preliminary Considerations

Looking at the loop (lines 2–8 of Alg. 1), we observe that the only dependency between loop iterations is the trace abstraction represented by automaton A. The update of the abstraction in line 8 aggregates the results of the different iterations and therefore, should not be done in parallel. Nevertheless, the aggregation itself is commutative, i.e., for the abstraction itself it does not matter in which order we use the interpolant automata to refine it. Furthermore, we notice that the check in line 5, which inspects the feasibility of trace π, is independent of abstraction A. In addition, when generating the interpolant automaton we only use the abstraction A to make the generation more efficient. Hence, using an older, out-of-date abstraction, which accepts more traces, is sound but may not be as efficient as using the latest one. Consequently, we can retrieve the most recent abstraction right before generating the interpolant automaton. For the emptiness check in line 3 and the search of a trace $\pi \in \mathcal{L}(A)$, it is also sound to use an out-of-date abstraction. However, using an out-of-date abstraction may likely result in performing unnecessary work, e.g., more refinement iterations, which one could avoid when using the latest abstraction. Further taking into account that we experienced that the feasibility check (line 5) and the interpolant automaton generation (line 7) are the most expensive tasks in each iteration, we decide to consider those

Algorithm 2 Coordinator that directs parallel trace abstraction

Input: program automaton $A_{\mathcal{P}}$

1: $A = A_{\mathcal{P}}$; $\mathcal{S}_{\pi_{\text{ex}}} = \emptyset$;
2: **while** True **do**
3: **if** $\mathcal{L}(A) == \emptyset$ **then** return SAFE;
4: **while** idleWorkerAvailable() $\wedge$ $\mathcal{L}(A) \setminus \mathcal{S}_{\pi_{\text{ex}}} \neq \emptyset$ **do**
5: select $\pi \in \mathcal{L}(A) \setminus \mathcal{S}_{\pi_{\text{ex}}}$
6: $\mathcal{S}_{\pi_{\text{ex}}} = \mathcal{S}_{\pi_{\text{ex}}} \cup \{\pi\}$;
7: giveTaskToWorker(π, A);
8: waitUntilAtLeastOneWorkerResultAvailable();
9: **while** workerResultAvailable() **do**
10: (A_π, result) = getWorkerResult();
11: **if** result == SAT **then** return UNSAFE;
12: $A = A \setminus A_\pi$;

two tasks for parallel execution. Thereby, we concede that different threads may need to maintain memory-expensive out-of-date copies of the abstraction. This is acceptable because from our experience sequential trace abstraction much more often runs out of time than it runs out of memory.

B. Procedure of Parallel Trace Abstraction

To realize our parallelization, we use a central coordinator (Alg. 2) in combination with several workers (Alg. 3). The coordinator directs the verification procedure. To this end, it runs a modified version of the sequential trace abstraction procedure (Alg. 1). Differences are highlighted in gray and mainly represent synchronization with workers. In addition, it maintains a set $\mathcal{S}_{\pi_{\text{ex}}}$, which tracks the traces already assigned for analysis, and uses it to avoid exploring the same trace π twice.[3] Like the sequential procedure, the coordinator (Alg. 2) initializes and iteratively updates the trace abstraction (lines 1 and 12). Also, it checks the stopping criteria of the verification procedure (lines 3 and 11). To maintain the set $\mathcal{S}_{\pi_{\text{ex}}}$, Alg. 2 initializes it with an empty set of already analyzed traces in line 1 and updates it in line 6 after selecting a trace to analyze. Furthermore, we adapt the trace selection (line 5) such that it only selects traces that are in the current trace abstraction and have not been assigned for analysis yet, i.e., are not in set $\mathcal{S}_{\pi_{\text{ex}}}$. In contrast to the sequential procedure, Alg. 2 distributes the selected traces to the available workers for analysis and relies on the workers' results to update its trace abstraction.

The workers (Alg. 3) process the traces. They perform the feasibility check of the traces (line 1). For traces proven infeasible, they also compute the interpolant automaton (line 3) required for refinement. Hence, they perform the steps in lines 5 and 7 of the sequential verification procedure (Alg. 1). In addition, they make the result (the result of the feasibility check and the interpolant automaton) available for the coordinator (line 5). To ensure that they provide a structurally valid result, they construct an automaton accepting the empty language in line 4 if the trace is feasible, i.e., no interpolant automaton is required.

[3]Note that in contrast to sequential trace abstraction, which either stops exploration when detecting that a trace is feasible or removes the infeasible trace from the trace abstraction before selecting the next trace, our parallel trace abstraction selects traces while the analysis of traces selected earlier may not have finished.

Algorithm 3 Worker for trace processing

Input: trace π, trace abstraction A with $\pi \in \mathcal{L}(A)$

1: (result, interpolants) = checkSAT(φ_π);
2: **if** result = UNSAT **then**
3: A_π = generateAutomaton(π, interpolants, A);
4: **else** $A_\pi = (\{q_0\}, \emptyset, q_0, \emptyset)$;
5: publishResult(A_π, result);

To coordinate with the workers, each iteration of the coordinator contains a phase to distribute the work to the workers (lines 4–7) followed by a phase to collect and process the workers' results (lines 8–12). Each phase is realized with a while loop. The loop distributing work stops when no further worker or trace to analyze is available, while the loop considering worker results stops if no result is available. Furthermore, we avoid busy waiting by explicitly waiting until some result becomes available in line 8.

Using the coordinator together with one worker basically performs sequential trace abstraction, but distributed on two threads. When we use more than one worker, we need to take particular care of the trace selection. To avoid redundant work, we already ensure that a trace is only selected once. Ideally, we would like to select diverse traces, e.g., traces with different reasons for infeasibility. We discuss trace selection in more detail in the next section.

C. Procedure for Diverse Trace Selection

Now, we discuss our trace selection (i.e., line 5 of Alg. 2). The selection must ensure that it does not select a trace from $\mathcal{S}_{\pi_{\text{ex}}}$ and ideally it should select a trace with a new reason for infeasibility. The more the newly selected trace differs from the previously selected traces $\mathcal{S}_{\pi_{\text{ex}}}$, the likelier it will be that it provides a different reason for infeasibility. With this in mind, we aim to steer our selection such that it selects traces that diverge early from previously selected traces.

Algorithm 4 shows our selection procedure, which implements a recursive search procedure through the abstraction. The algorithm gets the trace abstraction A to search through, the state q where in the abstraction to continue the search, the already selected prefix π, and the set of relevant, analyzed traces $\mathcal{S}_{\text{rt}}$, i.e., the previously selected traces with prefix π that are not yet known to be infeasible. Thus, line 5 of Alg. 2 calls the algorithm with the current trace abstraction, the abstraction's initial state q_0, an empty prefix trace ε, and set $\mathcal{S}_{\pi_{\text{ex}}} \cap \mathcal{L}(A)$ of traces already assigned for analysis but not yet known to be infeasible.

