Sound Bit-Precise Numerical Domains*

Tushar Sharma[1] and Thomas Reps[1,2] **

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. This paper tackles the challenge of creating a numerical abstract domain that can identify affine-inequality invariants while handling overflow in arithmetic operations over bit-vector data-types. The paper describes the design and implementation of a class of new abstract domains, called the *Bit-Vector-Sound, Finite-Disjunctive* ($BVSFD$) domains. We introduce a framework that takes an abstract domain $\mathcal{A}$ that is sound with respect to mathematical integers and creates an abstract domain $BVS(\mathcal{A})$ whose operations and abstract transformers are sound with respect to machine integers. We also describe how to create abstract transformers for $BVS(\mathcal{A})$ that are sound with respect to machine arithmetic. The abstract transformers make use of an operation $WRAP(av, v)$—where $av \in \mathcal{A}$ and v is a set of program variables—which performs wraparound in av for the variables in v.

To reduce the loss of precision from $WRAP$, we use finite disjunctions of $BVS(\mathcal{A})$ values. The constructor of finite-disjunctive domains, $FD_k(\cdot)$, is parameterized by k, the maximum number of disjunctions allowed.

We instantiate the $BVS(FD_k)$ framework using the abstract domain of *polyhedra* and *octagons*. Our experiments show that the analysis can prove 25% of the assertions in the SVCOMP loop benchmarks with $k = 6$, and 88% of the array-bounds checks in the SVCOMP array benchmarks with $k = 4$.

1 Introduction

This paper tackles the challenges of implementing a bit-precise relational domain capable of expressing program invariants. The paper describes the design and implementation of a new framework for abstract domains, called the *Bit-Vector-Sound Finite-Disjunctive* ($BVSFD$) domains, which are capable of capturing useful program invariants such as inequalities over bit-vector-valued variables.

The need for bit-vector invariants. The polyhedral domain [10] (denoted as $POLY$) is capable of expressing relational affine inequalities over rational (or real) variables. Previous research [26, 27, 42, 21, 37, 29] has also provided weaker forms

* Supported, in part, by a gift from Rajiv and Ritu Batra; by DARPA under cooperative agreement HR0011-12-2-0012; by NSF under grant CCF-0904371; DARPA MUSE award FA8750-14-2-0270 and DARPA STAC award FA8750-15-C-0082; and by the UW-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors, and do not necessarily reflect the views of the sponsoring agencies.

** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.

of polyhedral domains that are capable of expressing some affine inequalities. For instance, the octagon abstract domain [26] (denoted as *OCT*) can express only relational inequalities involving at most two variables where the coefficients on the variables are only allowed to be plus or minus one. However, the native machine-integer data-types used in programs (e.g., `int`, `unsigned int`, `long`, etc.) perform bit-vector arithmetic, and arithmetic operations wrap around on overflow. Thus, the underlying point space used in the aforementioned abstract domains does not faithfully model bit-vector arithmetic, and consequently the conclusions drawn from an analysis based on these domains are, in general, unsound, unless special steps are taken [41][8].

Example 1. The following C-program fragment incorrectly computes the average of two `int`-valued variables [5]:

```
int low, high, mid;
assume(0 <= low <= high);
mid = (low + high)/2;
assert(0<=low<=mid<=high);
```

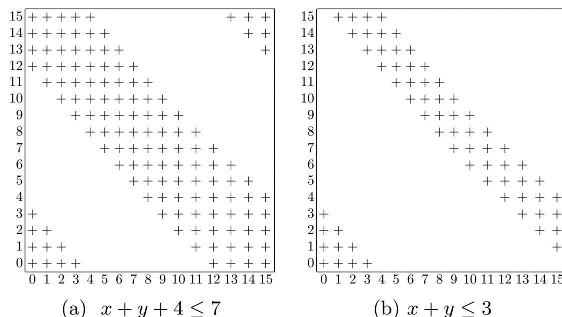
A static analysis based on polyhedra or octagons would draw the wrong conclusion that the assertion always holds. In particular, assuming 32-bit `int`s, when the sum of `low` and `high` is greater than 2^{31}-1, the sum overflows, and the resulting value of `mid` is smaller than `low`. Consequently, there exist runs in which the assertion fails. These runs are overlooked when the polyhedral domain is used for static analysis because the domain fails to take into account the bit-vector semantics of program variables. □

The problem that we wish to solve is not one of merely *detecting* overflow—e.g., to restrain an analyzer from having to explore what happens after an overflow occurs. On the contrary, our goal is to be able to track soundly the effects of arithmetic operations, including wrap-around effects of operations that overflow. This ability is useful, for instance, when analyzing code generated by production code generators, such as dSPACE TargetLink [12], which use the "compute-through-overflow" technique [14]. Furthermore, clever idioms for bit-twiddling operations, such as the ones explained in [43], sometimes rely on overflow [11].

Challenges in dealing with bit-vectors. Some of the ideas used in designing an inequality domain for reals do not carry over to ones designed for bit-vectors. First, in bit-vector arithmetic, additive constants cannot be cancelled on both sides of an inequality, as illustrated in the following example.

Example 2. Let x and y be 4-bit unsigned integers. Figures 1a and 1b depict the solutions in bit-vector arithmetic of the inequalities $x+y+4 \leq 7$ and $x+y \leq 3$, respectively. Although $x+y+4 \leq 7$ and $x+y \leq 3$ are syntactically quite similar, their solution spaces are quite different. In particular, because of wrap-around of values computed on the left-hand sides using bit-vector arithmetic, one cannot just subtract 4 from both sides to convert the inequality $x + y + 4 \leq 7$ into $x + y \leq 3$. □

Second, in bit-vector arithmetic, positive constant factors cannot be cancelled on both sides of an inequality; for example, if x and y are 4-bit bit-vectors, then $(4, 4)$ is in the solution set of $2x + 2y \leq 4$, but not of $x + y \leq 2$.



(a) $x + y + 4 \le 7$ (b) $x + y \le 3$

Fig. 1. Each + represents a solution of the indicated inequality in 4-bit unsigned bit-vector arithmetic.

While some simple domains do exist that are capable of representing certain kinds of inequalities over bit-vectors (e.g., intervals with a congruence constraint, sometimes called "strided-intervals" [34, 38, 3, 28]), such domains are non-relational; that is, they are not capable of expressing relations among several variables. On the other hand, there exist relational bit-precise domains [30, 6], but they cannot express inequalities. The abstract domain in [40] can handle certain kind of bit-vector inequalities, but it needs the client to provide a template for inequalities. Moreover, the domain is incapable of expressing simple inequalities of form $x \le y$, because they have variables on both side of the inequality.

Simon et al. [41] introduced sound wrap-around to ensure that the polyhedral domain is sound over bit-vectors. The wrap-around operation is called selectively on the abstract-domain elements while calculating the fixpoint. The operation is called selectively to preserve precision while not compromising on soundness with respect to the concrete semantics. The Verasco static analyzer [17] provides a bit-precise parametrized framework for abstract domains using the wrap-around operation. This approach has two disadvantages:

- The wrap-around operation almost always loses information due to calls on join: the convex hull of the elements that did not overflow with those that did usually does not satisfy many inequality constraints.
- They do not show how to create abstract transformers automatically.

