Hierarchical analysis of loops with relaxed abstract transformers

Banghu Yin[1], Liqian Chen[1], Jiangchao Liu[1], Ji Wang[1,2]
[1]School of Computer Science, National University of Defense Technology, Changsha, China
[1,2]State Key Laboratory of High Performance Computing, National University of Defense Technology, Changsha, China

Numerical computation is often involved in software of embedded control systems, cyber-physical systems, artificial neural network systems, big data processing systems, etc. Automatically discovering numerical loop invariants is fundamental for checking the safety of such software. Abstract interpretation provides a framework to automatically discover sound invariants but which may be not precise enough due to over-approximations. One major source of precision loss is due to the limited linear expressiveness of most widely-used numerical abstract domains and the widening operation. This becomes more serious when analyzing all variables simultaneously as a whole for programs that involve non-linear behaviors. Based on the observation that the dependency among variables in a loop can be hierarchical, in this paper, we propose a hierarchical static analysis to analyze a loop by utilizing relaxed abstract transformers. The main idea is to first partition all variables involved in a loop into different hierarchical layers, then compute invariants over the variables layer by layer in a bottom-up manner. During the iterative process, the computed invariants over lower-layer variables are then used to relax transfer functions when analyzing the higher-layer variables. One benefit of our method lies in that it can generate linear invariants to soundly enclose non-linear behaviors in a loop. Finally, we present encouraging experimental results on benchmark programs involving non-linear behaviors.

***Index Terms*—Static analysis, abstract interpretation, loop invariant, hierarchical analysis, relaxing operator, abstract transformer**

I. Introduction

SOFTWARE in traditional embedded control systems or nowadays cyber-physical systems, artificial neural network systems, big data processing systems, often involves numerical computation in their source code, due to their dependence on the underlying mathematical, physical, or computational model. Moreover, many kinds of such numerical software play an important role in safety-critical fields, which suggests the importance of their dependability. Hence, there is a need for numerical static analysis techniques and tools to discover numerical invariants automatically. Since the strongest invariants are not computable in general, we need to conduct sound over-approximations. Abstract interpretation [1] is a theory of sound approximation of the semantics of computer programs, and provides a general framework for static analysis.

Static analysis based on abstract interpretation essentially over-approximates the concrete semantics of programs by abstract semantics, and then computes a super-set of all reachable program states at every program point. It is a popular techniques to check for run-time errors in programs. However, it may also raise false alarms, due to the underlying over-approximation. To reduce the rate of false alarms, one immediate idea is improving the precision of the analysis, which remains challenging.

The main causes of the precision loss of abstract interpretation based static analysis lie in two aspects: (1) Most widely used numerical abstract domains only have linear, convex expressiveness (such as the abstract domains of interval [1], octagon [2], polyhedra [3]) or limited non-convex expressiveness(such as the abstract domains of zonotope [4], congruence [5] and donut [6]). On contrast, the set of reachable program states are often non-convex or non-linear; (2) The widening operator on which abstract interpretation relies to ensure the convergence of fixpoint iteration for computing loop invariants, may bring severe precision loss because widening often aggressively weakens unstable predicates in each iteration. In particular, when using an abstract domain of linear constraints to analyze a program, one would easily get infinite boundaries for those variables whose values are essentially non-linear with respect to other variables.

Manuscript received 12-Jun-2018; revised 22-Oct-2018 and 8-Apr-2019. This work is supported by the National Key R&D Program of China (No. 2017YFB1001802), and the NSFC Program(Nos. 61872445, 61532007). Corresponding author: L. Chen (email: lqchen@nudt.edu.cn).

```
1 void main(){
2  int x = -10;
3  int y = 0;
4  while(x <= 20){
5    x = x + 1;
6    y = y + x; // y = y + x * x * x;
7  }
8 }
```

Fig. 1. A motivating example.

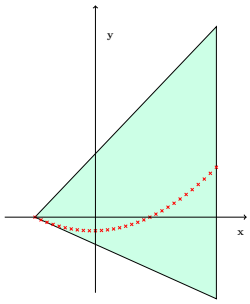

Fig. 2. Loop invariants of the motivating example.

Consider the program in Figure 1, which computes the sum of all integers from -9 to 21 through a loop and stores the result

into variable $\mathbf{y}$. At line 5, the most precise invariant would be: $\mathbf{y} = (\mathbf{x}-9)*(\mathbf{x}+10)/2 \wedge -10 \leq \mathbf{x} \leq 20$. All program states satisfying this invariant is shown as the red crosses in Figure 2. From this quadratic invariant, one can conclude that the value of $\mathbf{y}$ is always between -45 and 165 at line 5. Thus there is no integer overflow at line 6. However, a static analyzer based on linear constraint abstract domains (e.g., polyhedra abstract domain) may fail to infer any boundary information on variable $\mathbf{y}$ (i.e., it infers $\mathbf{y} \in [-\infty, +\infty]$). As a consequence, a false alarm of integer overflow would be raised at line 6. To infer non-linear invariants, several specific non-linear abstract domains [7] [8] have been proposed, some of which have been shown their usefulness in real-word applications [9]. However, such domains often rely on predefined templates or limit the degree of polynomials (e.g., quadratic) before conducting analysis. Limiting the degree of polynomials to be 2 would work for the case of the motivating example. However, if we modify the statement at line 6 to $\mathbf{y} = \mathbf{y} + \mathbf{x} * \mathbf{x} * \mathbf{x}$; these work would fail to infer new polynomial invariants with higher degree.

In this paper, we propose to use a hierarchical analysis approach to improve the precision of loop analysis, with the help of relaxed abstract transformers. More clearly, we first partition all variables involved in a loop into different hierarchical layers, and then compute loop invariants over variables layer by layer, from lower-layer to higher-layer. During the process of the whole analysis, the computed invariants over lower-layer variables (that we call "partial invariants"), are used to relax transfer functions when analyzing higher-layer variables. During the analysis, we make use of relaxing abstract transformers to reduce unstable ingredients, and to keep more stable ingredients. One benefit of our method lies in that it can generate linear invariants through conventional linear abstract domains to soundly enclose non-linear behaviors in a loop.

For the motivating example, our approach can generate the following linear loop invariants using the polyhedra abstract domain: $\{\mathbf{x} \leq 20 \ \wedge \ -\mathbf{x} \leq 10 \ \wedge \ -9\mathbf{x} - \mathbf{y} \leq 90 \ \wedge \ -21\mathbf{x} + \mathbf{y} \leq 210\}$. These constraints over-approximate all reachable program states in the loop with a reasonable precision (every variable is bounded). To illustrate this, we depict the state space represented by these constraints as the green triangle region in Figure 2. Before the statement at line 6 ($\mathbf{y} = \mathbf{y} + \mathbf{x}$;), our analysis can infer $\{\mathbf{x} \in [-9, 21] \ \wedge \ \mathbf{y} \in [-270, 630]\}$ following the above linear loop invariants. Thus it proves that there is no integer overflow at line 6.

The rest of the paper is organized as follows. Section II illustrates the main framework of our approach. Section III introduces hierarchical variable dependency graph and the corresponding program slicing techniques. Section IV presents relaxing techniques on transfer functions. Section V discusses soundness and precision of analysis via relaxing transformers. Section VI provides experimental results on benchmarks. Section VII discusses related work. Finally, conclusions as well as future work are given in Section VIII.

II. Overview

Standard abstract interpretation computes the invariants on all the variables in the loop simultaneously. This may cause much loss of precision, since widening is applied on abstract states that may involve many potentially unstable variables at the same time. Especially for programs with non-linear behaviors (e.g., the motivating example in Figure 1), static analysis using abstract domain with linear expressiveness could easily lose boundary information on variables that are non-linear with respective to other variables. However, it is common to find hierarchical dependency relations among variables in loop bodies. For example, the values of loop control variables (such as loop counters) usually do not depend on other variables in the loop. Other variables in the loop may depend on loop control variables directly or indirectly. In Figure 1, the value of variable $\mathbf{x}$ only depends on itself and that of $\mathbf{y}$ depends on $\mathbf{x}$. One idea is to compute the invariants on $\mathbf{x}$ and then $\mathbf{y}$, in order to reduce the unstable variables during widening. Widening is an important technique to ensure the convergency of the fix-point iteration procedure over a given abstract domain in abstract interpretation. Consider the following code segment {x=0; while(true) x=x+1;}. Suppose that we use abstract interpretation with the Interval domain to analyze this code. During the fix-point iteration procedure, after the loop iterates one (two) times, collected interval bounds at the loop head are x=[0,1] (x=[0,2]) respectively, which are unstable. It is easy to see that this iteration for this example, the fix-point iteration procedure will not terminate without widening operation. If we perform widening after the second iteration, the widening operation of the interval abstract domain keeps stable lower or upper bounds, and convert unstable upper (lower) bound to $+\infty$ ($-\infty$), and thus will results in x=[0,$+\infty$], which is in fact the invariant (stable bounds). In this paper, we use the standard widening operators defined in [1][3] for our analysis.

