Learning Fast and Precise Numerical Analysis

Jingxuan He
Department of Computer Science
ETH Zurich, Switzerland
jingxuan.he@inf.ethz.ch

Gagandeep Singh
Department of Computer Science
ETH Zurich, Switzerland
gsingh@inf.ethz.ch

Markus Püschel
Department of Computer Science
ETH Zurich, Switzerland
pueschel@inf.ethz.ch

Martin Vechev
Department of Computer Science
ETH Zurich, Switzerland
martin.vechev@inf.ethz.ch

Abstract

Numerical abstract domains are a key component of modern static analyzers. Despite recent advances, precise analysis with highly expressive domains remains too costly for many real-world programs. To address this challenge, we introduce a new data-driven method, called LAIT, that produces a faster and more scalable numerical analysis without significant loss of precision. Our approach is based on the key insight that sequences of abstract elements produced by the analyzer contain redundancy which can be exploited to increase performance without compromising precision significantly. Concretely, we present an iterative learning algorithm that learns a neural policy that identifies and removes redundant constraints at various points in the sequence. We believe that our method is generic and can be applied to various numerical domains.

We instantiate LAIT for the widely used Polyhedra and Octagon domains. Our evaluation of LAIT on a range of real-world applications with both domains shows that while the approach is designed to be generic, it is orders of magnitude faster on the most costly benchmarks than a state-of-the-art numerical library while maintaining close-to-original analysis precision. Further, LAIT outperforms hand-crafted heuristics and a domain-specific learning approach in terms of both precision and speed.

CCS Concepts: • **Theory of computation** → **Program analysis**; **Program verification**; **Abstraction**; • **Computing methodologies** → **Neural networks**.

Keywords: Abstract interpretation, Numerical domains, Machine learning, Performance optimization

ACM Reference Format:
Jingxuan He, Gagandeep Singh, Markus Püschel, and Martin Vechev. 2020. Learning Fast and Precise Numerical Analysis. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20), June 15–20, 2020, London, UK*. ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3385412.3386016

1 Introduction

In modern static analysis, numerical abstract domains are typically used to *automatically* infer numerical invariants for proving critical safety properties of various applications including standard imperative programs [7, 22, 46], untrusted kernels [19], and machine learning classifiers [18]. Developing a practically useful abstract domain is very challenging as one needs to carefully balance a fundamental tradeoff between domain *expressivity* and *complexity*. For example, Polyhedra [14] is the most expressive linear relational domain. However, it also comes with a worst-case exponential asymptotic time and space complexity. Over the years, various works introduced relational domains with weaker expressivity to boost analysis speed. Examples include Octahedron [11], TVPI [41], Octagon [34], Zones [33], and Zonotope [20].

Recently, there has been an increased interest in techniques that speed up existing numerical domains without losing precision (i.e., the strength of inferred invariants) [9, 10, 16, 28, 32, 45, 48]. Among these, online decomposition [13, 45] provides a general solution for multiple domains. However, there are still cases where the analysis based on online decomposition can be expensive. Orthogonally, another line of research based on machine learning has investigated creative heuristic methods to selectively lose some analysis precision for gaining performance [25, 26, 36, 42]. While promising, many of these techniques tend to be heavily specialized to the particular domains they consider. A key challenge then is developing learning-based methods which are both generic (applicable to different numerical domains), yet can provide significant benefits in terms of performance while incurring minimal precision loss.

Lait: generic learning-based approximations. In this work, we present a new generic learning-based method, called Lait (Learning-based Abstract Interpretation Transformers). The basic idea is to consider a generic approximation of abstract transformers based on removing constraints from the formula, and to then learn a neural classifier that identifies constraints to remove from the output of certain abstract transformers (we consider joins). The key insight enabling Lait to achieve close-to-original analysis precision is that sequences of abstract elements produced by a numerical static analyzer can contain *substantial redundancy* in the form of constraints not needed for computing the final invariants. This means that a classifier can learn how to selectively reduce this intermediate redundancy to boost analysis speed significantly, while still producing the same final result as that of the original sequence.

