Sound Bit-Precise Numerical Domains

Tushar Sharma

University of Wisconsin

THE UNIVERSITY of WISCONSIN MADISON

Thomas Reps

University of Wisconsin

GrammaTech

Verification via Abstract Interpretation

Program

↓ *Abstract Domain*

Numerical Abstract Domains (Polyhedra, Octagons, ..)

Abstraction

↓ *Fixpoint Analysis*

UNSOUND!

Program Invariants

Incorrect midpoint example

unsigned low , high , mid;

assume (low <= high);

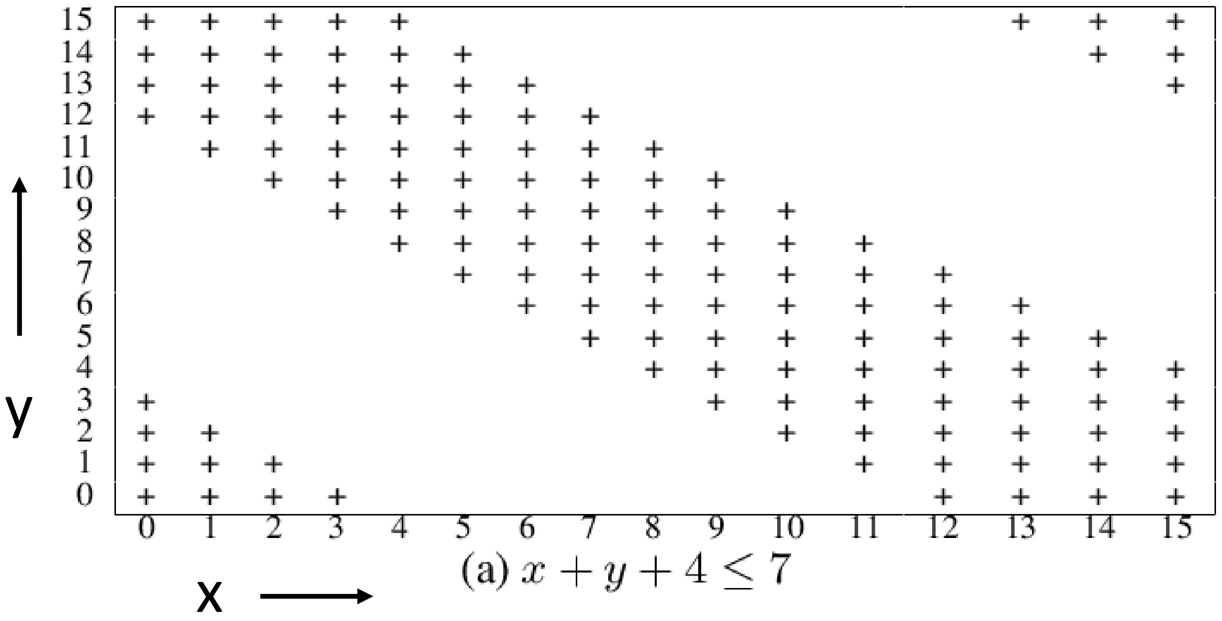
mid = (low + high) /2;

Possible overflow

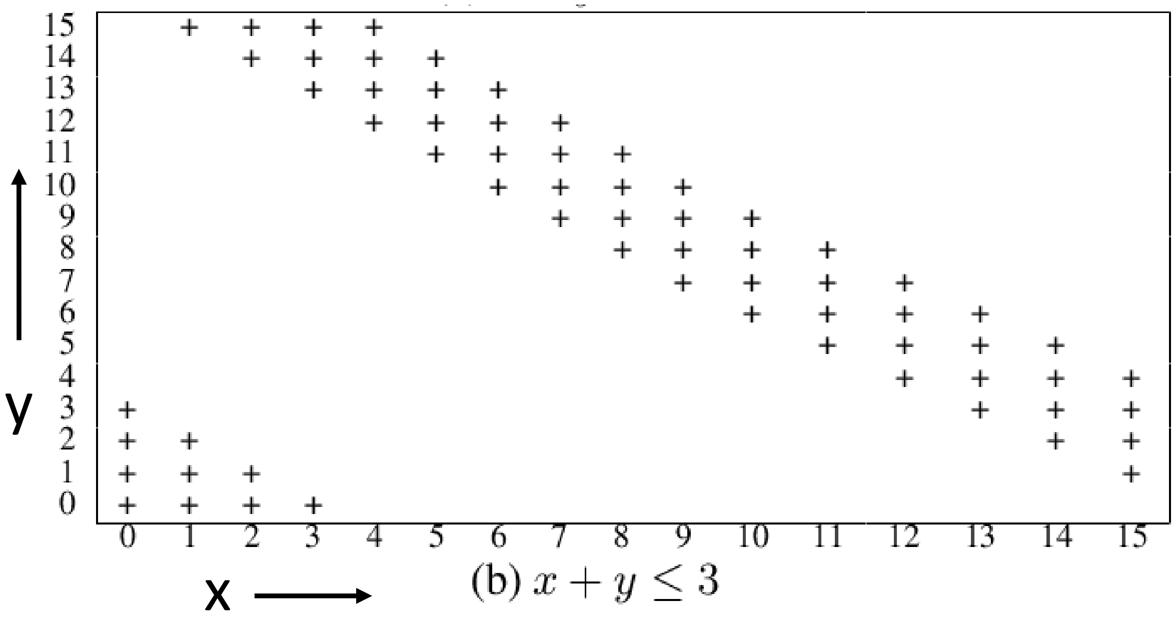
assert (low <= mid <= high);

Polyhedral analysis unsoundly says that the assert holds, *mid* might become less than low in case *low + high* overflows.