To select traces that diverge early from previously selected traces, Alg. 4 prioritizes successor transitions that have been reached less often via the prefix π currently considered by the

Algorithm 4 Recursive algorithm `search` to select diverse traces from a trace abstraction

Input: trace abstraction $A = (Q, \delta, q_0, F)$, search state q, selected prefix π, relevant set of analyzed traces $\mathcal{S}_{\text{rt}}$

```
1:  if S_rt = ∅ then
2:    if acceptingStateReachableFrom(q, A) then
3:      return π ∘ shortestPathToAcceptingStateFrom(q, A);
4:    else return null;
5:  succ = list({(q, ·, ·) ∈ δ});
6:  sort(succ, (q, op, q') → |{π' ∈ S_rt | π ∘ op ⪯ π'}|);
7:  while !empty(succ) do
8:    (q, op, q') = dequeue(succ);
9:    if π ∘ op ∉ S_rt then
10:     π_r = search(A, q', π ∘ op, {π' ∈ S_rt | π ∘ op ⪯ π'});
11:     if π_r != null then return π_r;
12: return null;
```

search procedure. If no traces have been selected yet that use the currently considered prefix π, i.e., $\mathcal{S}_{\text{rt}}$ is empty, lines 1–4 return the shortest trace with prefix π that is in the language of the abstraction or null if none exist. The prioritization itself is realized in lines 5–12 of Alg. 4. First, line 5 builds a list of all transitions that start in the current search state q. Next, the algorithm sorts these transitions by how likely they lead to a new trace. We approximate the likelihood with the number of previously selected traces that are continuations of the trace built from the currently selected prefix π and the transition's operation. Thereby, a higher number indicates a smaller likelihood. Thereafter, the while loop realizes the actual prioritized search. In each iteration, line 8 removes the transition with the largest likelihood (smallest number of continuations) that has not yet been explored. The check in line 9 ensures that we skip the transition whenever the transition's operation together with the currently selected trace results in a previously selected trace. In all other cases, we recursively search a trace not previously selected in line 10. The recursive call searches for a continuation of prefix trace π extended with the current transition's operation. Therefore, it continues the search with the successor q' of the currently considered transition and adapts the search to those relevant traces that agree with the new prefix. If the recursive call detects a new trace ($\pi_r \neq$ null), we return it in line 11. However, if the current transition cannot lead us to a new trace, the next loop iteration tries the next transition. If we fail to generate a new trace with any transition leaving the current search state q, we return null in line 12 and backtrack. Note that Alg. 4 terminates with a non-null result as long as there exists a trace accepted by trace abstraction A, which is not in the initial set $\mathcal{S}_{\text{rt}}$ of analyzed traces. Algorithm 2 guarantees this whenever it selects a new trace.

D. Implementation

We integrate our approach for parallel trace abstraction into ULTIMATE AUTOMIZER [19], a Java-based software verification tool that already supports sequential trace abstraction. Following our conceptual approach, we reuse the exsiting components of sequential trace abstraction for parallel trace abstraction but need to make them ready for parallel execution. In addition, we add the parallelization and adapt the trace search.

To make the components of trace abstraction ready for parallel execution, we must use multiple SMT solver instances in parallel. Since ULTIMATE AUTOMIZER links SMT formulae to specific SMT solver instances and uses SMT formulae to represent operations in its automaton abstraction, in interpolant automata, and in error traces, we need to copy the corresponding SMT formulae when exchanging the abstraction, an interpolant automaton, or an error trace between the coordinator and a worker. Moreover, we use different SMT solver instances for trace checking and interpolation in addition to supporting multiple interpolation strategies [20], [21]. Thus, a worker uses several SMT solver instances, which further increases the complexity of the exchange. Additionally, ULTIMATE AUTOMIZER selects the interpolation strategies it applies to analyze an error trace based on already explored similar traces and the strategies applied to them [21]. Hence, we ensure that the coordinator and workers exchange this information as well.

To realize the parallelization, we use the `Runnable` interface to implement Alg. 3, which processes a given trace π, and Furthermore, we realize the workers with a fixed thread pool, which is provided as an `ExecutorService`. Note that the user can configure the number of threads in the pool via a parameter. To start a new trace processing and to retrieve the results, we combine the `ExecutorService` instance with a `CompletionService` instance. The coordinator uses the `CompletionService` instance to initiate a new trace processing and to retrieve the results from a finished trace processing without busy waiting.

To allow for a better comparison between sequential and parallel trace abstraction, our implementation of parallel trace abstraction aims to include all traces π in its analysis that the sequential trace abstraction analyzes. To this end, we use a slightly modified search for diverse traces. In the first iteration of each execution of the loop in lines 4–7 of Alg. 2, we first apply breadth-first search, the search strategy of sequential trace abstraction, to detect a trace π accepted by the current trace abstraction (i.e., $\pi \in \mathcal{L}(A)$). If π has not been found before (i.e., $\pi \notin \mathcal{S}_{\pi_{\text{ex}}}$), we distribute π to a worker and continue with the next iterations of the loop, which all perform our proposed search (Alg. 4). Otherwise, we apply our proposed search. However, we do not compute $\mathcal{S}_{\pi_{\text{ex}}} \cap \mathcal{L}(A)$ when calling the search but overapproximate it. To this end, we remove a trace π from $\mathcal{S}_{\pi_{\text{ex}}}$ when we process the result of its analysis. Also, we do not check $\mathcal{L}(A) \setminus \mathcal{S}_{\pi_{\text{ex}}} \neq \emptyset$ but limit each search with Alg. 4.[4] Since we do not limit our BFS search, the BFS search ensures that we find an error trace if all workers are idle, i.e., we know $\mathcal{L}(A) \setminus \mathcal{S}_{\pi_{\text{ex}}} \neq \emptyset$.

[4]In practice, we use a limit of 5 seconds, which causes a timeout in 0.1% of the searches in our experiments.

TABLE I
PER TOOL CONFIGURATION, NUMBER OF CORRECTLY AND INCORRECTLY REPORTED PROPERTY VIOLATIONS (ALARMS) AND PROPERTY ADHERENCES (PROOFS)

	SEQ-UA	PAR-1-UA	PAR-2-UA	PAR-4-UA	PAR-6-UA	DSS
correct alarms	1128	1140	1241	1243	**1245**	291
correct proofs	3377	3695	3747	**3860**	3858	1263
incorrect alarms	0	0	0	0	0	317
incorrect proofs	0	0	0	0	0	62

IV. EVALUATION

The goal of our parallelization is to reduce the response time and to increase the effectiveness of the approach. Hence, our evaluation primarily examines the effect of our parallelization on response time and solved tasks and compares our parallelization with a recent parallelization technique that uses a software model checking technique similar to trace abstraction.