We instantiate Lait for speeding up analysis with the popular and expensive Polyhedra and Octagon domains. Note that Lait is also applicable to other domains such as Zones. We evaluated the performance and precision of Lait on a large number of challenging real-world benchmarks including Linux device drivers. Our results demonstrate that Lait can indeed boost analysis speed without significant loss of precision. Moreover, although Lait employs a generic method for losing precision, it produces better results than existing solutions including hand-made heuristics and a domain-specific learning method based on a state-of-the-art numerical library Elina [1].

Main contributions. Our contributions are:

- A generic approach for removing redundancy based on approximating the join transformer that can be instantiated for existing numerical domains. (Section 5.2)
- A learning framework for generating approximate transformers, called Lait. Our framework constructs a neural policy that employs graph-based structured prediction for removing redundant constraints. (Section 5.3)
- An instantiation and a complete implementation[1] of Lait on the Polyhedra and Octagon domains based on online decomposition. The instantiation includes the definition of features and weighted dependency graphs that enable structured prediction. (Section 6)
- An evaluation showing the effectiveness of Lait for speeding up both Polyhedra and Octagon analysis on a large set of real-world programs. For Polyhedra, Lait achieves orders of magnitude speedup while maintaining a mean precision of 98% w.r.t. Elina, the state-of-the-art library for numerical domains. For Octagon (which has lower complexity), Lait yields a mean speedup of 1.3x and a mean precision of 99%. Despite its generality, Lait is also faster and more precise than a prior learning-based method and hand-crafted heuristics. (Section 7)

[1]We include the learned Lait transformers in a new version of Elina [1].

2 Overview

In this section, we provide an overview of Lait on a small illustrative example. Figure 1 (a) shows the control flow graph (CFG) of a program with variables x, y, m and n. The program assumes that $y \geq 0$ and $x = n$ initially. Then it executes a *while* loop containing an if-else branch and an assignment statement. We consider the task of computing a loop invariant for the example program using the Polyhedra domain. We demonstrate that approximate analysis with Lait produces the same invariant as the precise analysis.