We introduce a class of abstract domains, called $BVS(\mathcal{A})$, that is sound with respect to bitvectors whenever $\mathcal{A}$ is sound with respect to mathematical integers. The $\mathcal{A}$ domain can be any numerical abstract domain. For example, it can be the polyhedral domain, which can represent useful program invariants as inequalities. We also describe how to create abstract transformers for $BVS(\mathcal{A})$ that are sound with respect to bitvectors. For $v \subseteq Var$ and $av \in \mathcal{A}$, we denote the result by $WRAP_v(av)$; the operation performs wraparound on av for variables in v. We give an algorithm for $WRAP_v(av)$ that works for any relational abstract domain (see §4.1). We use a finite number of disjunction of $\mathcal{A}$ elements—captured in the domain $FD_k(\mathcal{A})$—to help retain precision. The finite disjunctive domain is parametrized by the maximum number of disjunctions allowed in the domain

(referred to as k). Note that k=1 is the same as convex polyhedra with wrap-around [41].

Example 3. Consider the following C-program fragment, which correctly computes the average of two `int`-valued variables [5]:

```
low = (x<=y)? x:y;
high = (x<=y)? y:x;
mid = low+(high-low)/2;
assert((x<=mid<=y)||(y<=mid<=x));
```

In this example, the domain of convex polyhedra with wrap-around is insufficient to prove the assertion. The reason is that the assertion is a disjunction of two polyhedral invariants. However, $BVS(FD_k(POLY))$ for k=2 is able to prove the assertion. □

Problem statement. These challenges lead to the following problem statement:

> *Given a relational numeric domain over integers, capable of expressing inequalities, (i) provide an automatic method to create a relational abstract domain that can capture inequalities over bit-vector-valued variables; (ii) create sound bit-precise abstract transformers; and (iii) use them to identify inequality invariants over a set of program variables.*

Related work and contributions. Our work incorporates a number of ideas known from the literature, including

- the use of relational abstract domains [10, 25, 21, 27, 42, 33] that are sound over mathematical integers and capable of expressing inequalities.
- the use of a wrap-around operation [41, 7, 2] to ensure that the abstraction is sound with respect to the concrete semantics of the bitvector operations.
- the use of finite disjunctions [36, 1, 15] over abstract domains to obtain more precision.
- the use of instruction reinterpretation [16, 31, 32, 24, 22, 13] to obtain an abstract transformer automatically for an edge from a basic block to its successor.

Our contribution is that we put all of these to work together in a parametrized framework, along with a mechanism to increase precision by performing wrap-around on abstract values lazily.

- We propose a framework for abstract domains, called $BVSFD_k(\mathcal{A})$, to express bit-precise relational invariants by performing wrap-around over abstract domain $\mathcal{A}$ and using disjunctions to retain precision. This abstract domain is parametrized by a positive value k, which provides the maximum number of disjunctions that the abstract domain can make use of.
- We provide a generic technique via reinterpretation to create the abstract transformer for the path through a basic block to a given successor, such that the transformer incorporates lazy wrap-around.
- We present experiments to show how the performance and precision of $BVSFD_k$ analysis changes with the tunable parameter k.

§2 introduces the terminology and notation used in the rest of the paper. §3 demonstrates our framework with the help of an example. §4 introduces the $BVSFD_k$ abstract-domain framework, and formalizes abstract-transformer generation for the framework. §5 presents experimental results. §6 concludes.

2 Terminology

Abstract domains and vocabularies. Let abstract domain $\mathcal{A}$ and concrete domain $\mathcal{C}$ be related by a Galois connection $\mathcal{G} = \mathcal{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{A}$. Often $\mathcal{A}$ is really a family of abstract domains, in which case $\mathcal{A}[V]$ denotes the specific instance of $\mathcal{A}$ that is defined over vocabulary V, where V is a tuple of variables $(v_1, v_2, ..., v_n)$. Each variable v_i also has an associated size in bits, denoted by $s(v_i)$. The domain $\mathcal{C}$ is the powerset of the set of concrete states.

2.1 Concretization.

Given an abstract value $A \in \mathcal{A}[V]$, where V consists of n variables $(v_1, v_2, ..., v_n)$, the concretization of A, denoted by $\gamma_{\mathcal{A}[V]}(A)$, is the set of concrete states covered by A.

A concrete state σ is a mapping from variables to their concrete values, $\sigma : V \to \Pi_{v \in V} BV^{s(v)}$, where $s(v)$ is the size of variable v in bits and BV^b is a bitvector with b bits.

$$\gamma_{\mathcal{A}[V]}(A) = \bigcup_{(a_1, a_2, ..., a_n) \in A} \mu_V(bv_1, bv_2, ..., bv_n), \textbf{where } bv_i = a_i \% 2^{s(v_i)} \textbf{ for } i \in 0..n$$

$\mu_V(bv_1, bv_2, ..., bv_n)$ takes a tuple of bitvectors corresponding to vocabulary V and returns all concrete stores where each variable v_i in V has the value bv_i.

2.2 Wrap-around operation.

The wrap-around operation, denoted by $WRAP^{ty}_{V'}(A)$, takes an abstract-domain value $A \in \mathcal{A}[V]$, a subset V' of the vocabulary V, and the desired type ty for vocabulary V'. It returns an abstract value A' such that the wrap-around behavior of the points in A is soundly captured. This operation is performed by displacing the concrete values in $\gamma_{\mathcal{A}[V]}(A)$ that are outside the bitvector range for ty in vocabulary V' to the correct bitvector range by appropriate linear transformations.

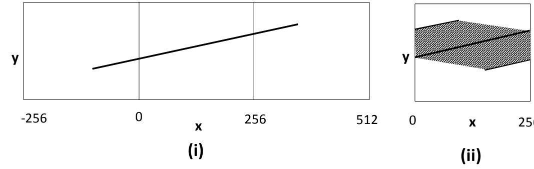


Fig. 2. Wrap-around on variable x, treated as an unsigned char.

For example, in Fig. 2, the result of calling wrap-around on the line in (i) for variable x leads to an abstract value that is the abstraction of the points in the three line segments in (ii). For the abstract domain of polyhedra, that abstraction is the shaded area in (ii).

2.3 Soundness.

An abstract value $A \in \mathcal{A}[V]$ is sound with respect to a set of concrete values $C \in \mathcal{C}$, if $\gamma_{\mathcal{A}[V]} \supseteq C$.

2.4 $\mathcal{L}(ELang)$: A Concrete language featuring finite integer arithmetic.

We borrow the simple language featuring finite-integer arithmetic from § 2.1 of [41] (with minor syntactic changes). An ELang program is a sequence of basic blocks with execution starting from the first block. Each basic block consists of a sequence of statements and a list of control-flow instructions.

$$
\begin{array}{rcl}
\langle ELang \rangle & :: & (Block)* \\
\langle Block \rangle & :: & l : (\langle Stmt \rangle \ \mathbf{;})^* \ \langle Next \rangle \\
\langle Next \rangle & :: & \mathbf{jump}\ l\mathbf{;} \\
 & | & \mathbf{if}\ v\ \langle Op \rangle_{\langle Type \rangle}\ \langle Expr \rangle\ \mathbf{then\ jump}\ l\ \mathbf{;}\ \langle Next \rangle \\
\langle Op \rangle & :: & < \mid \leq \mid = \mid \neq \mid \geq \mid > \\
\langle Expr \rangle & :: & n \mid n * v + \langle Expr \rangle \\
\langle Stmt \rangle & :: & v = \langle Expr \rangle \\
 & | & v{:}\langle Type \rangle = v{:}\langle Type \rangle \\
\langle Type \rangle & :: & (\mathbf{uint} \mid \mathbf{int})\ \langle Size \rangle \\
\langle Size \rangle & :: & \mathbf{1} \mid \mathbf{2} \mid \mathbf{4} \mid \mathbf{8}
\end{array}
$$

The statements are restricted to an assignment of a linear expression or a cast operation. The control-flow instruction consists of either a jump statement, or a conditional that is followed by more control-flow instructions. The assignment and condition instructions expect the variable and the expression involved to have the same type.