Bit-vector arithmetic is challenging



(a) $x + y + 4 \leq 7$



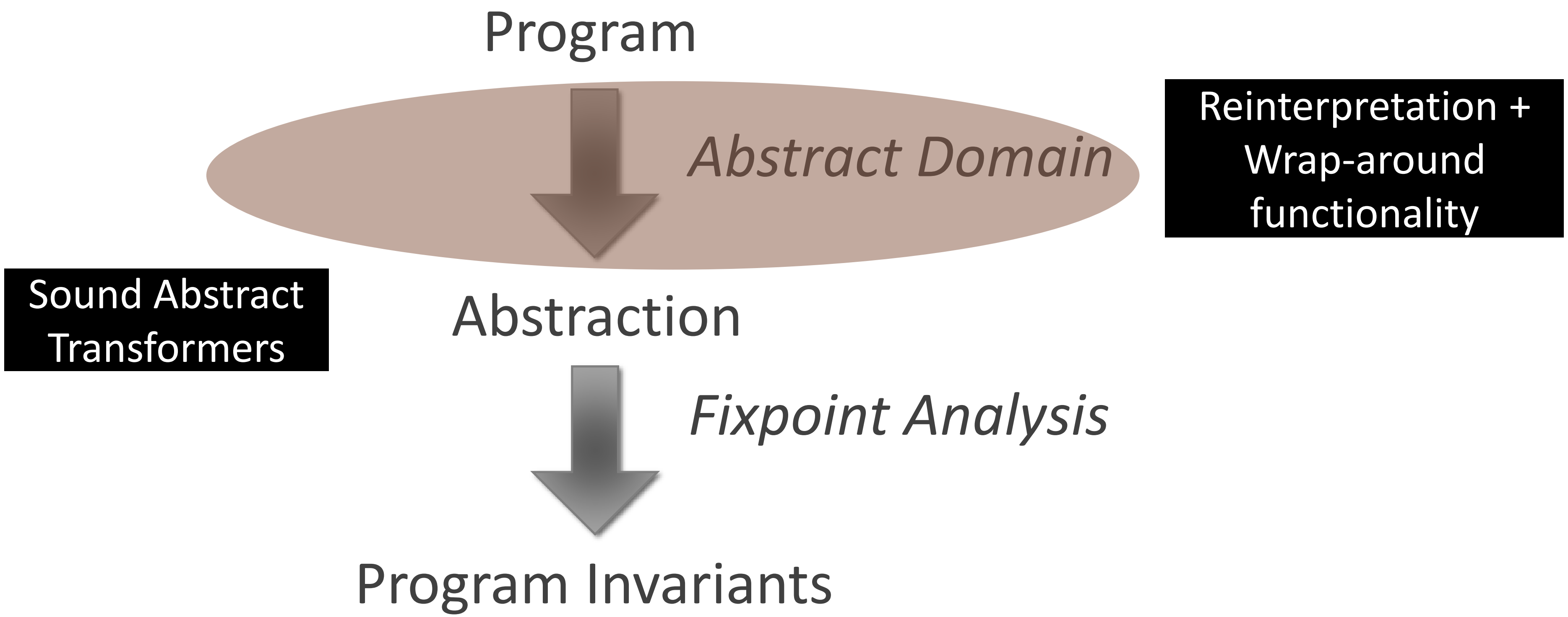
(b) $x + y \leq 3$

Problem Statement

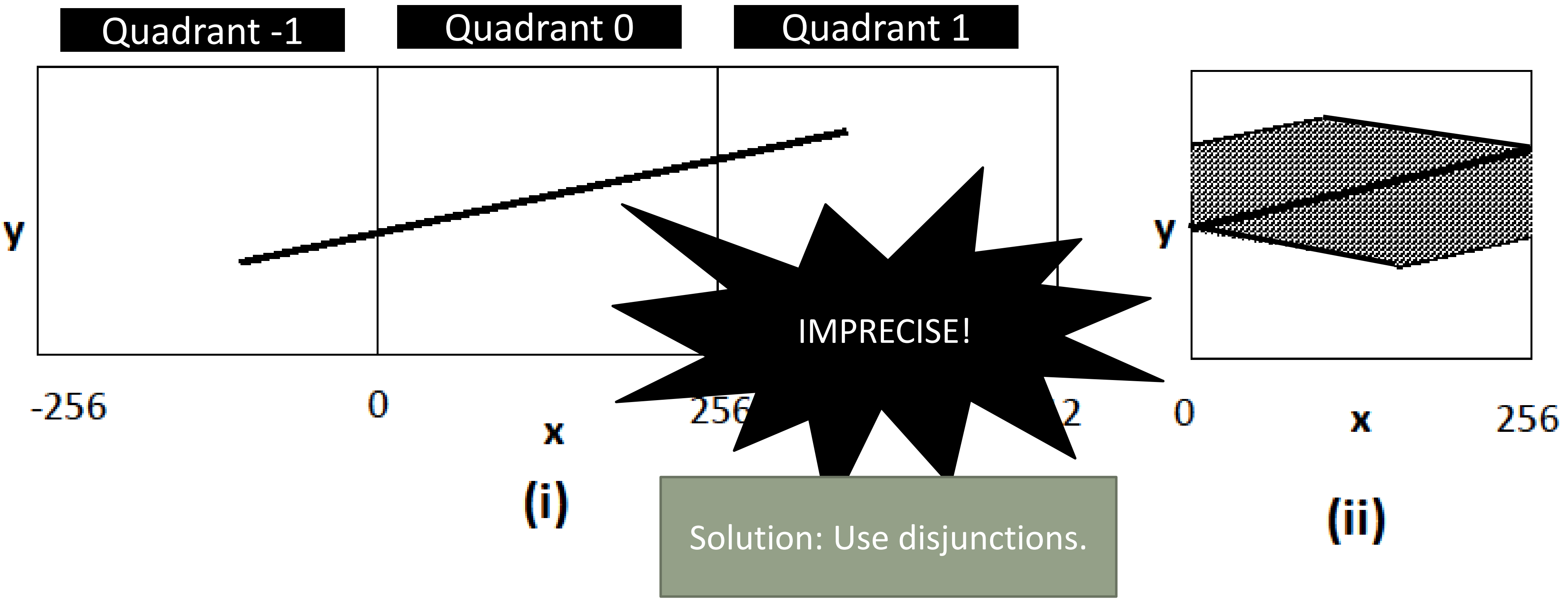
Given a relational numeric domain over integers, capable of expressing inequalities:

- Provide an automatic method to create a similar domain over bitvectors.
- Create sound bit-precise abstract transformers.

Key Idea

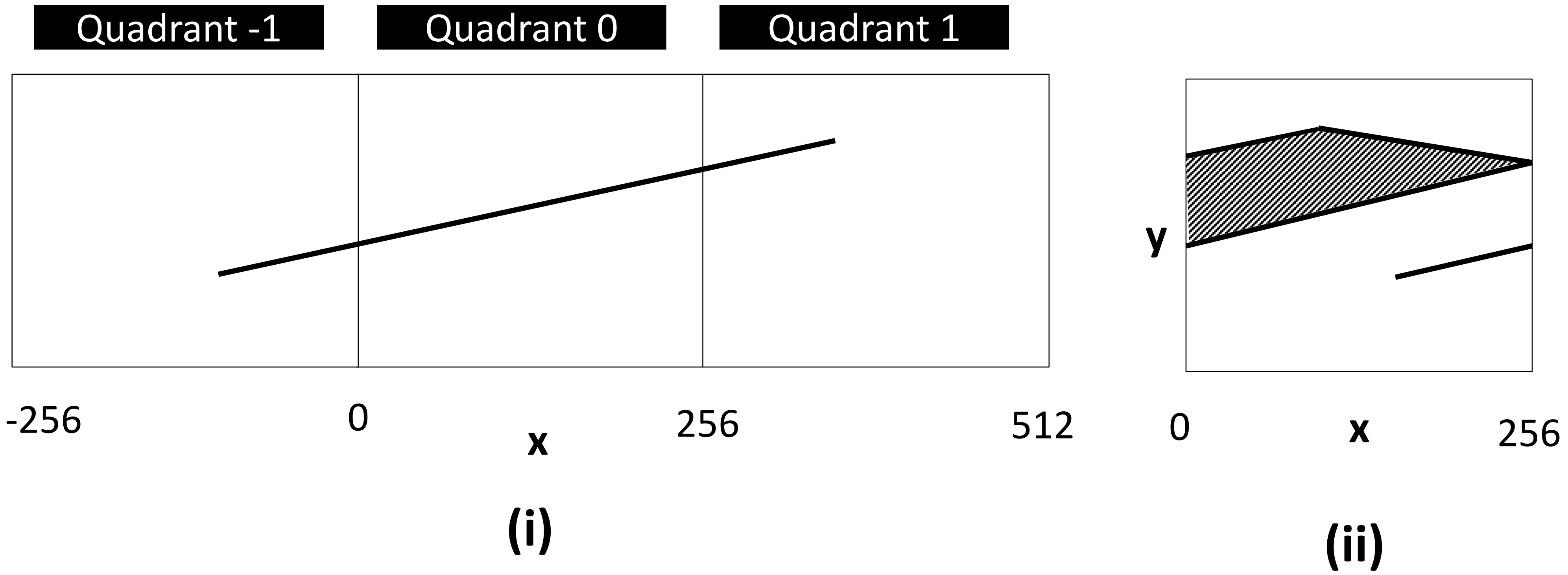

Program
Abstract Domain
Reinterpretation + Wrap-around functionality
Sound Abstract Transformers
Abstraction
Fixpoint Analysis
Program Invariants