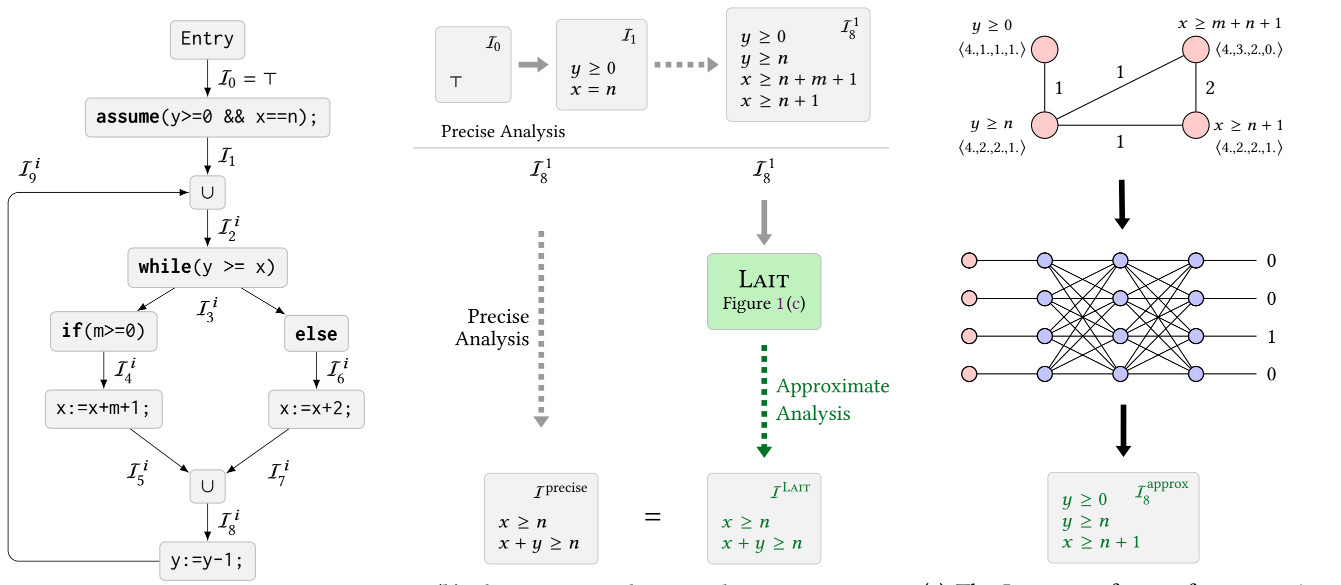
2.1 Precise Analysis

The analysis associates a polyhedron, represented by a conjunction of linear constraints, with each edge in Figure 1 (a). The polyhedron over-approximates the concrete values of the program variables before executing the statement in the successor node of the edge. Initially, the analysis sets the polyhedron $\mathcal{I}_0$ to $\top$ and the rest to $\bot$. For the polyhedra inside the loop, we use superscripts to distinguish between the polyhedra produced at different loop iterations.

First iteration. Now we execute the precise analysis step by step until the end of the first iteration of the loop:

1. It propagates $\mathcal{I}_0$ to the `assume` statement and intersects the constraints $y \geq 0$ and $x = n$ with $\mathcal{I}_0$, resulting in $\mathcal{I}_1 = \{y \geq 0, x = n\}$.
2. At the loop head, it considers the polyhedra from the entry ($\mathcal{I}_1$) and exit ($\mathcal{I}_9^0$) of the while loop. It applies the join ($\sqcup$) transformer: $\mathcal{I}_2^1 = \mathcal{I}_1 \sqcup \mathcal{I}_9^0 = \mathcal{I}_1 \sqcup \bot = \mathcal{I}_1$.
3. It enters the loop and sets $\mathcal{I}_3^1 = \{y \geq 0, x = n, y \geq x\}$ by intersecting $\mathcal{I}_2^1$ with the loop condition $y \geq x$.
4. It propagates $\mathcal{I}_3^1$ to the two branches of the `if-else`. In the `if`-branch, the analysis intersects $\mathcal{I}_3^1$ with the branch condition $m \geq 0$ and computes $\mathcal{I}_4^1 = \{y \geq 0, x = n, y \geq n, m \geq 0\}$. Next, it applies the assignment transformer for `x:=x+1+m` on $\mathcal{I}_4^1$ and obtains $\mathcal{I}_5^1 = \{y \geq 0, x = n+m+1, y \geq n, m \geq 0\}$. The analysis handles the `else` branch similarly and outputs $\mathcal{I}_7^1 = \{y \geq 0, x = n+2, y \geq n, m \leq 0\}$.
5. It exits the `if-else` branches and joins the branch outputs to compute $\mathcal{I}_8^1 = \{y \geq 0, x \geq n+m+1, x \geq n+1, y \geq n\}$.
6. It applies the assignment transformer for `y:=y-1` on $\mathcal{I}_8^1$ and gets $\mathcal{I}_9^1 = \{y \geq -1, x \geq n+m+1, x \geq n+1, y \geq n-1\}$. $\mathcal{I}_9^1$ represents an over-approximation of the concrete values of the program variables at the loop exit after executing one iteration of the loop.

Second iteration. The analysis starts from the loop head again with the updated polyhedra from the first iteration. It computes $\mathcal{I}_2^2 = \mathcal{I}_2^1 \sqcup \mathcal{I}_9^1 = \{y \geq -1, x \geq n, x+y \geq n\}$. Then the widening ($\triangledown$) transformer is applied on $\mathcal{I}_2^1$ and $\mathcal{I}_2^2$ for faster convergence towards a fixed point. This yields the polyhedron $\mathcal{I}^{\text{precise}} = \mathcal{I}_2^1 \triangledown \mathcal{I}_2^2 = \{x \geq n, x+y \geq n\}$. This polyhedron is then propagated through the while loop again



(a) The CFG of an example program.

