Fast Polyhedra Abstract Domain

Gagandeep Singh Markus Püschel Martin Vechev

Department of Computer Science
ETH Zurich, Switzerland
{gsingh,pueschel,martin.vechev}@inf.ethz.ch

Abstract

Numerical abstract domains are an important ingredient of modern static analyzers used for verifying critical program properties (e.g., absence of buffer overflow or memory safety). Among the many numerical domains introduced over the years, Polyhedra is the most expressive one, but also the most expensive: it has worst-case exponential space and time complexity. As a consequence, static analysis with the Polyhedra domain is thought to be impractical when applied to large scale, real world programs.

In this paper, we present a new approach and a complete implementation for speeding up Polyhedra domain analysis. Our approach does not lose precision, and for many practical cases, is orders of magnitude faster than state-of-the-art solutions. The key insight underlying our work is that polyhedra arising during analysis can usually be kept decomposed, thus considerably reducing the overall complexity.

We first present the theory underlying our approach, which identifies the interaction between partitions of variables and domain operators. Based on the theory we develop new algorithms for these operators that work with decomposed polyhedra. We implemented these algorithms using the same interface as existing libraries, thus enabling static analyzers to use our implementation with little effort. In our evaluation, we analyze large benchmarks from the popular software verification competition, including Linux device drivers with over 50K lines of code. Our experimental results demonstrate massive gains in both space and time: we show end-to-end speedups of two to five orders of magnitude compared to state-of-the-art Polyhedra implementations as well as significant memory gains, on all larger benchmarks. In fact, in many cases our analysis terminates in seconds where prior code runs out of memory or times out after 4 hours.

We believe this work is an important step in making the Polyhedra abstract domain both feasible and practically usable for handling large, real-world programs.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program analysis; F.2.1 [*Numerical Algorithms and Problems*]: Computations on matrices

General Terms Verification, Performance

Keywords Numerical program analysis, Abstract interpretation, Partitions, Polyhedra decomposition, Performance optimization

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

POPL'17, January 15–21, 2017, Paris, France
© 2017 ACM. 978-1-4503-4660-3/17/01...$15.00
http://dx.doi.org/10.1145/3009837.3009885

1. Introduction

Abstract interpretation is a general framework for static program analysis that defines sound and precise abstractions for the program's (potentially infinite) concrete semantics. The abstract semantics of a program require an *abstract domain* for capturing the properties of interest. Examples of useful program properties involve the program's heap [22, 26], numerical values [6, 14, 16, 17, 19, 30], termination [27, 28], and many others. An abstract interpreter is obtained by defining the effect of statements and expressions in the programming language on the abstract domain. In this paper, we focus on numerical abstract domains which capture numerical relationships between program variables. These relationships are important for proving the absence of buffer overflow, division by zero, and other properties. Thus, numerical domains are an important ingredient of modern static analyzers [4, 9].

Expressivity vs. cost In an ideal setting, one would simply use the most expressive domain to analyze a program, i.e., Polyhedra [6]. However, the Polyhedra domain comes with a worst case exponential complexity in both space and time. Thus, an analyzer using Polyhedra can easily fail to analyze large programs by running out of memory or by timing out. Because of this, the Polyhedra domain is often thought to be impractical, and thus, over the years, researchers have designed domains that limit its expressivity in exchange for better asymptotic complexity. Examples include Octagon [19], Zone [17], Pentagon [16], SubPolyhedra [14] and Gauge [30]. Unfortunately, limited expressivity can make the over-approximation too imprecise for proving the desired property.

Our work In this work, we revisit the basic assumption that Polyhedra is impractical for static analysis. We present a new approach which enables the application of Polyhedra to large, realistic programs, with speedups ranging between two to five orders of magnitude compared to the state-of-the-art. We note that our approach does not lose precision yet can analyze programs beyond the reach of current approaches.

The key insight of our approach is that the set of program variables partitions into subsets such that linear constraints only exist between variables in the same subset [10, 25]. We leverage this observation to decompose a large polyhedron into a set of smaller polyhedra, thus reducing the asymptotic complexity of the Polyhedra domain. However, maintaining decomposition online is challenging because over 40 Polyhedra operators change the partitions dynamically and in non-trivial ways: subsets can merge, split, grow, or shrink during analysis. Note that an exact partition cannot be computed a priori as otherwise the approach loses precision [4].

To ensure our method does not lose precision, we develop a theoretical framework that asserts how partitions are modified during analysis. We then use this theory to design new abstract operators for Polyhedra. Interestingly, our framework can be used for decomposing other numerical domains, not only Polyhedra.

Main contributions Our main contributions are:

- A theoretical framework for decomposing Polyhedra analysis. This framework allows for efficient maintenance of decomposition throughout the analysis *without* losing precision.
- New algorithms for Polyhedra operators which leverage decomposition based on the theory. The algorithms are further optimized using novel optimizations exploiting sparsity.
- A complete implementation of our Polyhedra operators[1] providing the same interface as APRON [13], enabling existing analyzers to immediately benefit from our work.
- An evaluation of the effectiveness of our approach showing massive gains in both space and time over state-of-the-art approaches on a large number of benchmarks, including Linux device drivers. For instance, we obtain a 170x speedup on the largest benchmark containing $>$ 50K lines of code. In many other cases, the analysis with our approach terminates whereas other implementations abort without result.

2. Background

In this section, we first introduce the necessary background on Polyhedra domain analysis. We present two ways to represent polyhedra and define the Polyhedra domain including its operators. We conclude the section by discussing their asymptotic complexity.

Notation Lower case letters $(a, b, \ldots)$ represent column vectors and integers $(g, k, \ldots)$. Upper case letters A, D represent matrices whereas O, P, Q are polyhedra. Greek letters $(\alpha, \beta, \ldots)$ represent scalars and calligraphic letters $(\mathcal{P}, \mathcal{C}, \ldots)$ are used for sets.

2.1 Representation of Polyhedra

Let $x = (x_1, x_2, \ldots x_n)^T$ be a column vector of program variables. A convex closed polyhedron $P \subseteq \mathbb{Q}^n$ that captures linear constraints among variables in x can be represented in two equivalent ways: the *constraint representation* and the *generator representation* [21]. Both are introduced next.

Constraint representation This representation encodes a polyhedron P as an intersection of:

- A finite number of closed half spaces of the form $a^T x \leq \beta$.
- A finite number of subspaces of the form $d^T x = \xi$.

Collecting these yields matrices A, D and vectors of rational numbers b, e such that the polyhedron P can be written as:

$$P = \{x \in \mathbb{Q}^n \mid Ax \leq b \text{ and } Dx = e\}. \tag{1}$$

The associated *constraint set* $\mathcal{C}$ of P is defined as $\mathcal{C} = \mathcal{C}_P = \{Ax \leq b, Dx = e\}$.

Generator representation This representation encodes the polyhedron P as the convex hull of:

- A finite set $\mathcal{V} \subset \mathbb{Q}^n$ of vertices v_i.
- A finite set $\mathcal{R} \subseteq \mathbb{Q}^n$ representing rays. $r_i \in \mathbb{R}$ are direction vectors of infinite edges of the polyhedron with one end bounded. The rays always start from a vertex in $\mathcal{V}$.
- A finite set $\mathcal{Z} \subseteq \mathbb{Q}^n$ representing lines[2]. $z_i \in \mathbb{Z}$ are direction vectors of infinite edges of the polyhedron with both ends unbounded. Each such line passes through a vertex in $\mathcal{V}$.

[1] http://elina.ethz.ch

[2] one dimensional affine subspaces.

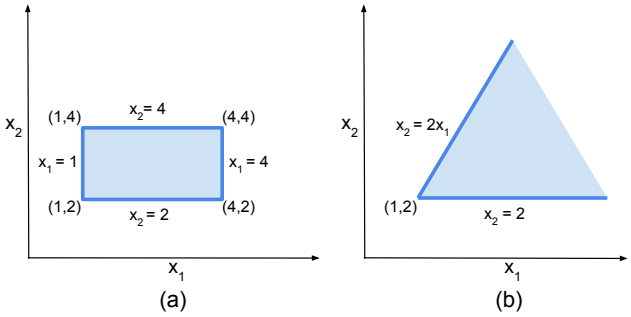

Figure 1: Two representations of polyhedron. (a) Bounded polyhedron; (b) unbounded polyhedron.

```
if(*){y:=2x-1;} else{y:=2x-2;}
assert(y<=2x);
```

Figure 2: Code with assertion for static analysis.

Thus, every $x \in P$ can be written as:

$$x = \sum_{i=1}^{|\mathcal{V}|} \lambda_i v_i + \sum_{i=1}^{|\mathcal{R}|} \mu_i r_i + \sum_{i=1}^{|\mathcal{Z}|} \nu_i z_i, \tag{2}$$

where $\lambda_i, \mu_i \geq 0$, and $\sum_{i=1}^{|\mathcal{V}|} \lambda_i = 1$. The above vectors are the *generators* of P and are collected in the set $\mathcal{G} = \mathcal{G}_P = \{\mathcal{V}, \mathcal{R}, \mathcal{Z}\}$.

Example 2.1. Fig. 1 shows two examples of both representations for polyhedra. In Fig. 1(a) the polyhedron P is bounded and can be represented as either the intersection of four closed half spaces or as the convex hull of four vertices:

$$\mathcal{C} = \{-x_1 \leq -1, x_1 \leq 4, -x_2 \leq -2, x_2 \leq 4\}, \text{or}$$
$$\mathcal{G} = \{\mathcal{V} = \{(1,2), (1,4), (4,2), (4,4)\}, \mathcal{R} = \emptyset, \mathcal{Z} = \emptyset\}.$$

Note that the sets of rays $\mathcal{R}$ and lines $\mathcal{Z}$ are empty in this case.

In Fig. 1(b), the polyhedron P is unbounded and can be represented either as the intersection of two closed half planes or as the convex hull of two rays starting at vertex $(1, 2)$:

$$\mathcal{C} = \{-x_2 \leq -2, x_2 \leq 2x_1\}, \text{or}$$
$$\mathcal{G} = \{\mathcal{V} = \{(1,2)\}, \mathcal{R} = \{(1,2), (1,0)\}, \mathcal{Z} = \emptyset\}.$$

To reduce clutter, we abuse notation and often write $P = (\mathcal{C}, \mathcal{G})$ since our algorithms, introduced later, maintain both representations. Both $\mathcal{C}$ and $\mathcal{G}$ represent *minimal* sets, i.e., they do not contain redundancy.

2.2 Polyhedra Domain

The Polyhedra domain is commonly used in static analysis to prove safety properties in programs like the absence of buffer overflow, division by zero and others. It is a fully relational numerical domain, i.e., can encode all possible linear constraints between program variables. Thus, it is more expressive than weakly relational domains such as Octagons [19], Pentagons [16] or Zones [17], which restrict the set of linear inequalities. The restrictions limit the set of assertions that can be proved using these domains. For example, the assertion in the code in Fig. 2 cannot be expressed using weakly relational domains whereas Polyhedra can express and prove the property. The expressivity of the Polyhedra domain comes at higher cost: it has asymptotic worst-case exponential complexity in both time and space.

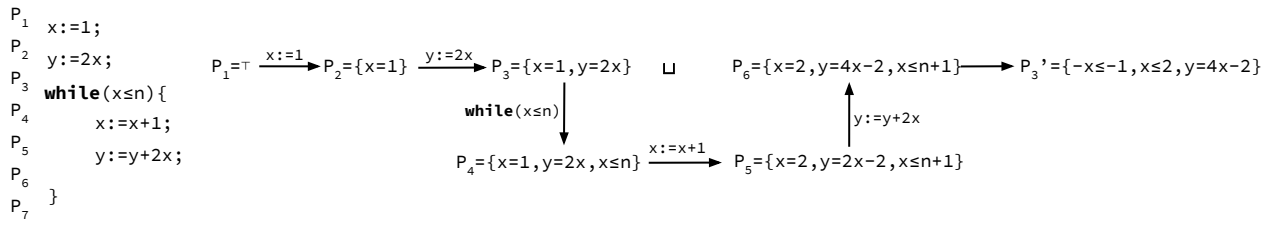

Figure 3: Polyhedra domain analysis (first iteration) on the example program on the left. The polyhedra are shown in constraint representation.

The Polyhedra abstract domain consists of the *polyhedra lattice* $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ and a set of operators. $\mathcal{P}$ is the set of convex closed polyhedra ordered by standard inclusion: $\sqsubseteq = \subseteq$. The least upper bound ($\sqcup$) of two polyhedra P and Q is the convex hull of P and Q, which, in general, is larger than the union $P \cup Q$. The greatest lower bound ($\sqcap$) of P and Q is simply the intersection $P \cap Q$. The top element $\top = \mathbb{Q}^n$ in the lattice is encoded by $\mathcal{C} = \emptyset$ or generated by n lines. The bottom element ($\bot$) is represented by any unsatisfiable set of constraints in $\mathcal{C}$ or with $\mathcal{G} = \emptyset$.

