CPA-DF: A Tool for Configurable Interval Analysis to Boost Program Verification

Dirk Beyer
LMU Munich, Germany

Po-Chun Chien
LMU Munich, Germany

Nian-Ze Lee
LMU Munich, Germany

Abstract—Software verification is challenging, and auxiliary program invariants are used to improve the effectiveness of verification approaches. For instance, the $k$-induction implementation in CPA-DF, an award-winning framework for program analysis, uses invariants produced by a configurable data-flow analysis to strengthen induction hypotheses. This invariant generator, CPA-DF, uses arithmetic expressions over intervals as its abstract domain and is able to prove some safe verification tasks alone. After extensively evaluating CPA-DF over SV-Benchmarks, the largest publicly available suite of C safety-verification tasks, we discover that its potential as a stand-alone analysis or a sub-analysis in a parallel portfolio for combined verification approaches has been significantly underestimated: (1) as a stand-alone analysis, CPA-DF finds almost as many proofs as the plain $k$-induction implementation without auxiliary invariants; (2) as a sub-analysis running in parallel to the plain $k$-induction implementation, CPA-DF boosts the portfolio verifier to solve a comparable amount of tasks as the heavily-optimized $k$-induction implementation with invariant injection. Our detailed analysis reveals that dynamic precision adjustment is crucial to the efficiency and effectiveness of CPA-DF. To generalize our results beyond CPA-DF, we use CoVeriTeam, a platform for cooperative verification, to compose three portfolio verifiers executing CPA-DF and one of three other state-of-the-art software verifiers in parallel. Surprisingly, running CPA-DF merely in parallel to these established tools can further boost the number of correct results up to more than 20%.

Demonstration video: https://youtu.be/l7UG-vhTL_4

I. INTRODUCTION

As the technology evolves, software computational systems are becoming increasingly complicated. Assuring the safety of these systems has always been a crucial research field, and numerous verification tools and algorithms have been proposed. To keep up with the development pace of software systems, cooperative verification [1] aims at combining the strengths of different tools to form a more powerful verifier. One prominent example is to augment $k$-induction with an invariant generator. Auxiliary invariants prune out unreachable program states and can strengthen the induction hypotheses, greatly improving the proof-finding capability of $k$-induction [2,3,4].

CPA-DF, an award-winning software analyzer, is one of the many tools implementing the cooperative $k$-induction method [3]. It uses a configurable data-flow analysis CPA-DF based on expressions over intervals to generate invariants. While profiling the performance of $k$-induction, we observe that CPA-DF is able to construct a proof, an inductive invariant that implies the safety property, on its own for many verification tasks. A question naturally arises: How much of the performance boost by combining $k$-induction with CPA-DF actually comes from the parallel execution, but not from the invariant injection? Surprisingly, we find out that by running the two analyses purely in parallel, we can solve almost as many tasks as running the two cooperatively (with invariant injection). In fact, nearly 90% of the boosting effect of this cooperative approach in CPA-DF is achieved by the parallel execution. That is, without the need of a more complicated mechanism to communicate invariants, a simple portfolio is already very effective. This motivates us to further investigate (1) the capability of CPA-DF and (2) whether this boosting effect can be generalized to other tools. The main engineering challenge was to decouple the integrated invariant-generation component CPA-DF from $k$-induction and use it modularly with other components or tools. Thanks to the high flexibility and adaptivity of the CPA framework [5] and CoVeriTeam [6], we were able to overcome the challenge elegantly.

Use Cases. In this paper, we demonstrate two major use cases of CPA-DF: (1) as a stand-alone analyzer, and (2) as a complementary performance booster in a portfolio-based verifier. We envision that CPA-DF could be beneficial to both verification-tool users and developers.

Novelty and Contributions. We discover a novel aspect of the existing invariant-generation component of CPA-DF and decouple it into a stand-alone static analyzer CPA-DF. The presented analyzer is highly configurable and can find a comparable amount of proofs as plain $k$-induction. To demonstrate the usability of the tool, we conducted a large-scale evaluation on more than 9000 verification tasks. We show that in CPA-DF, running CPA-DF as a parallel component along with $k$-induction achieves a performance boost similar to that of a complicated cooperative approach. This boosting effect can also be observed by pairing CPA-DF with another well-established tool as a portfolio-based verifier. The finding implies that a parallel portfolio is a simple yet very effective way to improve performance, almost as good as complex combinations that usually require huge engineering effort.

Tool Availability. We contribute the open-source data-flow analysis tool CPA-DF (see Section "Data-Availability Statement"), which can be used as a component to boost the performance of other verifiers.