(b) The precise analysis vs. the approximate analysis with LAIT.

(c) The LAIT transformer for constraint removal using graph neural networks.

Figure 1. Overview of our learned approximate join transformer LAIT on an illustrative example.

and the analysis computes $\mathcal{I}_2^3$ in the next loop iteration. For space reasons, we omit the computation of the third iteration.

The analysis then checks if the new polyhedron $\mathcal{I}_2^3$ is included in $\mathcal{I}^{\text{precise}}$ using the inclusion ($\sqsubseteq$) transformer. Here, the $\sqsubseteq$ transformer returns true and the analysis terminates yielding the loop invariant $\{x \geq n, x + y \geq n\}$. Note that this invariant cannot be computed with weaker domains such as Interval, Octagon, or TVPI, since the constraint $x + y \geq n$ cannot be represented in these domains.

2.2 Analysis with LAIT

Now we explain the analysis with our learned LAIT-enabled approximate join transformer on the same program.

Key insight: learning to remove redundant constraints. Our key insight is that the join transformer may create a large number of redundant constraints, which can be removed without affecting the computed loop invariant. Since the complexity of Polyhedra transformers is exponential in the number of constraints, this removal can yield considerable speedups without precision loss. The main challenge is to identify which constraints can be removed. If the removed constraints are relevant for the resulting fixed point, imprecision is introduced and will carry over for the subsequent program statements. The policy for constraint removal should be *adaptive* as the redundancy of a given constraint is determined by the dynamic state of the analysis.

In this work, we adopt a data-driven approach, implemented in LAIT, which is based on a neural classifier. The learning algorithm drives LAIT to identify redundant constraints to be removed based on the semantic, structural information of the analysis state. Next, we describe how LAIT works on the example program.

Approximate analysis with LAIT. We compare the precise analysis and our approximate analysis in Figure 1(b) (we use green for the approximate analysis). The approximate analysis computes the same polyhedra as the precise analysis until the join of $\mathcal{I}_5^1$ and $\mathcal{I}_7^1$ where it invokes LAIT.

LAIT first calls the precise join and obtains $\mathcal{I}_8^1 = \mathcal{I}_5^1 \sqcup \mathcal{I}_7^1$. Then, it removes constraints from $\mathcal{I}_8^1$. We show the steps in Figure 1(c). First, LAIT extracts a feature vector for each constraint in $\mathcal{I}_8^1$ and weighted edges between constraints, yielding a weighted dependency graph (top of Figure 1(c)). Defined in Section 6.1, the features capture polyhedron and constraint shape, and the edges indicate relationships between constraints (the number of shared variables in our case). Second, the graph is passed to a pre-trained structured prediction based classifier (in our case a graph convolutional neural network, depicted in the middle of Figure 1(c)). The classifier makes a binary decision for each constraint: whether to keep or remove it. To learn such a classifier, we propose a specialized algorithm in Section 5.3. In our example, the predictor decides to remove the constraint $x \geq m + n + 1$ and obtains the approximation $\mathcal{I}_8^{\text{approx}} = \{y \geq 0, x \geq n + 1, y \geq n\}$ of $\mathcal{I}_8^1$ (bottom of Figure 1(c)).

The approximate analysis continues with the assignment transformer for y:=y-1 and $\mathcal{I}_8^{\text{approx}}$ to produce $\mathcal{I}_9^1 = \{y \geq -1, x \geq n + 1, y \geq n\text{-}1\}$. The join transformer is then applied to compute $\mathcal{I}_2^2 = \mathcal{I}_2^1 \sqcup \mathcal{I}_9^1 = \{y \geq -1, x \geq n, x + y \geq n\}$. For this join, LAIT decides not to remove any constraints. Note that this is the same result as in the precise analysis, i.e., the removed constraint had no effect on the precision. However, the inputs of these two transformers in the approximate analysis have fewer constraints and so the analysis runs faster. In real-world benchmarks, a large number of transformers can benefit from such constraint removal.