3 Overview

In this section, we motivate and illustrate the design of our analysis using the *BVSFD* domain. The two important steps in abstract interpretation (AI) are:

1. Abstraction: The abstraction of the program is constructed using the abstract domain and abstract semantics.
2. Fixpoint analysis: Fixpoint iteration is performed on the abstraction of the program to provide invariants.

In the typical setup of AI, the set of states that can arise at each program point in the program is safely represented by the abstract-domain element found as the fixed point. This setup can be used to prove assertions. When the abstract-domain elements are themselves abstract transformers, the results provide function summaries or loop summaries [9, 39]. In principle, summaries can be computed offline for large libraries of code so that client static analyses can use them to provide verification results efficiently.

A static analyzer needs a way to construct abstract transformers for the concrete operations in the programs. Reinterpretation [16, 31, 32, 24, 22, 13] provides an automatic way to construct abstract transformers. For an analysis that provides function summaries or loop summaries, the fixpoint analysis is performed using equality, join ($\sqcup$), and abstract-composition ($\circ$) operations on abstract transformers.

Example 4. This example illustrates a function f that takes two 32-bit integers x and y at different rates, and resets their values to zero in case y is negative or overflows to a negative value. The function summary that we would like to obtain states that the relationship $x' \leq y'$ holds. Here, the unprimed and primed

variables denote the pre-state vocabulary variables and the post-state vocabulary variables, respectively.

```
  L0: f(int x, int y) {
  L1:   assume(x<=y)
  L2:   while(*) {
  L3:     if(*)
  L4:       x=x+1, y=y+1
  L5:     y=y+1
  L6:     if (y<=0)
  L7:       x=0, y=0
  L8:   }
 END: }
```

This example illustrates that merely detecting overflow would not be useful to assert the $x <= y$ relationship at the end of the function. □

3.1 Creation of Abstract Transformers

Consider the analysis for Ex. 4 with the abstract domain $BVSFD_2(OCT)$. The first step involves constructing the abstraction of the concrete operations in the program as abstract transformers.

For instance, the abstract transformer for the concrete operations starting from node $L0$ and ending at node $L2$, denoted by $\tau^{\sharp}_{L0 \to L2}$, is defined as

$$\{m \leq x, y \leq M \wedge x' = x \wedge y' = y \wedge x' \leq y'\},$$

where m and M represent the minimum and maximum values for a signed 32-bit integer, respectively. The constraints $\{m \leq x, y \leq M\}$ are the bounding constraints on the pre-state vocabulary that are added because $L0$ is the entry point of the function, and the function expects three 32-bit signed values x and y as input. The equality constraints $\{x' = x, y' = y\}$ specify that the variables x and y are unchanged. Finally, the constraint $\{x' \leq y'\}$ is added as a consequence of the *assume* call.

Now consider other concrete transformations, such as $L4 \to L5$ and $L5 \to L6$. For the transformation $L4 \to L5$, the values for x' and y' might overflow because of the increment operations at $L4$. Consequently, the value of the incoming variable y in the transformation $L5 \to L6$ might have overflowed as well. There are two ways to design the abstract transformer to deal with these kind of scenarios: 1) a naive eager approach, 2) a lazy approach.

Eager Abstract Transformers. In the naive eager approach, the abstract transformers are created such that the pre-state vocabulary is always bounded as per the type requirements. For this example, that would mean that the pre-state vocabulary variables x and y are bounded in the range $[m, M]$. Consequently, the abstract transformers for $L4 \to L5$ and $L5 \to L6$ are:

- $\tau^{\sharp E}_{L4 \to L5} = \{m \leq x, y \leq M \wedge x' = x + 1 \wedge y' = y + 1\}$
- $\tau^{\sharp E}_{L5 \to L6} = \{m \leq x, y \leq M \wedge x' = x \wedge y' = y + 1\}$.

Because the eager approach expects the pre-state vocabulary to be bounded, an abstract-composition operation $a_1 \circ a_2$, where a_1 and a_2 are abstract transformers, needs to call the *WRAP* operation (§2.2) for the entire post-state vocabulary of a_2, for correctness. For instance, let $a_1 = \{m \leq u \leq M \wedge u' = u\}$ and $a_2 = \{m \leq u \leq M \wedge u' = M + 1\}$. The abstract transformer a_1 preserves u

and a_2 changes the value of u' to $M+1$. The composition of these operations matches the pre-state vocabulary of a_1 with the post-state vocabulary of a_2, by renaming them to the same temporary variables and performing a meet. For this particular example, it will perform $\{m \leq u'' \leq M \wedge u' = u''\} \sqcap \mathit{WRAP}_{u''}(\{m \leq u \leq M \wedge u'' = M+1\})$, where it has matched the pre-state vocabulary variable u of a_1 with the post-state vocabulary variable u' of a_2, by renaming them both to a temporary variable u''. Note that in the absence of the *WRAP* operation on the post-state vocabulary of a_2, the meet operation above will return the empty element $\bot$. This result would be unsound because the value of u' in a_2 should have overflowed to m.

Now consider the composition $\tau^{\sharp E}_{L5\to L6} \circ \tau^{\sharp E}_{L4\to L5}$. After matching, composition will perform the meet of:

- $\{m \leq x'', y'' \leq M \wedge x' = x'' \wedge y' = y''+1\}$
- $\mathit{WRAP}_{\{x'',y''\}}\{m \leq x, y \leq M \wedge x'' = x+1 \wedge y'' = y+1\}$

The result of *WRAP* will be a join of four values. The four values are the combinations of cases where x'' and y'' might or might not overflow. As a result, the final composition will give an abstract transformer that overapproximate those four values. For $\mathit{BVSFD}_2(\mathit{OCT})$, it will result in a loss of precision because it cannot express the disjunction of these four values precisely.