Inspired by the observation above, our analysis first constructs a hierarchical variable dependency graph on the variables involved in the loop, and then carries out an iterative refinement method to compute the loop invariants incrementally. In the hierarchical variable dependency graph, the variables that do not depend on others are put in the lowest layer, and the variables that need the same number of intermediate nodes to reach the variables of the lowest layer are put in the same layer. With this graph, our method carries out an iterative refinement analysis as follows: (1) Our analysis first computes the partial invariants on the lowest layer variables (those do not depend on other variables). To do that, the original loop in the program is sliced, so that only those statements involving the lowest layer variables are left. A regular abstract domain is then used to compute the invariants of the sliced loop; (2) Our method makes use of the already computed partial invariants on lower-layer variables to compute the invariants on higher-layer variables. The iterative process continues until reaching the highest layer. More clearly, when analyzing the i-th layer variables, the original program is sliced, so that only variables in layers lower than or equal to i are left. Our analysis utilizes computed partial invariants on variables in layers lower than i to to relax the expressions in the sliced program. After relaxing, there will be fewer unstable ingredients (e.g., variables, expressions, etc, where "unstable" means that their valuations change between two iterations), which potentially makes the widening more precise.

One key point of our method is designing a proper relaxing operator, which should provide relaxed transfer functions with more stable constraints before widening. In the loop body, some statements can contain unstable expressions (i.e., their valuations change between two iterations) that cannot be expressed precisely by the employed abstract domain (e.g., polyhedra abstract domain cannot express squared or higher-order constraints). These expressions are the main target to relax. Our analysis substitutes these target expressions with over-approximations derived from the computed partial loop invariants. The above process is called "semantic relaxing". It is worth mentioning that the partial loop invariants needed by "semantic relaxing" only involve lower-layer variables rather than all variables. In practice, they only need to constrain those lower-layer variables that appear in the target expression of relaxing.

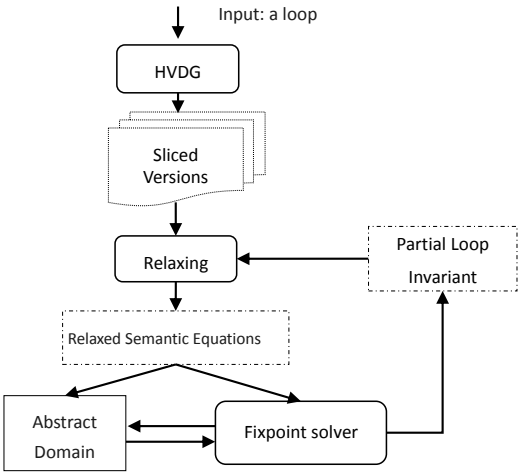

Fig. 3. The main framework of our method.

Our method improves the analysis precision of abstract interpretation by combining hierarchical analysis and abstract transformers relaxing. (1) Hierarchical analysis can help to generate more precise invariants for lower-layer variables. This observation can be indicated by our motivating example. If we take abstract interpreter (e.g., Interproc) with the polyhedra domain to analyze it as a whole, we can only get x=[-oo, 20] at the loop head. The lower bound of x is missed during the widening operation, since the constraints relating x and y are unstable. However, if we analyze the program without assignment "y=y+x" in loop body (i.e., the program in Figure 4(a)), we can get x=[-10,20] at loop head, which is more precise. (2) Relaxing abstract transformers can help to get more precise constraints between higher-layer and lower-layer variables. For our motivating example, if relaxing technique is not applied after first-pass analysis, although bounded range can be inferred for x, we still can not get finite bounds for y during the second-pass analysis on the program shown in Figure 4(b)).

Figure 3 shows the framework of our method. It mainly contains the following four steps: building a hierarchical variable dependency graph, program slicing based on variable hierarchy, semantic relaxing based on partial invariants, and fix-point solving. These steps will be described in detail in the subsequent sections.

```
1void main(){
2 int x=-10;
3
4 while(x<=20){
5   x=x+1;
6
7 }
8}
```

(a)

```
1void main(){
2 int x=-10;
3 int y=0;
4 while(x<=20){
5   x=x+1;
6   y=y+x;
7 }
8}
```

(b)

```
1void main(){
2int x=-10;
3int y=0;
4while(x<=20){
5  x=x+1;
6  y=y+[-9,21];
7 }
8}
```

(c)

Fig. 4. Slicing and relaxing of our motivating example.

As we have shown in Section I, standard abstract interpretation without using our technique fails to infer any boundary information on variable $\mathbf{y}$ for our motivating example. We now illustrate how our method analyzes the motivating example in Figure 1. First, our analysis infers data and control dependencies between all the variables in the loop body (i.e., variables $\mathbf{x}$ and $\mathbf{y}$). In this example, it finds that the variable $\mathbf{x}$ only depends on itself and the variable $\mathbf{y}$ depends on $\mathbf{x}$ as well as $\mathbf{y}$. Thus our analysis builds a so-called hierarchical variable dependency graph with two variables $\mathbf{x}$ and $\mathbf{y}$, where $\mathbf{x}$ is placed in layer 0 and $\mathbf{y}$ in layer 1. Next, the program is first sliced into program P_0 with only the layer 0 variable $\mathbf{x}$, as shown in Figure 4(a). Since P_0 only involves the lowest-layer variable, no relaxing is needed. Then we use the classic polyhedra abstract domain to compute the partial loop invariants in P_0, which obtains $\mathbf{x} \in [-9, 21]$ at line 6. Then, our analysis slices the program into P_1 (as shown in Figure 4(b)) with the variables in layer 0 and 1. Note that P_1 is actually the original program. In P_1, the assignment at line 6 can be relaxed by the partial invariants computed from the previous step. That is, the appearance of variable $\mathbf{x}$ is replaced by its range [-9,21]. The relaxed program P_1 is shown in Figure 4(c). Note that now the variable y in layer 1 only depends on itself. Using the polyhedra abstract domain to analyze the program P_1, our method can obtain the following loop invariant at line 5:

$$\mathbf{x} \leq 20 \ \wedge \ -\mathbf{x} \leq 10 \ \wedge \ -9\mathbf{x} - \mathbf{y} \leq 90 \ \wedge \ -21\mathbf{x} + \mathbf{y} \leq 210$$

The linear invariants above are the final invariants computed by our method. They are more precise than the invariants computed by the standard non-hierarchical analysis using the polyhedra polyhedra domain, which infers no boundary on variable $\mathbf{y}$.

III. Slicing based on Hierarchical Variable Dependency Graph

This section introduces Hierarchical Variable Dependency Graph (HVDG), based on which we perform program slicing. First, we clarify some definitions. Then, we talk about how to construct the Hierarchical Variable Dependency Graph. Finally, we illustrate how to slice.

A. Variable Dependency

Let v_1 and v_2 be two program variables in a loop. We say v_1 is *directly control dependent* on v_2, if there exists an

assignment s to v_1 which appears in a branch of a conditional statement whose branch condition involves v_2. Variable v_1 is said to be *directly data dependent* on v_2 if v_2 appears in the right-hand expression of an assignment to v_1. We say variable v_1 *directly depends* on v_2 if v_1 is directly control or data dependent on v_2. Suppose there exists a variable sequence $v_0, v_1, \ldots, v_n (n >= 2)$, and for every i $(0 \leq i < n)$, we have v_{i+1} directly depends on v_i. Then we say v_n *indirectly depends* on v_0. Variable v_1 *depends on* variable v_2 if v_1 directly or indirectly depends on v_2.

B. Hierarchical Variable Dependency Graph

Let Var denote the set of all program variables in a loop. Let A and B be two subsets of Var. We say A (directly) depends on B if there exists $v_1 \in A$, $v_2 \in B$, such that v_1 (directly) depends on v_2. Given a program loop, we define its corresponding Variable Dependency Graph (VDG) as a directed acyclic graph (DAG) $G = (N, E)$, where each node $n \in N$ corresponds to a subset of Var (denoted as Var_n), and each edge $e \in E$ from node n_1 to node n_2 represents that Var_{n_2} directly depends on Var_{n_1}. For the sake of concision, even if Var_n depends on Var_n, we do not add a loop on node n. Note that such convention does not affect our results because we aim at deriving the dependencies between different variables.

Given a DAG $G = (N, E)$, a layering of G is a partition of its node set N into a sequence of disjoint subsets (called layers) $N_0, N_1, ..., N_{h-1}$, such that if $(u, v) \in E$ where $u \in N_i$ and $v \in N_j$ then $i < j$. For each i $(0 \leq i \leq h-1)$, N_i is called the i-th layer (or layer i). A DAG with a layering is called a layered (or hierarchical) graph [10]. The height of a layered graph is the number of layers h. Note that, since layered graph has represented the direction (i.e., the dependency relation) between nodes by layers, it is treated as an undirected graph for the sake of simplicity.