A. Experimental Setting

Tool Configurations. Our evaluation runs four different configurations of our parallel trace abstraction, namely PAR-1-UA, PAR-2-UA, PAR-4-UA, and PAR-6-UA. They differ in the size of the fixed thread pool. PAR-1-UA uses one worker thread and describes the sequential baseline of our approach. The other three configurations use two, four, and six worker threads. Since even the sequential configuration PAR-1-UA uses a coordinator and worker thread, we also run ULTIMATE AUTOMIZER with the sequential configuration SEQ-UA used in the 14th competition on software verification (SV-COMP25). For all five configurations, we use ULTIMATE AUTOMIZER version 0.3.0-474959ed[5]. Moreover, we use DSS, a recent state-of-the-art technique for parallel model checking, which employs distributed summary synthesis [7]. For DSS, we use the version provided in the artifact [22] accompanying the paper.

Verification Tasks. Our evaluation considers verification tasks from the software verification benchmark[6] (SVBenchmark), which is used by the international competition on software verification (SV-COMP) [23]. A verification task in SV-COMP consists of a program and a property. We restrict our evaluation to all tasks from SV-COMP 2025 that consider (a) sequential C programs, the programs supported by trace abstraction in ULTIMATE AUTOMIZER, and (b) property `unreach-call`, a safety property which states that certain error locations (encoded as calls to the function `reach_error`) must not be reached. However, we exclude one task for which the SV-COMP community recently determined that it contains undefined behavior, i.e., it became invalid. In total, we consider 14 559 tasks, 11 157 of them fulfill the property while 3 402 of them violate the property.

Evaluation Environment. The machines, we use, contain 33 GB of memory and an Intel Xeon E3-1230 v5 CPU with 8 processing units and a frequency of 3.40 GHz. Their operating system is a 64-bit Ubuntu 24.04 with Linux kernel 6.8 and the installed Java version is OpenJDK 21.0.7. We limit each run of any tool configuration on a verification task to 8 CPU cores, 15 min of wall time and 30 GB of memory and enforce the limits with BENCHEXEC [24] (version 3.18).

[5]Note that SEQ-UA solves slightly more tasks in this version than in its version submitted to SV-COMP 2025.

[6]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp25-final

B. Impact of Parallelization on Effectiveness

In this section, we study the impact of our parallelization on effectiveness, i.e., the number of (correctly) solved tasks. To this end, let us look at Tab. I, which shows for every tool configuration (column) the number of correctly and incorrectly reported alarms (property violations) and proofs (property satisfactions). When studying the trace abstraction configurations (first five columns), we observe that PAR-1-UA performs better than the SV-COMP 2025 configuration SEQ-UA. One reason is that our new search strategy allows us to recover from exceptions that occur during trace processing, i.e., instead of stopping the verification process, we free the worker to process other traces and keep the error trace in $\mathcal{S}_{\pi_{\text{ex}}}$ to ensure that it is not selected again. Further studying Tab. I, we observe that none of our configuration reports any incorrect results and with increasing number of threads the number of solved alarms and proofs typically increases. The only exception is PAR-6-UA, which solves less proofs than PAR-4-UA. We investigate this further and observe that PAR-1-UA can correctly solve 2 219 of 4 835 tasks by analyzing less than six traces, i.e., PAR-6-UA does not provide any benefits while causing additional overhead, which may cause timeouts. When restricting the set of tasks for which PAR-1-UA analyzes at least six traces, i.e., all configurations have its worth, PAR-6-UA has the largest number of correct proofs.

> Effectiveness typically increases when the number of workers increases.

C. Effect of Parallelization on Response Time

Next, we study whether parallelization may reduce the response time (i.e., wall time). To observe the immediate effect of parallelization on response time, we compare configurations for trace abstraction that only differ in the number of worker threads. Hence, we compare the wall time of our approach with one worker (PAR-1-UA) with the configurations PAR-2-UA, PAR-4-UA, and PAR-6-UA that utilize 2, 4, and 6 workers. In addition, we compare PAR-2-UA, PAR-4-UA, and PAR-6-UA with

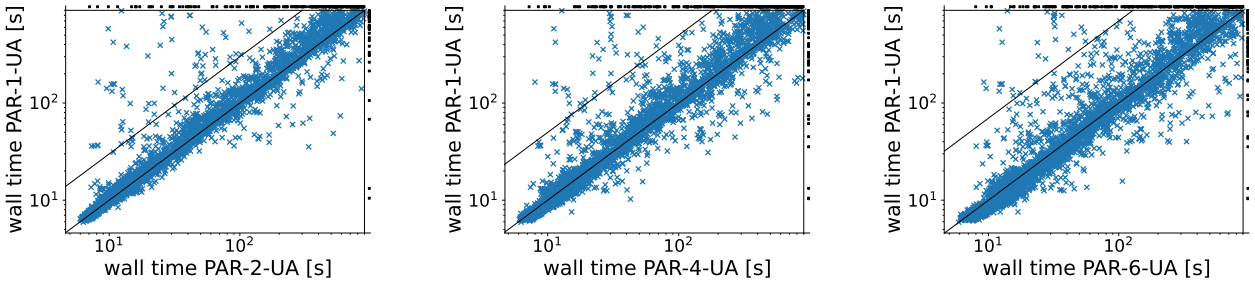


Fig. 3. Scatter plot that compares the wall time of PAR-2-UA (left), PAR-4-UA (middle), and PAR-6-UA (right) with the wall time of PAR-1-UA (y-axes) on tasks solved by both configurations. Additionally, points outside the rectangle represent tasks either not solved by PAR-1-UA (top) or PAR-2-UA, PAR-4-UA, and PAR-6-UA (right)

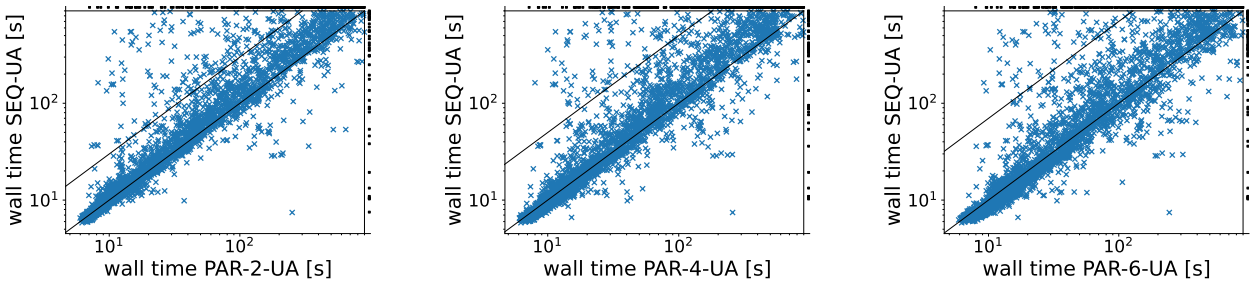


Fig. 4. Scatter plot that compares the wall time of PAR-2-UA (left), PAR-4-UA (middle), and PAR-6-UA (right) with the wall time of SEQ-UA (y-axes) on tasks solved by both configurations. Additionally, points outside the rectangle represent tasks either not solved by SEQ-UA (top) or PAR-2-UA, PAR-4-UA, and PAR-6-UA (right)