Operators The operators used in the Polyhedra domain analysis model the effect of various program statements such as assignments and conditionals on the program states approximated by polyhedra. A standard implementation of the Polyhedra domain contains more than 40 operators [2, 13]. We introduce the most frequently used operators in Polyhedra domain:

Inclusion test: this operator tests if $P \sqsubseteq Q$ for the given polyhedra P and Q.

Equality test: this operator tests if two polyhedra P and Q are equal by double inclusion.

Join: this operator computes $P \sqcup Q$, i.e., the convex hull of P and Q.

Meet: this operator computes $P \sqcap Q = P \cap Q$.

Widening: as the polyhedra lattice has infinite height, the analysis requires *widening* to accelerate convergence. The result of the widening operator [7] $P \nabla Q$ contains constraints from $\mathcal{C}_Q$ that are either present in $\mathcal{C}_P$ or that can replace a constraint in $\mathcal{C}_P$ without changing P. Using the constraint representation it is defined as:

$$\mathcal{C}_{P \nabla Q} = \begin{cases} \mathcal{C}_Q, \text{ if } P = \bot; \\ \mathcal{C}'_P \cup \mathcal{C}'_Q, \text{ otherwise;} \end{cases} \tag{3}$$

where:

$$\mathcal{C}'_P = \{c \in \mathcal{C}_P \mid \mathcal{C}_Q \models c\},$$
$$\mathcal{C}'_Q = \{c \in \mathcal{C}_Q \mid \exists c' \in \mathcal{C}_P, \mathcal{C}_P \models c \text{ and } ((\mathcal{C}_P \setminus c') \cup \{c\}) \models c'\}.$$

where $\mathcal{C} \models c$ tests whether the constraint c can be *entailed* by the constraints in $\mathcal{C}$.

Next we introduce the operators corresponding to program statements. For simplicity, we assume that the expression δ on the right hand side of both conditional and assignment statements is affine i.e., $\delta = a^T x + \epsilon$, where $a \in \mathbb{Q}^n$, $\epsilon \in \mathbb{Q}$ are constants. Non-linear expressions can be approximated by affine expressions using the techniques described in [18].

Conditional: Let $\otimes \in \{\leq, =\}$, $1 \leq i \leq n$, and $\alpha \in \mathbb{Q}$, the conditional statement $\alpha x_i \otimes \delta$ adds the constraint $(\alpha - a_i)x_i \otimes \delta - a_i x_i$ to the constraint set $\mathcal{C}$.

Assignment: The operator for an assignment $x_i := \delta$ first adds a new variable x'_i to the polyhedron P and then augments $\mathcal{C}$ with the constraint $x'_i - \delta = 0$. The variable x_i is then projected out from the constraint set $\mathcal{C} \cup \{x'_i - \delta = 0\}$. Finally, the variable x'_i is renamed back to x_i.

2.3 Polyhedra Domain Analysis: Example

Fig. 3 shows a simple program that computes the sum of the first n even numbers where a polyhedron P_ℓ is associated with each line ℓ in the program. Each P_ℓ approximates the program state before executing the statement at line ℓ. Here, we work only with the constraint representation of polyhedra. The analysis proceeds iteratively by selecting the polyhedron at a given line, say P_1, then applying the operator for the statement at that program point (`x:=1` in this case) on that polyhedron, and producing a new polyhedron, in this case P_2. The analysis terminates when a fixed point is reached, i.e., when further iterations do not modify any polyhedra.

First iteration Initially, polyhedra P_1 is top ($\top$). Next, the analysis applies the operator for `x:=1` to P_1, producing P_2. The set $\mathcal{C}_1$ is empty and the operator adds constraint $x = 1$ to obtain P_2. The next statement assigns to y. Since $\mathcal{C}_2$ does not contain any constraint involving y, the operator for `y:=2x` adds $y = 2x$ to obtain P_3. Next, the conditional statement for the loop is processed: that operator adds the constraint $x \leq n$ to obtain polyhedron P_4. The assignment statement `x:=x+1` inside the loop assigns to x which is already present in the set $\mathcal{C}_4$. Thus, a new variable x' is introduced and constraint $x' - x - 1 = 0$ is added to $\mathcal{C}_4$ producing:

$$\mathcal{C}'_5 = \{x = 1, y = 2x, x \leq n, x' - x - 1 = 0\}$$

The operator then projects out x from $\mathcal{C}'_5$ to produce:

$$\mathcal{C}''_5 = \{x' = 2, y = 2x' - 2, x' \leq n + 1\}$$

Variable x' is then renamed to x to produce the final set for P_5:

$$\mathcal{C}_5 = \{x = 2, y = 2x - 2, x \leq n + 1\}$$

The next assignment `y:=y+2x` is handled similarly to produce P_6.

Next iterations The analysis then returns to the head of the while loop and propagates the polyhedron P_6 to that point. To compute the new program state at the loop head, it now needs to compute the union of P_6 with the previous polyhedron P_3 at that point. Since the union of convex polyhedra is usually not convex, it is approximated using the join operator ($\sqcup$) to yield the polyhedron P'_3.

The analysis then checks if the new polyhedron P'_3 at the loop head is included in P_3 using inclusion testing ($\sqsubseteq$). If yes, then no new information was added and the analysis terminates. However, here, $P'_3 \not\sqsubseteq P_3$ and so the analysis continues. After several iterations, the widening operator (∇) may be applied at the loop head instead of the join to accelerate convergence.

2.4 Operators and Asymptotic Complexity

The asymptotic complexity of Polyhedra operators depends on how a polyhedron is represented, as shown in Table 1. In the table, n is

Table 1: Asymptotic time complexity of Polyhedra operators with different representations.

Operator	Constraint	Generator	Both
Inclusion ($\sqsubseteq$)	$O(m\,\mathrm{LP}(m,n))$	$O(g\,\mathrm{LP}(g,n))$	$O(ngm)$
Join ($\sqcup$)	$O(nm2^{n+1})$	$O(ng)$	$O(ng)$
Meet ($\sqcap$)	$O(nm)$	$O(ng2^{n+1})$	$O(nm)$
Widening (∇)	$O(m\,\mathrm{LP}(m,n))$	$O(g\,\mathrm{LP}(g,n))$	$O(ngm)$
Conditional	$O(n)$	$O(ng2^{n+1})$	$O(n)$
Assignment	$O(nm^2)$	$O(ng)$	$O(ng)$

the number of variables, m is the number of constraints in $\mathcal{C}$, $g = |\mathcal{V}| + |\mathcal{R}| + |\mathcal{Z}|$ is the number of generators in $\mathcal{G}$ and $\mathrm{LP}(m,n)$ is the complexity of solving a linear program with m constraints and n variables. For binary operators like join, meet and others, m and g denote the maximum of the number of constraints and generators in P and Q. The column *Constraint* shows the cost of computing the constraint set for the result from the input constraint set(s); the column *Generator* analogously for the generators. The column *Both* shows the cost of computing one of the representations for the result when both representations are available for the input(s).

Operators vs. representations Table 1 shows operators, such as meet ($\sqcap$), which are considerably more efficient to compute using the constraint representation whereas other operators, such as join ($\sqcup$), are cheaper using the generator representation. Operators such as inclusion testing ($\sqsubseteq$) are most efficient when one of the two participating polyhedron is represented via constraints and the other via generators. As a result, popular libraries such as NewPolka [13] and PPL [2] maintain both representations of polyhedra during analysis. In this paper we follow the same approach and each polyhedron P is represented as $P = (\mathcal{C}, \mathcal{G})$.

Maintaining both representations requires conversion. For example, the meet of two polyhedra can be efficiently computed by taking the union of the respective constraints. Conversion is then required to compute the corresponding generator representation of the result. As is common, we use Chernikova's [5, 15] algorithm but with our own optimized implementation (Section 5.1) for converting from the constraint to the generator representation and vice-versa. The conversion algorithm also minimizes both representations.

Conversion between representations When both representations are available, all Polyhedra operators become polynomial (last column of Table 1) and Chernikova's algorithm becomes the bottleneck for the analysis as it has worst case exponential complexity for conversion in either direction. We refer the reader to [5, 15] for details of the algorithm. There are two approaches for reducing the cost of these conversions: *lazy* and *eager*.

The *lazy* approach computes the conversion only when required to amortize the cost over many operations. For example, in Fig. 3, there are a number of conditional checks and assignments in succession so one can keep working with the constraint representation and compute the generator one only when needed (e.g., at the loop head when join is needed). The *eager* approach computes the conversion after every operation. Chernikova's algorithm is incremental, which means that for operators which add constraints or generators such as meet ($\sqcap$), join ($\sqcup$), conditional and others, the conversion needs to be computed only for the added constraints or generators. Because of this, in some cases eager can be faster than lazy. Our operators are compatible with both approaches, however in this paper we use the eager approach.

3. Polyhedra Decomposition

We next present the key insight of our work and show how to leverage it to speedup program analysis using Polyhedra. Our observation is that the polyhedra arising in program analysis usually do not relate all program variables through a constraint. This allows us to decompose a large polyhedron into a set of smaller polyhedra, which reduces both space and time complexity of the analysis without affecting its precision. For example, the n-dimensional hypercube requires 2^n generators whereas with decomposition only $2n$ generators are required. Further, the decomposition allows the expensive polyhedra operators to operate on smaller polyhedra, making them cheaper without losing precision.

We first introduce our notation for partitions. Then, we introduce the theoretical underpinning of our work: the interaction between the Polyhedra domain operators and the partitions.

3.1 Partitions

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be the set of n variables. For a given polyhedron, $\mathcal{X}$ can be partitioned into subsets $\mathcal{X}_k$ we call *blocks* such that constraints only exist between variables in the same block. Each unconstrained variable x_i yields a singleton block $\{x_i\}$. We refer to this unique, finest partition as $\pi = \pi_P = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$.

Example 3.1. Consider

$$\mathcal{X} = \{x_1, x_2, x_3\} \text{ and}$$
$$P = \{x_1 + 2x_2 \leq 3\}.$$

Here, $\mathcal{X}$ is partitioned into two blocks: $\mathcal{X}_1 = \{x_1, x_2\}$ and $\mathcal{X}_2 = \{x_3\}$. Now consider

$$P = \{x_1 + 2x_2 \leq 3, 3x_2 + 4x_3 \leq 1\}.$$

Here, the partition of $\mathcal{X}$ has only one block $\mathcal{X}_1 = \{x_1, x_2, x_3\}$.

The partition π_P decomposes the polyhedron P defined over $\mathcal{X}$ into a set of smaller polyhedra P_k which we call *factors*. Each factor P_k is defined only over the variables in $\mathcal{X}_k$. The polyhedron P can be recovered from the factors P_k by computing the union of the constraints $\mathcal{C}_{P_k}$ and the Cartesian product of the generators $\mathcal{G}_{P_k}$. For this, we introduce the $\bowtie$ operator defined as:

$$\begin{aligned} P &= P_1 \bowtie P_2 \bowtie \ldots \bowtie P_r \\ &= (\mathcal{C}_{P_1} \cup \mathcal{C}_{P_2} \ldots \cup \mathcal{C}_{P_r}, \mathcal{G}_{P_1} \times \mathcal{G}_{P_2} \ldots \times \mathcal{G}_{P_r}). \end{aligned} \tag{4}$$

Example 3.2. The polyhedron P in Fig. 1 (a) has no constraints between variables x_1 and x_2. Thus, $\mathcal{X} = \{x_1, x_2\}$ can be partitioned into blocks: $\pi_P = \{\{x_1\}, \{x_2\}\}$ with corresponding factors $P_1 = (\mathcal{C}_{P_1}, \mathcal{G}_{P_1})$ and $P_2 = (\mathcal{C}_{P_2}, \mathcal{G}_{P_2})$ where:

$$\mathcal{C}_{P_1} = \{-x_1 \leq -1, x_1 \leq 4\} \qquad \mathcal{C}_{P_2} = \{-x_2 \leq -2, x_2 \leq 4\}$$
$$\mathcal{G}_{P_1} = \{\{(1), (4)\}, \emptyset, \emptyset\} \qquad \mathcal{G}_{P_2} = \{\{(2), (4)\}, \emptyset, \emptyset\}$$

The original polyhedron can be recovered from P_1 and P_2 as $P = P_1 \bowtie P_2 = (\mathcal{C}_{P_1} \cup \mathcal{C}_{P_2}, \mathcal{G}_{P_1} \times \mathcal{G}_{P_2})$.

The set $\mathcal{L}$ consisting of all partitions of $\mathcal{X}$ forms a *partition lattice* $(\mathcal{L}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. The elements π of the lattice are ordered as follows: $\pi \sqsubseteq \pi'$, if every block of π is included in some block of π' (π "is finer" than π'). This lattice contains the usual operators of *least upper bound* ($\sqcup$) and *greatest lower bound* ($\sqcap$). In the partition lattice, $\top = \{\mathcal{X}\}$ and $\bot = \{\{x_1\}, \{x_2\}, \ldots, \{x_n\}\}$.