Wrap around operation

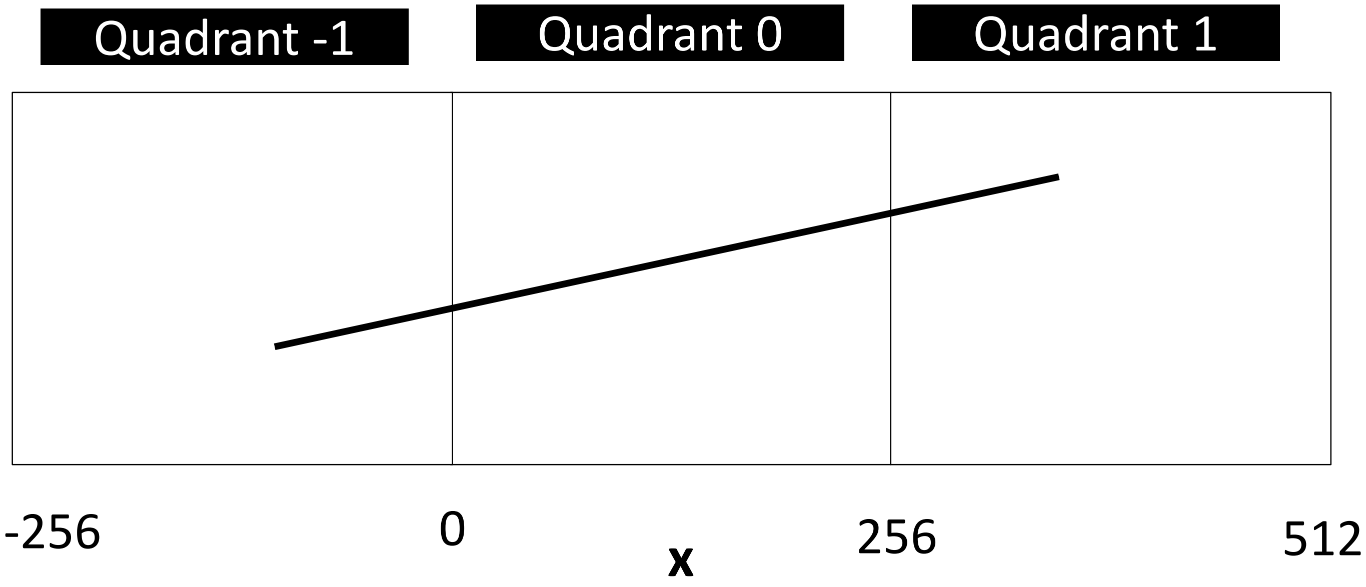


A. SIMON AND A. KING. TAMING THE WRAPPING OF INTEGER ARITHMETIC. IN SAS, 2007.

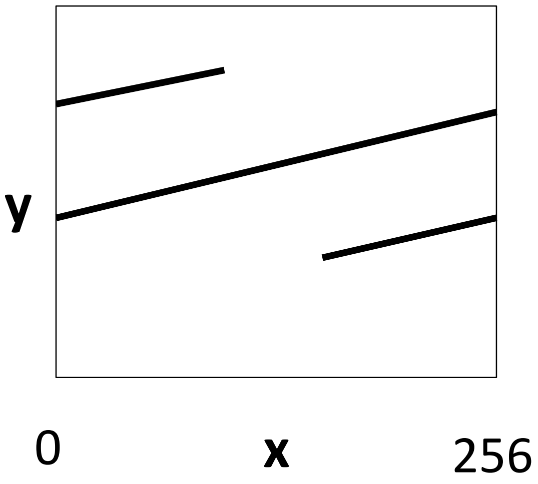
Maximum allowed of disjunctions = 2



Maximum allowed of disjunctions $\geq$ 3



(i)



(ii)

Abstract Domain Constructors

- **Bit-Vector-Sound Constructor (BVS)**
  - For a base domain $\mathcal{A}$, such that $\mathcal{A}$ is sound over integers, BVS[$\mathcal{A}$] constructs a bit-precise version of the domain.
  - Sound over bitvectors and imprecise.
- **Finite-Disjunctive Domain Constructor (FD_k)**
  - For a base domain $\mathcal{A}$, such that $\mathcal{A}$ is sound over integers, and a parameter k, FD_k[$\mathcal{A}$] constructs a finite-disjuctive version of the domain.
  - Number of disjunction in any element $a \in FD_k[\mathcal{A}]$ cannot exceed k.
  - Unsound over bitvectors (sound over integers) and precise.
- **Bit-Vector-Sound Finite-Disjunctive Domain ($BVSFD_k$)**
  - $BVSFD_k[\mathcal{A}]$ domain is constructed as $BVS[FD_k[\mathcal{A}]]$.
  - Sound over bitvectors and precise.

Contributions

- Introduce a generic framework to construct sound abstract domains **$BVSFD_k$** over bitvectors, by performing wrap-around over abstract domain A and using **disjunctions** to retain precision.
- 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.

Abstract-Domain Interface

Type	Operation	Description
$\mathcal{A}$	$\top$	Top element
$\mathcal{A}$	$\bot$	Bottom element
$\mathcal{A}$	$(a_1 == a_2)$	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
$\mathcal{A}$	$\mathbb{C}(le_1 \leq le_2)$	Construct abstract value
$\mathcal{D}$	$D(a_1, a_2)$	Distance heuristic

Abstract Interpretation with $BVSFD_k$

- **Abstract Transformers Generation**
  - Eager Abstract Transformers
  - Lazy Abstract Transformers

- **Fixed-point Iteration**
  - Uses join (widen at loop headers) and abstract composition to get procedure summaries

Abstract Transformers Generation

```
  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:  }
```

Desired property: x <= y relationship is true at the end of the function.

Abstract Transformers Generation

```
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: }
```

L0→L2: $(m \leq x, y \leq M) \wedge (x' = x \wedge y' = y) \wedge x' \leq y'$

m is minimum signed integer -2147483648.

M is maximum signed integer 2147483647.

Eager Abstract Transformers

Need 4 disjunctions for the following scenarios:

1) Variables x'',y'' don't overflow
2) Both variables x'',y'' overflow
3) Variable y'' overflows, but x'' does not
4) Variable x'' overflows, but y'' does not

```
L0:
L7:
L8:    }
END: }
```

y in eager abstract

ays bounded in [m,

M) ^ (x'=x+1 ^ y'=y+1)

x,y≤M) ^ (x'=x ^ y'=y+1)

act composition (L4→L5) o (L5→L6):

$$(m \leq x'',y'' \leq M) \wedge (x'=x'' \wedge y'=y''+1)$$

$$\sqcap$$

$$\mathrm{WRAP}_{\{x'',y''\}}(\,(m \leq x,y \leq M) \wedge (x''=x+1 \wedge y''=y+1)\,)$$

Eager Abstract Transformers

- Forces call to wrap at each composition operation.
- Addition and multiplication over integers preserves concretization over bitvectors.
- For example, $(M+1)\%(2^{32}) = m\%(2^{32})$.

Avoid adding bounds until there is a cast or guard operation!

Lazy Abstract Transformers

```
  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:   }
```

Pre-vocabulary in lazy abstract transformers **need not be** bounded in [m, M].