the sequential configuration SEQ-UA. Figures 3 and 4 show one scatter plot for each of the parallel configurations PAR-2-UA, PAR-4-UA, and PAR-6-UA. Each scatter plot in Fig. 3 compares the wall time of the respective parallel configuration (x-axis) with the wall time of PAR-1-UA (y-axis) while Fig. 4 compares the wall time of the respective parallel configuration (x-axis) with the wall time of SEQ-UA (y-axis). Thereby, each scatter plot contains one data point (x, y) for each task correctly solved by at least one of the two compared configurations, i.e., 5 036 (5 036) tasks for PAR-2-UA, 5 178 (5 161) tasks for PAR-4-UA, and 5 203 (5 172) tasks for PAR-6-UA in Fig. 3 (Fig. 4). (Black) points above the top border show tasks that PAR-1-UA and SEQ-UA do not solve, respectively. Similarly, tasks right of the right border show tasks that the respective configuration using multiple worker threads does not solve. Depending on the scatter plot in Fig. 3, PAR-1-UA fails to solve between 201 and 368 tasks, while PAR-2-UA, PAR-4-UA, and PAR-6-UA fail to solve 48, 75, and 100 tasks, respectively. Sometimes, a configuration with multiple worker threads fails while PAR-1-UA solves the tasks close to the timeout. Thus, the overhead for parallelization may cause the failure. In other cases, we observe that the configuration with multiple worker threads analyzes more traces and still fails. Hence, we believe that the order in which the trace abstraction is refined may differ. This may lead to different trace abstractions such that the relevant traces that need to be analyzed to succeed will not be selected and analyzed early enough. Still, the number of failures of PAR-1-UA for which the parallelization succeeds are much larger than vice versa. We make a similar observation when comparing against SEQ-UA. Namely, SEQ-UA fails to solve between 531 and 667 tasks while PAR-2-UA, PAR-4-UA, and PAR-6-UA fail to solve 48, 58, and 69 tasks, respectively.

Next, let us look at tasks solved by both of the compared configurations, i.e., the points in the rectangle. All three scatter plots in Fig. 3 contain points below the diagonal, i.e., PAR-1-UA responds faster, and above the diagonal, i.e., PAR-1-UA responds slower. More importantly, there are points in each plot that are above the line parallel to the diagonal. This line represents the upper bound for the speedup achievable according to Amdahl's law.[7] For a point to be above that upper bound, the parallelization needs to skip some work done by PAR-1-UA. For example, due to parallelization we may find a property violation earlier, find a better loop abstraction earlier, or in general get to a smaller abstraction earlier, which may allow us to skip processing some traces considered by PAR-1-UA. In detail, PAR-2-UA responds faster for 49% (2 369 of 4 787) of the commonly solved tasks and achieves a speedup larger than 3 for 67 tasks. Over all commonly solved tasks, PAR-2-UA's mean and median wall time is 87 s and 17 s compared to 101 s and 17 s for PAR-1-UA. Furthermore, PAR-4-UA responds faster for 43% (2 002 of 4 760) of the commonly solved tasks and achieves a speedup larger than 5 for 42 tasks. Its mean and median wall time is 84 s and 17 s compared to 99 s and 17 s for PAR-1-UA. PAR-6-UA responds faster 34% (1 619 of 4 735) of the commonly solved tasks and achieves

[7]The upper bound of the speedup is identical with the thread number.

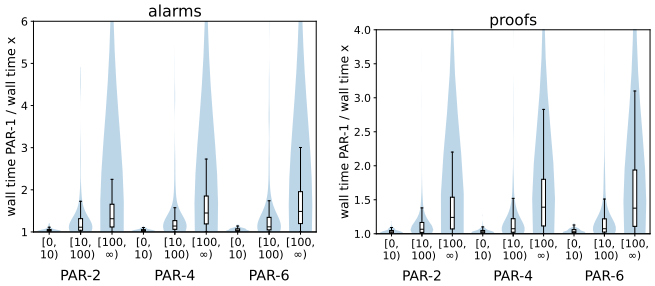


Fig. 5. Speed up on tasks for which PAR-1-UA responds more slowly. Grouped by the wall time of PAR-1-UA into the intervals $[0, 10)$, $[10, 100)$, and $[100, \infty)$. On the left, we only consider tasks with property violations (alarms) while on the right we only consider tasks that adhere to the specification (proofs).

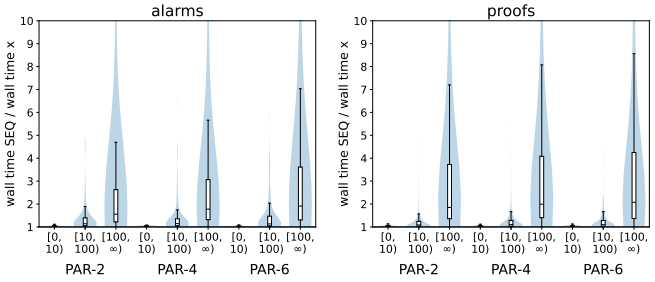


Fig. 6. Speed up on tasks for which SEQ-UA responds more slowly. Grouped by the wall time of SEQ-UA into the intervals $[0, 10)$, $[10, 100)$, and $[100, \infty)$. On the left, we only consider tasks with property violations (alarms) while on the right we only consider tasks that adhere to the specification (proofs).

a speedup larger than 7 for 31 tasks. Its mean and median wall time is 86 s and 18 s compared to 97 s and 17 s for PAR-1-UA. Moreover, we observe that if parallelization is slower, it will typically not take that much longer. Indeed, it only takes between 0.68 and 1.99 additional seconds in the median and between 9.58 and 14.59 additional seconds on average. When not profiting from parallelization, we do not lose much.

Comparing the three parallel configurations with SEQ-UA (Fig. 4), we observe similar but slightly better results. PAR-2-UA, PAR-4-UA, and PAR-6-UA respond faster for 55% (2 482 of 4 457), 45% (2 019 of 4 447), and 38% (1 683 of 4 436) of the commonly solved tasks, respectively, and achieve a speedup above Amdahl's law for 170, 95, and 64 tasks, respectively. Over all commonly solved tasks, the three parallel configurations require 67-69 s in the mean and 15-16 s in the median while SEQ-UA takes 91-93 s in the mean and 16 s in the median. Also when SEQ-UA is faster, the parallel configurations only take between 0.55 and 1.70 additional seconds in the median and between 9.22 and 11.59 additional seconds on average.