Example 3.3. For example,

$$\{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\} \sqsubseteq \{\{x_1, x_2, x_3\}, \{x_4\}, \{x_5\}\}$$

Now consider,

$$\pi = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}\} \text{ and}$$
$$\pi' = \{\{x_1, x_2, x_3\}, \{x_4\}, \{x_5\}\}$$

Then,

$$\pi \sqcup \pi' = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}\} \text{ and}$$
$$\pi \sqcap \pi' = \{\{x_1, x_2\}, \{x_3\}, \{x_4\}, \{x_5\}\}$$

Definition 3.1. We call a partition π *permissible* for P if there are no variables x_i and x_j in different blocks of π related by a constraint in P, i.e., if $\pi \sqsupseteq \pi_P$.

Note, that the finest partition $\pi_\top$ for the top ($\top$) and the bottom ($\bot$) polyhedra is the bottom element in the partition lattice, i.e., $\pi_\top = \pi_\bot = \bot$. Thus, every partition is permissible for these.

3.2 Operators and Partitions

We now describe the effects of Polyhedra operators on partitions. The partition for the output of operators such as meet, conditionals, and assignment is computed from the corresponding partitions of input polyhedra P and Q. For the join however, to ensure we do not end up with a trivial and imprecise partition, we need to examine P and Q (discussed later in the section). Our approach to handling the join partition is key to achieving significant analysis speedups.

Inclusion test We observe that if $P \sqsubseteq Q$ and $P \neq \bot$, then variables in different blocks of π_P cannot be in the same block of π_Q. This yields

Lemma 3.1. Let P and Q be two polyhedra satisfying $P \sqsubseteq Q$ and $P \neq \bot$. Then $\pi_Q \sqsubseteq \pi_P$.

Meet The constraint set for the meet $P \sqcap Q$ is the union $\mathcal{C}_P \cup \mathcal{C}_Q$. Thus, overlapping blocks $\mathcal{X}_i \in \pi_P$ and $\mathcal{X}_j \in \pi_Q$ will merge into one block in $\pi_{P \sqcap Q}$. This yields

Lemma 3.2. Let P and Q be two polyhedra with $P \sqcap Q \neq \bot$. Then $\pi_{P \sqcap Q} = \pi_P \sqcup \pi_Q$.

Conditional and assignment The conditional and assignment statements ($x_i := \delta$ and $\alpha x_i \otimes \delta$) create new constraints between program variables. Thus, to compute the partitions for the outputs of these operators, we first compute a block $\mathcal{B}$ which contains all variables affected by the statement. Let $\mathcal{A}$ be the set of all variables x_j with $a_j \neq 0$ in $\delta = a^T x + \epsilon$, then $\mathcal{B} = \mathcal{A} \cup \{x_i\}$. To express the fusion incurred by $\mathcal{B}$, we introduce the following:

Definition 3.2. Let π be a partition and $\mathcal{B} \subseteq \mathcal{X}$, then $\pi \uparrow \mathcal{B}$ is the finest partition π' such that $\pi \sqsubseteq \pi'$ and $\mathcal{B}$ is a subset of an element of π'.

As discussed, the operator for the conditional statement $\alpha x_i \otimes \delta$ adds constraint $(\alpha - a_i)x_i \otimes \delta - a_i x_i$ to $\mathcal{C}_P$ to produce the set $\mathcal{C}_O$ for the output O. Thus, in π_O, all blocks $\mathcal{X}_i \in \pi_P$ that overlap with $\mathcal{B}$ will merge into one, whereas non-overlapping blocks remain independent. Thus, we get the following lemma for calculating π_O.

Lemma 3.3. Let P be the input polyhedra and let $\mathcal{B}$ be the block corresponding to the conditional $\alpha x_i \otimes \delta$. If $O \neq \bot$, then $\pi_O = \pi_P \uparrow \mathcal{B}$.

π_O for the output O of the operator for the assignment $x_i := \delta$ can be computed similarly to that of the conditional operator.

Lemma 3.4. Let P be the input polyhedra and let $\mathcal{B}$ be the block corresponding to an assignment $x_i := \delta$. Then $\pi_O = \pi_P \uparrow \mathcal{B}$.

Widening Like the join, the partition for widening (∇) depends not only on partitions π_P and π_Q, but also on the exact form of P and Q. Thus, the resulting partition is not guaranteed to be optimal but only permissible. By definition, the constraint set for $P \nabla Q$ contains only constraints from Q. Thus, the partition for $P \nabla Q$ satisfies

Lemma 3.5. For polyhedra P and Q, $\pi_{P \nabla Q} \sqsubseteq \pi_Q$.

Note that the widening operator can potentially remove all constraints containing a variable, making the variable unconstrained. Thus, in general, $\pi_{P \nabla Q} \neq \pi_Q$.

Join Let $\mathcal{C}_P = \{A_1 x \leq b_1\}$ and $\mathcal{C}_Q = \{A_2 x \leq b_2\}$[3] and $\mathcal{Y} = \{x'_1, x'_2, \ldots, x'_n, \lambda\}$, then the constraint set $\mathcal{C}_{P \sqcup Q}$ for the join of P and Q can be computed by projecting out variables $y_i \in \mathcal{Y}$ from the following set $\mathcal{S}$ of constraints:

$$\mathcal{S} = \{A_1 x' \leq b_1 \lambda, A_2(x - x') \leq b_2(1 - \lambda), -\lambda \leq 0, \lambda \leq 1\}. \tag{5}$$

The Fourier-Motzkin elimination algorithm [12] is used for this projection. The algorithm starts with $\mathcal{S}_0 = \mathcal{S}$ and projects out variables iteratively one after another so that $\mathcal{C}_{P \sqcup Q} = \mathcal{S}_{n+1}$. Let $\mathcal{S}_{i-1}$ be the constraint set obtained after projecting out the first $i-1$ variables in $\mathcal{Y}$. Then $y_i \in \mathcal{Y}$ is projected out to produce $\mathcal{S}_i$ as follows:

$$\begin{aligned}
\mathcal{S}^+_{y_i} &= \{c \mid c \in \mathcal{S}_{i-1} \text{ and } a_i > 0\},\\
\mathcal{S}^-_{y_i} &= \{c \mid c \in \mathcal{S}_{i-1} \text{ and } a_i < 0\},\\
\mathcal{S}^0_{y_i} &= \{c \mid c \in \mathcal{S}_{i-1} \text{ and } a_i = 0\},\\
\mathcal{S}^\pm_{y_i} &= \{\mu c_1 + \nu c_2 \mid (c_1, c_2) \in \mathcal{S}^+_{y_i} \times \mathcal{S}^-_{y_i} \text{ and } \mu a_{1_i} + \nu a_{2_i} = 0\},\\
\mathcal{S}_i &= \mathcal{S}^0_{y_i} \cup \mathcal{S}^\pm_{y_i}.
\end{aligned} \tag{6}$$

Each iteration can potentially produce a quadratic number of new constraints, many of which are redundant. The redundant constraints are removed for efficiency.

The partition of $P \sqcup Q$ depends in non-trivial ways on P and Q. In particular, $\pi_{P \sqcup Q}$ has no general relationship to either $\pi_P \sqcup \pi_Q$ or $\pi_P \sqcap \pi_Q$. The following example illustrates this:

Example 3.4. Let

$$\begin{aligned}
P &= \{\{x_1 - x_2 \leq 0, x_1 \leq 0\}, \{x_3 = 1\}\} \text{ and}\\
Q &= \{\{x_1 \leq 2\}, \{x_3 = 0\}\} \text{ with}\\
\pi_P &= \{\{x_1, x_2\}, \{x_3\}\} \text{ and}\\
\pi_Q &= \{\{x_1\}, \{x_2\}, \{x_3\}\}.
\end{aligned}$$

In this case we have,

$$\begin{aligned}
P \sqcup Q &= \{\{x_1 + 2x_3 \leq 2, -x_3 \leq 0, x_3 \leq 1\}\} \text{ and}\\
\pi_{P \sqcup Q} &= \{\{x_1, x_3\}, \{x_2\}\}.
\end{aligned}$$

However,

$$\begin{aligned}
\pi_P \sqcup \pi_Q &= \{\{x_1, x_2\}, \{x_3\}\} \text{ and}\\
\pi_P \sqcap \pi_Q &= \{\{x_1\}, \{x_2\}, \{x_3\}\}.
\end{aligned}$$

Thus, neither $\pi_P \sqcup \pi_Q$ nor $\pi_P \sqcap \pi_Q$ are permissible partitions for $P \sqcup Q$.

The theorem below identifies a case which enables us to compute a non-trivial permissible partition for $P \sqcup Q$. The theorem states that we can "transfer" a block from the input partitions to the output partition under certain conditions. It is a key enabler for the speedups shown later.

Theorem 3.6. *Let P and Q be two polyhedra with the same permissible partition $\pi = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$ and let π' be a permissible partition for the join, that is, $\pi_{P \sqcup Q} \sqsubseteq \pi'$. If for any block $\mathcal{X}_k \in \pi$, $P_k = Q_k$, then $\mathcal{X}_k \in \pi'$.*

Proof. Since both P and Q are partitioned according to π, the constraint set in (5) can be written for each $\mathcal{X}_k$ separately:

$$\{A_{1k} x'_k \leq b_{1k} \lambda, A_{2k}(x_k - x'_k) \leq b_{2k}(1 - \lambda), -\lambda \leq 0, \lambda \leq 1\}. \tag{7}$$

[3] We assume equalities are encoded as symmetric pairs of opposing inequalities for simplicity.

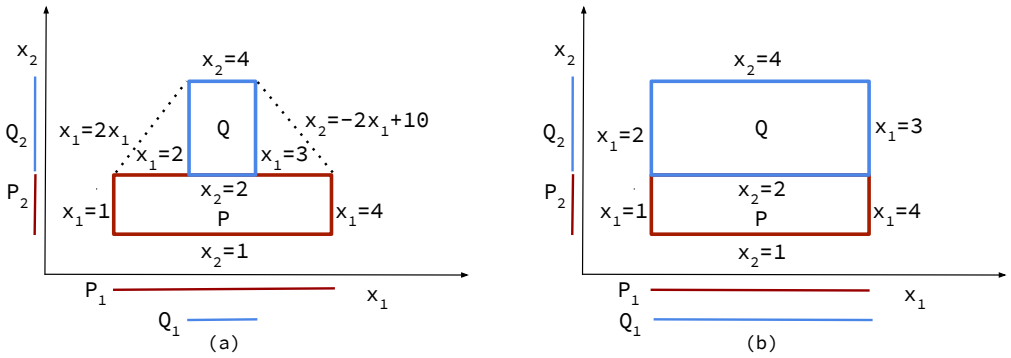

Figure 4: Two examples of $P \sqcup Q$ with $\pi_P = \pi_Q = \{\{x_1\}, \{x_2\}\}$. (a)$P_1 \neq Q_1, P_2 \neq Q_2$; (b) $P_1 = Q_1, P_2 \neq Q_2$.

where x_k is column vector for the variables in $\mathcal{X}_k$. λ occurs in the constraint set for all blocks. For proving the theorem, we need to show that no variable in $\mathcal{X}_k$ will have a constraint with a variable in $\mathcal{X}_{k'} \in \pi$ after join. The variables in $\mathcal{X}_k$ can have a constraint with the variables in $\mathcal{X}_{k'}$ only by projecting out λ. Since $P_k = Q_k$, $\mathcal{C}_{P_k}$ and $\mathcal{C}_{Q_k}$ are equivalent, we can assume $A_{1k} = A_{2k}$ and $b_{1k} = b_{2k}$.[4] Inserting this into (7) we get

$$\{A_{1k}x'_k \leq b_{1k}\lambda, A_{1k}(x_k - x'_k) \leq b_{1k}(1-\lambda), -\lambda \leq 0, \lambda \leq 1\}. \quad (8)$$

The result of the projection is independent of the order in which the variables are projected out. Thus, we can project out λ last. For proving the theorem, we need to show that it is possible to obtain all constraints for $\mathcal{C}_{P_k \sqcup Q_k}$ before projecting out λ in (8). We add $A_{1k}x'_k \leq b_{1k}\lambda$ and $A_{1k}(x_k - x'_k) \leq b_{1k}(1-\lambda)$ in (8) to project out all x'_k and obtain:

$$\{A_{1k}x_k \leq b_{1k}, -\lambda \leq 0, \lambda \leq 1\}. \quad (9)$$

Note that the constraint set in (9) does not contain all constraints generated by the Fourier-Motzkin elimination. Since $P_k = P_k \sqcup P_k$, we have $\mathcal{C}_{P_k \sqcup Q_k} = \mathcal{C}_{P_k}$ and $\mathcal{C}_{P_k}$ is included in the constraint set of (9), thus the remaining constraints generated by the Fourier-Motzkin elimination are redundant. In (9), all constraints among the variables in $\mathcal{X}_k$ are free from λ, therefore projecting out λ does not create new constraints for the variables in $\mathcal{X}_k$. Thus, there cannot be any constraint from a variable in $\mathcal{X}_k$ to a variable in $\mathcal{X}_{k'}$. □

The proof of the theorem also yields the following result.