A Hierarchical Variable Dependency Graph (HVDG) is an undirected layered graph derived from a variable dependence graph (VDG), denoted as $\mathcal{G} = (N, E)$, wherein there is an undirected edge $e \in E$ connecting nodes n_1, n_2 and node $n_1 \in N$ is put in the next higher layer of $n_2 \in N$, if Var_{n_1} directly depends on Var_{n_2}. Note that the nodes in $\mathcal{G}$ with no input edges are placed in the lowest layer (layer 0), and the other nodes are placed as low layer as possible under the constraints of variable dependencies. Actually, HVDG is a sort of Hasse graph derived from VDG. The nodes in HVDG are ordered from bottom to top w.r.t. the dependencies of the variable sets. According to the hierarchical relationship between different nodes in HVDG, we can derive the hierarchical relationship between the variables directly. Obviously, assume the height of HVDG is h, then all the variables in the loop can be divided into h layers according to the variable dependencies. If node n appears in the i-th layer of HVDG $(0 \leq i \leq h-1)$, then we say all the variables in Var_n are the i-th layer variables of the loop.

We construct the HVDG for a loop in four steps :

1) Constructing original VDG: If variable v appears in the loop, then create a node n in VDG denoting the variable set $\{v\}$. If variable v_2 directly depends on v_1, then draw a directed edge from node n_1 (denoting $\{v_1\}$) to node n_2 (denoting $\{v_2\}$);
2) Merging nodes: If there exists a directed edge from n_1 to n_2 and a directed edge from n_2 to n_1, then merge n_1 and n_2 as a new node n_3, where $Var_{n_3} = Var_{n_1} \cup Var_{n_2}$. In addition, the other edges from or to n_1 and n_2 are transferred to the output or input edges of n_3. In other words, if Var_{n_1} and Var_{n_2} depends on each other, we merge them;
3) Deleting redundant edges: Let n_0, n_1, ..., n_k be nodes in VDG. If there exists an edge from n_0 to n_k and for each $i(0 \leq i \leq k-1)$ there exists an edge from n_i to n_{i+1}, then we delete the directed edge from n_0 to n_k. It means that if Var_{n_k} both directly and indirectly depends on Var_{n_0}, we remove their direct dependency in the graph.
4) Deriving HVDG from VDG: We keep all the nodes of VDG (after the previous 3 steps, which is a DAG) as the nodes of HVDG. If there is a directed edge from node n_1 to n_2 in VDG, we add an indirected edge between n_1, n_2 and put n_1 in the next higher layer of n_2. Note that, for those nodes with no input edges in VDG, we put them in the lowest layer in HVDG. For the other nodes, we put them as low layer as possible under the constraints of variable dependencies.

Note that the proposed notion of HVDG in this paper is different from the concept of conventional program dependence graph (PDG). First, our HVDG describes the dependency between variables, while PDG describes the dependency between statements. Second, our HVDG organizes the whole graph in a hierarchical way while PDG does not.

```
1 while(t<=1000)
2 {
3  t=t + 1;
4  x=x+y+2;
5  y=y+2*x;
6  if(z+y>100)
7   z=z+y+1;
8  else
9   z=x+u;
10}
```

Fig. 5. An illustrating example for variable dependency.

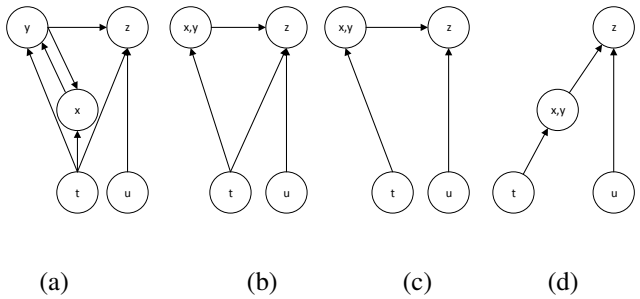

Fig. 6. Constructing HVDG for the program shown in Fig. 5.

Example 1. Consider the program snippet in Figure 5. In this program, we can see that variable x is directly control

dependent on variable t (from statements at lines 1&4), x is directly data dependent on y (from the statement at line 4), and z indirectly depends on x (from statements at lines 5&7). The initial VDG is shown in Figure 6(a). Figure 6(b) gives the resulting VDG after merging nodes $\{x\}$ and $\{y\}$. Figure 6(c) is the resulting VDG after deleting the edge from node $\{t\}$ to node $\{z\}$. Figure 6(d) gives the final HVDG. Based on this HVDG, we could see that t and u are the 0th layer variables, x and y is 1st layer variable, and z is the 2nd layer variable.

C. HVDG based Program slicing

We first introduce some notations before detailing the workflow of slicing. Suppose the variables in a loop have $n+1$ layers in the corresponding HDVG, and $Var_{\leq i}$ denotes the variables from layer 0 to layer i. Given a statement s, $USE[s]$ represents the variables read in the s, while $DEF[s]$ stands for the variables written in s.

For i-th layer, we perform the slicing [11] with respect to $Var_{\leq i}$, i.e., the variables in and below layer i in HDVG. More precisely, it is a two-staged work: 1) For an assignment statement s, we keep it in the resulting sliced program if $DEF[s] \subseteq Var_{\leq i}$, otherwise we remove it. 2) For a branch condition s, we keep it if $USE[s] \subseteq Var_{\leq i}$, otherwise we replace it with a non-deterministic condition $brandom$. In addition, we insert a $skip$ statement for an empty branch.

Note that, when $i = n$, $Var_{\leq i}$ contains all the variables in the loop, then the slicing result will be the original program.

Example 2. Consider the program snippet in Figure 5 again. For the slicing criterions $Var_{\leq 0}$, $Var_{\leq 1}$, and $Var_{\leq 2}$, the slicing results will be the programs shown in Figure 7(a), Figure 7(b), and Figure 7(c) respectively.

```
1while(t<=1000)
2{
3 t=t+1;
4
5
6
7
8
9
10}
```

(a)

```
1while(t<=1000)
2{
3 t=t+1;
4 x=x+y+2;
5 y=y+2*x;
6 if(brandom)
7  skip;
8 else
9  skip;
10}
```

(b)

```
1while(t<=1000)
2{
3 t=t+1;
4 x=x+y+2;
5 y=y+2*x;
6 if(z+y>100)
7  z=z+y+1;
8 else
9  z=x+u;
10}
```

(c)

Fig. 7. Slicing based on HVDG.

IV. Relaxing Based on Partial Loop Invariants

In this section, we introduce the relaxing operator which is used to over-approximate abstract transformers. It aims to reduce the number of unstable ingredients (such as variables, expressions, etc.), thus improving the precision of loop analysis. This section first gives the definition and soundness condition of relaxing operator, and then introduces some general strategies for arbitrary abstract domains.

A. Relaxing Operator

We first introduce some notations. Let $\mathbb{I} \in \{\mathbb{R}, \mathbb{Q}, \mathbb{Z}\}$ be the set of numerical values, Var be the set of all program variables in the loop, $\rho \in \mathbb{S} = Var \to \mathbb{I}$ be an environment which assigns each variable a value.

We assume that our target programs support interval expressions e. More clearly, we have the following syntax for expressions:

$$\begin{array}{rcll} e & := & x & x \in Var \\ & | & [a,b] & a \in \mathbb{I} \cup \{-\infty\}, b \in \mathbb{I} \cup \{+\infty\} \\ & | & e \oplus e & \oplus \in \{+,-,\times,\div\} \end{array}$$

This syntax covers expressions of scalar values (when $a = b$). The semantics of interval expressions are standard [12]. Here we just show some of the semantics that are relevant in this paper. The semantics of an expression $[\![e]\!] : \mathbb{S} \mapsto \mathcal{P}(\mathbb{I})$ maps an environment to a set of values. The semantics of an assignment statement $[\![x := e]\!] : \mathcal{P}(\mathbb{S}) \to \mathcal{P}(\mathbb{S})$ updates the store of x by the valuation of expression e. Let $S \subseteq \mathbb{S}$, the semantics of expressions and assignment statements are defined as follows.

$$\begin{array}{lcl} [\![x]\!](\rho) & := & \{\rho(x)\} \\ [\![[a,b]]\!](\rho) & := & \{v \in \mathbb{I} \mid a \leq v \leq b\} \\ [\![e_0 \oplus e_1]\!](\rho) & := & \{v_0 \oplus v_1 \mid v_0 \in [\![e_0]\!](\rho), v_1 \in [\![e_1]\!](\rho)\} \\ [\![x := e]\!](S) & := & \{\rho[x \mapsto v] \mid \rho \in S, v \in [\![e]\!](\rho)\} \end{array}$$

We define the following partial order between expressions: $e_0 \preceq e_1 \Leftrightarrow \forall \rho \in \mathbb{S}, [\![e_0]\!](\rho) \subseteq [\![e_1]\!](\rho)$. This partial order defines the inclusion relationship between the valuation of expressions. During the iteration of a loop, the valuation of the variables always satisfies loop invariants denoted as Inv, i.e., $\rho \in Inv$ where $Inv \subseteq \mathbb{S}$. Then we define the following partial order under the given loop invariants Inv:

$$Inv \vdash e_0 \preceq e_1 \Leftrightarrow \forall \rho \in Inv, [\![e_0]\!](\rho) \subseteq [\![e_1]\!](\rho)$$

If $Inv \vdash e_0 \preceq e_1$, replacing $[\![x := e_0]\!]$ with $[\![x := e_1]\!]$ is called *relaxing* on transfer function under the given loop invariants Inv. Such operator is called a relaxing operator (denoted as $\Re$).