Lazy Abstract Transformers. The eager approach to creating abstract transformers forces a call to *WRAP* at each compose operation. The lazy approach can avoid unneccessary calls to *WRAP* by not adding any bounding constraints to the pre-state vocabulary. The abstract transformer $\tau^{\sharp L}_{L4\to L5}$ is defined as $\{x' = x+1 \wedge y' = y+1\}$ and $\tau^{\sharp L}_{L5\to L6}$ is defined as $\{x' = x \wedge y' = y+1\}$. The abstract transformer $\tau^{\sharp L}_{L4\to L5}$ is sound, because the concretization of the abstract transformer, denoted by $\gamma(\tau^{\sharp L}_{L4\to L5})$, overapproximates the collecting concrete semantics for $L4 \to L5$ (see proposition 1 in [41]). A similar argument can be made for the abstract transformer $\tau^{\sharp L}_{L5\to L6}$. The composition $\tau^{\sharp L}_{L5\to L6} \circ \tau^{\sharp L}_{L4\to L5}$ gives $\{x' = x+1 \wedge y' = y+2\}$. Thus, the lazy abstract transformer can retain precision by avoiding unnecessary calls to *WRAP*. However, one cannot avoid calling *WRAP* for every kind of abstract transformer and still maintain soundness. Consider the abstract transformer $\tau^{\sharp L}_{L5\to L7}$, which is similar to $\tau^{\sharp L}_{L5\to L6}$, but must additionally handle the branch condition for $L6 \to L7$. Defining it in a similar vein as $L4 \to L5$ will result in $\{x' = x \wedge y' = y+1 \wedge y' \leq 0\}$. While the first three constraints are sound, the fourth constraint representing the branch condition, is unsound with respect to the concrete semantics. The reason is that y' might overflow to a negative value, in which case the condition evaluates to true and the branch to $L7$ is taken. However, the abstract transformer does not capture that behavior and is, therefore, unsound with respect to the concrete semantics. To achieve soundness in the presence of a branch condition, the following steps are performed for each variable v' involved in a branch condition:

- A backward dependency analysis is performed to find the subset V_b of the pre-state vocabulary on which v' depends. For the edge $L5 \to L6$, only y' is involved in a branch condition. The backward-dependency analysis yields

$V_b = \{y\}$, because the only pre-state vocabulary variable that y' depends on is y.

- The bounding constraints for the vocabulary V_b are added to the abstract transformer. For example, after adding bounding constraints, $\tau^{\sharp L}_{L5 \to L6}$ becomes $\{m \leq y \leq M \wedge x' = x \wedge y' = y + 1 \wedge y' \leq 0\}$.
- The wrap operation is called on the abstract transformer for the variable v'. For the edge $L5 \to L6$, this step will soundly set the abstract transformer to the disjunction of
  - $\{m \leq y \leq M - 1 \wedge x' = x \wedge y' = y + 1 \wedge y' \leq 0\}$
  - $\{y = M - 1 \wedge x' = x \wedge y' = m\}$

  Note that the abstract domain $BVSFD_2(OCT)$ can precisely express the above disjunction because the number of disjunctions is ≤ 2.

Our analysis uses the lazy approach to create abstract transformers, because it can provide more precise function summaries. Fig. 3 illustrates the lazy abstract transformers generated for Ex. 4.

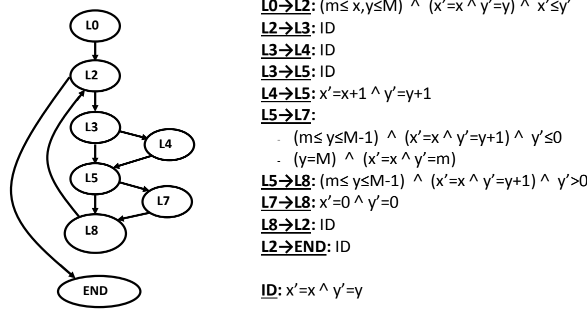


Fig. 3. Lazy abstract transformers with the $BVSFD_2(POLY)$ domain for Ex. 4. ID refers to the identity transformation.

3.2 Fixed-point computation

To obtain function summaries, an iterative fixed-point computation needs to be performed. Tab. 1 provides some snapshots of the fixpoint analysis with the $BVSFD_2(OCT)$ domain for Ex. 4.

To simplify the discussion, we focus only on three program points: $L2$, $L5$, and $L7$. Each row in the table shows the intermediate value of path summaries from $L0$ to each of the three program points. Quiescence is discovered during the fifth iteration. The abstract value in row (i) and column $L2$ shows the intermediate path summary for $L2$ calculated after one iteration of the analysis. It states that the pre-state vocabulary variables x and y are bounded and neither of them has been modified, because at this iteration the analysis has not considered the paths that go through the loop. At row (i) and column $L5$, the domain precisely captures the disjunction of two paths arising at the conditional at $L3$. The abstract value at row (i), column $L7$ is obtained as the composition of the abstract transformer $L5 \to L7$ with the path summary at $L5$ in row (i). The abstract-composition operations for abstract values with disjunctions performs abstract composition for all pairs of abstract transformers in the arguments, and

Table 1. Snapshots in the fixed-point analysis for Ex. 4 using the $BVSFD_2(OCT)$ domain. $B_{v_1,v_2,..,v_n}$ are the bounding constraints for the variables v_1,v_2,..v_n.

Node	L2	L5	L7
(i)	• $B_{x,y} \wedge (x' = x) \wedge (y' = y)$	• $B_{x,y} \wedge (x' = x) \wedge (y' = y)$ • $B_{x,y} \wedge (x' = x + 1) \wedge (y' = y + 1)$	• $B_{x,y} \wedge (m \leq y' \leq 0) \wedge (x \leq x' \leq x + 1) \wedge (y \leq y' \leq y + 2) \wedge (x' \leq y')$ • $B_{x,y} \wedge (m \leq y' \leq m + 1) \wedge (x \leq x' \leq x + 1)$
(ii)	• $B_{x,y,y'} \wedge (x \leq x' \leq x + 1) \wedge (y \leq y' \leq y + 2) \wedge (x' \leq y')$ • $B_{x,y} \wedge (x' = 0) \wedge (y' = 0)$	• $B_{x,y} \wedge (x \leq x' \leq x + 2) \wedge (y \leq y' \leq y + 3) \wedge (x' \leq y')$ • $B_{x,y} \wedge (0 \leq x' \leq 1) \wedge (y' = x')$	• $B_{x,y} \wedge (m \leq y' \leq 0) \wedge (x \leq x' \leq x + 2) \wedge (y \leq y' \leq y + 4) \wedge (x' \leq y')$ • $B_{x,y} \wedge (m \leq y' \leq m + 3) \wedge (x \leq x' \leq x + 2)$
(iii)	• $B_{x,y,y'} \wedge (x \leq x' \leq x + 2) \wedge (y \leq y' \leq y + 4) \wedge (x' \leq y')$ • $B_{x,y} \wedge (0 \leq x' \leq 1) \wedge (0 \leq y' \leq 2) \wedge (x' \leq y')$	• $B_{x,y} \wedge (x \leq x' \leq x + 3) \wedge (y \leq y' \leq y + 5) \wedge (x' \leq y')$ • $B_{x,y} \wedge (0 \leq x' \leq 2) \wedge (0 \leq y' \leq 4) \wedge (x' \leq y')$	• $B_{x,y} \wedge (m \leq y' \leq 0) \wedge (x \leq x' \leq x + 3) \wedge (y \leq y' \leq y + 6) \wedge (x' \leq y')$ • $B_{x,y} \wedge (m \leq y' \leq m + 5) \wedge (x \leq x' \leq x + 3)$
(iv)	• $B_{x,y,y'} \wedge (x \leq x') \wedge (y \leq y') \wedge (x' \leq y')$ • $B_{x,y,y'} \wedge (0 \leq x') \wedge (x' \leq y')$	• $B_{x,y} \wedge (x \leq x') \wedge (y \leq y') \wedge (x' \leq y')$ • $B_{x,y} \wedge (0 \leq x') \wedge (x' \leq y')$	• $B_{x,y} \wedge (m \leq y' \leq 0) \wedge (x \leq x') \wedge (y \leq y') \wedge (x' \leq y')$ • $B_{x,y,y'} \wedge (m \leq y' \leq 0)$

then does a join on that set of values. To obtain the abstract transformer for $L7$ in row (i), it computes the join of the following values:

1. $B_x \wedge (m \leq y \leq M - 1) \wedge (x' = x) \wedge (y' = y + 1) \wedge (y' \leq 0)$
2. $B_x \wedge (m \leq y \leq M - 2) \wedge (x' = x + 1) \wedge (y' = y + 2) \wedge (y' \leq 0)$
3. $B_x \wedge (y = M) \wedge (x' = x) \wedge (y' = m)$
4. $B_x \wedge (M \leq y \leq M - 1) \wedge (x' = x) \wedge (m \leq y' \leq m + 1)$

Our abstract-domain framework uses a distance heuristic (see §4.1) to merge abstract values that are closest to each other. For this particular case, the abstract transformers (1) and (2) describe the scenarios where y' does not overflow, and are merged to give the first disjunct of row (i), column $L7$. Similarly, the abstract transformers (3) and (4) describe the scenarios where y' overflows, and are merged to give the second disjunct of row (i), column $L7$.