Corollary 3.1. If Theorem 3.6 holds, then P_k (and Q_k) is a factor of $P \sqcup Q$.

Example 3.5. Fig. 4 shows two examples of $P \sqcup Q$ where both P and Q have the same partition $\pi_P = \pi_Q = \{\{x_1\}, \{x_2\}\}$. In Fig. 4(a),

$$P = \{\{x_1 = 1, x_1 = 4\}, \{x_2 = 1, x_2 = 2\}\},$$
$$Q = \{\{x_1 = 2, x_1 = 3\}, \{x_2 = 2, x_2 = 4\}\}.$$

In this case, $P_1 \neq Q_1$ and $P_2 \neq Q_2$, thus $P \sqcup Q$ contains constraints $x_2 = 2x_1$ and $x_2 = -2x_1 + 10$ relating x_1 and x_2, i.e., $\pi_{P \sqcup Q} = \{\{x_1, x_2\}\}$.

In Fig. 4(b),

$$P = \{\{x_1 = 1, x_1 = 4\}, \{x_2 = 1, x_2 = 2\}\},$$
$$Q = \{\{x_1 = 1, x_1 = 4\}, \{x_2 = 2, x_2 = 4\}\}.$$

[4] One can always perform a transformation so that $A_{1k} = A_{2k}$ and $b_{1k} = b_{2k}$ holds.

In this case, $P_1 = Q_1$. Thus, by Theorem 3.6, $\{x_1\} \in \pi_{P \sqcup Q}$, i.e., $\pi_{P \sqcup Q} = \{\{x_1\}, \{x_2\}\}$.

4. Polyhedra Domain Analysis with Partitions

After presenting the theoretical background, we now discuss how we integrate partitioning in the entire analysis flow. The basic idea is to perform the analysis while maintaining the variable set partitioned, and thus the occurring polyhedra decomposed, as fine-grain as possible. The results from the previous section show that the main Polyhedra operators can indeed maintain the partitions, even though these partitions change during the analysis. Crucially, under certain assumptions, even the join produces a non-trivial partitioned output. Note that there are no guarantees that for a given program, the partitions do not become trivial (i.e., equal to $\{\mathcal{X}\}$); however, as our results later show, this is typically not the case and thus significant speedups are obtained. This should not be surprising: in complex programs, not all variables used are related to each other, However, there will be groups of variables that indeed develop relationships and these groups may change during execution. Our approach identifies and maintains such groups.

Maintaining precision We emphasize that partitioning the variable set and thus decomposing polyhedra and operators working on polyhedra, does not affect the overall precision of the result. That is, we neither lose nor gain precision in our analysis compared to prior approaches which do not use online partitioning. *The granularity of a partition only affects the cost, i.e., runtime and memory space*, required for the analysis, but not the precision of its results.

We now briefly discuss the data structures used for polyhedra and the maintenance of permissible partitions throughout the analysis. For the remainder of the paper, permissible partitions will be denoted with $\overline{\pi}_P \sqsupseteq \pi_P$. The following sections then provide more details on the respective operators.

4.1 Polyhedra Encoding

For a given polyhedron, NewPolka and PPL store both, the constraint set $\mathcal{C}$ and the generator set $\mathcal{G}$, each represented as a matrix. We follow a similar approach adapted to our partitioned scenario. Specifically, assume a polyhedron P with permissible partition $\overline{\pi}_P = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$, i.e., associated factors $\{P_1, P_2, \ldots, P_r\}$, where $P_k = (\mathcal{C}_{P_k}, \mathcal{G}_{P_k})$. The blocks of $\overline{\pi}_P$ are stored as a linked list of variables and the polyhedron as a linked list of factors. Each factor is stored as two matrices. We do not explicitly store the factors and the blocks for the unconstrained variables. For example, $\top$ is stored as $\emptyset$.

Table 2: Asymptotic time complexity of Polyhedra operators with decomposition.

Operator	Decomposed
Inclusion ($\sqsubseteq$)	$O(\sum_{i=1}^{r} n_i g_i m_i)$
Join ($\sqcup$)	$O(\sum_{i=1}^{r} n_i g_i m_i + n_{\max} g_{\max})$
Meet ($\sqcap$)	$O(\sum_{i=1}^{r} n_i m_i)$
Widening (∇)	$O(\sum_{i=1}^{r} n_i g_i m_i)$
Conditional	$O(n_{\max})$
Assignment	$O(n_{\max} g_{\max})$

4.2 Operators and Permissible Partitions

The results in Section 3.2 calculated for each input polyhedra P, Q with partitions π_P, π_Q either the best (finest) or a permissible partition of the output polyhedron O of an operator. Inspection shows that each result can be adapted to the case where the input partitions are only permissible. In this case, the output partition is likewise only permissible.

Lemma 4.1. Given permissible input partitions $\overline{\pi}_P$ and $\overline{\pi}_Q$, Lemmas 3.2–3.5 and Theorem 3.6 yield permissible partitions for the outputs of operators. Specifically, using prior notation:

i) *Meet:* $\overline{\pi}_{P \sqcap Q} = \overline{\pi}_P \sqcup \overline{\pi}_Q$ is permissible if $P \sqcap Q \neq \bot$, otherwise $\bot$ is permissible.
ii) *Conditional:* $\overline{\pi}_P \uparrow \mathcal{B}$ is permissible if $O \neq \bot$, otherwise $\bot$ is permissible.
iii) *Assignment:* $\overline{\pi}_P \uparrow \mathcal{B}$ is permissible.
iv) *Widening*: $\overline{\pi}_{P \nabla Q} = \overline{\pi}_Q$ is permissible.
v) *Join:* Let $\overline{\pi} = \overline{\pi}_P \sqcup \overline{\pi}_Q$ and $\mathcal{U} = \{\mathcal{X}_k \mid P_k = Q_k, \mathcal{X}_k \in \overline{\pi}\}$. Then the following is permissible:

$$\overline{\pi}_{P \sqcup Q} = \mathcal{U} \ \cup \bigcup_{\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}} \mathcal{T}$$

Table 2 shows the asymptotic time complexity of the Polyhedra operators decomposed with our approach. For simplicity, we assume that for binary operators both inputs have the same partition. In the table, r is the number of blocks in the partition, n_i is the number of variables in the i-th block, g_i and m_i are the number of generators and constraints in the i-th factor respectively. It holds that $n = \sum_{i=1}^{r} n_i$, $m = \sum_{i=1}^{r} m_i$ and $g = \prod_{i=1}^{r} g_i$. We denote the number of variables and generators in the largest block by $n_{\max}$ and $g_{\max}$, respectively. Since we follow the eager approach for conversion, both representations are available for inputs, i.e., the second column of Table 2 corresponds to column *Both* in Table 1. We do not show the cost of conversion.

Fig. 5 shows a representative program annotated with Polyhedra invariants at each program point. The program contains five variables u, v, x, y, z and has two conditional if-statements. It can be seen that the Polyhedra at different program points can be decomposed and thus the Polyhedra operators benefit from the complexity reduction. For example, the assignment operator for `x:=2y` and the conditional operator for `x==y` need to operate only on the factor corresponding to the block $\{x, y\}$. The assignment operator for `u:=3v` and the conditional operator for `u==v` benefit similarly. We next discuss the algorithms for core operators using partitions.

5. Polyhedra Operators

In this section, we describe our algorithms for the main Polyhedra operators. For each operator, we first describe the base algorithm, followed by our adaptation of that algorithm to use partitions. We also discuss useful code optimizations for our algorithms. We follow an eager approach for the conversion, thus the inputs and the output have both $\mathcal{C}$ and $\mathcal{G}$ available. Join is the most challenging

$\mathcal{P}_1 : \top$
`x:=5;`
$\mathcal{P}_2 : \{\{x = 5\}\}$
`u:=3;`
$\mathcal{P}_3 : \{\{x = 5\}, \{u = 3\}\}$
`if(x==y){`
$\mathcal{P}_4 : \{\{x = 5, x = y\}, \{u = 3\}\}$
`x:=2y;`
$\mathcal{P}_5 : \{\{y = 5, x = 2y\}, \{u = 3\}\}$
`}`
$\mathcal{P}_6 : \{\{-x \leq -5, x \leq 10\}, \{u = 3\}\}$
`if(u==v){`
$\mathcal{P}_7 : \{\{-x \leq -5, x \leq 10\}, \{u = 3, u = v\}\}$
`u :=3v;`
$\mathcal{P}_8 : \{\{-x \leq -5, x \leq 10\}, \{v = 3, u = 3v\}\}$
`}`
$\mathcal{P}_9 : \{\{-x \leq -5, x \leq 10\}, \{-u \leq -3, u \leq 9\}\}$
`z:=x + u;`
$\mathcal{P}_{10} : \{\{-x \leq -5, x \leq 10, -u \leq -3, u \leq 9, -z \leq -8, z \leq 19\}\}$

Figure 5: Example of complexity reduction through decomposition for Polyhedra analysis on an example program.

operator to adapt with partitions as the partition for the output depends on the exact form of the inputs. Our algorithms rely on two auxiliary operators, *conversion* and *refactoring*, which we describe first.

5.1 Auxiliary Operators

We apply code optimizations to leverage sparsity in the conversion algorithm which makes our conversion faster. Refactoring is frequently required by our algorithms to make the inputs conform to the same partition.

Conversion operator An expensive step in Chernikova's algorithm is the computation of a matrix-vector product which is needed at each iteration of the algorithm. We observed that the vector is usually sparse, i.e., it contains mostly zeros, thus we need to consider only those entries in the matrix which can be multiplied with the non-zero entries in the vector. Therefore at the start of each iteration, we compute an index for the non-zero entries of the vector. The index is discarded at the end of the iteration. This code optimization significantly reduces the cost of conversion.

We also vectorized the matrix-vector product using AVX intrinsics, however it does not provide as much speedup compared to leveraging sparsity by keeping the index.

Refactoring Let P and Q be defined over the same set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$, and let $\overline{\pi}_P = \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$, $\overline{\pi}_Q = \{\mathcal{X}_{Q_1}, \mathcal{X}_{Q_2}, \ldots, \mathcal{X}_{Q_q}\}$ be permissible partitions for P and Q respectively and $\mathcal{B} \subseteq \mathcal{X}$. Usually $\overline{\pi}_P \neq \overline{\pi}_Q$, thus an important step for the operators such as meet, inclusion testing, widening and join is refactoring the inputs P and Q so that the factors correspond to the same partition $\overline{\pi}$ which is simply the least upper bound $\overline{\pi}_P \sqcup \overline{\pi}_Q$.

Similarly, usually $\mathcal{B} \notin \overline{\pi}_P$ for the conditional and the assignment operators. Thus, P is refactored according to $\overline{\pi} = \overline{\pi}_P \uparrow \mathcal{B}$.

P is refactored by merging all factors P_i whose corresponding blocks $\mathcal{X}_{P_i}$ are included inside the same block $\mathcal{X}_j$ of $\overline{\pi}$. The merg-

Algorithm 1 Refactor P with partition $\overline{\pi}_P$ based on $\overline{\pi}$

function REFACTOR($P, \overline{\pi}_P, \overline{\pi}$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $\overline{\pi} \leftarrow \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$
  for $k \in \{1, 2, \ldots, r\}$ **do**
    $P'_k := \top$
  for $i \in \{1, 2, \ldots, p\}$ **do**
    $k := j,\ s.t., \mathcal{X}_{P_i} \subseteq \mathcal{X}_j, \mathcal{X}_j \in \overline{\pi}$
    $P'_k := P'_k \bowtie P_i$
  $P' := \{P'_1, P'_2, \ldots, P'_r\}$
return P'

ing is performed using the $\bowtie$ operator defined in (4). Refactoring is shown in Algorithm 1. We will use r to denote the number of blocks in $\overline{\pi}$.

Example 5.1. Consider[5]:

$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$,
$P = \{\{x_1 = x_2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 1\}, \{x_6 = 2\}\}$,
$Q = \{\{x_1 = 2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 2\}, \{x_6 = 3\}\}$, with
$\overline{\pi}_P = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\}$ and
$\overline{\pi}_Q = \{\{x_1, x_2, x_4\}, \{x_3\}, \{x_5\}, \{x_6\}\}$.