The $\Re$ Operator is sound in the following sense.

Theorem 1 (Soundness of Relaxing)**.** *If* $Inv \vdash e_0 \preceq e_1$*, then*

$$[\![x := e_0]\!](Inv) \subseteq [\![x := e_1]\!](Inv)$$

Proof. Following the definition of $Inv \vdash e_0 \preceq e_1$, thus $\forall \rho \in Inv, [\![e_0]\!](\rho) \subseteq [\![e_1]\!](\rho)$. Following the definition of $[\![x := e]\!]$, $[\![x := e_0]\!](\rho) \subseteq [\![x := e_1]\!](\rho)$. □

B. Relaxing Strategies

In this subsection, we introduce several relaxing strategies. These strategies are heuristic and may fit into different programs. All the strategies are under the following two assumptions.

- The program sequence $P_0, P_1, \ldots, P_n$ has been obtained by slicing based on HVDG, and the partial loop invariants $Inv^\sharp_{i-1}$ has been already computed by analyzing relaxed program $P_{i-1} (1 \leq i \leq n)$ with our analysis. In this section, we show the strategies of relaxing P_i;
- The employed abstract domain $D^\sharp$ provides an operator $bound\_exp(e)$ that finds the lower and upper bounds of an expression e in an abstract environment.

1) Relaxing based on bounds of expressions

The basic idea of this relaxing strategy is utilizing the partial invariants $Inv_{i-1}^{\sharp}$ to relax all new statements in P_i compared to P_{i-1}. We use s to denote a new statement, e to denote the righthand expression of s, and let $MBE(e)$ be the set of all the "maximum" bounded expressions in e that only involves variables in $Var_{\leq i-1}$. Note that in this paper, an expression is seen as a syntactic tree, and the sub-expressions are also defined along the syntactic tree. More formally, $MBE(e)$ can be deductively defined as following:

$$MBE(e)=\begin{cases}\emptyset & if\ (e \in Var \wedge bound\_exp(e) = \top) \\ & \quad \vee\ e \in Intvs \\ e & if\ e \notin Intvs \wedge bound\_exp(e) \neq \top \\ MBE(e_1) \cup MBE(e_2) & if\ bound\_exp(e) = \top \wedge e = e_1 \oplus e_2\end{cases}$$

where $Intvs$ is the set of all interval constants and $\oplus \in \{+, -, \times, \div\}$.

For each $e' \in MBE(e)$, its appearance is replaced by its value range orderly, which is obtained from $Inv_{i-1}^{\sharp}$ by $bound\_exp(e')$. This replacement results in relaxed transfer functions for those new statements. Note that we do not include unbounded expressions in $MBE(e)$. In other words, if $bound\_exp(e')$ returns $\top$, our analysis does not relax e', since it brings no benefit for improving precision. That is to say our analysis only relaxes statements on bounded expressions. This strategy is called Bounded Expression Strategy (BES).

In some abstract domains, operator $bound\_exp$ can be expensive in time and space. For the sake of efficiency, we define a simpler strategy called Bounded Variable Strategy (BVS). This strategy follows a similar process, but relaxing $BV(e)$ rather than $MBE(e)$, where $BV(e)$ is a set of variables in $V_{\leq i-1}$ and also appearing in e. This means that it relaxes e by replacing each low-layer variables with intervals, rather than considering the maximum expressions. Compared with BES, BVS is more efficient, but less precise.

Example 3. Suppose $Inv_{i-1}^{\sharp} = \{5 \leq x + y \leq 10 \wedge x \geq 0 \wedge y \geq 0 \wedge z = [-\infty, +\infty]\}$ the new statement in P_i is $s : \{t = x * t + x + y + z;\}$ and only t is the variable in the i-th layer. Then, we have $MBE(x * t + x + y + z) = \{x, x+y\}$ (Note that, if "y+z" is also bounded, we will also add "y+z" into $MBE(x * t + x + y + z)$). If our analysis utilizes BES to relax s, we finally get relaxed statement $s' : \{t = [0, 10]t+z+[5, 10];\}$. If we take BVS, since $bound\_exp(x) = [0, 10]$, $bound\_exp(y) = [0, 10], bound\_exp(z) = [-\infty, +\infty]$, we finally get relaxed statement $s' : \{t = [0, 10]t+z+[0, 20];\}$.

2) Relaxing based on interval linearization

In this subsection, we define a relaxing strategy that is not as general as those defined in the last subsection, but is more precise on specific cases. This relaxing strategy is called relaxing based on interval linearization (RIL). RIL first relaxes expressions only involving lower layer variables into interval affine forms [13]: $i_0 + \Sigma_k(i_k \times v_k)$, where i_k is an interval and v_k is an variable. When encountering a product of two variables ($x_1 \times x_2$), where neither of x_1 and x_2 is constant, we need to choose one of the factor to be replaced by its interval range. The choice of which factor to be replaced may influence the analysis result. Our heuristic strategy is to replace the factor with narrower interval range. For more heuristic strategies, we refer the readers to [14].

Example 4. Suppose y and z are lower-layer variables, $Inv_{i-1}^{\sharp} = \{y = [c, d] \wedge z = [e, f]\}$, and the new statement in P_i is $s : \{x = y * y + z;\}$ compared to P_{i-1}. If we take the BES or BVS strategies, our analysis will get $s' : \{x = [c, d][c, d] + [e, f];\}$. If we take RIL to relax this expression to interval affine form, we will get $s' : \{x = [c, d] * y + z;\}$.

The interval affine forms can be handled by the existing interval polyhedra domain [15]. We may further adopt linearization techniques [14] to further relax interval affine forms into linear forms as $i'_0 + \Sigma_k(c_k \times v_k)$, where c_k is constant. In practice, c_k usually takes the middle point of i_k as its value. Then the linear form can be dealt with by conventional numerical abstract domains of linear constraints (such as the polyhedra abstract domain).

Example 5. Continue Example 4. Given the assignment $\{x = [c, d] * y + z;\}$, we can derive a liner form, such as $\{x = ((c + d)/2) * y + z + [m, n]\}$; where $[m, n] = [(c - d)/2, [d - c]/2] \boxtimes [e, f]$, and $\boxtimes$ is interval arithmetic multiplication. We can then fit it into the polyhedra abstract domain as $Inv_i^{\sharp} = \{x - (c + d) * y/2 - z <= n \wedge x - (c + d) * /2 - z >= m\}$, which contains linear constraints between x, y, z.

Note that, the interval linearization technique in our method is different from the standard interval linearization technique used in [14], which uses variables interval ranges of current iteration (not loop invariants) to perform interval linearization, while we utlize variable bounds derived from invariants.

3) Relaxing based on one-variable interval linear form

The previous strategies only consider local information on a sole statement. We can also design relaxing strategies that consider multiple statements as a whole (such as statements in block). In this part, we propose relaxing strategies of such kind. They are all based on one-variable interval linear form (OVILF).

a) Assignments in one-variable interval linear form: The following assignment form is called one-variable interval linear form (OVILF)

$$v_k = i_1 \times v_k + i_2;$$

where i_1 and i_2 are interval constants.

If in an assignment to a current layer variable, the right-hand side expression contains only lower-layer variables and the left-hand variable, then this assignment can be transformed into OVILF. This transformation can be easily obtained by relaxing all lower-layer variables into intervals. Our strategy assumes that assignments to current layer variables can be transformed into OVILF. Furthermore, these OVILFs can be transformed into the form such that in a loop the assignment to each variable appears only once. To reach this target form, we use symbolic techniques [16] [17] to transfer several OVILF assignments to v_k into one. Now we show how two assignments to v_k are combined into one. This process can be repeated to deal with the case of more than two assignments. If the two assignments are sequential in the loop, as shown in Figure 8, with symbolic propagation techniques, these two assignments are replaced with the following OVILF assignment

$$v_k = (i_1 \boxtimes i_3)v_k + (i_2 \boxtimes i_3 \boxplus i_4);$$

where $\boxplus$, $\boxtimes$ represent interval addition and multiplication operations.

```
1 while(cond){
2  ...;
3  v_k = i_1 v_k + i_2;
     //where i_1, i_2 are interval constants;
4  v_k = i_3 v_k + i_4;
     //where i_3, i_4 are interval constants;
5 }
```

Fig. 8. Sequential assignments inside a loop.

If the two assignments are in different branches of the same conditional block, as shown in Figure 9. We delete these two statements, and insert a new assignment as follows to the end of conditional block.

$$v_k = (i_1 \sqcup i_3)v_k + (i_2 \sqcup i_4);$$

where $\sqcup$ is the union of two intervals.

```
1 while(cond){
2  ...;
3  if(*)
4     v_k = i_1 v_k + i_2;
      //where i_1, i_2 are interval constants;
5  else
6     v_k = i_3 v_k + i_4;
      //where i_3, i_4 are interval constants;
7 }
```

Fig. 9. Assignments in different branches inside a loop.