L4→L5: (x'=x+1 ^ y'=y+1)

L5→L6: (x'=x ^ y'=y+1)

(L4→L5) o (L5→L6): (x'=x+1 ^ y'=y+2)

Lazy Abstract Transformers

```
  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:   }
```

Step 1: Backward dependency analysis to find subset of pre-vocabulary on which y' depends.
Answer: y

Step 2: Add bounding constraints for y.

L5→L7: (x'=x ^ y'=y+1) ^ …

Lazy Abstract Transformers

```
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: }
```

Step 1: Backward dependency analysis to find subset of pre-vocabulary on which y' depends.
Answer: y

Step 2: Add bounding constraints for y.

Step 3: Perform Wrap on y'.

L5→L7: $(m \le y \le M) \wedge (x' = x \wedge y' = y+1)$

Lazy Abstract Transformers

```
  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:   }
```

Step 1: Backward dependency analysis to find subset of pre-vocabulary on which y' depends.
Answer: y

Step 2: Add bounding constraints for y.

Step 3: Perform Wrap on y'.

L5→L7: $\text{WRAP}_{\{y'\}}((m \le y \le M) \wedge (x' = x \wedge y' = y+1))$

Lazy Abstract Transformers

```
  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:   }
```

Step 1: Backward dependency analysis to find subset of pre-vocabulary on which y' depends.
Answer: y

Step 2: Add bounding constraints for y.

Step 3: Perform Wrap on y'.

Step 4: Add guard to the abstract value.

L5→L7: ((m≤y≤M-1) ^ (x'=x ^ y'=y+1)**)** ⊔
((y=M) ^ (x'=x ^ y'=m)**)**

Lazy Abstract Transformers

```
  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:   }
```

Step 1: Backward dependency analysis to find subset of pre-vocabulary on which y' depends.
Answer: y

Step 2: Add bounding constraints for y.

Step 3: Perform Wrap on y'.

Step 4: Add guard to the abstract value.

$$\textbf{L5} \rightarrow \textbf{L7:}\ \big((m \le y \le M-1) \wedge (x' = x \wedge y' = y+1) \wedge (y' <= 0)\big) \sqcup \big((y = M) \wedge (x' = x \wedge y' = m)\big)$$

Implementation

Program *(LLVM Bitcode)*

Reinterpretation

Abstract Domain

*(*Parma Polyhedra Library)

Abstract Transformers

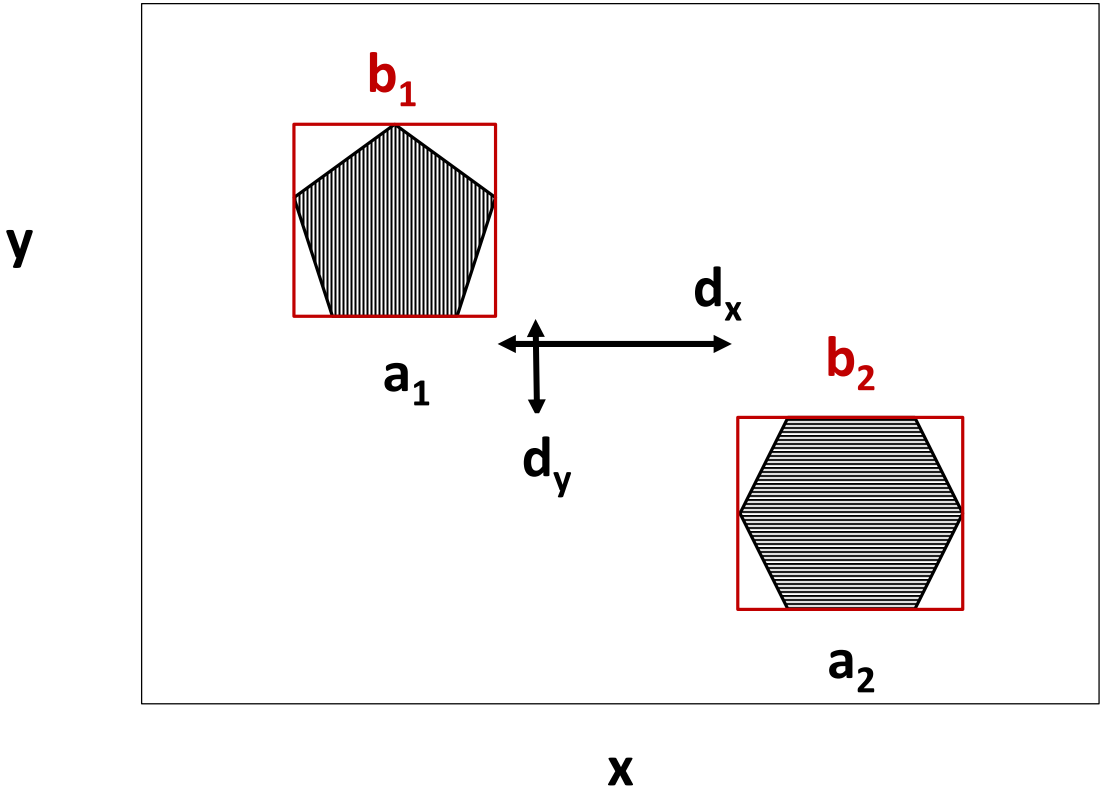
(Weighted Pushdown System)

Fixpoint Analysis

(WALi poststar + pathsummary)