In the second iteration, shown in row (ii), the first disjunct for $L2$ is the join of the effect of first iteration of the loop, where x and y are incremented, with the old value, where x and y are unchanged. Additionally, the second disjunct in row (ii), column $L2$ captures the case where both x and y are set to 0 at program point $L7$. Iteration (iii) proceeds in a similar manner, and finally the value at $L2$ saturates due to widening. The value of $L2$ at iteration (iv) is propagated to the end of the function to give the following function summary:

- $B_{x,y,y'} \wedge (x \leq x') \wedge (y \leq y') \wedge (x' \leq y')$
- $B_{x,y,y'} \wedge (0 \leq x') \wedge (x' \leq y')$

Algorithm 1 Wrap for a single variable

```
1: function WRAP(a, v, ty)
2:     if a is ⊥ then
3:         return ⊥
4:     (m, M) ← Range(ty)
5:     s ← (M − m) + 1
6:     [l, u] ← GetBounds(a, v)
7:     if l ≠ −∞ ∧ u ≠ ∞ then
8:         ⟨q_l, q_u⟩ ← ⟨⌊(l − m)/s⌋ , ⌊(u − m)/s⌋⟩
9:     b ← C(m ≤ v) ⊓ C(v ≤ M)
10:    if l = −∞ ∨ u = ∞ ∨ (q_u − q_l) > t then
11:        return RM_{v}(a) ⊓ b
12:    else
13:        return ⋃_{q∈[q_l,q_u]} ((a ▷ v := v − qs) ⊓ b)
```

Type	Operation	Description
$\mathcal{A}$	$\top$	*top element*
$\mathcal{A}$	$\bot$	*bottom element*
bool	$(a_1 == a_2)$	*equality*
$\mathcal{A}$	$(a_1 \sqcap a_2)$	*meet*
$\mathcal{A}$	$(a_1 \sqcup a_2)$	*join*
$\mathcal{A}$	$(a_1 \nabla a_2)$	*widen*
$\mathcal{A}$	$\pi_W(a)$	*project on vocabulary W*
$\mathcal{A}$	$RM_W(a)$	*remove vocabulary W*
$\mathcal{A}$	$\rho(a_1, v_1, v_2)$	*rename variable v_1 to v_2*
$\mathcal{A}$	$\mathcal{C}(le_1 \ op \ le_2)$	*construct abstract value*
$set[\mathcal{A}]$	$WRAP_W^{ty}(a_1)$	*wrap vocabulary W*
D	$\mathcal{D}(a_1, a_2)$	*distance*

Fig. 4. Abstract-domain interface for $\mathcal{A}$.

Thus, the function summary enables us to establish that $x' \leq y'$ is true at the end of the function.

4 The *BVSFD* Abstract-Domain Framework

In this section, we present the intuition and formalism behind the design and implementation of the *BVSFD* abstract-domain framework.

4.1 Abstract-Domain Constructors

BVSFD uses of the following abstract-domain constructors:

- Bit-Vector-Sound Constructor: This constructor, denoted by $BVS[\mathcal{A}]$, takes an arbitrary abstract domain and constructs a bit-precise version of the domain that is sound with respect to the concrete semantics. It needs the base domain $\mathcal{A}$ to provide a *WRAP* operator.
- Finite-Disjunctive Constructor: This constructor, denoted by $FD_k[\mathcal{A}]$, takes an abstract domain $\mathcal{A}$ and a parameter k, and constructs a finite-disjunctive version of the domain, where the number of disjunctions in any abstract value should not exceed k. This constructor uses a distance measure, denoted by $\mathcal{D}$, to determine which disjuncts are combined when the number of disjunctions exceeds k.

The $BVSFD_k[\mathcal{A}]$ domain is constructed as $BVS[FD_k[\mathcal{A}]]$. Fig. 4 shows the interface that the base abstract domain $\mathcal{A}$ needs to provide to instantiate the $BVSFD_k[\mathcal{A}]$ framework.

The first seven operations are standard abstract-domain operations. The remove-vocabulary operation $RM_W(\mathcal{A})$, can be implemented as $\pi_{V-W}(\mathcal{A})$, where V is the full vocabulary. The rename operation $\rho(\mathcal{A}, v_1, v_2)$ can be easily implemented in most abstract-domain implementations through simple variable renaming and/or variable-order permutation. The construct operation, denoted by $\mathcal{C}$, constructs an abstract value from the linear constraint $le_1 = le_2$, where le_1 and le_2 are linear expressions, and operation op $\in \{=, \leq, \geq\}$. This operation is available for any numeric abstract domain that can capture linear constraints. If the domain cannot express a specific type of linear constraint (for instance, the octagon domain cannot express linear constraints with more than two variables),

it can safely return $\top$. The *WRAP* operation is similar to the wrap operation in [41], except that it returns a set of abstract-domain values, whose disjunction correctly captures the wrap-around behavior. The *WRAP* operation from [41] is modified to return a *set* of abstract-domain values by placing values in a set instead of calling join. Alg. 1 shows how wrap is performed for a single variable. It takes the abstract value a and perform wrap-around on variable v, treated as type ty. Line 4 obtains the range for a type, and line 5 calculates the size of that range. Line 6 obtains the range of v in abstract value a. This operation can be implemented by projecting a on v and reading the resultant interval. Lines 7-8 calculate the range of the quadrants for the variable v. Line 9 computes the bounding constraints on v, treated as type ty. Line 10 compares the number of quadrants to a threshold t. If the number of quadrants exceeds t, the result is computed by removing constraints on v in a using the RM operation, and adding the bounding constraints to the final result. Otherwise, for each quadrant, the appropriate value is computed by displacing the quadrants to the correct range. The displacing of the abstract value a for the quadrant q, denoted by $a \rhd v := v - qs$, is implemented as $RM_{\{u\}}(\rho(a, v, u) \sqcap \mathcal{C}(v = u - qs))$. We used $t = 16$ in the experiments reported in §5.