Inside a loop body, we conduct the transformation into OVILF from the inner most branches to the outmost ones by utilizing the above symbolic combinations repeatedly. Finally, we can get the target form such that inside a loop there exists only one OVILF for each variable.

b) Analyzing loops with one-variable interval linear assignments: This part introduces the method for solving fix-point for OVILF (i.e., $v_k = i_1 \times v_k + i_2$;). In general, if i_1 is not one of the three constants 1, -1, 0, then the value of v_k is always exponential to loop counter variables. Conventional abstract interpretation with linear constraint abstract domain can hardly find bounds on such variables. We propose a quick fix-point solving technique called *formula method* to deal with such cases.

① Utilizing Formula Method for the general term

Our analysis first introduces a counter n to the loop by inserting $n = 0$; before the loop, and $n = n + 1$; at the end of the loop. Even n may be control dependent on other variables in HVDG, we still consider n as the 0th layer variable. Suppose that for the i-th layer variable x, the loop body only contains one OVILF assignment on x as following:

$$x = [a, b]x + [c, d]; (a, b, c, d \in \mathbb{I}, and\ a \leq b, c \leq d)$$

Assume that the initial value of x is x_0, and that x_n represents the interval range of x after n iterations. Then by substitution of x, we obtain the following formula:

$$x_n = [a, b]^n x_0 + ([a, b]^{n-1} + [a, b]^{n-2} + \cdots + 1)[c, d] \quad (1)$$

Formula (1) needs a lot of interval addition and multiplication operations. Therefore, we further simplify it based on the sign of a.

If $a \geq 0$, then for each $0 \leq i \leq n$, $a^i \leq b^i$ holds. Then Formula (1) can be simplified to the following formula:

$$x_n = [a^n, b^n]x_0 + [g, h][c, d] \quad (2)$$

where

$$[g, h] = \begin{cases} [\frac{1-a^n}{1-a}, \frac{1-b^n}{1-b}] & a, b \neq 1 \\ [n, \frac{1-b^n}{1-b}] & a = 1, b \neq 1 \\ [\frac{1-a^n}{1-a}, n] & a \neq 1.b = 1 \\ [n, n] & a, b = 1 \end{cases}$$

If $a < 0$, then for every $0 \leq i \leq n$, there is no deterministic relation between a^i and b^i, and the value of x_i may change its sign during the iterations. Thus Formula (2) does not work for this case. We propose a more general formula with some loss of precision. To do that, we first relax Formula (1) into the following form:

$$x_n = [-t^n, t^n]x_0 + ([-t^{n-1}, t^{n-1}] + [-t^{n-2}, t^{n-2}] + \ldots + 1)[c, d]$$

where $t = max\{|a|, |b|\}$. Then the above formula can be simplified to following formula:

$$x_n = [-t^n, t^n]x_0 + [j, k][c, d] \quad (3)$$

where

$$[j, k] = \begin{cases} [-\frac{1-t^n}{1-t}, \frac{1-t^n}{(1-t}] & t \neq 1 \\ [-n, n] & t = 1 \end{cases}$$

Example 6. In the program in Figure 10, variable y has exponential relation with x. From HVDG, we find x and n (loop counter inserted by our method) are the $0th$ layer variables, and y is 1st layer variable. Our analysis utilizes the polyhedra domain to compute the loop invariants of the sliced program with $0th$ layer variables x and n, and get invariant $\{x = [0, 10] \wedge n = [0, 5]\}$ at loop head. Then the assignment $\{y = 2 * y + x;\}$ is relaxed to $\{y = 2 * y + [2, 12];\}$. We cannot get the upper bound of y by conventional widening operator. But with Formula (2), our analysis gets $y = [1, 404]$ at line 3 quickly.

```
1 int x=0,y=1,n=0
2 while(x<=10){
3  x=x+2;
4  y=2*y+x;
5  n=n+1;
6 }
```

Fig. 10. An example with exponential invariants.

② Loop bound computation

For the inserted loop counter n, conventional abstract interpretation sometimes can generate linear invariants between n and other $0th$ layer variables, so the upper bound of n may be inferred from the invariant directly. Take the program in Figure 10 for instance, the upper bound of n can be inferred as 5. However, if the upper bound of n cannot be inferred directly from the invariant, our method will treat n as a symbolic constant, and keep it in the formula, and derive the upper bound of n by constraints solver.

```
1 int x=0,y=0,n=0;
2 while((y-x)<100){
3  x=x+2;
4  if(*)
5   y=2*y+1;
6  else
7   y=3*y+2;
8  n=n+1;
9 }
```

Fig. 11. An example for computing loop bounds.

Example 7. We use the program in Figure 11 to illustrate the process of inferring the upper bound of loop counter n. Our method first relaxes the statements in the loop to OVILF: $\{x = x + 2; y = [2, 3] * y + [1, 2];\}$, and then obtains $\{x_n = 2n; y_n = [2^n - 1, 3^n - 1];\}$ by the formula method. Following loop condition $y - x \leq 100$, we find $[2^n - 2n - 1, 3^n - 2n - 1] \leq 100$. By solving this inequality on n, our method gets $n \leq 6$, with which we can get bounds for x and y, i.e., $\{x = [0, 12] \wedge y = [0, 728]\}$.

③ Further Optimization

Our formula method computes interval information on variables, which contains no relational constraints between variables. Now we propose a strategy that makes use of the interval information to further relax the transfer functions of all statements in a loop to the following form:

$$v_1 = cv_1 + i_1;$$
$$v_2 = cv_2 + i_2;$$

where i_1, i_2 are intervals, and c is constant.

For these relaxed statements, a linear invariant between v_1 and v_2 will be generated by conventional abstract interpretation with linear constraint abstract domains, such as the polyhedra abstract domain.

Example 8. Consider again the program in Figure 11. Based on the previous interval invariant $\{x = [0, 12] \wedge y = [0, 728]\}$ which is generated in Example 7, we perform further relaxing on previously relaxed statements $\{x = x + 2; y = [2, 3] * y + [1, 2];\}$ to obtain the above general form as $\{x = x + 2; y = y + [1, 1458];\}$ (where the coefficients of variable x and y in the right-hand expressions are both 1 after relaxing). Then we can utilize the polyhedra domain to perform analysis, which results in the following invariants:

$$\{2y \geq x \wedge y \leq 729x \wedge x \geq 0 \wedge x \leq 12 \wedge y \leq 728\}.$$

V. Soundness and Precision of Analysis via Relaxing Transformers

In this section, we discuss the soundness and precision of our hierarchical analysis using relaxed transformers.

First, the soundness of the analysis using relaxed transformers is straightforward. In Section IV, Theorem 1 has shown the soundness of the relaxed transformer itself, that is, given the same abstract pre-state, the resulting post-state given by the relaxed transformer will be an over-approximation of that given by the concrete transfer semantics. And all the other abstract operations that are required during the analysis reuse the same operations provided by the regular abstract domains, including meet, join, widening, etc. Then, the framework of abstract interpretation guarantees the soundness of the whole analysis using relaxed transformers.

This paper mainly aims to leverage the technique of relaxing abstract transformers to improve the precision of the analysis. One may wonder why we could improve the precision of the final analysis results via relaxing transformers, since relaxing transformers itself means a kind of precision loss. It is known that the widening operator in the abstract interpretation framework may not be monotone [18]. First, the widening operator is not monotone with respective to the left parameter. In other words, even though $I_1 \sqsubseteq I_2$, we may not have $I_1 \triangledown I \sqsubseteq I_2 \triangledown I$. E.g., suppose $I_1 = [0, 1], I_2 = [0, 2], I = [0, 2]$, which implies that $I_1 \sqsubseteq I_2$. However, we have $I_1 \triangledown I = [0, +\infty] \not\sqsubseteq I_2 \triangledown I = [0, 2]$. Also, the widening operator is not monotone with respective to the second parameter. E.g., suppose we use the polyhedra abstract domain with standard widening, and $I = \{0 \geq x - y \geq -1, x \geq 0\}, I_1 = \{-x + 2y \geq 0, 2x - y \geq -2\}, I_2 = \{x \geq 0, y \geq 0\}$, which implies $I_1 \sqsubseteq I_2$. However, we have $I \triangledown I_1 = \top \not\sqsubseteq I \triangledown I_2 = \{x \geq 0\}$.

In other words, due to the above fact that the widening operator is not monotone, the whole analysis using relaxed abstract transformers may get more precise final results than that using the original abstract transformers. Of course, using relaxed abstract transformers may also get less precise final results. In fact, when using a proper relaxing operator, although it may cause more precision loss temporally at that statement, the relaxing operator may utilize the already computed partial invariants to reduce unstable ingredients, and helps to get more precise final results after the whole analysis. Take our motivating example in Figure 1 for example,

VI. Experiment and Evaluation

We have implemented a prototype static analyzer RelaxAIer to support our hierarchical analysis using relaxed abstract transformers, on top of Interproc [19], which is one of the available and widely used invariant generation tools based on abstract interpretation [20][21][22]. Compared with Interproc, RelaxAIer has used the same abstract domain library and fixpoint solver library. We use Box (an implementation of the interval abstract domain) and Polka (an implementation of the polyhedra abstract domain) from the APRON numerical abstract domain library [23] in our experiment.

A. Evaluation Setup

We evaluate RelaxAIer along the following two research questions:

(1) RQ1: How frequently can loop variables in non-linear programs be partitioned into multiple layers?