Next, let us study those tasks for which PAR-2-UA, PAR-4-UA, and PAR-6-UA respond faster, respectively. Our goal is to investigate whether the benefit of parallelization is different for proof and alarm detection as well as for more difficult tasks, i.e., tasks for which PAR-1-UA (SEQ-UA) takes longer. Figures 5 and 6 show two plots each, the left plot considers alarms and the right considers proofs. Each plot contains three combined violin and box plots per configuration that uses multiple worker threads. Each combined violin and box plot considers tasks solved by both, the respective configuration with multiple worker threads and the respective sequential configuration, PAR-1-UA in Fig. 5 and SEQ-UA in Fig. 6. However, the combined plots only consider those commonly solved tasks for which PAR-1-UA (SEQ-UA) is slower wrt. wall time and show the distribution of the speedup (i.e., wall time of PAR-1-UA (SEQ-UA) divided by wall time of the other configuration) for tasks which PAR-1-UA (SEQ-UA) solves in less than 10 s of wall time, in 10-100 s of wall time, and more than 100 s of wall time. For each parallel configuration, we observe that in the alarms and proofs plots in Fig. 5 and 6 the violins and boxes become larger from left to right and the whiskers become higher, i.e., we may be able to save more wall time when the verification is complex and takes longer. However, the variance for more complex tasks is also higher. Furthermore, we observe that the variance of the tasks solved within a specific time interval increases when using more worker threads. It seems that with more worker threads the possible speedup depends more and more on the benchmark.

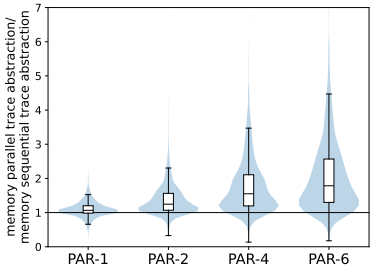


Fig. 7. Memory increase of the four configurations of parallel trace abstraction compared to sequential trace abstraction SEQ-UA

> Parallelization can significantly reduce response time, particularly for time-consuming tasks, while exhibiting negligible negative effects when no improvements are achieved.

D. Effect of Parallelization on Memory Usage

Now, we briefly study how much more memory parallel trace abstraction requires due to copies of the abstraction, etc. To this end, Fig. 7 shows one combined violin and box plot for each configuration of parallel trace abstraction that shows the distribution of the ratio of used peak memory of that configuration and that of the sequential configuration SEQ-UA on all task solved by both configurations. When looking at the figure, we observe that for all configurations the median and average increase is smaller or equal two and typically significantly below the number of workers. We even observe ratios below one, i.e., the parallel configuration requires less memory. In these cases, the parallel configuration skips work–

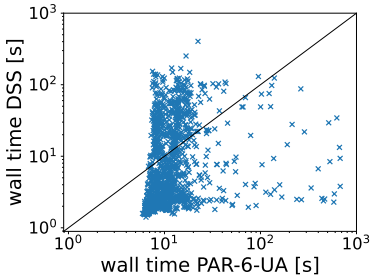


Fig. 8. Comparing the wall time of our approach PAR-6-UA (x-axis) and competitor DSS (y-axis)

similar to the effect that leads to speedups exceeding those predicted by Amdahl's law.

Memory usage of parallel trace abstraction scales well.

E. Comparison Against Existing Parallelization Techniques

Finally, we compare our parallelization to state-of-the-art parallelization approaches. We are aware of two recent, relevant parallelization approaches: ranged program analysis [6] and distributed summary synthesis (DSS) [7]. Like us, both may use a verification procedure based on predicates, CEGAR, and interpolation. We exclude ranged program analysis because we technically failed to port it to Ubuntu 24.04 (the OS environment for our experiments), Since DSS makes use of all 8 available cores, we decide to compare DSS to our configuration PAR-6-UA, which uses the most threads.

Effectiveness. First, we compare the number of solved tasks, which are shown in column PAR-6-UA and DSS of Tab. I. We observe that PAR-6-UA correctly detects about four-times more alarms and three-times more proofs than DSS while DSS reports much more incorrect results, both alarms and proofs. One reason for the higher number of incorrect results is DSS's limited support for pointers and arrays, which is known to cause incorrect results.

Efficiency. To compare the efficiency in terms of response time (wall time), we restrict our comparison to the 1 395 of all 14 559 tasks that our approach (i.e., PAR-6-UA) and DSS solved correctly. Figure 8 shows a scatter plot that contains one point (x, y) for each of the 1 395 tasks that compares the wall time of our approach (x) with the wall time of DSS (y). We observe that there exist points above and below the diagonal, i.e., our approach solves the task faster than DSS and vice versa. A detailed analysis reveals that DSS uses less time for 66% (915 of 1 395) of the tasks correctly solved by DSS and PAR-6-UA. On these 915 tasks, DSS outperforms PAR-6-UA by a median of 7.1 s and a mean of 18 s. Conversely, PAR-6-UA is faster on the remaining 480 tasks correctly solved by DSS and PAR-6-UA, with a median improvement of 21 s and a mean improvement of 31 s. While DSS is more often faster on the tasks correctly solved by DSS and PAR-6-UA, PAR-6-UA is faster by a bigger margin.

Our approach is significantly more effective than DSS. On tasks solved by both approaches, DSS is more frequently faster but when PAR-6-UA outperforms DSS, it does so by a larger margin.

F. Threats to Validity

Our implementation of the parallelization may contain bugs. We think it is unlikely since we never observed any incorrect result. Still, our results may not generalize. Despite using all sequential C tasks from SV-COMP that consider the `unreach-call` safety property, which encode different safety properties, the set may not reflect all real-world sequential programs and safety properties. Also, results may differ with other resource limits.

V. Related Work

We are aware of one other software verification approach [25] that parallelizes the CEGAR refinement step. The approach performs scheduling constraint based abstraction refinement, a verification approach for concurrent programs, which is based on bounded model checking (BMC). Its parallelization runs multiple instances of the verification procedure in parallel and lets each instance use a random search strategy parameterized by the process ID to detect different counterexamples (i.e., syntactic paths to property violations). The refinement results (i.e., scheduling constraints) are exchanged via a shared data structure. In contrast, we consider a different verification technique and verify sequential programs. Furthermore, we run the analyses of counterexamples (including interpolation and abstraction refinement) not the complete verification procedure in parallel. Additionally, we perform a more sophisticated counterexample search.

Instead of parallelizing the refinement steps, Lopes and Rybalchenko [10] parallelize the exploration of the abstract state space in predicate-abstract model checking. Also, there are approaches [8], [26], [9] that parallelize the state-space construction in explicit model checking. Parallelized state-space construction may even use GPUs [27], [28], [29]. In context of static analysis, e.g., dataflow analyses, some approaches parallelize the computation of abstract facts [11], [30], [31], [12] while in LTL model checking, several approaches [32], [33], [34] parallelize the emptiness check of the language described by the product of system automaton and negated specification. Also, some approaches run complete algorithmic blocks of an analysis in parallel like base- and step-case in k-induction [35] or invariant generation for k-induction [36]. Furthermore, verifiers may use implementations of decision diagrams [37] or SAT and SMT solvers [38] that can operate in parallel, e.g., use multiple threads to perform an operation.