In this case, $\overline{\pi}$ is:

$$\overline{\pi} = \overline{\pi}_P \sqcup \overline{\pi}_Q = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}.$$

We find that both blocks $\overline{\pi}_{P_1} = \{x_1, x_2\}$ and $\overline{\pi}_{P_2} = \{x_3, x_4\}$ of $\overline{\pi}_P$ are included in the first block of $\overline{\pi}_P \sqcup \overline{\pi}_Q$, thus P_1 and P_2 are merged using the $\bowtie$ operator. We merge Q_1 and Q_2 similarly. The resulting P' and Q' are shown below:

$$P' = \{P_1 \bowtie P_2, \{x_5 = 1\}, \{x_6 = 2\}\} \text{ and}$$
$$Q' = \{Q_1 \bowtie Q_2, \{x_5 = 2\}, \{x_6 = 3\}\}$$

where,

$$P_1 \bowtie P_2 = \{x_1 = x_2, x_2 = 2, x_3 \leq 2\} \text{ and}$$
$$Q_1 \bowtie Q_2 = \{x_1 = 2, x_2 = 2, x_3 \leq 2\}$$

After explaining refactoring, we now present our algorithms for the Polyhedra operators with partitions.

5.2 Meet ($\sqcap$)

For the double representation, $\mathcal{C}_{P \sqcap Q}$ is the union of the constraints of the input polyhedra P and Q , i.e., $\mathcal{C}_{P \sqcap Q} = \mathcal{C}_P \cup \mathcal{C}_Q$. If $\mathcal{C}_{P \sqcap Q}$ is unsatisfiable, then $P \sqcap Q = \bot$. $\mathcal{G}_{P \sqcap Q}$ is obtained by incrementally adding the constraints in $\mathcal{C}_Q$ to the polyhedron defined by $\mathcal{G}_P$ through the conversion operator. If $\mathcal{C}_{P \sqcap Q}$ is unsatisfiable, then the conversion returns $\mathcal{G}_{P \sqcap Q} = \emptyset$, .

Meet with partitions Our algorithm first computes the same partition $\overline{\pi}_P \sqcup \overline{\pi}_Q$. P and Q are then refactored according to this partition using Algorithm 1 to obtain P' and Q'. If $P'_k = Q'_k$, then $\mathcal{C}_{P'_k} \cup \mathcal{C}_{Q'_k} = \mathcal{C}_{P'_k}$ and therefore we add P'_k to O, otherwise we add $\mathcal{C}_{P'_k} \cup \mathcal{C}_{Q'_k}$ to $\mathcal{C}_{P \sqcap Q}$.

If $P'_k = Q'_k$, then no conversion is required, otherwise the constraints in $\mathcal{C}_{Q'_k}$ are incrementally added to the polyhedron defined by $\mathcal{G}_{P'_k}$ through the conversion. If the conversion algorithm returns $\mathcal{G}_{P'_k \sqcap Q'_k} = \emptyset$, then we set $P \sqcap Q = \bot$. We know from Section 4 that $\overline{\pi}_{P \sqcap Q} = \overline{\pi}_P \sqcup \overline{\pi}_Q$ if $P \sqcap Q \neq \bot$, otherwise $\overline{\pi}_{P \sqcap Q} = \bot$.

[5] We show only constraints for simplicity.

Algorithm 2 Polyhedra Meet

function MEET($P, Q, \overline{\pi}_P, \overline{\pi}_Q$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $Q \leftarrow \{Q_1, Q_2, \ldots, Q_q\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $\overline{\pi}_Q \leftarrow \{\mathcal{X}_{Q_1}, \mathcal{X}_{Q_2}, \ldots, \mathcal{X}_{Q_q}\}$
  $P' :=$ refactor$(P, \overline{\pi}_P, \overline{\pi}_P \sqcup \overline{\pi}_Q)$
  $Q' :=$ refactor$(Q, \overline{\pi}_Q, \overline{\pi}_P \sqcup \overline{\pi}_Q)$
  $O = \emptyset$
  for $k \in \{1, 2, \ldots, r\}$ **do**
    if $P'_k = Q'_k$ **then**
      O.add(P'_k)
    else
      $\mathcal{C} :=$ remove_common_con$(\mathcal{C}_{P'_k} \cup \mathcal{C}_{Q'_k})$
      $\mathcal{G} :=$ incr_chernikova$(\mathcal{C}, \mathcal{C}_{P'_k}, \mathcal{G}_{P'_k})$
      if $\mathcal{G} = \emptyset$ **then**
        $O := \bot$
        $\overline{\pi}_O := \bot$ **return**
      O.add$((\mathcal{C}, \mathcal{G}))$
  $\overline{\pi}_O := \overline{\pi}_P \sqcup \overline{\pi}_Q$

Code optimization $\mathcal{C}_{P'_k}$ and $\mathcal{C}_{Q'_k}$ usually contain a number of common constraints. The generators in $\mathcal{G}_{P'_k}$ already correspond to the constraints that occur in both $\mathcal{C}_{P'_k}$ and $\mathcal{C}_{Q'_k}$. Thus, these constraints can be removed. This further reduces the cost of the conversion.

The check for common constraints can create an overhead as in the worst case we have to compare each vector in $\mathcal{C}_{Q'_k}$ with all vectors in $\mathcal{C}_{P'_k}$. To reduce this overhead, for a given vector in $\mathcal{C}_{Q'_k}$, we keep track of the vector index which caused the equality check to fail for the previous vector in $\mathcal{C}_{P'_k}$. For the next vector in $\mathcal{C}_{P'_k}$, we first compare the vector values at this index as the next vector, if not equal, is also likely to fail this check. The pseudo code for our meet operator is shown in Algorithm 2.

5.3 Inclusion ($\sqsubseteq$)

For the double representation, $P \sqsubseteq Q$ holds if all generators in $\mathcal{G}_P$ *satisfy* all constraints in $\mathcal{C}_Q$. A vertex $v \in \mathcal{V}_P$ satisfies the constraint set $\mathcal{C}_Q$ if $Av \leq b$ and $Dv = e$. A ray $r \in \mathcal{R}_P$ satisfies $\mathcal{C}_Q$ if $Ar \leq 0$ and $Dr = 0$. A line $z \in \mathcal{Z}_P$ satisfies $\mathcal{C}_Q$ if $Az = 0$ and $Dz = 0$.

Inclusion testing with partitions In our algorithm, we refactor P and Q according to the same partition $\overline{\pi}_P \sqcup \overline{\pi}_Q$. We only refactor the generators of P and the constraints of Q according to $\overline{\pi}_P \sqcup \overline{\pi}_Q$, obtaining $\mathcal{G}_{P'}$ and $\mathcal{C}_{Q'}$ respectively. We then check for each block $\mathcal{X}_k$ in $\overline{\pi}_P \sqcup \overline{\pi}_Q$ if all generators in $\mathcal{G}_{P'_k}$ satisfy $\mathcal{C}_{Q'_k}$.

Code optimization The result of the inclusion testing operator is usually negative, so we first check the smaller factors for inclusion. Thus, the factors are sorted in the order given by the product of the number of generators in $\mathcal{G}_{P'_k}$ and the number of constraints in $\mathcal{C}_{Q'_k}$. The pseudo code for our inclusion testing operator is shown in Algorithm 3.

5.4 Conditional

For the double representation, the operator for the conditional statement $\alpha x_i \otimes \delta$ adds the constraint $c = (\alpha - a_i)x_i \otimes \delta - a_i x_i$ to the constraint set $\mathcal{C}_P$, producing $\mathcal{C}_O$. If $\mathcal{C}_O$ is unsatisfiable, then $O = \bot$. $\mathcal{G}_O$ is obtained by incrementally adding the constraint c to the polyhedron defined by $\mathcal{G}_P$ though the conversion. The conversion returns $\mathcal{G}_O = \emptyset$, if $\mathcal{C}_O$ is unsatisfiable.

Algorithm 3 Inclusion testing

function INCLUSION($P, Q, \overline{\pi}_P, \overline{\pi}_Q$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $Q \leftarrow \{Q_1, Q_2, \ldots, Q_q\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $\overline{\pi}_Q \leftarrow \{\mathcal{X}_{Q_1}, \mathcal{X}_{Q_2}, \ldots, \mathcal{X}_{Q_q}\}$
  $\mathcal{G}_{P'} :=$ refactor_gen($\mathcal{G}_P, \overline{\pi}_P, \overline{\pi}_P \sqcup \overline{\pi}_Q$)
  $\mathcal{C}_{Q'} :=$ refactor_con($\mathcal{C}_Q, \overline{\pi}_Q, \overline{\pi}_P \sqcup \overline{\pi}_Q$)
  sort_by_size($\mathcal{G}_{P'}, \mathcal{C}_{Q'}$)
  for $k \in \{1, 2, \ldots, r\}$ **do**
    if $P'_k \not\sqsubseteq Q'_k$ **then return** false
return true

Algorithm 4 Conditional operator

function CONDITIONAL($P, \overline{\pi}_P, stmt$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $stmt \leftarrow \alpha x_i \otimes \delta$
  $\mathcal{B} :=$ extract_block($stmt$)
  $P' :=$ refactor($P, \overline{\pi}_P, \overline{\pi}_P \uparrow \mathcal{B}$)
  $O := \emptyset$
  $\overline{\pi}_O := \overline{\pi}_P \uparrow \mathcal{B}$
  for $k \in \{1, 2, \ldots, r\}$ **do**
    if $\mathcal{B} \subseteq \overline{\pi}_{O_k}$ **then**
      $\mathcal{C} := \mathcal{C}_{P'_k} \cup \{(\alpha - a_i)x_i \otimes \delta - a_i x_i\}$
      $\mathcal{G} :=$ incr_chernikova($\mathcal{C}, \mathcal{C}_{P'_k}, \mathcal{G}_{P'_k}$)
      if $\mathcal{G} = \emptyset$ **then**
        $O := \bot$
        $\overline{\pi}_O := \bot$ **return**
      O.add($(\mathcal{C}, \mathcal{G})$)
    else
      O.add(P'_k)

Conditional operator with partitions Our algorithm refactors P according to $\overline{\pi}_P \uparrow \mathcal{B}$, producing P'. The constraint c is added to the constraint set $\mathcal{C}_{P'_k}$ of the factor corresponding to the block $\mathcal{X}_k \in \overline{\pi}_P \uparrow \mathcal{B}$ containing $\mathcal{B}$, producing $\mathcal{C}_{O_k}$. $\mathcal{G}_{O_k}$ is obtained by incrementally adding the constraint c to the polyhedron defined by $\mathcal{G}_{P'_k}$. If the conversion algorithm returns $\mathcal{G}_{O_k} = \emptyset$, then we set $O = \bot$. As shown in Section 4, $\overline{\pi}_O = \overline{\pi}_P \uparrow \mathcal{B}$ if $O \neq \bot$, otherwise $\overline{\pi}_O = \bot$. The pseudo code for our conditional operator is shown in Algorithm 4.

5.5 Assignment

In Section 2, the operator for the assignment $x_i := \delta$, where $\delta = a^T x + \epsilon$, was defined using the constraint set $\mathcal{C}_P$ of P. For the double representation, the operator works on the generator set $\mathcal{G}_P = \{\mathcal{V}_P, \mathcal{R}_P, \mathcal{Z}_P\}$. The generators $\mathcal{G}_O = \{\mathcal{V}_O, \mathcal{R}_O, \mathcal{Z}_O\}$ for the output are given by:

$$\begin{aligned} \mathcal{V}_O &= \{v' \mid v'_i = a^T v + \epsilon, v \in \mathcal{V}_P\}, \\ \mathcal{R}_O &= \{r' \mid r'_i = a^T r, r \in \mathcal{R}_P\}, \\ \mathcal{Z}_O &= \{z' \mid z'_i = a^T z, z \in \mathcal{Z}_P\}. \end{aligned} \tag{10}$$

If the assignment is invertible, i.e., if $a_i \neq 0$ (for example `x:=x+1`), the constraint set $\mathcal{C}_O$ can be calculated by backsubstitution. Let x'_i be the new value of x_i after assignment, then $x'_i = a^T x + \epsilon$. Thus, putting $x_i = (x'_i - \sum_{j \neq i} a_j x_j - \epsilon)/a_i$ for x_i in all constraints of the set $\mathcal{C}_P = \{Ax \leq b, Dx = e\}$ and renaming x'_i to x_i, we get the constraint set $\mathcal{C}_O$. For the non-invertible assignments, the conversion algorithm is applied on all generators in $\mathcal{G}_O$.