(2) RQ2: How effective is our technique in improving the precision of abstract interpretation?

To address these research questions, we have conducted experiments on a benchmark consisting of 46 programs on Table I, 35 of which are taken from the whole set of NLA (Non-linear Algorithmic) Benchmark [24] [25], and 3 of which are taken from [26] (the program namely mul2) and [27] (the programs namely prod, petter) and the last 8 of which are from

[28]. These programs often appear as sets of test cases in the work of non-linear invariant generation [29] [30] [31] [32] [33]. Although all these program are within 100 lines of code, they implement non-linear mathematical functions and are very subtle to analyze or verify. Many of them contain symbolic inputs (whose value ranges are not restricted). To conduct experimental comparisons, we thus assume a set of fixed parameter values as symbolic inputs for these programs. In addition, 5 programs have infinite loops (such as $while(true)$). These infinite loops cause all variables in the programs to take infinite bounds. For the sake of experimental comparisons, we have modified the loop conditions for these 5 programs, so that the numbers of loop iterations are bounded. RelaxAIer is compared with Interproc to evaluate the effectiveness of our technique in improving the precision of abstract interpretation.

B. Evaluation Results

The result of our experiments are shown on Table I. The "Disc." column gives a short description of the functionality of the program. The "#V" column gives the number of variables. The "#D" column reports the maximum degree of underlying strongest loop invariant on loop control variable, where "p" means polynomial invariant, the number ahead of "p" means the maximum degree of polynomial invariant, and "e" means exponential invariant. The "#H" column provides height of HVDG (i.e., the number of hierarchical layers over variables). The "Interproc" column reports the results obtained by Interproc when using the polyhedra abstract domain. "RelaxAIer" reports the result of our tool with previous designed relaxing strategies. "#B" column reports the number of bounded variables that the analysis infers at the loop head. "T(ms)" column reports the time overhead of generating invariant in millisecond when the analysis runs on a virtual machine (using VMware Workstation) , with a guest OS of Ubuntu 14.04 (2GB Memory), host OS of Windows 8, and a 3.2 GHz quad-core Intel(R) Core(TM) i7-5500U host CPU. "P" column compares the precision of the resulting loop invariants generated by Interproc and RelaxAIer. A "$>$" ("$<$", "$=$") indicates that Interproc outputs stronger (weaker, equivalent) invariants than RelaxAIer.

1) RQ1 – Frequency of layered loop variables in programs

Utilizing the hierarchical dependency relationships among variables is one basic idea of our approach. Note that "#H=1" means that the loop variables cannot be partitioned into layers in programs. From the "#H" column in Table I, we can see that 40 out of 46 programs (around 87.0%) have at least 2 hierarchical layers among variables. We have further manually checked the source code of most programs, and found that only a few variables (i.e., 0th layer variables) count for controlling the number of loop iterations, while most of the functionality of the program are implemented via higher-layer variables. This observation further provides a confirmation from practice that the dependency among variables in a loop can be hierarchical.

2) RQ2 – Effectiveness of our technique in improving precision of abstract interpretation

Considering the accuracy, comparing the "P" columns in Table I, we find that RelaxAIer improves precision for 39(84.8%) out of 46 programs. The main reasons for these improvements lie in that for these programs with non-linear invariants, the analysis precision of Interproc is limited by the linear expressiveness of the polyhedra abstract domain. And the widening operator may cause severe loss of precision, and fails to find finite boundaries for variables, which is one of the main sources of high false alarm rate when checking for run-time errors. Our hierarchical analysis with relaxed abstract transformers makes it more possible to enclose the non-linear invariants through polyhedral invariants, and infer finite boundaries for more variables. Comparing the "#B" columns between Interproc and RelaxAIer, we find that for 35 out of 46 programs (76.1%), RelaxAIer can infer finite boundaries for more variables compared to Interproc. And for these 35 programs, RelaxAIer also infers more precise invariants. For the remaining 11 programs, RelaxAIer and Interproc result in the same number of bounded variables. For these 11 programs, we then further compare the accuracy of the loop invariants by comparing the resulting interval range for each variable. And we highlight the comparison results for these 11 programs in the column "P". RelaxAIer get more precise variable ranges for 4 programs (column "P" is "$<$"). For 5 programs (4 of which involve only linear invariants after providing fixed values for symbolic inputs, i.e., #D is 1p), RelaxAIer is as precise as Interproc in variable ranges (column "P" is "$=$"). And RelaxAIer get less precise variable ranges for 2 programs (column "P" is "$>$"). For these 2 programs, parts of the most accurate loop invariant can be represented by linear equalities, which can not be discovered any more after relaxing transfer functions in RelaxAIer.

Example 9. We take the program in Figure 12 to show this precision loss. This program is simplified from "hard.c" in our experiment. If we analyze it as a whole, we can find the invariant at the loop head using the polyhedra domain:$\{\mathbf{x} \leq 128 \wedge -\mathbf{x} \leq 1 \wedge -\mathbf{x}+\mathbf{y} = 128\}$, thus we know $\mathbf{y} = [-127, 0]$ at the loop head. However, if we relax the value of $\mathbf{x}$ into $[1, 128]$ at line 6 (Note that this range is got by the first-pass analysis on $\mathbf{x}$ only), we find that the relationship between $\mathbf{x}$ and $\mathbf{y}$ (i.e., $-\mathbf{x} + \mathbf{y} = 128$) does not hold any more. Thus, we can only get $\mathbf{y} = [-\infty, 0]$ at the loop head, which is less precise than $\mathbf{y} = [-127, 0]$.

```
1 void main(){
2   int x = 128;
3   int y = 0;
4   while(x >= 1){
5     x = x/2;
6     y = y-x;
7   }
8 }
```

Fig. 12. An example with precision loss for RelaxAIer.

In addition, for 6 programs with 1-layer HVDG (whose "#H" is 1 in Table I), we also implemented a strategy to utilize the partial invariants of 0-th layer variables to relax the statements in the loop, and RelaxAIer can increase the number of bounded variables of 4 programs.

Example 10. Among these 4 programs, we take the program in Figure 13 for example to show this improvement, which is simplified from "fermat2.c" in our experiment. For this

TABLE I
EXPERIMENTAL RESULTS.

Program	Disc.	#V	#D	#H	Interproc		RelaxAIer		P
					#B	T(ms)	#B	T(ms)	
mul2	product	2	1p	2	2	8	2	16	=
prod	product	3	e	3	0	28	3	84	<
cohendiv	division	6	e	3	4	28	6	96	<
mannadiv	division	3	1p	2	3	16	3	36	=
divbin	division	3	e	3	0	36	3	128	<
hard	division	4	e	2	4	28	4	120	>
wensley	division	4	e	2	4	36	4	96	>
dijkstra	square root	4	e	2	0	40	4	284	<
sqrt	square root	3	2p	2	0	12	3	32	<
z3sqrt	square root	3	e	1	2	24	2	44	=
freire1	square root	2	2p	1	0	8	2	8	<
freire2	cubic root	3	3p	1	0	16	3	36	<
euclidex1	extended gcd	9	e	3	0	96	4	420	<
euclidex2	extended gcd	6	e	2	0	36	0	72	<
euclidex3	extended gcd	11	e	4	0	96	3	312	<
fermat	divisor	3	2p	1	0	36	0	112	<
fermat2	divisor	3	2p	1	2	28	3	84	<
knuth	divisor	5	e	1	0	124	5	300	<
lcm	lcm	4	e	2	0	28	0	46	<
lcm2	lcm	4	e	2	0	32	0	68	<
illinois	protocol	4	3p	3	0	116	2	176	<
berkeley	protocol	4	3p	3	0	60	3	188	<
firefly	protocol	4	3p	3	0	76	2	140	<
mesi	protocol	4	3p	3	0	52	3	220	<
moesi	protocol	5	3p	3	0	72	4	136	<
prod4br	product	4	e	3	0	168	4	244	<
readers_writers	simulation	3	1p	2	0	36	0	76	=
cohencu	cubic sum	4	3p	4	2	16	4	76	<
petter	power sum	2	6p	2	1	8	2	16	<
ps1	power sum	3	1p	2	3	8	3	20	=
ps2	power sum	3	2p	3	0	8	3	36	<
ps3	power sum	3	3p	3	0	12	3	40	<
ps4	power sum	3	4p	3	0	12	3	40	<
ps5	power sum	3	5p	3	0	12	3	44	<
ps6	power sum	3	6p	3	0	12	3	44	<
geo1	geo series	3	e	2	0	16	3	60	<
geo2	geo series	3	e	2	0	20	3	64	<
geo3	geo series	3	e	2	0	24	3	68	<
BCK2011_gauss	power sum	4	2p	2	0	16	4	40	<
BCK2011_strength_reduction	power sum	6	2p	3	0	72	4	276	<
BCK2011_strength_reduction_linear	power sum	6	2p	3	0	60	6	168	<
CFD17-add-const_product	power sum	4	2p	3	0	32	4	128	<
fibonacci_information_flow	fibonacci	3	e	2	0	32	1	48	<
exp_with_linear_inner_loop	exponential	4	e	3	0	28	2	52	<
exp_add_linear	exponential	3	e	3	0	16	2	56	<
exp_add_loop_variable	exponential	2	e	2	0	12	1	20	<
Total		178		110	27	1752	129	4870	

program, $\mathbf{y}$ depends on $\mathbf{x}$ and $\mathbf{x}$ depends on $\mathbf{y}$, so "#H=1". If we use Interproc with the polyhedra domain to analyze it, we get $\mathbf{x} = [0, +\infty], \mathbf{y} = [1, 128]$ at the loop head. However, if we use RelaxAIer, for the first-pass analysis (we analyze the whole program), we can get $\mathbf{x} = [2, +\infty]$ at line 6. Then we relax $\mathbf{y} = \mathbf{y} - \mathbf{x}$ into $\mathbf{y} = \mathbf{y} - [2, +\infty]$, and perform the second-pass analysis. This time, we can get the more precise invariant for $\mathbf{x}$, i.e., $\mathbf{x} = [2, 129]$ at the loop head, which shows that $\mathbf{x}$ is bounded now.

```
1 void main(){
2   int x = 0;
3   int y = 128;
4   while(y > 0){
5     x = x+2;
6     y = y-x;
7   }
8 }
```

Fig. 13. An one-layer example with precise result for RelaxAIer.