Program Invariants

Distance Heuristic



$D(a_1, a_2) = d_x + d_y$

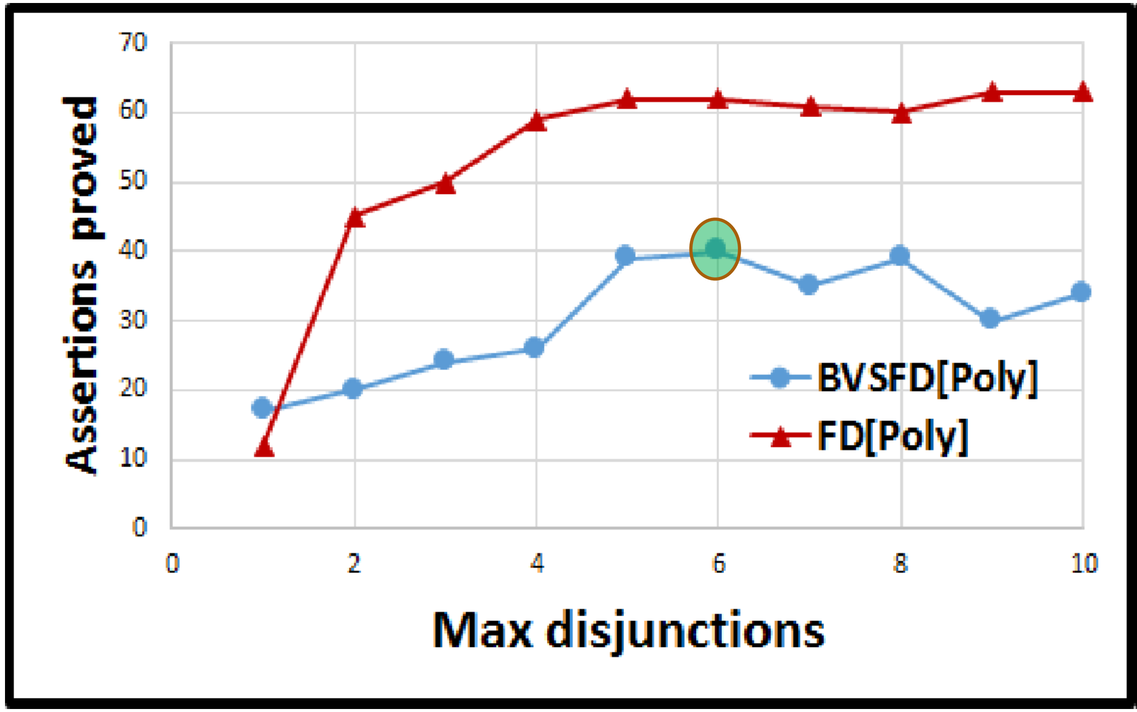
Experimental Evaluation

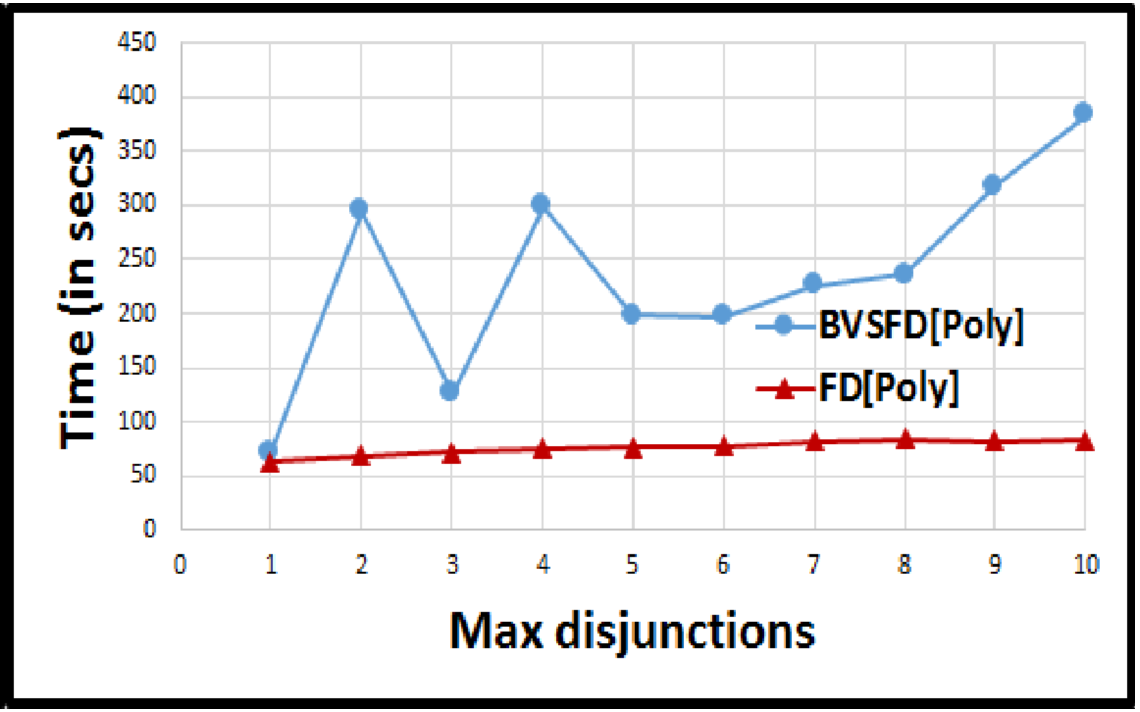
- Performance and precision comparison of the bit-precise disjunctive-inequality domain $BVSFD_k$ for different values of k.
- Base domains of octagons and polyhedra.
- Assertion proving and array-bounds checking.
- Time out of 200 seconds for each example.