We implement $\mathcal{D}(a_1, a_2)$ by converting a_1 and a_2 into the strongest boxes b_1 and b_2 that overapproximate a_1 and a_2, and computing the distance between b_1 and b_2. A box is essentially a conjunction of intervals on each variable in the vocabulary. We measure the distance between two boxes as a tuple (d_1, d_2), where d_1 is the number of incompatible intervals, and d_2 is the sum of the distances between *compatible* intervals. Two intervals are considered to be *incompatible* if one is unbounded in a direction that the other one is not. For example, intervals $[0, \infty]$ and $[-7, \infty]$ are compatible, but $[0, 17]$ and $[-7, \infty]$, and $[-\infty, 12]$ and $[-7, \infty]$ are not. The *distance* between two compatible intervals is 0 if their intersection is non-empty; otherwise, it is the difference of the lower bound of the higher interval and the upper bound of the lower interval. For example, the distance between $[0, 11]$ and $[17, 21]$ is $(17 - 11) = 6$. Given two distances $d = (d_1, d_2)$ and $d' = (d'_1, d'_2)$, $d > d'$ iff either (i) $d_1 > d'_1$, or (ii) $d_1 = d'_1 \wedge d_2 > d'_2$. If the number of disjunctions in an abstract value exceeds parameter k, the abstract-domain constructor $FD_k[\mathcal{A}]$ merges (using join) the pair of abstract-domain elements that are closest as measured by the distance measure.

4.2 Abstract Transformers

In this section, we describe how the abstract transformers are generated using reinterpretation [16, 31, 32, 24, 22]. The reinterpretation consists of a domain of abstract transformers $BVSFD_k[\mathcal{A}[V; V']]$, a domain of abstract integers $BVSFD_k^{\mathrm{INT}}[\mathcal{A}[t; V]]$, and operations to lookup a variable's value in the post-state of an abstract transformer and to create an updated version of a given abstract transformer [13]. Here, V denotes the pre-state vocabulary variables, V' denotes the post-state vocabulary variables, and t denotes a temporary variable not in V or V'. Given blocks $B : [l : s_1; ...s_n; nxt]$ and $B' : [l' : t'; ...t_n; nxt]$ in an *ELang* program (see §2.4), where B' is a successor of B, reinterpretation of B can provide an abstract transformer for the transformation that starts from the

first instruction in B and ends in the first instruction in B', denoted by $B \rightarrow B'$.

Rule 1 in Fig. 5 specifies how abstract-transformer evaluation for basic-block pairs feeds into abstract-transformer evaluation on a sequence of statements. The evaluation on a sequence of statements starts with the identity abstract transformer, denoted by *id*. Rule 2 states that the abstract transformer for a sequence of instruction can be broken down into an abstract transformer for a smaller sequence of instruction, by recursively performing statement-level abstract interpretation $[\![.]\!]^{\sharp}_{Stmt}$ on the first instruction in the sequence. In this rule and subsequent $[\![.]\!]^{\sharp}_{Next}$ and $[\![.]\!]^{\sharp}_{Stmt}$ rules, "a" denotes the intermediate abstract transformer value. It starts as *id* at the beginning of the instruction sequence, and gets updated or accessed by assignment and control-flow statements in the sequence.

Rules 3,4, and 5 handle control-flow statements. Rule 3 delegates the responsibility of executing the last instruction in the statement sequence to $[\![.]\!]^{\sharp}_{Next}$. Rule 4 deals with unconditional-jump instructions. The label is checked against a goal label and either $\perp$ or the current transformer a is returned accordingly. Rule 5 handles conditional branching. It conjoins the input transformer with p in the true case and n in the false case. p and n are calculated by performing abstract versions of op and $!op$, respectively, on the sound abstract integers corresponding to v_{Int} and v_{exp}. Here, $!op$ denotes the negation of the op symbol. For example, negation of $\leq$ is $>$. The sound version of an abstract integer is created by calling $LazyWrap^{type}$ (see rule 20). This function is the key component behind lazy abstract-transformer generation (see §3.1). In our implementation, we compute $DependentVoc_t(i)$ by looking at the constraints in i and returning the vocabulary subset $d \subseteq V$ that depend on t. B_d refers to the bounding constraints on the variables in vocabulary d.

Rules 6 and 7 handle assignment statements. Assignment to a linear expression merely performs a post-state-vocabulary update on the current abstract transformer "a." Note that this rule does not call $WRAP$ even though the result of computing exp can go out of bounds. Rule 7 handles the cast operation. $s(t_1)$ and $s(t_2)$ gets the size for the types t_1 and t_2, respectively. For a downcast operation, it performs simple update. In the case of upcast, $LazyWrap^{type}$ is called to preserve soundness (see Section 6 of [41]).

Rules 8, 9, 10, and 11 handle reinterpretation of expressions. Rules 8, 10, and 11 delegate computation to the corresponding abstract-integer operations. Rule 9 performs a variable lookup in the current value of abstract transformer "a."

Rules 12 to 20 deal with the operations on abstract integers in $BVSFD_k^{\mathrm{INT}}[\mathcal{A}[t;V]]$. Rule 12 constructs an abstract integer from a constant. Rule 13 finds out if a variable is a constant. This operation is used by abstract multiplication (Rule 14) to determine if the multiplication of two abstract integers is linear or not. Rules 14-18 use vocabulary-removal(RM) and variable-rename operations (ρ) to ensure that the vocabulary of the output is $\{t\} \cup V$.

Rules 21 and 22 are variable lookup and update operations. Lookup takes an abstract transformer $a \in BVSFD_k$ and a variable $v' \in V'$, and returns the abstract integer $i \in BVSFD_k^{\mathrm{INT}}$ such that the relationship of t with V in i is the

Basic Block:

$$[\![B \rightarrow B']\!]^\sharp_{Block} = [\![[s_1;...;s_n;nxt]]\!]^\sharp_{Seq}(id, l') \tag{1}$$

$$[\![[s_1;...;s_n;nxt]]\!]^\sharp_{Seq}(a, l') = [\![[s_2;...;s_n;nxt]]\!]^\sharp_{Seq}(([\![s_1]\!]^\sharp_{Stmt} a), l') \tag{2}$$

Control Flow:

$$[\![[nxt]]\!]^\sharp_{Seq}(a, l') = [\![nxt]\!]^\sharp_{Next}(a, l') \tag{3}$$

$$[\![\textbf{jump } l'']\!]^\sharp_{Next}(a, l') = \textbf{ if } l'' \textit{ is } l' \textbf{ then } a \textbf{ else } \bot \tag{4}$$

$$[\![\textbf{if } v\ op_{type}\ exp \textbf{ then jump } l''; nxt]\!]^\sharp_{Next}(a, l') = \tag{5}$$

$$\textbf{if } l'' \textit{ is } l' \textbf{ then } a \sqcap p \textbf{ else } a \sqcap n, \textbf{where}$$

$$p = v_{Int}\ op_{type}\ exp_{Int},\ n = v_{Int}\ !op\ exp_{Int},$$

$$v_{Int} = LazyWrap^{type}([\![v]\!]^\sharp_{Expr} a),\ exp_{Int} = LazyWrap^{type}([\![exp]\!]^\sharp_{Expr} a)$$

Assignments:

$$[\![v = exp]\!]^\sharp_{Stmt} a = update(a, v', [\![exp]\!]^\sharp_{Expr} a) \tag{6}$$

$$[\![v_1 : t_1 = v_2 : t_2]\!]^\sharp_{Stmt} a = \textbf{if } s(t_1) \le s(t_2) \tag{7}$$

$$\textbf{then } update(a, v_1', [\![v_2]\!]^\sharp_{Expr} a)$$

$$\textbf{else } update(a, v_1', LazyWrap^{t_1}([\![v_2]\!]^\sharp_{Expr} a))$$

Expressions:

$$[\![n]\!]^\sharp_{Expr} a = const_int(n) \tag{8}$$

$$[\![v]\!]^\sharp_{Expr} a = lookup(v', a) \tag{9}$$

$$[\![exp_1 * exp_2]\!]^\sharp_{Expr} a = mult([\![exp_1]\!]^\sharp_{Expr} a, [\![exp_2]\!]^\sharp_{Expr} a) \tag{10}$$

$$[\![exp_1 + exp_2]\!]^\sharp_{Expr} a = add([\![exp_1]\!]^\sharp_{Expr} a, [\![exp_2]\!]^\sharp_{Expr} a) \tag{11}$$