Overall, in the last row of Table I, we show that totally 178 bounded variables appear in these 46 programs. Interproc can infer finite boundaries for 27 variables (15.2%), while RelaxAIer can infer finite boundaries for 129 variables (72.5%), which is 4.78X of that of Interproc. Hence, the experimental results show that the invariants generated by our method

is obviously more precise than that given by standard non-hierarchical analysis.

Considering the time overhead, since our method performs multiple passes of constructing relaxed semantic equations and fix-point solving, the time overhead is greater than the standard non-hierarchical analysis which only takes one-pass analysis. From the results shown in Table I, we find that RealxAIer takes 4870 milliseconds to analyze these 46 programs, whose average overhead is 105.9 milliseconds, while Interproc takes 1752 milliseconds, whose average overhead is 38.1 milliseconds. The average overhead of RelaxAIer is 2.78X of that of Interproc. However, for these relatively small programs, time is not a big issue.

Threats to validity. We identified the following threats to the validity of our experiments: (1) Sample size: our experimental programs are all from benchmark suites, which mainly contain small-sized non-linear programs. (2) Underlying analyzers: the efficiency and precision of computing invariants rely on the underlying abstract interpretation based analyzers (our implementation is based on Interproc), especially the chosen of abstract domains. For our experiments, we mainly use a combination of the interval abstract domain and the polyhedra abstract domain.

VII. Related Work

To improve the precision of static analysis by abstract interpretation, Miné [14] proposed several generic symbolic enhancement methods that can be applied to numerical abstract domains, including linearization and symbolic constant propagation. These symbolic techniques also help to compensate for the lack of non-linear transfer functions for those abstract domains of linear constraints. During loop analysis, these techniques rely on the immediate environments computed during iterations to simplify expressions. Since these immediate environments are unstable, it may cause much precision loss during widening. By contrast, our method utilizes partial loop invariants on lower-layer variables to relax transfer functions on higher-layer variables. Since our relaxed transfer functions involve less unstable ingredients (such as variables, expressions, etc.), our method turns to suffer less precision loss during widening.

Much existing work has addressed on improving the precision of fix-point solving (e.g., by enhancing widening operator), such as widening with thresholds, widening delaying [9], lookahead widening [34], intertwining widening and narrowing [35], etc. Our approach uses the chaotic iteration strategy [36] with traditional widening technique. Our work does not focus on enhancing fix-point solving directly, but by relaxing the abstract transformers before conducting the fix-point computation. The aforementioned improvement techniques over fix-point solving are orthogonal to our approach, and they can be applied during the fix-point solving period, on top of the relaxed abstract transformers generated by our approach.

Generating non-linear loop invariants has been a challenging topic for years. Many techniques like Gröbner bases [26], abstract interpretation [37], interpolating theorem [38], Guess-and-Check [27], counterexample-guided [33], and solving semi-algebraic systems [39][40][41] have been proposed. These works do not fully solve this challenge. In [27], Sharma et al. proposed an algorithm for computing algebraic equation invariants. Their method first guesses a candidate invariant by date driven analysis and then check whether this candidate invariant is indeed a true invariant. This Guess-and-Check procedure iterates until a true invariant is found. However, this method can only generate polynomial equation invariants, and it is necessary to presume a small degree in advance due to the fact that their overhead increases sharply with growing degree. Our method does not directly compute non-linear invariant, but uses approximated linear invariants that are captured by linear constraint abstract domains to soundly enclose non-linear invariants. Since our method only relies on conventional linear constraint abstract domains, it is more general and efficient than those methods generating polynomial invariants.

Gulavani et al. [42] used non-linear axioms such as log, multiplication, square root and exponentiation to enhance the expressiveness of the polyhedra domain. Their loop invariant is used to handle the bounds of non-linear invariants in complexity analysis. Recently, Kincaid et al. [28] proposed a wedge abstract domain to obtain non-linear invariants in programs. The wedge domain treats non-polynomial terms as independent dimensions, and describes the linearity of non-polynomial terms through the linear constraints expressed by the polyhedra domain. Both of these methods introduce non-linear elements into the polyhedra domain to express non-linear invariants. The precision of their generated non-linear invariants depends on the precision of linear invariants derived by the polyhedra domain. Our method is devoted to generating more precise linear loop invariants using conventional linear constraint abstract domains through hierarchical analysis with relaxed transfer functions, which is also helpful to improve the precision of the methods in [42] [28].

The arithmetic-geometric domain [43] infers bounds on the values of variables in terms of a clock counter (which models the program execution time). The main idea is to approximate the maximum value of each variable X by an expression of the affine form $[X \mapsto a \times X + b]^{(n)}(M)$, where a, b are non-negative real numbers, M is the initial value, n is the maximum value of the clock counter. This domain has been successfully applied in certification of huge embedded software involving floating-point rounding errors. Our relaxing based on one-variable interval linear form shares the similar idea as [43]. One of the main differences lies in that our approach abstracts all assignment statements to variable X (even in different branches) inside a loop into one statement of one-variable interval linear form $X = i_1 \times X + i_2$, where i_1 and i_2 are interval constants and then uses Formula Method for the general term to compute the bounds of variables in terms of the number of loop iterations. However, the arithmetic-geometric domain [43] maintains the affine form in the abstract domain (although it abstracts floating-point computation into interval linear form) and performs analysis statement-by-statement using abstract interpretation.

Recently, there have been some new advances in utilizing abstract acceleration [21] [44] rather than widening to compute loop invariants. This method models all the assignments with no branches by a square matrix, and converts a loop of n

iterations into a n-th power of the matrix. Then thanks to the Jordan decomposition of the linear transformers, the matrix can be abstracted by the polyhedra domain. Their method can also be used to approximate the boundaries of non-linear loop invariants. However, their method also has several limitations. E.g., it targets only linear loops, i.e., loops containing linear assignments and guards. If there is a branch in the loop, two matrices are constructed and analyzed separately, which may cause combinatorial explosion. By contrast, our work can handle non-linear assignment statements and loops with multiple branches inside.

Hakjoo et al. [45] utilized the control and data dependency between statements to exploit the sparsity of the program, in order to improve the scalability of their analysis. In this paper, we focus on utilizing the dependency relations between variables (rather than statements) to improve the precision of loop analysis. More recently, Singh et al. [46] proposed approaches by decomposing the transformers to improve the efficiency of sub-polyhedra abstract domains without compromising their precision. They decompose the transformers along with conditions for checking whether the decomposed transformers lose precision with respect to the original transformers. In this paper, we focus on improving the overall precision of the whole analysis and relax transformers which may cause precision loss locally.

VIII. Conclusion and Future work

We present a hierarchical analysis to infer loop invariants based on relaxed abstract transformers. First, we introduce a so-called hierarchical variable dependency graph, to organize all the variables in the loop. The approach then analyzes a refined series of sliced versions of the original loop, which are generated with respect to different layers of variables from lower to higher. The so-called partial invariants over lower-layer variables are then used to relax transfer functions when analyzing the higher-layer variables. To this end, we propose three strategies for relaxing abstract transformers based on the already computed partial invariants. The key idea behind relaxing abstract transformers is to relax unstable ingredients into stable ones such that the overall analysis will finally get more precise results. We also explain the soundness guarantee and discuss the precision of using relaxed abstract transformers. Finally, we present encouraging experimental results on the benchmark programs involving non-linear behaviors, which shows that our approach can get more precise invariants than the standard approach.

For the future work, we will extend our approach to support nested loops and consider more relaxing strategies. Also, we plan to apply the idea of hierarchical analysis and relaxing in more applications such as complexity/resource bound analysis, WCET analysis, etc.

References

[1] P. Cousot and R. Cousot, "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL'77*. ACM, 1977, pp. 238–252.

[2] A. Miné, "The octagon abstract domain," *Higher-order and symbolic computation*, vol. 19, no. 1, pp. 31–100, 2006.