Assertion proving benchmarks

	SVCOMP'16 loop benchmarks with true assertions		
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**

Assertions Proving with Polyhedra



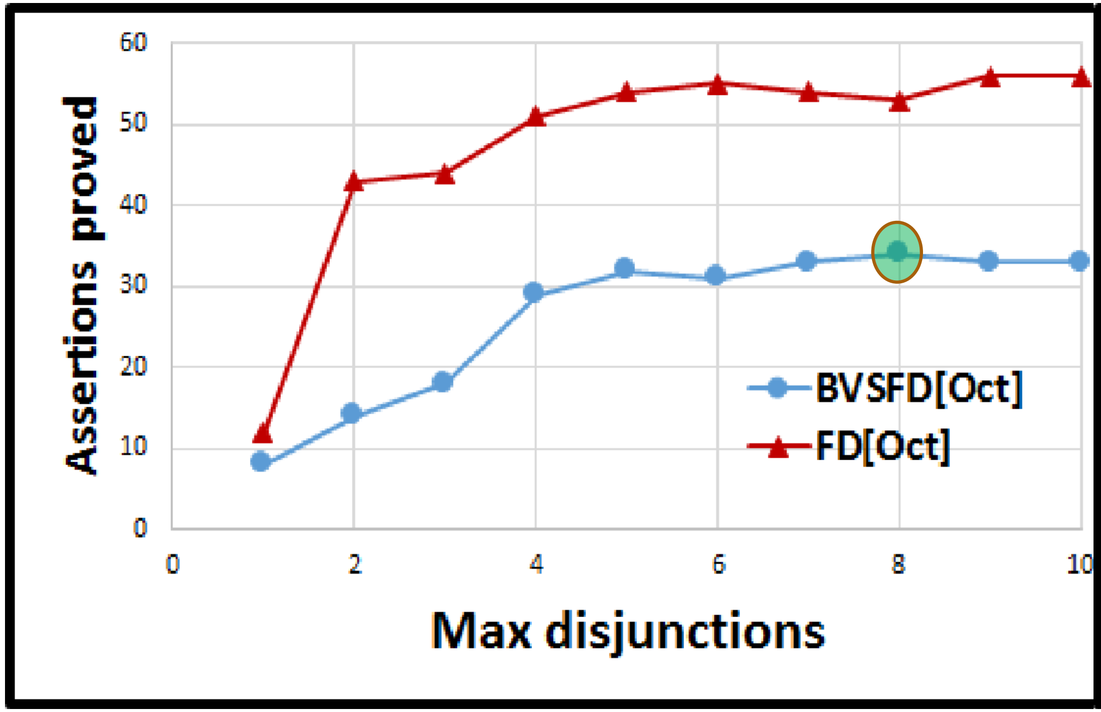


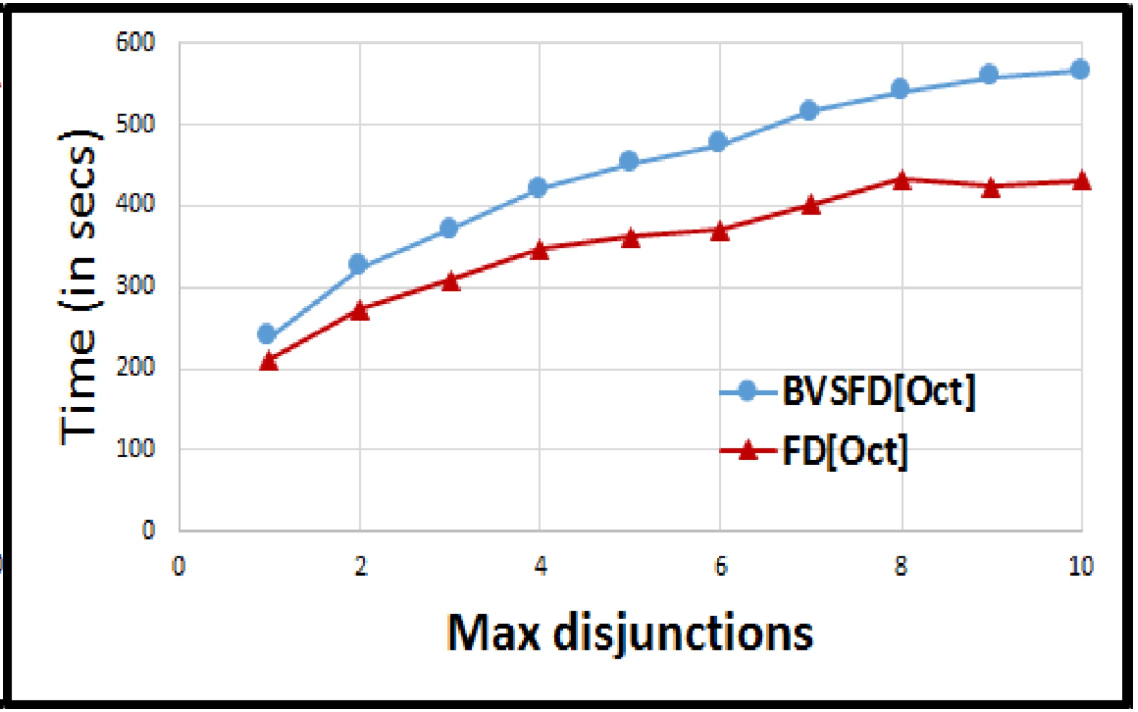
44-142% of the assertions of FD[Poly]

40/157 assertions

1.1-4.6 times slower

Assertions Proving with Octagons





33-67% of the assertions of FD[Oct]