Abstract Integers:

$$const_int(n) = \mathcal{C}(t = n) \tag{12}$$

$$get_const(i) = \textbf{if } \pi_{\{t\}}(i) \textbf{ is } \{t = n\} \textbf{ then } (true, n) \textbf{ else } (false, 0), \tag{13}$$

$$mult(i_i, i_2) = \textbf{let } (b, n) = get_const(i_i) \textbf{ in} \tag{14}$$

$$(\textbf{if } b \textbf{ then } RM_{\{t'\}}(\rho(i_1, t, t') \sqcap \mathcal{C}(t = n * t')) \textbf{ else } \top)$$

$$add(i_i, i_2) = RM_{\{t', t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t = t' + t'')\big) \tag{15}$$

$$i_i\ op\ i_2 = RM_{\{t', t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t'\ op\ t'')\big), \tag{16}$$

$$\textbf{where } op \in \{=, \le, \ge\}$$

$$i_i > i_2 = RM_{\{t', t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t' \ge t'' + 1)\big) \tag{17}$$

$$i_i < i_2 = RM_{\{t', t''\}}\big(\rho(i_1, t, t') \sqcap \rho(i_2, t, t'') \sqcap \mathcal{C}(t' \le t'' - 1)\big) \tag{18}$$

$$i_i \ne i_2 = (i_i < i_2) \sqcup (i_1 > i_2) \tag{19}$$

$$LazyWrap^{type}(i) = WRAP^{type}_{\{t\}}(i \sqcap \mathcal{C}(B_d)), \textbf{where } d = DependentVoc_t(i) \tag{20}$$

Variable lookup and update:

$$lookup(a, v') = \pi_{V \cup \{t\}}(a \sqcap \{t = v'\}) \tag{21}$$

$$update(a, v', i) = RM_{\{v'\}}(a) \sqcap RM_{\{t\}}(i \sqcap \{v' = t\}) \tag{22}$$

Fig. 5. Reinterpretation semantics for $\mathcal{L}(ELANG)$.

same as the relationship of v' with V in a. The variable-update operation works in the opposite direction. Update takes an abstract transformer $a \in BVSFD_k$, a variable $v' \in V'$, and $i \in BVSFD_k^{\text{INT}}$, and returns $a' \in BVSFD_k$ such that the relationship of v' with V in a' is the same as the relationship of t with V in i, and all the other relationships that do not involve v' remain the same.

5 Experimental Evaluation

In this section, we compare the performance and precision of the bit-precise disjunctive-inequality domain $BVSFD_k$ for different values of k. We perform this comparison for the base domains of octagons and polyhedra. The abstract transformers for the *BVSFD* domain were automatically synthesized for each path through a basic block to one of its successor by using reinterpretation (see §4.2). We also perform array-out-of-bounds checking to quantify the usefulness of the precision gain for different values of k. The experiments were designed to shed light on the following questions:

1. How much does the performance of the analysis degrade as k is increased?
2. How much does the precision of the analysis increase as k is increased?
3. What is the value of k beyond which no further precision is gained?
4. What is the effect of adding sound bit-precise handling of variables on performance and precision?

5.1 Experimental Setup

Given a C file, we first create the corresponding LLVM bitcode [20, 23]. We then feed the bitcode to our solver, which uses the WALi [18] system to create a Weighted Pushdown System (WPDS) corresponding to the LLVM CFG. The transitions in the WPDS correspond to CFG edges from a basic block to one of its successors. The semiring weights on the edges are abstract transformers in the $BVSFD_k$ abstract domain. We then perform interprocedural analysis by performing post* followed by the path-summary operation [35] to obtain overapproximating function summaries and an overapproximation of the reachable states at all branch points. We used EWPDS merge functions [19] to preserve local variables across call sites. We used the *Pointset_Powerset* [1] framework in the Parma Polyhedra Library [2, 33] to implement the $FD_k[\mathcal{A}]$ constructor. For each example used in the experiments, we use a timeout of 200 seconds.

5.2 Assertion Checking

For this set of experiments, we picked the subset of the SVCOMP [4] loop benchmarks for which all assertions hold. Because our analyzer is sound, we are interested in the percentage of true assertions that it can verify. Tab. 2 provides information about the benchmarks that we used. We performed the analysis on these examples and performed assertion checking by checking whether the program points corresponding to assertion failures had the bottom abstract state.

Fig. 6a and Fig. 6b show the performance and precision numbers, respectively, for the loop SVCOMP benchmarks, with *POLY* as the base domain. The results answer the experimental questions as follows:

1. With two exceptions, at $k = 2$ and $k = 4$, the performance steadily decreases as the number of maximum allowed disjunctions k is increased. The analysis

Table 2. Information about the loop benchmarks containing true assertions, a subset of the SVCOMP benchmarks.

Benchmark	examples	instructions	assertions
loop-invgen	18	2373	90
loop-lit	15	1173	16
loops	34	3974	32
loop-acceleration	19	1001	19
total	86	8521	158

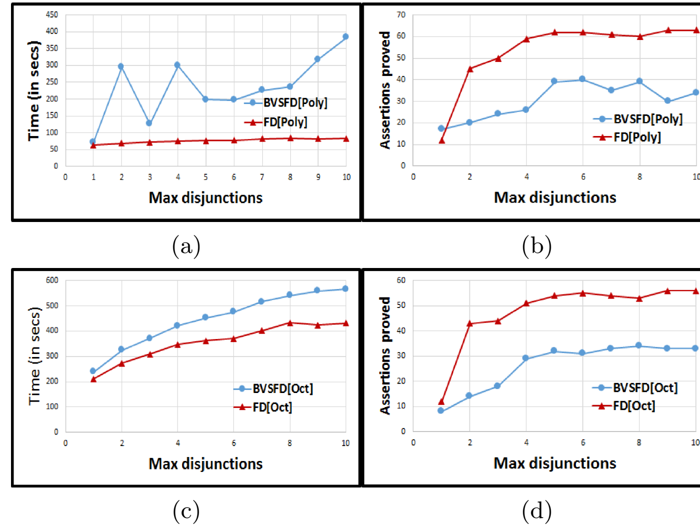


Fig. 6. Precision and performance numbers for SV-COMP loop benchmarks.

times for $k = 2$ and $k = 4$ do not fit the trend because one example times out for $k = 2$ or $k = 4$, but does not time out for $k = 3$ or $k = 5$. This behavior can be attributed to the non-monotonic behaviors of the finite-disjunctive join and widening operations.

2. The precision, measured as the number of proved assertions, increases from $k = 1$ to $k = 6$. From $k = 7$ onwards the change in precision is haphazard.
3. The analysis achieves the best precision at $k = 6$, where it proves 40 out of 157 assertions.
4. The sound analysis using $BVSFD_k[POLY]$ is 1.1-4.6 times slower than the unsound analysis using $FD_k[POLY]$, and is able to prove 44-142% of the assertions obtained with $FD_k[POLY]$.