Parallelizing the verification algorithm may be complex. Thus, other approaches split the verification task into subtasks and analyzes them in parallel. One option is to split the set of program paths, which is e.g., applied to dataflow analyses [39],

BMC [40], [41], [42], software model checking [6], [43], [44], [45], and symbolic execution [46], [47], [48]. Another strategy is to decompose the program into code blocks, as e.g., done in dataflow analyses [49], [50], verifiers like Bolt [51], Coverity [52], BAM [5], or DSS [7], and static application security testing [53]. When one needs to analyze multiple properties, e.g., different assertions, one can also analyze the individual properties in parallel [54], [55], [56].

Even simpler than splitting the verification task is it to run different verifiers or different verifier configurations in parallel. Approaches like PredatorHP [57], Nacpa [58], or CoVeriTeam's parallel portfolio [4] run different approaches in parallel but in isolation. In contrast, some approaches [59], [3], [60], [61], [62] that run multiple SAT or SMT based verification instances in parallel may share information, e.g., perform lemma sharing. Finally, swarm verification [2], [63] and parallel randomized state-space search [1] aim to diversify the search through the state space, e.g., by letting various instances use different search orders.

While it is not straightforward to combine our approach with other algorithmic parallelization techniques, splitting the verification task or running different verifier instances on a verification task is orthogonal to our parallelization approach and can easily be combined with our approach.

VI. Conclusion

Trace abstraction is an automata-based software verification technique, which checks safety properties of software. Its original verification algorithm implemented in Ultimate Automizer has not been designed to work in parallel. To let trace abstraction benefit from modern compute hardware, in particular multi-threading, we therefore propose a parallelization approach for trace abstraction. Our approach parallelizes the abstraction refinement, i.e., our approach analyzes multiple, distinct syntactic paths, which lead to an error location, in parallel. Thereby, each analysis checks the feasibility of the respective path and in case of infeasibility computes an interpolant automaton to refine the trace abstraction. A coordinator uses a specialized search procedure to find diverse paths that lead to an error location, distributes the found paths to available workers, and applies the refinement to the global trace abstraction. Since our parallelization is orthogonal to parallel portfolio approaches, which run multiple verifiers in parallel, or parallelization approaches that split the verification task, we can freely combine it with those approaches.

We evaluated the implementation of our parallel trace abstraction in Ultimate Automizer on a large set of C verification tasks from the SVBenchmark. Thereby, we observe that parallelization increases the effectiveness, i.e., typically, we solve more tasks with more workers. Nevertheless, parallelization will only make sense if the number of traces analyzed by sequential trace abstraction is at least as large as the number of workers. Furthermore, we observe that in particular for longer running tasks parallelization may reduce the wall time significantly. For several tasks, the speedup for the wall time is significantly larger than the upper bound given by Amdahl's law for task parallelization, i.e., parallel trace abstraction may allow us to skip some of the verification steps performed by sequential trace abstraction. The value of parallel trace abstraction is further confirmed by its success in SV-COMP 2026 [17].

Data-Availability Statement. All evaluation data and a reproduction package of our experiments is available on Zenodo [64].

Acknowledgment. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 496852682.

References

[1] M. B. Dwyer, S. G. Elbaum, S. Person, and R. Purandare, "Parallel randomized state-space search," in *Proc. ICSE*. IEEE, 2007, pp. 3–12. [Online]. Available: https://doi.org/10.1109/ICSE.2007.62

[2] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification," in *Proc. ASE*. IEEE, 2008, pp. 1–6. [Online]. Available: https://doi.org/10.1109/ASE.2008.9

[3] S. Chaki and D. Karimi, "Model checking with multi-threaded IC3 portfolios," in *Proc. VMCAI*, ser. LNCS 9583. Springer, 2016, pp. 517–535. [Online]. Available: https://doi.org/10.1007/978-3-662-49122-5_25

[4] D. Beyer, S. Kanav, and C. Richter, "Construction of verifier combinations based on off-the-shelf verifiers," in *Proc. FASE*, ser. LNCS 13241. Springer, 2022, pp. 49–70. [Online]. Available: https://doi.org/10.1007/978-3-030-99429-7_3

[5] D. Beyer and K. Friedberger, "Domain-independent multi-threaded software model checking," in *Proc. ASE*. ACM, 2018, pp. 634–644. [Online]. Available: https://doi.org/10.1145/3238147.3238195

[6] J. Haltermann, M.-C. Jakobs, C. Richter, and H. Wehrheim, "Parallel program analysis via range splitting," in *Proc. FASE*, ser. LNCS 13991. Springer, 2023, pp. 195–219. [Online]. Available: https://doi.org/10.1007/978-3-031-30826-0_11

[7] D. Beyer, M. Kettl, and T. Lemberger, "Decomposing software verification using distributed summary synthesis," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1307–1329, 2024. [Online]. Available: https://doi.org/10.1145/3660766

[8] H. Garavel, R. Mateescu, and I. M. Smarandache, "Parallel state space construction for model-checking," in *Proc. SPIN*, ser. LNCS 2057. Springer, 2001, pp. 217–234. [Online]. Available: https://doi.org/10.1007/3-540-45139-0_14

[9] S. Evangelista, L. M. Kristensen, and L. Petrucci, "Multi-threaded explicit state space exploration with state reconstruction," in *Proc. ATVA*, ser. LNCS 8172. Springer, 2013, pp. 208–223. [Online]. Available: https://doi.org/10.1007/978-3-319-02444-8_16

[10] N. P. Lopes and A. Rybalchenko, "Distributed and predictable software model checking," in *Proc. VMCAI*, ser. LNCS 6538. Springer, 2011, pp. 340–355. [Online]. Available: https://doi.org/10.1007/978-3-642-18275-4_24

[11] J. Rodriguez and O. Lhoták, "Actor-based parallel dataflow analysis," in *Proc. CC*, ser. LNCS 6601. Springer, 2011, pp. 179–197. [Online]. Available: https://doi.org/10.1007/978-3-642-19861-8_11

[12] Z. Sun, D. Xu, Y. Zhang, Y. Qi, Y. Wang, Z. Zuo, Z. Wang, Y. Li, X. Li, Q. Lu, W. Peng, and S. Guo, "BigDataflow: A distributed interprocedural dataflow analysis framework," in *Proc. FSE*. ACM, 2023, pp. 1431–1443. [Online]. Available: https://doi.org/10.1145/3611643.3616348

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Proc. CAV*, ser. LNCS 1855. Springer, 2000, pp. 154–169. [Online]. Available: https://doi.org/10.1007/10722167_15

[14] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *Proc. SAS*, ser. LNCS 5673. Springer, 2009, pp. 69–85. [Online]. Available: https://doi.org/10.1007/978-3-642-03237-0_7

[15] ——, "Software model checking for people who love automata," in *Proc. CAV*, ser. LNCS 8044. Springer, 2013, pp. 36–52. [Online]. Available: https://doi.org/10.1007/978-3-642-39799-8_2

[16] M. Barth, D. Dietsch, M. Heizmann, and M.-C. Jakobs, "Ultimate paralizer: Parallel trace abstraction (competition contribution)," in *Proc. TACAS*, ser. LNCS 16506. Springer, 2026, pp. 595–599. [Online]. Available: https://doi.org/10.1007/978-3-032-22749-2_38