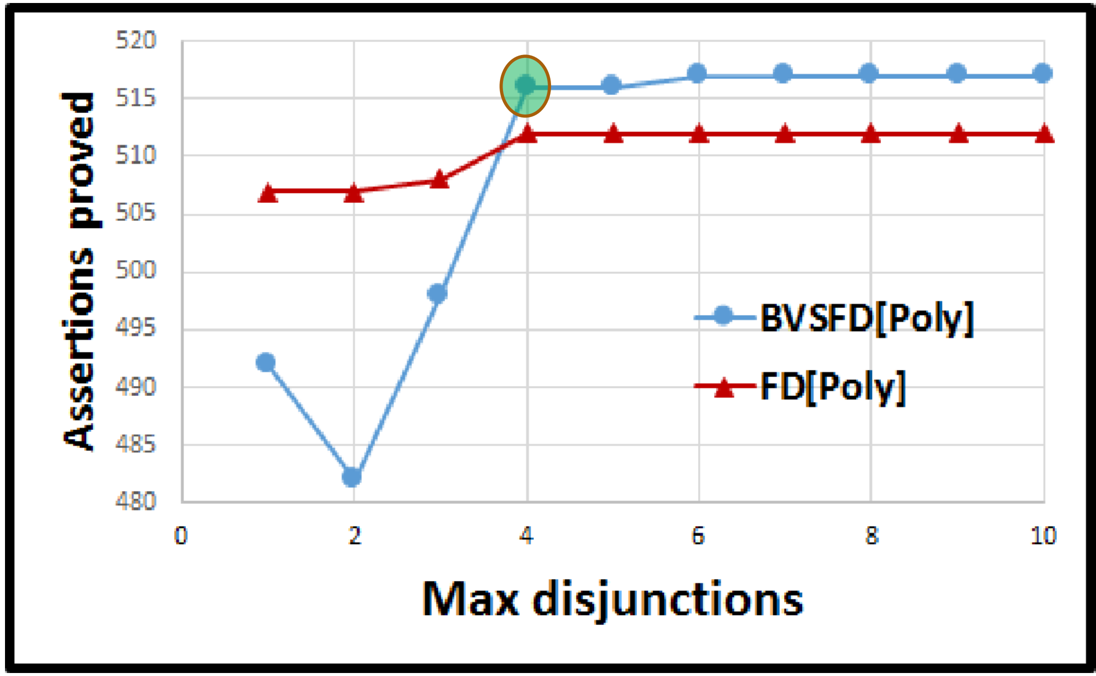
34/157 assertions

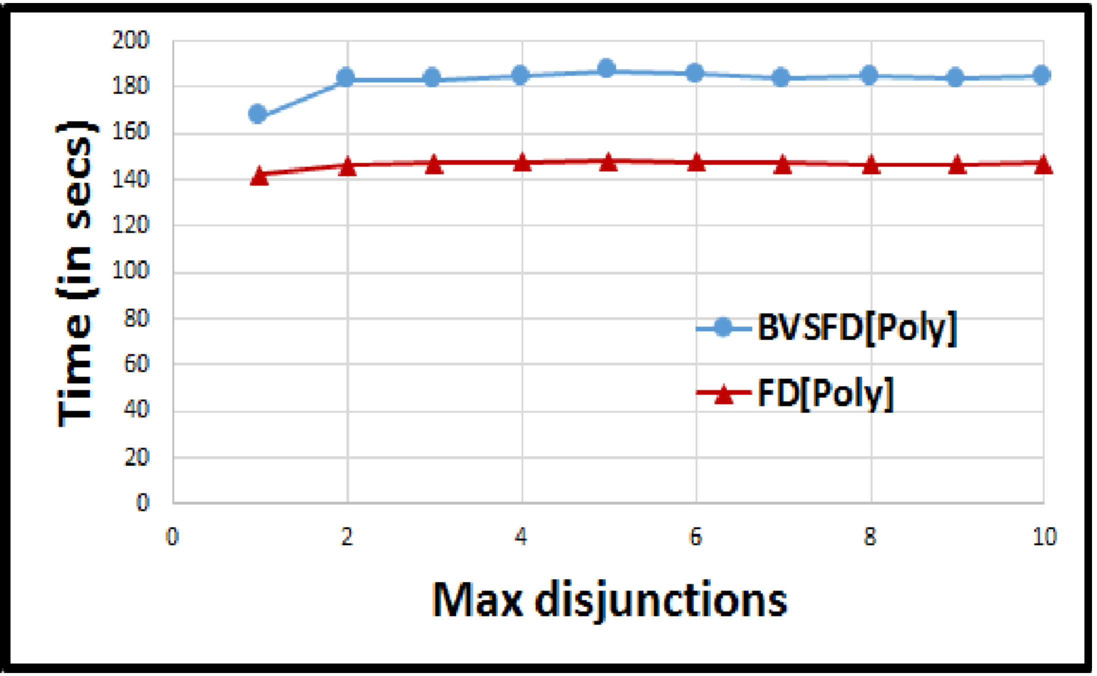
1.1-1.3 times slower

Array-Bounds Checking

- Added bounds checking for each array access and update
- SVCOMP’16 Array benchmarks
- 88 examples, 14,742 instructions and 598 array-bounds checks

Array-Bounds Checking with Polyhedra





95-101% of the assertions

515/598 assertions

1.18-1.26 times slower

Conclusion

Questions?

- Introduce a generic framework to construct sound abstract domains $\mathbf{BVSFD_k}$.
- We provide a generic technique via *reinterpretation* to create the abstract transformer with enhanced precision via **lazy** wrap-around.
- Our experiments show that the analysis can prove:
  - 25% of the assertions in the SVCOMP loop benchmarks with $\mathbf{BVSFD_6[Poly]}$.
  - 22% of the assertions in the SVCOMP loop benchmarks with $\mathbf{BVSFD_8[Oct]}$.
  - 88% of the array-bounds checks in the SVCOMP array benchmarks with $\mathbf{BVSFD_4[Poly]}$.
  - Empirical results show that values of k in the range 3-8 provide best results.