Fig. 6c and Fig. 6d show the performance and precision numbers, respectively, for the loop SVCOMP benchmarks, with OCT as the base domain. The results answer the experimental questions as follows:

1. The performance steadily decreases with increase in k.
2. The precision, measured as the number of proved assertions, increases from $k = 1$ to $k = 5$. From $k = 5$ onwards the precision is essentially unchanged.
3. The analysis achieves the best precision at $k = 8$, where it proves 34 out of 157 assertions.

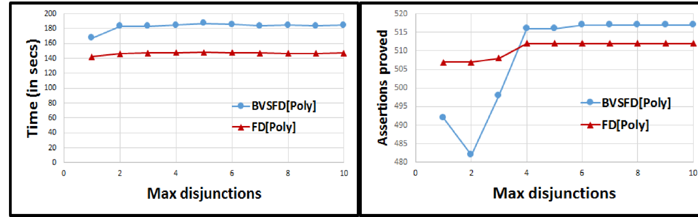


Fig. 7. Precision and performance numbers for SV-COMP array benchmarks with *POLY* as the base domain.

4. The sound analysis using $BVSFD_k[OCT]$ is 1.1-1.3 times slower than the unsound analysis using $FD_k[OCT]$, and is able to prove 33-67% of the assertions obtained with $FD_k[OCT]$.

In our experiments, we found the $BVSFD_k[OCT]$-based analysis to be 1-3.3x slower than $BVSFD_k[POLY]$. This slowdown occurs because the maximum vocabulary size in abstract transformers is ≤ 12, and abstract operations for octagons are slower than that of polyhedra for such a small vocabulary size.

5.3 Array-Bounds Checking

We perform array-bound checking using invariants from the $BVSFD_k$ analysis. For each array access and update we create an error state that is reached when an array bound is violated. These array-bounds checks are verified by checking if the path summaries at the error states are $\perp$. There are 88 examples in the benchmark, with a total of 14,742 instructions and 598 array-bounds checks. Fig. 7 lists the number of array-bound checks proven for each application, for different values of k, for the SVCOMP array benchmarks.

The results answer the experimental questions as follows:

1. The performance of the analysis increases by 9% from $k = 1$ to $k = 2$. After $k = 2$, the performance stabilizes.
2. With one exception at $k = 2$, the precision—measured as the number of array-bounds checks proven—increases from $k = 1$ to $k = 4$. From $k = 4$ onwards the precision is essentially unchanged.
3. The analysis achieves the best precision at $k = 4$, where it proves 515 out of the 598 array-bounds checks.
4. The sound analysis using $BVSFD_k[POLY]$ is 1.18-1.26 times slower than the unsound analysis using $FD_k[POLY]$, and is able to prove 95-101% of the array-bound checks obtained with $FD_k[POLY]$.

6 Conclusion

The key contribution of the paper is to provide a framework for abstract domains that not only expresses bit-precise relational invariants, but enables improved precision by using a finite number of disjunctions. The maximum number of allowed disjunctions k in the analysis can be customized by the user. This framework, denoted by $BVSFD_k$, can be instantiated for any numerical abstract domain that can handle simple inequalities. We also provide a generic approach to create sound abstract transformers for $BVSFD_k$. Our experimental results with *polyhedra* and *octagons* illustrate the practical benefits of the approach.

References

1. R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. *STTT*, 8(4/5):449–466, 2006.
2. R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
3. G. Balakrishnan and T. Reps. WYSINWYX: What You See Is Not What You eXecute. *TOPLAS*, 2010.
4. D. Beyer. Software verification and verifiable witnesses - (report on sv-comp 2015). In *TACAS*, 2015.
5. J. Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. "googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html".
6. J. Brauer and A. King. Automatic abstraction for intervals using Boolean formulae. In *SAS*, 2010.
7. J. Brauer and A. King. Transfer function synthesis without quantifier elimination. *LMCS*, 8(3), 2012.
8. S. Bygde, B. Lisper, and N. Holsti. Fully bounded polyhedral analysis of integers with wrapping. In *Int. Workshop on Numerical and Symbolic Abstract Domains*, 2011.
9. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proc. 2nd. Int. Symp on Programming*, Paris, Apr. 1976.
10. P. Cousot and N. Halbwachs. Automatic discovery of linear constraints among variables of a program. In *POPL*, 1978.
11. W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *ICSE*, 2012.
12. dSPACE TargetLink. www.dspace.com/en/pub/home/products/sw/pcgs/targetli.cfm.
13. M. Elder, J. Lim, T. Sharma, T. Andersen, and T. Reps. Abstract domains of affine relations. *TOPLAS*, 2014.
14. H. L. Garner. Theory of computer addition and overflow. *IEEE Trans. on Computers*, C-27(4), Apr. 1978.
15. K. Ghorbal, F. Ivančić, G. Balakrishnan, N. Maeda, and A. Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In *VMCAI*, 2012.
16. N. Jones and A. Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *POPL*, pages 296–306, 1986.
17. J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie. A formally-verified c static analyzer. In *POPL*, 2015.
18. N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.
19. A. Lal, T. Reps, and G. Balakrishnan. Extended weighted pushdown systems. In *CAV*, 2005.
20. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Int. Symp. on Code Generation and Optimization*, 2004.
21. V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *VMCAI*, 2009.
22. J. Lim. *Transformer Specification Language: A system for generating analyzers and its applications.* PhD thesis, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, May 2011. Tech. Rep. 1689.

23. LLVM: Low level virtual machine. llvm.org.
24. K. Malmkjær. *Abstract Interpretation of Partial-Evaluation Algorithms.* PhD thesis, Dept. of Comp. and Inf. Sci., Kansas State Univ., Manhattan, Kansas, 1993.
25. A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 1006.
26. A. Miné. The octagon abstract domain. In *WCRE*, 2001.
27. A. Miné. A few graph-based relational numerical abstract domains. In *SAS*, pages 117–132, 2002.
28. A. Miné. Abstract domains for bit-level machine integer and floating-point operations. In *IJCAR*, pages 55–70, 2012.
29. D. Monniaux. Automatic modular abstractions for template numerical constraints. *LMCS*, 6(3), 2010.
30. M. Müller-Olm and H. Seidl. Analysis of modular arithmetic. *TOPLAS*, 2007.
31. A. Mycroft and N. Jones. A relational framework for abstract interpretation. In *Programs as Data Objects*, 1985.
32. F. Nielson. Two-level semantics and abstract interpretation. *Theor. Comp. Sci.*, 69:117–242, 1989.
33. PPL: The Parma polyhedra library. www.cs.unipr.it/ppl/.
34. T. Reps, G. Balakrishnan, and J. Lim. Intermediate-representation recovery from low-level code. In *PEPM*, 2006.
35. T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *SCP*, 58(1–2), 2005.
36. S. Sankaranarayanan, F. Ivančić, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS*, 2006.
37. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, 2005.
38. R. Sen and Y. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *MEMOCODE*, 2007.
39. M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.
40. T. Sharma, A. Thakur, and T. Reps. An abstract domain for bit-vector inequalities. TR-1789, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, 2013.
41. A. Simon and A. King. Taming the wrapping of integer arithmetic. In *SAS*, 2007.
42. A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *Int. Workshop on Logic Based Prog. Dev. and Transformation*, pages 71–89, 2002.
43. H. Warren, Jr. *Hacker's Delight.* Addison-Wesley, 2003.