[17] D. Beyer and J. Strejcek, "Evaluating software verifiers for C, Java, and SV-LIB - (report on SV-COMP 2026)," in *Proc. TACAS*, ser. LNCS 16506. Springer, 2026, pp. 461–502. [Online]. Available: https://doi.org/10.1007/978-3-032-22749-2_23

[18] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer, "Temporal-safety proofs for systems code," in *Proc. CAV*, ser. LNCS 2404. Springer, 2002, pp. 526–538. [Online]. Available: https://doi.org/10.1007/3-540-45657-0_45

[19] F. Schüssele, M. Bentele, D. Dietsch, M. Heizmann, X. Jiang, D. Klumpp, and A. Podelski, "Ultimate Automizer and the abstraction of bitwise operations - (competition contribution)," in *Proc. TACAS*, ser. LNCS 14572. Springer, 2024, pp. 418–423. [Online]. Available: https://doi.org/10.1007/978-3-031-57256-2_31

[20] D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski, "Craig vs. Newton in software model checking," in *Proc. FSE*. ACM, 2017, pp. 487–497. [Online]. Available: https://doi.org/10.1145/3106237.3106307

[21] M. Heizmann, Y. Chen, D. Dietsch, M. Greitschus, J. Hoenicke, Y. Li, A. Nutz, B. Musa, C. Schilling, T. Schindler, and A. Podelski, "Ultimate Automizer and the search for perfect interpolants - (competition contribution)," in *Proc. TACAS*, ser. LNCS 10806. Springer, 2018, pp. 447–451. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_30

[22] D. Beyer, M. Kettl, and T. Lemberger, "Reproduction package for FSE 2024 article 'Decomposing software verification using distributed summary synthesis' (version FSE24-proceedings)," 2024. [Online]. Available: https://doi.org/10.5281/zenodo.11563223

[23] D. Beyer and J. Strejcek, "Improvements in software verification and witness validation: SV-COMP 2025," in *Proc. TACAS*, ser. LNCS 15698. Springer, 2025, pp. 151–186. [Online]. Available: https://doi.org/10.1007/978-3-031-90660-2_9

[24] D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: Requirements and solutions," *STTT*, vol. 21, no. 1, pp. 1–29, 2019. [Online]. Available: https://doi.org/10.1007/s10009-017-0469-y

[25] L. Yin, W. Dong, W. Liu, and J. Wang, "Parallel refinement for multi-threaded program verification," in *Proc. ICSE*. IEEE, 2019, pp. 643–653. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00074

[26] G. J. Holzmann, "A stack-slicing algorithm for multi-core model checking," in *Proc. PDMC@CAV*, ser. Electronic Notes in Theoretical Computer Science, vol. 198, no. 1. Elsevier, 2007, pp. 3–16. [Online]. Available: https://doi.org/10.1016/j.entcs.2007.10.017

[27] S. Edelkamp and D. Sulewski, "External memory breadth-first search with delayed duplicate detection on the GPU," in *Proc. MoChArt*, ser. LNCS 6572. Springer, 2010, pp. 12–31. [Online]. Available: https://doi.org/10.1007/978-3-642-20674-0_2

[28] E. Bartocci, R. DeFrancisco, and S. A. Smolka, "Towards a GPGPU-parallel SPIN model checker," in *Proc. SPIN*. ACM, 2014, pp. 87–96. [Online]. Available: https://doi.org/10.1145/2632362.2632379

[29] A. Wijs and D. Bosnacki, "GPUexplore: Many-core on-the-fly state space exploration using GPUs," in *Proc. TACAS*, ser. LNCS 8413. Springer, 2014, pp. 233–247. [Online]. Available: https://doi.org/10.1007/978-3-642-54862-8_16

[30] K. Dewey, V. Kashyap, and B. Hardekopf, "A parallel abstract interpreter for JavaScript," in *Proc. CGO*. IEEE, 2015, pp. 34–45. [Online]. Available: https://doi.org/10.1109/CGO.2015.7054185

[31] S. K. Kim, A. J. Venet, and A. V. Thakur, "Deterministic parallel fixpoint computation," *Proc. ACM Program. Lang.*, vol. 4, no. POPL, pp. 14:1–14:33, 2020. [Online]. Available: https://doi.org/10.1145/3371082

[32] L. Brim, I. Cerná, P. Krcál, and R. Pelánek, "Distributed LTL model checking based on negative cycle detection," in *Proc. FST TCS*, ser. LNCS 2245. Springer, 2001, pp. 96–107. [Online]. Available: https://doi.org/10.1007/3-540-45294-X_9

[33] J. Barnat and I. Cerná, "Distributed breadth-first search LTL model checking," *Formal Methods Syst. Des.*, vol. 29, no. 2, pp. 117–134, 2006. [Online]. Available: https://doi.org/10.1007/s10703-006-0009-y

[34] S. Evangelista, L. Petrucci, and S. Youcef, "Parallel nested depth-first searches for LTL model checking," in *Proc. ATVA*, ser. LNCS 6996. Springer, 2011, pp. 381–396. [Online]. Available: https://doi.org/10.1007/978-3-642-24372-1_27

[35] T. Kahsai and C. Tinelli, "PKind: A parallel k-induction based model checker," in *Proc. PDMC*, ser. EPTCS, vol. 72, 2011, pp. 55–62. [Online]. Available: https://doi.org/10.4204/EPTCS.72.6

[36] D. Beyer, M. Dangl, and P. Wendler, "Boosting k-induction with continuously-refined invariants," in *Proc. CAV*, ser. LNCS 9206. Springer, 2015, pp. 622–640. [Online]. Available: https://doi.org/10.1007/978-3-319-21690-4_42

[37] T. van Dijk and J. van de Pol, "Multi-core decision diagrams," in *Handbook of Parallel Constraint Reasoning*, Y. Hamadi and L. Sais, Eds. Springer, 2018, pp. 509–545. [Online]. Available: https://doi.org/10.1007/978-3-319-63516-3_13

[38] A. E. J. Hyvärinen and C. M. Wintersteiger, "Parallel satisfiability modulo theories," in *Handbook of Parallel Constraint Reasoning*, Y. Hamadi and L. Sais, Eds. Springer, 2018, pp. 141–178. [Online]. Available: https://doi.org/10.1007/978-3-319-63516-3_5

[39] E. Sherman and M. B. Dwyer, "Structurally defined conditional data-flow static analysis," in *Proc. TACAS*, ser. LNCS 10806. Springer, 2018, pp. 249–265. [Online]. Available: https://doi.org/10.1007/978-3-319-89963-3_15

[40] M. K. Ganai and A. Gupta, "Tunneling and slicing: Towards scalable BMC," in *Proc. DAC*. ACM, 2008, pp. 137–142. [Online]. Available: https://doi.org/10.1145/1391469.1391507