II. RELATED WORK

Our work is mainly related to data-flow analysis and cooperative verification.
Data-Flow Analysis. Data-flow analysis computes the information flow throughout the control-flow graph of a program. It is used by compilers for tracking the reaching definitions and performing constant propagation [7]. Depending on the used abstract domain, it can analyze different aspects of a program. Domains of explicit values [7] and intervals [8, 9], are commonly used to overapproximate numerical values of program variables. Compared to model checking, data-flow analysis is usually less precise but more efficient [10].

Cooperative Verification. Cooperative verification approaches [1] combine analyses with different strengths. Using auxiliary invariants to confine the state space of the main analysis has been studied for many algorithms, including $k$-induction [2, 3, 4], predicate abstraction [11, 12], and IC3/PDR [13]. In contrast to cooperative approaches demanding tight integration between analyses, verifiers can also be combined easily as a sequential or parallel portfolio.

III. CONFIGURABLE INTERVAL ANALYSIS

Configurable Program Analysis with Precision. A configurable program analysis with dynamic precision adjustment (CPA+) [10] specifies an abstract domain and a set of precisions used to inspect a program. The precision is adjustable to make the analysis either efficient but coarse or precise but time-consuming. A CPA+ $\mathbb{I}$ with an initial abstract state $e_0$ and a precision $\pi$ is given to the CPA+ algorithm (Algorithm 2.1 in the referred publication [10]) to construct a set of reachable abstract states under precision $\pi$.

A CPA+ with Arithmetic Expressions over Intervals. CPA-DF relies on a CPA+ using arithmetic expressions over intervals as its abstract domain. This interval CPA+ $\mathbb{I}$ was first used as a generator of auxiliary invariants to boost the performance of $k$-induction [3]. The complete description of $\mathbb{I}$ can be found in the technical report [14]. We briefly summarize the major features of $\mathbb{I}$ below.

In the abstract domain of $\mathbb{I}$, every program variable is mapped to an arithmetic expression over intervals, e.g., $[l_1, u_1] \cup [l_2, u_2]$, where $l_i$ and $u_i$ are numerical values representing the lower and upper bounds of an interval, respectively. Compared to a plain interval analysis that only tracks a single interval for each variable, $\mathbb{I}$ is able to represent complex ranges of variables and hence more precise. A precision $\pi$ for $\mathbb{I}$ includes a set of important variables and a Boolean flag toggling whether widening [9] is applied or not. Two abstract states are merged only if their interval expressions match over all important variables, and the merged state will take the union of the interval expressions of the two abstract states. The widening operation further relaxes the abstraction of a merged state by computing the upper and lower bounds and assigning a single interval to each variable.

Dynamic Precision Adjustment. CPA-DF combines the interval CPA+ $\mathbb{I}$ with a precision-refinement strategy to achieve an efficient and effective data-flow analysis. The procedure is described in Alg. 1. It takes as input the interval CPA+ $\mathbb{I}$, an initial abstract state $e_0$, an initial precision $\pi_0$, and a safety property $P$.

**Algorithm 1 Interval analysis with precision refinement**

**Input:** the interval CPA+ $\mathbb{I}$, an initial abstract state $e_0$, an initial precision $\pi_0$, and a safety property $P$

**Output:** safe if $P$ holds for all reachable abstract states, or unknown if the analysis is inconclusive

1: $\pi := \pi_0$
2: do
3: $R := \text{CPA+}(\mathbb{I}, e_0, \pi)$
4: if $\forall s \in R : s \models P$ then
5: return safe
6: $\pi := \text{RefinePrecision}(\mathbb{I}, \pi)$
7: while $\pi \neq \text{nil}$
8: return unknown

IV. EXPERIMENTAL EVALUATION

We demonstrate the two major use cases of CPA-DF presented in Sect. I by investigating the research questions below:

- **RQ1:** Can dynamic precision adjustment improve CPA-DF as a stand-alone analysis for finding proofs?
- **RQ2:** Can a parallel portfolio of CPA-DF and the plain $k$-induction implementation compete with the optimized implementation with invariant injection in CPAchecker?
- **RQ3:** Can CPA-DF complement other state-of-the-art software verifiers as a sub-analysis in a parallel portfolio?

**Benchmark Set.** To answer the research questions, we used the verification tasks from the 2023 Intl. Competition on Software Verification (SV-COMP’23). We selected only verification tasks whose safety property is the reachability of calls to an error function. The overall benchmark set consists of a total of 9537 verification tasks, where 3153 tasks contain a known specification violation, and the remaining 6386 tasks are considered safe.