Algorithm 5 Assignment operator

function ASSIGNMENT($P, \overline{\pi}_P, stmt$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $stmt \leftarrow x_i := a^T x + \epsilon$
  $\mathcal{B} :=$ extract_block($stmt$)
  $P' :=$ refactor($P, \overline{\pi}_P, \overline{\pi}_P \uparrow \mathcal{B}$)
  $O := \emptyset$
  $\overline{\pi}_O := \overline{\pi}_P \uparrow \mathcal{B}$
  for $k \in \{1, 2, \ldots, r\}$ **do**
    if $\mathcal{B} \subseteq \overline{\pi}_{O_k}$ **then**
      $\mathcal{G} :=$ handle_assign($\mathcal{G}_{P'_k}, stmt$)
      if $a_i = 0$ **then**
        $\mathcal{C} :=$ backsubstitute($\mathcal{G}, stmt$)
      else
        $\mathcal{C} :=$ chernikova($\mathcal{G}$)
      O.add($(\mathcal{C}, \mathcal{G})$)
    else
      O.add(P'_k)

Assignment operator with partitions In our algorithm, we refactor P according to $\overline{\pi}_P \uparrow \mathcal{B}$, producing P'. We compute the new generators using (10) only for the factor P'_k corresponding to the block $\mathcal{X}_k \in \overline{\pi}_P \uparrow \mathcal{B}$ containing $\mathcal{B}$. The constraints are computed only for P'_k for both invertible and non-invertible assignments. This results in a large reduction of the operation count. As shown in Section 4, $\overline{\pi}_O = \overline{\pi}_P \uparrow \mathcal{B}$. The pseudo code for our assignment operator is shown in Algorithm 5. The `handle_assign` function applies (10) on $\mathcal{G}_{P'_k}$.

5.6 Widening (∇)

For the double representation, the widening operator requires the generators and the constraints of P and the constraints of Q. A given constraint $ax \otimes b$, where $\otimes \in \{\leq, =\}$, *saturates* a vertex $v \in \mathcal{V}$ if $av = b$, a ray $r \in \mathcal{R}$ if $ar = 0$, and a line $z \in \mathcal{Z}$ if $az = 0$.

For given constraint c and $\mathcal{G}$, the set $\mathcal{S}_{c,\mathcal{G}}$ is defined as:

$$\mathcal{S}_{c,\mathcal{G}} = \{g \mid g \in \mathcal{G} \text{ and } c \text{ saturates } g\}. \tag{11}$$

The standard widening operator computes for each constraint $c_p \in \mathcal{C}_P$, the set $\mathcal{S}_{c_p,\mathcal{G}_P}$ and for each constraint $c_q \in \mathcal{C}_Q$, the set $\mathcal{S}_{c_q,\mathcal{G}_P}$. If $\mathcal{S}_{c_q,\mathcal{G}_P} = \mathcal{S}_{c_p,\mathcal{G}_P}$ for any c_p, then c_q is added to the output constraint set $\mathcal{C}_{P \nabla Q}$. The widening operator removes the constraints from $\mathcal{C}_Q$, so the conversion is not incremental in the standard implementations. Recent work [24] allows incremental conversion when constraints or generators are removed.

Widening with partitions In our algorithm, we refactor P according to $\overline{\pi}_P \sqcup \overline{\pi}_Q$, producing P'. For a given constraint $c_q \in \mathcal{C}_{Q_i}$, we access the block $\mathcal{X}_k \in \overline{\pi}_P \sqcup \overline{\pi}_Q$ containing $\mathcal{X}_{Q_i}$ and compute $\mathcal{S}_{c_q,\mathcal{G}_{P'_k}}$. If this set is equal to $\mathcal{S}_{c_p,\mathcal{G}_{P'_k}}$ for any $c_p \in \mathcal{C}_{P'_k}$, then c_q is added to $\mathcal{C}_{O_i}$. If $\mathcal{C}_{O_i} = \mathcal{C}_{Q_i}$, then the conversion is not required, otherwise it is applied on all constraints in $\mathcal{C}_{O_i}$. As shown in Section 4, $\overline{\pi}_{P \nabla Q} = \overline{\pi}_Q$. The pseudo code for our widening operator is shown in Algorithm 6. The `saturate` function applies (11) on given c and $\mathcal{G}$.

To possibly improve the granularity for $\overline{\pi}_O$, we check if for any block $\mathcal{X}_k \in \overline{\pi}_{P \nabla Q}$, $\mathcal{C}_{O_k} = \emptyset$; if yes, then $\mathcal{X}_k$ is removed from

Algorithm 6 Polyhedra widening

function WIDENING($P, Q, \overline{\pi}_P, \overline{\pi}_Q$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $Q \leftarrow \{Q_1, Q_2, \ldots, Q_q\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $\overline{\pi}_Q \leftarrow \{\mathcal{X}_{Q_1}, \mathcal{X}_{Q_2}, \ldots, \mathcal{X}_{Q_q}\}$
  $P' :=$ refactor$(P, \overline{\pi}_P, \overline{\pi}_P \sqcup \overline{\pi}_Q)$
  $O := \emptyset$
  for $k \in \{1, 2, \ldots, r\}$ **do**
    for $c_p \in \mathcal{C}_{P'_k}$ **do**
      $\mathcal{S}_{c_p, \mathcal{G}_{P'_k}} :=$ saturate$(c_p, \mathcal{G}_{P'_k})$
  for $i \in \{1, 2, \ldots, q\}$ **do**
    $\mathcal{C}_{O_i} := \emptyset$
    $k := j,\ s.t., \mathcal{X}_{Q_i} \subseteq \mathcal{X}_j, \mathcal{X}_j \in \overline{\pi}_P \sqcup \overline{\pi}_Q$
    for $c_q \in \mathcal{C}_{Q_i}$ **do**
      $\mathcal{S}_{c_q, \mathcal{G}_{P'_k}} :=$ saturate$(c_q, \mathcal{G}_{P'_k})$
      if $\exists c_p \in \mathcal{C}_{P'_k}$, s.t., $\mathcal{S}_{c_q, \mathcal{G}_{P'_k}} = \mathcal{S}_{c_p, \mathcal{G}_{P'_k}}$ **then**
        $\mathcal{C}_{O_i} := \mathcal{C}_{O_i} \cup \{c_q\}$
    if $\mathcal{C}_{O_i} = \mathcal{C}_{Q_i}$ **then**
      O.add(Q_i)
    else
      $\mathcal{G}_{O_i} :=$ chernikova$(\mathcal{C}_{O_i})$
      O.add$((\mathcal{C}_{O_i}, \mathcal{G}_{O_i}))$

$\overline{\pi}_{P \triangledown Q}$ and replaced by a set of singleton blocks with each block corresponding to a variable in $\mathcal{X}_k$.

5.7 Join ($\sqcup$)

For the double representation, the generators $\mathcal{G}_O$ of the output $O = P \sqcup Q$ of the join are simply the union of the generators of the input polyhedra, i.e., $\mathcal{G}_O = \mathcal{G}_P \cup \mathcal{G}_Q$. $\mathcal{C}_O$ is obtained by incrementally adding the generators in $\mathcal{G}_Q$ to the polyhedron defined by $\mathcal{C}_P$.

Join with partitions In our join operator shown in Algorithm 8, we refactor P and Q according to $\overline{\pi}_P \sqcup \overline{\pi}_Q$, obtaining P' and Q' respectively. The join operator can create constraints between the variables in different blocks of $\overline{\pi}_P \sqcup \overline{\pi}_Q$. In the worst case, the join can merge all blocks into one to produce the $\top$ partition, which blows up the number of generators due to the Cartesian product in (4). However, in many cases common in the program analysis setting, the blocks of $\overline{\pi}_P \sqcup \overline{\pi}_Q$ need not be combined without sacrificing precision. Identifying such cases is key in our work for avoiding the exponential blowup observed by prior libraries [2, 13]. Theorem 3.6 identifies such cases.

Computing the generators for the join If $P'_k = Q'_k$ holds, then P'_k can be added to O by Corollary 3.1. Since no new generators are added, the conversion is not required for these. This results in a large reduction of the operation count for the conversion.

As in Section 4, $\overline{\pi} = \overline{\pi}_P \sqcup \overline{\pi}_Q, \mathcal{U} = \{\mathcal{X}_k \mid P'_k = Q'_k\}$. The factors in P' and Q' corresponding to the blocks $\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}$ are merged using the $\bowtie$ operator to produce $P'_\mathcal{T}$ and $Q'_\mathcal{T}$ respectively. Thus $\mathcal{G}_O = \{\mathcal{G}_{P'_{\mathcal{U}_1}}, \mathcal{G}_{P'_{\mathcal{U}_2}}, \ldots, \mathcal{G}_{P'_{\mathcal{U}_u}}, \mathcal{G}_{P'_\mathcal{T}} \cup \mathcal{G}_{Q'_\mathcal{T}}\}$ where $u = |\mathcal{U}|$. The pseudo code for this step is shown in Algorithm 7.

Computing the constraints for the join We know the constraint set for all factors corresponding to the blocks in $\mathcal{U}$. $\mathcal{C}_{P'_\mathcal{T} \cup Q'_\mathcal{T}}$ is obtained by incrementally adding the generators in $\mathcal{G}_{Q'_\mathcal{T}}$ to the polyhedron defined by $\mathcal{C}_{P'_\mathcal{T}}$.

Algorithm 7 Compute Generators for the join

function COMPUTE_GEN_JOIN($P', Q', \overline{\pi}_P \sqcup \overline{\pi}_Q$)
  Parameters:
    $P' \leftarrow \{P'_1, P'_2, \ldots, P'_r\}$
    $Q' \leftarrow \{Q'_1, Q'_2, \ldots, Q'_r\}$
    $\overline{\pi}_P \sqcup \overline{\pi}_Q \leftarrow \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$
  $\mathcal{U} := \emptyset$
  $\overline{\pi}_O := \emptyset$
  $P'_\mathcal{T} := Q'_\mathcal{T} := \top$
  $O := \emptyset$
  for $k \in \{1, 2, \ldots, r\}$ **do**
    if $P'_k = Q'_k$ **then**
      $\mathcal{U}$.add$(\mathcal{X}_k)$
      O.add(P'_k)
    else
      $\overline{\pi}_O := \overline{\pi}_O \cup \mathcal{X}_k$
      $P'_\mathcal{T} := P'_\mathcal{T} \bowtie P'_k$
      $Q'_\mathcal{T} := Q'_\mathcal{T} \bowtie Q'_k$
  $\overline{\pi}_O := \mathcal{U} \cup \overline{\pi}_O$
return $\overline{\pi}_O, O, P'_\mathcal{T}, Q'_\mathcal{T}$

Algorithm 8 Polyhedra join

function JOIN($P, Q, \overline{\pi}_P, \overline{\pi}_Q$)
  Parameters:
    $P \leftarrow \{P_1, P_2, \ldots, P_p\}$
    $Q \leftarrow \{Q_1, Q_2, \ldots, Q_q\}$
    $\overline{\pi}_P \leftarrow \{\mathcal{X}_{P_1}, \mathcal{X}_{P_2}, \ldots, \mathcal{X}_{P_p}\}$
    $\overline{\pi}_Q \leftarrow \{\mathcal{X}_{Q_1}, \mathcal{X}_{Q_2}, \ldots, \mathcal{X}_{Q_q}\}$
  $P' :=$ refactor$(P, \overline{\pi}_P, \overline{\pi}_P \sqcup \overline{\pi}_Q)$
  $Q' :=$ refactor$(Q, \overline{\pi}_Q, \overline{\pi}_P \sqcup \overline{\pi}_Q)$
  $(\overline{\pi}_O, O, P'_\mathcal{T}, Q'_\mathcal{T}) :=$ compute_gen_join$(P', Q', \overline{\pi}_P \sqcup \overline{\pi}_Q)$
  $\mathcal{G} :=$ remove_common_gen$(\mathcal{G}_{P'_\mathcal{T}} \cup \mathcal{G}_{Q'_\mathcal{T}})$
  $\mathcal{C} :=$ incr_chernikova$(\mathcal{G}, \mathcal{G}_{P'_\mathcal{T}}, \mathcal{C}_{P'_\mathcal{T}})$
  O.add$((\mathcal{C}, \mathcal{G}))$

Example 5.2. Consider

$\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$,
$P = \{\{x_1 = x_2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 1\}, \{x_6 = 2\}\}$,
$Q = \{\{x_1 = 2, x_2 = 2\}, \{x_3 \leq 2\}, \{x_5 = 2\}, \{x_6 = 3\}\}$ with
$\overline{\pi}_P = \{\{x_1, x_2\}, \{x_3, x_4\}, \{x_5\}, \{x_6\}\}$ and
$\overline{\pi}_Q = \{\{x_1, x_2, x_4\}, \{x_3\}, \{x_5\}, \{x_6\}\}$

In this case, the refactoring gives us,

$$\overline{\pi}_P \sqcup \overline{\pi}_Q = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\},$$
$$P' = \{\{x_1 = x_2, x_2 = 2, x_3 \leq 2\}, \{x_5 = 1\}, \{x_6 = 2\}\},$$
$$Q' = \{\{x_1 = 2, x_2 = 2, x_3 \leq 2\}, \{x_5 = 2\}, \{x_6 = 3\}\}.$$

We observe that only $P'_1 = Q'_1$, thus we add P'_1 to the join O and $\{x_1, x_2, x_3, x_4\}$ to $\mathcal{U}$. Applying Algorithm 7 we get,

$$\mathcal{T} = \{\{x_5\}, \{x_6\}\},$$
$$P'_\mathcal{T} = \{x_5 = 1, x_6 = 2\},$$
$$Q'_\mathcal{T} = \{x_5 = 2, x_6 = 3\},$$
$$O = \{\{x_1 = x_2, x_2 = 2, x_3 \leq 2\}\},$$
$$\overline{\pi}_O = \{\{x_1, x_2, x_3, x_4\}, \{x_5, x_6\}\}.$$

$\mathcal{G}_{Q'_\mathcal{T}}$ contains only one vertex $(2, 3)$. The conversion operator incrementally adds this vertex to the polyhedron defined by $\mathcal{C}_{\mathcal{P}'_\mathcal{T}}$.