[41] T. L. Nguyen, P. Schrammel, B. Fischer, S. La Torre, and G. Parlato, "Parallel bug-finding in concurrent programs via reduced interleaving instances," in *Proc. ASE*. IEEE, 2017, pp. 753–764. [Online]. Available: https://doi.org/10.1109/ASE.2017.8115686

[42] O. Inverso and C. Trubiani, "Parallel and distributed bounded model checking of multi-threaded programs," in *Proc. PPoPP*. ACM, 2020, pp. 202–216. [Online]. Available: https://doi.org/10.1145/3332466.3374529

[43] J. Haltermann, M.-C. Jakobs, C. Richter, and H. Wehrheim, "Parallel program analysis on path ranges," *Science of Computer Programming*, vol. 238, p. 103154, 2024. [Online]. Available: https://doi.org/10.1016/j.scico.2024.103154

[44] ——, "Ranged program analysis via instrumentation," in *Proc. SEFM*, ser. LNCS 14323. Springer, 2023, pp. 145–164. [Online]. Available: https://doi.org/10.1007/978-3-031-47115-5_9

[45] C. Richter, M. Chalupa, M.-C. Jakobs, and H. Wehrheim, "Cooperative software verification via dynamic program splitting," in *Proc. ICSE*. IEEE, 2025, pp. 2087–2099. [Online]. Available: https://doi.org/10.1109/ICSE55347.2025.00092

[46] J. H. Siddiqui and S. Khurshid, "Scaling symbolic execution using ranged analysis," in *Proc. SPLASH*. ACM, 2012, pp. 523–536. [Online]. Available: https://doi.org/10.1145/2384616.2384654

[47] M. Staats and C. S. Pasareanu, "Parallel symbolic execution for structural test generation," in *Proc. ISSTA*. ACM, 2010, pp. 183–194. [Online]. Available: https://doi.org/10.1145/1831708.1831732

[48] R. Qiu, S. Khurshid, C. S. Pasareanu, J. Wen, and G. Yang, "Using test ranges to improve symbolic execution," in *Proc. NFM*, ser. LNCS 10811. Springer, 2018, pp. 416–434. [Online]. Available: https://doi.org/10.1007/978-3-319-77935-5_28

[49] Q. Shi and C. Zhang, "Pipelining bottom-up data flow analysis," in *Proc. ICSE*. ACM, 2020, pp. 835–847. [Online]. Available: https://doi.org/10.1145/3377811.3380425

[50] Q. Stiévenart, N. V. Es, J. V. der Plas, and C. D. Roover, "A parallel worklist algorithm and its exploration heuristics for static modular analyses," *J. Syst. Softw.*, vol. 181, p. 111042, 2021. [Online]. Available: https://doi.org/10.1016/j.jss.2021.111042

[51] A. Albarghouthi, R. Kumar, A. V. Nori, and S. K. Rajamani, "Parallelizing top-down interprocedural analyses," in *Proc. PLDI*. ACM, 2012, pp. 217–228. [Online]. Available: https://doi.org/10.1145/2254064.2254091

[52] S. McPeak, C. Gros, and M. K. Ramanathan, "Scalable and incremental software bug detection," in *Proc. FSE*. ACM, 2013, pp. 554–564. [Online]. Available: https://doi.org/10.1145/2491411.2501854

[53] M. Christakis, T. Cottenier, A. Filieri, L. Luo, M. N. Mansur, L. Pike, N. Rosner, M. Schäf, A. Sengupta, and W. Visser, "Input splitting for cloud-based static application security testing platforms," in *Proc. FSE*. ACM, 2022, pp. 1367–1378. [Online]. Available: https://doi.org/10.1145/3540250.3558944

[54] G. Yang, Q. C. D. Do, and J. Wen, "Distributed assertion checking using symbolic execution," *ACM SIGSOFT Softw. Eng. Notes*, vol. 40, no. 6, pp. 1–5, 2015. [Online]. Available: https://doi.org/10.1145/2830719.2830729

[55] R. Kumar, T. Ball, J. Lichtenberg, N. Deisinger, A. Upreti, and C. Bansal, "CloudSDV enabling static driver verifier using Microsoft Azure," in *Proc. IFM*, ser. LNCS 9681. Springer, 2016, pp. 523–536. [Online]. Available: https://doi.org/10.1007/978-3-319-33693-0_33

[56] M. Marescotti, A. Gurfinkel, A. E. J. Hyvärinen, and N. Sharygina, "Designing parallel PDR," in *Proc. FMCAD*. IEEE, 2017, pp. 156–163. [Online]. Available: https://doi.org/10.23919/FMCAD.2017.8102254

[57] P. Peringer, V. Soková, and T. Vojnar, "PredatorHP revamped (not only) for interval-sized memory regions and memory reallocation (competition contribution)," in *Proc. TACAS*, ser. LNCS 12079. Springer, 2020, pp. 408–412. [Online]. Available: https://doi.org/10.1007/978-3-030-45237-7_30

[58] T. Lemberger and H. Wachowitz, "Nacpa: Native checking with parallel-portfolio analyses - (competition contribution)," in *Proc. TACAS*, ser. LNCS 15698. Springer, 2025, pp. 236–241. [Online]. Available: https://doi.org/10.1007/978-3-031-90660-2_18

[59] E. Ábrahám, T. Schubert, B. Becker, M. Fränzle, and C. Herde, "Parallel SAT solving in bounded model checking," in *Proc. FMICS*, ser. LNCS 4346. Springer, 2006, pp. 301–315. [Online]. Available: https://doi.org/10.1007/978-3-540-70952-7_21

[60] A. Champion, A. Mebsout, C. Sticksel, and C. Tinelli, "The Kind 2 model checker," in *Proc. CAV*, ser. LNCS 9780. Springer, 2016, pp. 510–517. [Online]. Available: https://doi.org/10.1007/978-3-319-41540-6_29

[61] A. Gacek, J. Backes, M. Whalen, L. G. Wagner, and E. Ghassabani, "The JKind model checker," in *Proc. CAV*, ser. LNCS 10982. Springer, 2018, pp. 20–27. [Online]. Available: https://doi.org/10.1007/978-3-319-96142-2_3

[62] M. Blicha, A. E. J. Hyvärinen, M. Marescotti, and N. Sharygina, "A cooperative parallelization approach for property-directed k-induction," in *Proc. VMCAI*, ser. LNCS 11990. Springer, 2020, pp. 270–292. [Online]. Available: https://doi.org/10.1007/978-3-030-39322-9_13

[63] G. J. Holzmann, R. Joshi, and A. Groce, "Swarm verification techniques," *IEEE Trans. Software Eng.*, vol. 37, no. 6, pp. 845–857, 2011. [Online]. Available: https://doi.org/10.1109/TSE.2010.110

[64] M. Barth and M.-C. Jakobs, "Artifact for "Multi-threaded software model checking via parallel trace abstraction refinement"," 2026. [Online]. Available: https://doi.org/10.5281/zenodo.19793493