**Experimental Setup.** All experiments were conducted on machines with a 3.4 GHz CPU (Intel Xeon E3-1230 v5, 8 processing units) and 33 GB of RAM. The operating system was

---

1In the implementation, a composite CPA of $\mathbb{I}$ and other supportive CPAs tracking program locations, pointers, and call stacks is used.
will need to process more abstract states and thus might not prevent abstract states from being merged: The analysis [18], whose archives can be automatically downloaded CPA-DF, the program under analysis. In this way, CPA-DF the highest (0 for the tasks solvable with both precisions. To automatically (16 although CPA-DF found fewer proofs than CPA-DF under both precisions, where precision (0 (16 CPA-DF used to find a proof under the precisions enabling widening (16 x, y) in the plot means that there is a task solvable by CPA-DF with x, y separated by both precisions, where precision (0, t) requires y seconds, and precision (16, t) requires x seconds. In this comparison, although CPA-DF with precision (0, t) found fewer proofs than precision (16, t) (1137 vs. 1583), it often found proofs faster for the tasks solvable with both precisions. To automatically search for a suitable precision for a verification task, CPA-DF dynamically refines the precision from the lowest one (0, t) to the highest (N, f), where N is the number of all variables in the program under analysis. In this way, CPA-DF is as precise as necessary and as efficient as possible. Overall, CPA-DF with dynamic precision adjustment solved more tasks than any other static precision in our experiments (column “DF” of Table I).

RQ2: Parallel Portfolio versus Invariant Injection. To compare the effect on k-induction between running CPA-DF as a sub-analysis in a parallel portfolio and as a generator for invariant injection, we evaluated four different analyses in CPA-CHECKER: the configurable interval analysis described in this paper (DF), the plain k-induction (KI) without auxiliary invariant, the parallel portfolio of KI and DF (KI||DF), and KI with induction hypotheses strengthened by auxiliary invariants injected from DF (KI+&nabla;DF) [3].

The experimental results are summarized in Table I. Additionally, Fig. 3 shows a quantile plot comparing the proof-finding capabilities of the assessed methods. A data point (x, y) in the plot indicates that x safe tasks are correctly proved by the respective algorithm within a CPU time bound of y seconds each. Observe that by combining DF with KI, either with communication (KI+&nabla;DF) or without (KI||DF), we can achieve a significant improvement over running a single analysis, meaning that the two analyses complement each other.

We also note that although KI||DF did not prove as many tasks as KI+&nabla;DF, it avoided the 15 wrong proofs2 caused by invariant injection. All of these wrong proofs came from tasks in the ReachSafety-Hardware category, where KI+&nabla;DF also found most additional proofs over KI||DF. In other words, the extra proofs found by KI+&nabla;DF in this category might be due to unsound overapproximation. If we exclude the tasks in this category, KI||DF and KI+&nabla;DF found 2551 and 2554 proofs in the remaining tasks, respectively. Our results show that a parallel portfolio of CPA-DF and k-induction already brings us most performance improvement that can be gained with invariant injection, which involves a complicated information exchange between analyses and is more error-prone.

RQ3: Boosting Program Verification with CPA-DF. To examine the boosting effect of CPA-DF on other tools, we paired ESBMC, SYMBIOTIC, and UAUTOMIZER from SV-COMP’23 with CPA-DF via COVERITeam to form three parallel portfolios, respectively. Table I contains the results of running the verifiers alone or in parallel with CPA-DF. A quantile plot comparing the performance of these tool combinations on the safe verification tasks is also shown in Fig. 4. At the expense of discovering fewer violations because of the CPU time spent by CPA-DF, it can boost the performance of the other tools by communicating their results.

TABLE I: Summary of the verification results on 9537 benchmark tasks

| Framework     | CPA-Checker | ESBMC E||DF | SYMBIOTIC S||DF | UAUTOMIZER U||DF |
|---------------|-------------|----------|-----------|----------------|----------------|
| Configuration | DF          | KI        | KI+&nabla;DF | KI||DF         | KI+&nabla;DF    | KI||DF         |
| Tasks         | 1719        | 10520     | 13153      | 3597           | 6537           | 7222           |
| Correct proofs| 3653        | 4636      | 1491       | 3578           | 5604           | 5554           |
| Wrong proofs  | 0           | 0         | 0          | 0              | 0              | 0              |
| Wrong alarms  | 2           | 10        | 1          | 1              | 0              | 0              |
| Unknowns      | 359         | 4075      | 4309       | 5604           | 5356           | 5554           |
| Solved by DF | 1719        | -         | -          | -              | -              | -              |

Fig. 1: Numbers of proofs found by CPA-DF with different precisions

64-bit Ubuntu 22.04, running Linux 5.15 and OpenJDK 17.0.5. Each verification task was limited to 4 processing units, 15 min of CPU time, and 15 GB of RAM. We used BENCHEXEC [15] for reliable benchmarking.