Thus, the factors O_1 and O_2 of $O = \{O_1, O_2\}$ are given by,

$$O_1 = \{x_1 = x_2, x_2 = 2, x_3 \leq 2\} \text{ and}$$
$$O_2 = \{-x_5 \leq -1, x_5 \leq 2, x_6 = x_5 + 1\}.$$

As shown in Section 4, $\overline{\pi}_{P \sqcup Q} = \mathcal{U} \cup \bigcup_{\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}} \mathcal{T}$. Note that we can have $\pi_O \neq \overline{\pi}_O$ even though $\overline{\pi}_P = \pi_P$ and $\overline{\pi}_Q = \pi_Q$. This is because the join operator will not have a constraint involving a variable x_i if either P or Q does not contain any constraint involving x_i. We illustrate this with an example below:

Example 5.3. Consider

$$P = \{\{x_1 = 0\}, \{x_2 - x_3 = 2, x_3 - x_4 = 3\} \text{ and}$$
$$Q = \{\{x_1 = 0\}, \{x_2 - x_4 = 5\}\} \text{ with}$$
$$\overline{\pi}_P = \pi_P = \{\{x_1\}, \{x_2, x_3, x_4\}\} \text{ and}$$
$$\overline{\pi}_Q = \pi_Q = \{\{x_1\}, \{x_2, x_4\}, \{x_3\}\}.$$

For this example, Algorithm 8 returns

$$O = \{\{x_1 = 0\}, \{x_2 - x_4 = 5\}\} \text{ and}$$
$$\overline{\pi}_O = \{\{x_1\}, \{x_2, x_3, x_4\}\}.$$

whereas $\pi_O = \{\{x_1\}, \{x_2, x_4\}, \{x_3\}\}$. Thus $\pi_O \sqsubseteq \overline{\pi}_O$.

Improving the granularity of $\overline{\pi}_O$ We lose performance since $\overline{\pi}_O$ is usually not the finest partition for O. To possibly improve the partition obtained, we perform a preprocessing step before applying Algorithm 7 in our join operator. If all variables of a block $\mathcal{X}_k \in \overline{\pi}_P \sqcup \overline{\pi}_Q$ are unconstrained in either P or Q, then the join does not require any constraints involving these variables. We replace $\mathcal{X}_k$ in $\overline{\pi}_P \sqcup \overline{\pi}_Q$ with a set of singleton blocks. This set has one block for each variable in $\mathcal{X}_k$. P'_k and Q'_k are not considered for the join.

If only a subset of variables of $\mathcal{X}_k \in \overline{\pi}_P \sqcup \overline{\pi}_Q$ are unconstrained in either P or Q, then we cannot remove the unconstrained variables from $\mathcal{X}_k$ as the join may require constraints involving the unconstrained variables. For example x_3 is unconstrained in Q in example 5.3. However, the constraints involving x_3 are required for the join or else we lose precision.

It is important to note that the key to keeping the cost of the join down is to reduce the application of the $\bowtie$ operator as it increases the number of generators exponentially, which in turn, increases the cost of the expensive conversion. The $\bowtie$ operator is applied in Algorithm 8 during refactoring and while merging factors corresponding to $\mathcal{T}$. In practice, $\overline{\pi}_P$ and $\overline{\pi}_Q$ are usually similar so the $\bowtie$ operator adds a small number of generators while refactoring.

In the program analysis setting, the join is applied at the loop head: P represents the polyhedron before executing the loop body and Q represents the polyhedron after executing the loop. The loop usually creates new constraints between a small number of variables. The factors corresponding to the blocks containing only the unmodified variables are equal, thus $|\bigcup_{\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}} \mathcal{T}|$ is small. Hence, the application of the $\bowtie$ operator while merging factors corresponding to $\mathcal{T}$ does not create an exponential number of new generators.

Comparison with static partitioning It is also worth noting that determining unmodified blocks before running the analysis requires knowledge of the partition at the start of the loop. Partitions computed based on dependence relation between program variables may not be permissible as the abstract semantics of the Polyhedra operators may relate more variables, resulting in precision loss. This is illustrated by the code in Fig. 6.

Here an analysis based on the dependence relation will yield the partition $\{\{x, z\}, \{y\}\}$ after the assignment `z:=x`, since the variables x and y are unrelated. However, the join due to the conditional if-statement creates constraint between x and y, thus $\pi_P = \{\{x, y, z\}\}$ which is computed by our analysis.

```
x:=0;
y:=0;
if(*){
        x++;
        y++;
}
z:=x;
```

Figure 6: Precision loss for static partitioning.

Complexity The performance of the join operator is dominated by the cost of the conversion. The conversion incrementally adds the generators corresponding to $\mathcal{G}_Q$ to the polyhedron defined by the constraints in $\mathcal{C}_P$. The worst case complexity of the conversion is exponential in the number of generators. For the join without partitioning, the number of generators can be $O(2^n)$ in the worst case. The join operator in Algorithm 8 applies the conversion only on the generators in $\mathcal{G}_{Q'_{\mathcal{T}}}$. Using the notation from Section 4, let $\mathcal{S} = \bigcup_{\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}} \mathcal{T}$ be the union of all blocks in $\overline{\pi}$ for which the corresponding factors are not equal, then the number of generators in $\mathcal{G}_{Q'_{\mathcal{T}}}$ can be $O(2^{|\mathcal{S}|})$ in the worst case. In practice, usually $2^{|\mathcal{S}|} \ll 2^n$ resulting in a huge reduction in operation count.

An alternative approach for computing $\mathcal{C}_O$ could be to use (5), however this is more expensive than applying the conversion. This is because the Fourier-Motzkin elimination can generate a quadratic number of new constraints for each variable that it projects out. Many of the generated constraints are redundant and should be removed to keep the algorithm efficient. Redundancy check is performed by calling a linear solver for every constraint which slows down the computation.

6. Experimental Evaluation

In this section, we evaluate the effectiveness of our decomposition approach for analyzing realistic programs. We implemented all of our algorithms in the ELINA library [1]. ELINA provides the same interface as APRON, thus, existing static analyzers using APRON can directly benefit from our approach with minimal effort.

We compare the performance of ELINA against NewPolka and PPL, both widely used state-of-the-art libraries for Polyhedra domain analysis. PPL uses the same basic algorithms as NewPolka, but uses a lazy approach for the conversion whereas NewPolka uses an eager approach. PPL is faster than NewPolka for some operators and slower for others. Like NewPolka, ELINA uses an eager approach. The experimental results of our evaluation show that the polyhedra arising during analysis can indeed be kept partitioned using our approach. We demonstrate dramatic savings in both time and memory across all benchmarks.

6.1 Experimental Setup

In ELINA we work with rational numbers encoded using 64-bit integers as in NewPolka and PPL. In the case of integer overflow, all libraries set the polyhedron to $\top$.

Platform All of our experiments were carried out on a 3.5 GHz Intel Quad Core i7-4771 Haswell CPU. The sizes of the L1, L2, and L3 caches are 256 KB, 1024 KB, and 8192 KB, respectively, and the main memory has 16 GB. Turbo boost was disabled for consistency of measurements. All libraries were compiled with gcc 5.2.1 using the flags `-O3 -m64 -march=native`.

Table 3: Speedup of Polyhedra domain analysis for ELINA over NewPolka and PPL.

Benchmark	Category	LOC	NewPolka		PPL		ELINA		Speedup ELINA vs.	
			time(s)	memory(GB)	time(s)	memory(GB)	time(s)	memory(GB)	NewPolka	PPL
firewire_firedtv	LD	14506	1367	1.7	331	0.9	0.4	0.2	3343	828
net_fddi_skfp	LD	30186	5041	11.2	6142	7.2	9.2	0.9	547	668
mtd_ubi	LD	39334	3633	7	MO	MO	4	0.9	908	>38
usb_core_main0	LD	52152	11084	2.7	4003	1.4	65	2	170	62
tty_synclinkmp	LD	19288	TO	TO	MO	MO	3.4	0.1	>4235	>1186
scsi_advansys	LD	21538	TO	TO	TO	TO	4	0.4	>3600	>3600
staging_vt6656	LD	25340	TO	TO	TO	TO	2	0.4	>7200	>7200
net_ppp	LD	15744	TO	TO	10530	0.15	924	0.3	>16	11.4
p10_l00	CF	592	841	4.2	121	0.9	11	0.8	76	11
p16_l40	CF	1783	MO	MO	MO	MO	11	3	>69	>24
p12_l57	CF	4828	MO	MO	MO	MO	14	0.8	>71	>15
p13_l53	CF	5816	MO	MO	MO	MO	54	2.7	>50	>26
p19_l59	CF	9794	MO	MO	MO	MO	70	1.7	>15	>4
ddv_all	HM	6532	710	1.4	85	0.5	0.05	0.1	12772	1700

Analyzer We use the *crab-llvm* analyzer which is part of the SeaHorn [9] verification framework. The analyzer is written in C++ and analyzes LLVM bitcode for C programs. It generates polyhedra invariants which are then checked for satisfiability with an SMT-solver. The analysis is intra-procedural and the time for analyzing different functions in the analyzed program varies.

6.2 Experimental Results

We measured the time and memory consumed for the Polyhedra analysis by NewPolka, PPL, and ELINA on more than 1500 benchmarks. We used a time limit of 4 hours and a memory limit of 12 GB for our experiments.

Benchmarks We tested the analyzer on the benchmarks of the popular software verification competition [3]. The competition provides benchmarks in different categories. We chose three categories which are suited for the analysis with a numerical domain: (a) Linux Device Drivers (LD), (b) Control Flow (CF), and (c) Heap Manipulation (HM). Each of these categories contains hundreds of benchmarks and invariants that cannot be expressed using weaker domains such as Octagon, Zone, or others.

Table 3 shows the time (in seconds) and the memory (in GB) consumed for Polyhedra analysis with NewPolka, PPL, and ELINA on 14 large benchmarks. These benchmarks were selected based on the following criteria:

- The analysis ran for > 10 minutes with NewPolka.
- There was no integer overflow during the analysis for the most time consuming function in the analyzed program.

During analysis, our algorithms obtain mathematically/semantically the exact same polyhedra as NewPolka and PPL, just represented differently (decomposed). In the actual implementation, since our representation contains different numbers, it is possible that ELINA produces an integer overflow before NewPolka or PPL. However, on the benchmarks shown in Table 3, NewPolka overflowed 296 times whereas ELINA overflowed 13 times. We also never overflowed on the procedures in the benchmarks that are most expensive to analyze (neither did NewPolka and PPL).

We show the speedups for ELINA over NewPolka and PPL which range from one to at least four orders of magnitude. In the table, the entry TO means that the analysis did not finish within 4 hours. Similarly, the entry MO means that the analysis exceeded the memory limit. In the case of a memory overflow or a time out, we provide a lower bound on the speedup, which is very conservative.

Table 3 also shows the number of lines of code for each benchmark. The largest benchmark is usb_core_main0 with 52K lines of code. ELINA analyzes this benchmark in 65 seconds whereas NewPolka takes > 3 hours and PPL requires > 1 hour. PPL performs better than NewPolka on 5 benchmarks whereas NewPolka has better performance than PPL on 2 benchmarks. Half of the benchmarks in the Linux Device Drivers category do not finish within the time and memory limit with NewPolka and PPL. net_ppp takes the longest to finish with ELINA (about 15 minutes).

All benchmarks in the Control Flow category run out of memory with both NewPolka and PPL except for p10_l00 which is also the smallest. This is because all benchmarks in this category contain a large number of join points which creates exponential number of generators for both libraries. With our approach, we are able to analyze all benchmarks in this category in ≤ 3 GB. There are > 500 benchmarks in this category not shown in Table 3 that run out of memory with both libraries whereas ELINA is able to analyze them.

There is only one large benchmark in the Heap Manipulation category. For it we get a 12722x speedup and also save 14x in memory over NewPolka. The gain over PPL is 1700x in time and 5x in memory.

We gathered statistics on the number of variables ($|\mathcal{X}|$), the size of the largest block ($|\mathcal{S}|$) in the respective partition, and its number of blocks (nb) after each join for all benchmarks. Table 4 shows *max* and *average* of these quantities. It can be seen that the number of variables in $\mathcal{S}$ is significantly smaller than in $\mathcal{X}$ resulting in complexity gains. The last column shows the fraction of the times the partition is trivial (equal to $\{\mathcal{X}\}$). It is very low and happens only when the number of variables is very small.