[3] P. Cousot and N. Halbwachs, "Automatic discovery of linear restraints among variables of a program," in *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, 1978, pp. 84–96.

[4] K. Ghorbal, E. Goubault, and S. Putot, "The zonotope abstract domain taylor1+," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 627–633.

[5] P. Granger, "Static analysis of arithmetical congruences," *International Journal of Computer Mathematics*, vol. 30, no. 3-4, pp. 165–190, 1989.

[6] K. Ghorbal, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta, "Donut domains: Efficient non-convex domains for abstract interpretation," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2012, pp. 235–250.

[7] J. Feret, "Static analysis of digital filters," in *European Symposium on Programming (ESOP'04)*, ser. LNCS, no. 2986. Springer-Verlag, 2004, springer-Verlag.

[8] P. Roux and R. Jobredeaux, "A generic ellipsoid abstract domain for linear time invariant systems," in *ACM International Conference on Hybrid Systems: Computation and Control*, 2012, pp. 105–114.

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival, "Combination of abstractions in the astrée static analyzer," in *Annual Asian Computing Science Conference*. Springer, 2006, pp. 272–300.

[10] P. Healy and N. S. Nikolov, "How to layer a directed acyclic graph," in *International Symposium on Graph Drawing*. Springer, 2001, pp. 16–30.

[11] M. Weiser, "Program slicing," in *Proceedings of the 5th international conference on Software engineering*. IEEE Press, 1981, pp. 439–449.

[12] R. E. Moore, R. B. KearFott, and M. J. Cloud, *Introduction to Interval Analysis*. SOciety for Industrial and Applied Mathematics, 2009.

[13] L. H. De Figueiredo and J. Stolfi, "Affine arithmetic: concepts and applications," *Numerical Algorithms*, vol. 37, no. 1-4, pp. 147–158, 2004.

[14] A. Miné, "Symbolic methods to enhance the precision of numerical abstract domains," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2006, pp. 348–363.

[15] L. Chen, A. Miné, J. Wang, and P. Cousot, "Interval polyhedra: An abstract domain to infer interval linear relationships," in *International Static Analysis Symposium*. Springer, 2009, pp. 309–325.

[16] A. Thakur and T. Reps, "A method for symbolic computation of abstract operations," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 174–192.

[17] T. Reps and A. Thakur, "Automating abstract interpretation," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2016, pp. 3–40.

[18] P. Cousot and R. Cousot, "Comparing the galois connection and widening/narrowing approaches to abstract interpretation," in *International Symposium on Programming Language Implementation and Logic Programming*. Springer, 1992, pp. 269–295.

[19] B. Jeannet, "Interproc analyzer for recursive programs with numerical variables," *INRIA, software and documentation are available at the following URL: http://pop-art. inrialpes. fr/interproc/interprocweb. cgi. Last accessed*, pp. 06–11, 2010.

[20] I. Dillig, T. Dillig, B. Li, and K. McMillan, "Inductive invariant generation via abductive inference," in *Acm Sigplan Notices*, vol. 48, no. 10. ACM, 2013, pp. 443–456.

[21] B. Jeannet, P. Schrammel, and S. Sankaranarayanan, "Abstract acceleration of general linear loops," in *ACM SIGPLAN Notices*, vol. 49, no. 1. ACM, 2014, pp. 529–540.

[22] J. Li, J. Sun, L. Li, Q. L. Le, and S.-W. Lin, "Automatic loop-invariant generation and refinement through selective sampling," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2017, pp. 782–792.

[23] B. Jeannet and A. Miné, "Apron: A library of numerical abstract domains for static analysis," in *International Conference on Computer Aided Verification*. Springer, 2009, pp. 661–667.

[24] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Dig: a dynamic invariant generator for polynomial and array invariants," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 23, no. 4, p. 30, 2014.

[25] E. Rodríguez-Carbonell and D. Kapur, "Automatic generation of polynomial invariants of bounded degree using abstract interpretation," *Science of Computer Programming*, vol. 64, no. 1, pp. 54–75, 2007.

[26] S. Sankaranarayanan, H. B. Sipma, and Z. Manna, "Non-linear loop invariant generation using gröbner bases," *ACM SIGPLAN Notices*, vol. 39, no. 1, pp. 318–329, 2004.

[27] R. Sharma, S. Gupta, B. Hariharan, A. Aiken, P. Liang, and A. V. Nori, "A data driven approach for algebraic loop invariants," in *European Symposium on Programming*. Springer, 2013, pp. 574–592.
[28] Z. Kincaid, J. Cyphert, J. Breck, and T. Reps, "Non-linear reasoning for invariant synthesis," *Proceedings of the ACM on Programming Languages*, vol. 2, no. POPL, p. 54, 2017.
[29] T. Nguyen, D. Kapur, W. Weimer, and S. Forrest, "Using dynamic analysis to generate disjunctive invariants," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 608–619.
[30] A. Humenberger, M. Jaroschek, and L. Kovács, "Invariant generation for multi-path loops with polynomial assignments," in *International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer, 2018, pp. 226–246.
[31] D. Cachera, T. Jensen, A. Jobin, and F. Kirchner, "Fast inference of polynomial invariants for imperative programs," Ph.D. dissertation, INRIA, 2011.
[32] E. Rodríguez-Carbonell and D. Kapur, "Generating all polynomial invariants in simple loops," *Journal of Symbolic Computation*, vol. 42, no. 4, pp. 443–476, 2007.
[33] T. Nguyen, T. Antonopoulos, A. Ruef, and M. Hicks, "Counterexample-guided approach to finding numerical invariants," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 605–615.
[34] D. Gopan and T. Reps, "Lookahead widening," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 452–466.
[35] G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani, "Efficiently intertwining widening and narrowing," *Science of Computer Programming*, vol. 120, pp. 1–24, 2016.
[36] F. Bourdoncle, "Efficient chaotic iteration strategies with widenings," in *Formal Methods in Programming and their Applications*. Springer, 1993, pp. 128–141.
[37] E. Rodríguez-Carbonell and D. Kapur, "An abstract interpretation approach for automatic generation of polynomial invariants," in *International Static Analysis Symposium*. Springer, 2004, pp. 280–295.
[38] A. Srikanth, B. Sahin, and W. R. Harris, "Complexity verification using guided theorem enumeration," *ACM SIGPLAN Notices*, vol. 52, no. 1, pp. 639–652, 2017.
[39] Y. Chen, B. Xia, L. Yang, and N. Zhan, "Generating polynomial invariants with DISCOVERER and QEPCAD," in *Formal Methods and Hybrid Real-Time Systems 2007*, ser. Lecture Notes in Computer Science, vol. 4700. Springer, 2007, pp. 67–82.
[40] Y. Feng, L. Zhang, D. N. Jansen, N. Zhan, and B. Xia, "Finding polynomial loop invariants for probabilistic programs," in *ATVA 2017*, ser. Lecture Notes in Computer Science, vol. 10482. Springer, 2017, pp. 400–416.
[41] W. Lin, M. Wu, Z. Yang, and Z. Zeng, "Verification for non-polynomial hybrid systems using rational invariants," *Comput. J.*, vol. 60, no. 5, pp. 675–689, 2017.
[42] B. S. Gulavani and S. Gulwani, "A numerical abstract domain based on expression abstraction and max operator with application in timing analysis," in *International Conference on Computer Aided Verification*. Springer, 2008, pp. 370–384.
[43] J. Feret, "The arithmetic-geometric progression abstract domain," in *International Workshop on Verification, Model Checking, and Abstract Interpretation*. Springer, 2005, pp. 42–58.
[44] C. Le Guernic, "Toward a sound analysis of guarded lti loops with inputs by abstract acceleration (extended version)," in *Static Analysis Symposium*, vol. 10422, 2017.
[45] H. Oh, K. Heo, W. Lee, W. Lee, D. Park, J. Kang, and K. Yi, "Global sparse analysis framework," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 36, no. 3, p. 8, 2014.
[46] G. Singh, M. Püschel, and M. T. Vechev, "A practical construction for decomposing numerical abstract domains," *PACMPL*, vol. 2, no. POPL, pp. 55:1–55:28, 2018.

Banghu Yin received the M.S. degree in computer science from National University of Defense Technology, changsha, China in 2014. He is currently working toward the Ph.D. degree at National University of Defense Technology, changsha, China.

His research interests include abstract interpretation, program analysis and verification.

Liqian Chen received the Ph.D. degree in computer science from National University of Defense Technology, changsha, China in 2010.

He is currently an Associate Professor in the School of Computer Science, National University of Defense Technology, changsha, China. His research interests include abstract interpretation, program analysis and verification.

Jiangchao Liu received the Ph.D. degree in École Normale Supérieure, Paris, France in 2018.

He is currently a research associate in the School of Computer Science, National University of Defense Technology, changsha, China. His research interests include abstract interpretation, program analysis and verification.

Ji Wang received the Ph.D. degree in computer science from National University of Defense Technology, changsha, China in 1995.

He is currently a Professor in the School of Computer Science, National University of Defense Technology, changsha, China. His research interests include abstract interpretation, program analysis and verification.