Revision 43678 on branch data-flow-exp-configs of CPA-CHECKER was used for the implementations of CPA-DF and k-induction. We used COVERITeam [6] at commit 838ee67ff to compose parallel portfolios. As representative state-of-the-art tools, we chose ESBMC [16], SYMBIOTIC [17], and UAUTOMIZER [18], whose archives can be automatically downloaded and installed by COVERITeam from the SV-COMP’23 artifacts on Zenodo [19]. The time limit of CPA-DF was set to 5 min in parallel portfolios to allocate enough time to the main analysis.

RQ1: Dynamic Precision Adjustment in CPA-DF. To study the effect of dynamic precision adjustment on the proof-finding capability of CPA-DF, we executed it with fixed precisions limiting the number of important variables to {0, 1, 4, 8, 16} and switched the widening operation on and off. Figure 1 shows the number of safe tasks that CPA-DF proved under each static precision, and the results of CPA-DF with dynamic precision adjustment are shown in column “DF” of Table I.

In Fig. 1, we observe that more proofs were found as the number of important variables increases. However, marking too many variables as important is suboptimal because this will prevent abstract states from being merged: The analysis will need to process more abstract states and thus might not finish timely. For the widening operation, we also observe a similar phenomenon: without the overapproximation performed by widening, the analysis found fewer proofs.

The scatter plot in Fig. 2 compares the CPU time CPA-DF used to find a proof under the precisions enabling widening and allowing 0 and 16 important variables, respectively. The two precisions are denoted as (0, t) and (16, t). A data point (x, y) in the plot means that there is a task solvable by CPA-DF under both precisions, where precision (0, t) requires y seconds, and precision (16, t) requires x seconds. In this comparison, although CPA-DF with precision (0, t) found fewer proofs than precision (16, t) (1137 vs. 1583), it often found proofs faster for the tasks solvable with both precisions. To automatically search for a suitable precision for a verification task, CPA-DF dynamically refines the precision from the lowest one (0, t) to the highest (N, f), where N is the number of all variables in the program under analysis. In this way, CPA-DF is as precise as necessary and as efficient as possible. Overall, CPA-DF with dynamic precision adjustment solved more tasks than any other static precision in our experiments (column “DF” of Table I).

2The issue (https://gitlab.com/sosy-lab/software/cpachecker/-/issues/1070) has been discovered by the community and is under investigation.
all three verifiers were able to find more proofs via having CPA-DF as a parallel component\(^3\). The extent of improvement depends on how much the sets of solvable tasks by the sub-analyses in the portfolio overlap. For example, there were 1530 tasks solvable by both UA\(\text{UTOMIZER}\) and CPA-DF, which already accounts for 89% of the tasks the latter could solve. Such a large overlap explains why UA\(\text{UTOMIZER}\)\(\|\)DF only had a marginal improvement over UA\(\text{UTOMIZER}\). However, 1282 (43%) proofs were found by CPA-DF in the portfolio, meaning that CPA-DF was able to solve these tasks faster than UA\(\text{UTOMIZER}\). By contrast, SYMBIOTIC benefited the most from running CPA-DF in parallel because the solvable sets of the two sub-analyses are more disjoint (only 767 tasks were solvable by both). From Fig. 4, we can also observe a more significant improvement on SYMBIOTIC than the other two verifiers.

### V. CONCLUSION

We presented CPA-DF, a data-flow analyzer based on interval expressions, originally developed as an auxiliary invariant generator integrated with the \(k\)-induction implementation in CPA\(\text{CHECKER}\). Decoupling CPA-DF as a stand-alone tool, we are able to execute it modularly with other tools. In our evaluation, we observed that executing CPA-DF in parallel to a main analysis is not only helpful for \(k\)-induction in CPA\(\text{CHECKER}\), which achieved a comparable performance as invariant injection, but also enhances other state-of-the-art verifiers. We envision verifier developers and users trying out CPA-DF to boost the performance of their tools. For future work, we plan to export invariants derived by CPA-DF in an exchangeable format to broaden its application.

**Data-Availability Statement.** The tool CPA-DF is implemented in the open-source framework CPA\(\text{CHECKER}\). The source code is available at https://svn.sosy-lab.org/software/cpachecker/branches/data-flow-exp-configs@43678, and a precompiled executable can be found in the supplementary package [20].

**REFERENCES**

\(^3\) Even though UA\(\text{UTOMIZER}\) was able to solve more tasks than UA\(\text{UTOMIZER}\)\(\|\)DF overall, if we apply the scoring scheme in SV-COMP, which assigns two (resp. one) points to each correct proof (resp. alarm), UA\(\text{UTOMIZER}\)\(\|\)DF obtains a higher score than UA\(\text{UTOMIZER}\).