The bottleneck for the analysis is the conversion applied on $\mathcal{G}_{P \sqcup Q}$ during the join operator. ELINA applies conversion on $\mathcal{G}_{P'_{\mathcal{T}} \sqcup Q'_{\mathcal{T}}}$ which contains variables from the set $\mathcal{S} = \bigcup_{\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}} \mathcal{T}$ whereas NewPolka and PPL apply conversion for all variables in the set $\mathcal{X}$. The first part of Fig. 7 plots the number of variables in $\mathcal{S}$ and in $\mathcal{X}$ for all joins during the analysis of the usb_core_main0 benchmark. $|\mathcal{X}|$ varies for all joins at different program points. It can be seen that the number of variables in $\mathcal{S}$ is close to the number of variables in $\mathcal{X}$ till join number 5000. Although the number of variables is large in this region, it is not the bottleneck for NewPolka and PPL as the number of generators is linear in the number of variables. We get a speedup of 4x mainly due to our conversion operator which leverages sparsity. The most expensive region of the analysis for both NewPolka and PPL is after join number 5000 where the number of generators grow exponentially. In this

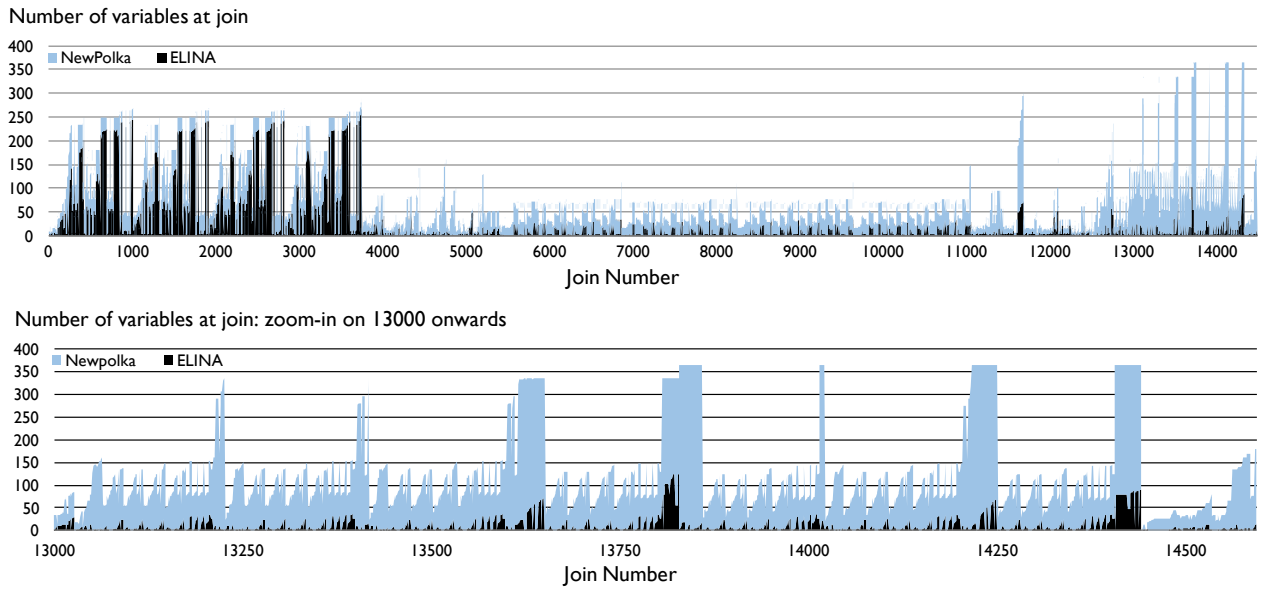

Figure 7: The join operator during the analysis of the `usb_core_main0` benchmark. The x-axis shows the join number and the y-axis shows the number of variables in $\mathcal{S} = \bigcup_{\mathcal{T} \in \overline{\pi} \setminus \mathcal{U}} \mathcal{T}$ (subset of variables affected by the join) and in $\mathcal{X}$. The first figure shows these values for all joins whereas the second figure shows it for one of the expensive regions of the analysis.

Table 4: Partition statistics for Polyhedra analysis with ELINA.

Benchmark	$\lvert \mathcal{X} \rvert$		$\lvert \mathcal{S} \rvert$		nb		trivial/total
	max	avg	max	avg	max	avg	
`firewire_firedtv`	159	80	24	5	31	6	10/577
`net_fddi_skfp`	589	111	89	24	89	15	76/5163
`mtd_ubi`	528	60	111	10	57	12	27/2518
`usb_core_main0`	365	72	267	29	61	15	80/14594
`tty_synclinkmp`	332	47	48	8	34	10	23/3862
`scsi_advansys`	282	67	117	11	82	19	11/2315
`staging_vt6656`	675	53	204	10	62	6	35/1330
`net_ppp`	218	59	112	33	19	5	1/2350
`p10_l00`	303	184	234	59	38	29	0/601
`p16_l40`	188	125	86	39	53	38	4/186
`p12_l57`	921	371	461	110	68	28	4/914
`p13_l53`	1631	458	617	149	78	28	5/1325
`p19_l59`	1272	476	867	250	65	21	9/1754
`ddv_all`	45	22	7	2	14	8	5/124

region, $\mathcal{S}$ contains 9 variables on average whereas $\mathcal{X}$ contains 54. The second part of Fig. 7 zooms in one of these expensive regions. Since the cost of conversion depends exponentially on the number of generators which in turn depends on the number of variables, we get a large speedup.

We also measured the effect of optimizations not related to partitioning on the overall speedup. The maximum difference was on the `net_ppp` benchmark which was 2.4x slower without the optimizations.

Remaining benchmarks Above we presented the results for 14 large benchmarks. The remaining benchmarks either finish or run out of memory in $<$ 10 minutes with NewPolka or the analysis produces an integer overflow in the most time consuming function. The bound on the speedup for these ranges from 2x to 76x.

7. Related Work

We next discuss the work most closely related to ours.

The concept of polyhedra partitioning has been explored before in [10]. Here, the partitions are based upon the decomposition of the matrix encoding the constraint representation of polyhedra. In this approach, the partitions are not maintained, instead the input polyhedra are partitioned for every operator, which carries significant overhead. This matrix based decomposition cannot be applied for the matrix encoding the generator set. Furthermore, the partitions are coarser at the join points as the authors do not detect equal factors which degrades performance. For example, using the partitions computed by this approach in ELINA, it takes $>$ 1 hour to analyze the `usb_core_main0` benchmark.

The authors of [23] observe that the polyhedra arising during analysis of their benchmarks show sparsity, i.e., a given variable occurs in only a few constraints. The authors work only with the constraint representation and exploit sparsity to speedup the join. In case the output becomes too large, the join is approximated. We implemented this approach in ELINA but without the approximation step so that we do not lose precision. For our benchmarks, we found that the performance of this approach degrades quickly due to frequent calls to the linear solver for redundancy removal.

Another work [20] decomposes polyhedra P and Q before applying the join into two factors $P = \{P_1, P_2\}$ and $Q = \{Q_1, Q_2\}$, such that $P_1 = Q_1$ and $P_2 \neq Q_2$. Thus, the conversion is only required for $\mathcal{G}_{P_2 \sqcup Q_2}$. This is similar to Theorem 3.6. However, the authors rely on syntactic equality between the constraints for identifying the factors, in contrast, Theorem 3.6 relies on the semantic equality. Further, their partition is coarser as it has only two blocks which increases the number of generators.

The authors of [8] observe that the analysis with the Zones domain on their benchmarks is usually sparse. They exploit sparsity by using graph based algorithms for the Zones domain operators. The benchmarks used by them are similar to those in this paper. However, the invariants produced by the Polyhedra domain are more precise.

The concept of maintaining decomposition to improve efficiency has been applied to the Octagon domain [4, 11, 25, 29]. However, the approaches in [4, 11, 29] can lose precision. Our prior work [25] is closest in spirit as it also maintains partitions dynamically. However, this maintenance is less challenging for the Octagon domain. Moreover, partitions computed at join points are coarser since [25] does not detect equal factors. Theorem 3.6 can thus be used to further improve that work. In contrast to the Octagon domain, none of the Polyhedra domain operators are compute bound, thus the analysis does not gain from vectorization.

8. Conclusion

We presented a theoretical framework, and its implementation, for speeding up Polyhedra domain analysis by orders of magnitude without losing precision. The key idea was to decompose the analysis and its operators to work on sets of smaller polyhedra thus reducing asymptotic time and space complexity. This was possible because in real-world programs the variable set partitions into independent groups, a form of locality. The challenge in maintaining these partitions was in their continuous change as the groups shrink or expand during analysis.

We provided a complete end-to-end implementation of the Polyhedra domain analysis within ELINA [1] which is compatible with APRON to enable easy integration into existing analyzers. Benchmarking against NewPolka and PPL on real-world programs including Linux device drivers and heap manipulations showed orders of magnitude speedup or successful completion where the others time-out or exceed memory.

Our theoretical framework for decomposition is generic and should be extendable to any numerical domain that maintains linear constraints between program variables, and even beyond program analysis to constraint synthesis, loop optimization frameworks, and others that use NewPolka or PPL for polyhedral computations.

Finally, we believe this paper is a significant step forward in making Polyhedra domain analysis practical for real-world use.

Acknowledgement

We would like to thank the anonymous reviewers for their constructive feedback. This research was supported by the Swiss National Science Foundation (SNF) grant number 163117.

References

[1] ELINA: ETH Library for Numerical Analysis. http://elina.ethz.ch.

[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. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[3] D. Beyer. Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 887–904, 2016.

[4] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 196–207, 2003.

[5] N. Chernikova. Algorithm for discovering the set of all the solutions of a linear programming problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282 – 293, 1968.

[6] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 84–96, 1978.

[7] R. Cousot, R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. *Science of Computer Programming*, 58(1):28 – 56, 2005.

[8] G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. *Exploiting Sparsity in Difference-Bound Matrices*, pages 189–211. 2016.

[9] A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The SeaHorn verification framework. In *Proc. Computer Aided Verification (CAV)*, pages 343–361, 2015.

[10] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design (FMSD)*, 29(1):79–95, 2006.

[11] K. Heo, H. Oh, and H. Yang. Learning a variable-clustering strategy for Octagon from labeled data generated by a static analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 237–256, 2016.

[12] J. L. Imbert. Fourier's elimination: Which to choose? *Principles and Practice of Constraint Programming*, pages 117–129, 1993.

[13] B. Jeannet and A. Miné. APRON: A library of numerical abstract domains for static analysis. In *Proc. Computer Aided Verification (CAV)*, volume 5643, pages 661–667, 2009.

[14] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 5403, pages 229–244, 2009.

[15] H. Le Verge. A note on Chernikova's algorithm. Technical Report 635, IRISA, 1992.

[16] F. Logozzo and M. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *Proc. Symposium on Applied Computing*, pages 184–188, 2008.

[17] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proc. Programs As Data Objects (PADO)*, pages 155–172, 2001.

[18] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. European Symposium on Programming (ESOP)*, pages 3–17, 2004.

[19] A. Miné. The octagon abstract domain. *Higher Order and Symbolic Computation*, 19(1):31–100, 2006.

[20] A. Miné, E. Rodriguez-Carbonell, and A. Simon. Speeding up polyhedral analysis by identifying common constraints. *Electronic Notes in Theoretical Computer Science*, 267(1):127 – 138, 2010.

[21] T. S. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The double description method. In *Proc. Contributions to the theory of games, vol. 2*, pages 51–73. 1953.

[22] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 105–118, 1999.

[23] A. Simon and A. King. Exploiting sparsity in polyhedral analysis. In *Proc. Static Analysis Symposium (SAS)*, pages 336–351, 2005.

[24] A. Simon, A. Venet, G. Amato, F. Scozzari, and E. Zaffanella. Efficient constraint/generator removal from double description of polyhedra. *Electronic Notes in Theoretical Computer Science*, 307:3 – 15, 2014.

[25] G. Singh, M. Püschel, and M. Vechev. Making numerical program analysis fast. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 303–313, 2015.

[26] A. Toubhans, B.-Y. E. Chang, and X. Rival. Reduced product combination of abstract domains for shapes. In *Proc. Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 375–395, 2013.

[27] C. Urban and A. Miné. An abstract domain to infer ordinal-valued ranking functions. In *Proc. European Symposium on Programming (ESOP)*, pages 412–431, 2014.

[28] C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. In *Proc. Static Analysis Symposium (SAS)*, pages 302–318, 2014.

[29] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 231–242, 2004.

[30] A. J. Venet. The Gauge domain: Scalable analysis of linear inequality invariants. In *Proc. Computer Aided Verification (CAV)*, pages 139–154, 2012.