The subsequent widening produces the same loop invariant $\mathcal{I}^{\text{LAIT}} = \mathcal{I}_2^1 \triangledown \mathcal{I}_2^2 = \{x \geq n, x + y \geq n\}$ as the precise analysis ($\mathcal{I}^{\text{LAIT}} = \mathcal{I}^{\text{precise}}$). After obtaining $\mathcal{I}^{\text{LAIT}}$, we run the precise analysis for one iteration to generate precise invariants for all program points. Any assertions dischargeable by the precise analysis can also be discharged by our approximate analysis since they produce the same invariants.

3 Background

We provide the necessary background on numerical abstract domains, online decomposition for fast numerical analysis, and the graph neural network model.

3.1 Numerical Abstract Domains

A numerical abstract domain $\mathcal{D}$ consists of abstract elements $\mathcal{I}$ defined over the set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and a set of abstract transformers. An abstract element describes the abstract state, which over-approximates the possible concrete states obtained by assigning numerical values to the variables in $\mathcal{X}$. The abstract transformers operate on abstract elements to compute over-approximations of concrete state changes due to program statements.

In this paper, we focus on sub-polyhedra domains where an abstract element is represented by a conjunction of a finite number of linear constraints $C = \{c_1, c_2, \ldots, c_m\}$ relating the variables in $\mathcal{X}$. We describe in greater detail two example domains considered in this paper: the Polyhedra [14] and the Octagon [34] domains, both of which are widely used in modern program analysis and can be expensive in practice.

Polyhedra. Each constraint c in the Polyhedra abstract domain is in the following form:

$$\sum_{i=1}^{n} a_i x_i \circ b, \quad a_i \in \mathbb{Z}, x_i \in \mathcal{X}, \circ \in \{=, \leq, \geq\}, \text{and } b \in \mathbb{Q}.$$

It is the most expressive linear relational domain but also the most expensive with a worst-case exponential time and space complexity.

Octagon. Each constraint c in the Octagon domain is restricted to contain at most two variables with the coefficients taken from the set $\mathcal{K} = \{-1, 0, 1\}$. Formally, c takes the form:

$$a_i x_i + a_j x_j \leq b, \quad a_i, a_j \in \mathcal{K}, x_i, x_j \in \mathcal{X}, \text{and } b \in \mathbb{R} \cup \{\infty\}.$$

The restriction reduces the time and space complexity of the Octagon domain to cubic and quadratic respectively.

Join. The join transformer ($\sqcup$) over-approximates the union of abstract elements originating from program branches. It is usually the most expensive transformer among about 40 transformers required for analyzing programs in modern languages. For example, the join transformer of the Polyhedra domain has worst-case exponential time complexity in terms of both the number of input constraints and the number of variables. It is also the most expensive transformer in the Octagon domain as it requires a closure operation with cubic time complexity. In practice, the join transformer may introduce a large number of (potentially redundant) constraints to the analysis causing a performance bottleneck.

3.2 Online Decomposition

Online decomposition [13, 23, 43–45] is based on the observation that the variable set $\mathcal{X}$ can often be decomposed w.r.t. an abstract element $\mathcal{I}$ into multiple small, non-related blocks on which transformers can operate independently. Formally, it constructs a partition $\overline{\pi}_{\mathcal{I}} = \{\mathcal{X}_1, \mathcal{X}_2, \ldots, \mathcal{X}_r\}$ of $\mathcal{X}$, where each $\mathcal{X}_i \subseteq \mathcal{X}$ is a block. $\overline{\pi}_{\mathcal{I}}$ requires that every constraint in $\mathcal{I}$ only relates variables in the same block. Accordingly, $\mathcal{I} = \mathcal{I}(\mathcal{X})$ can be decomposed into smaller factors $\mathcal{I}(\mathcal{X}_i)$ on which transformers operate much faster. Online decomposition has enabled orders of magnitude speedup for sub-polyhedra domains [45] including Polyhedra [23, 44] and Octagon[43].

The main challenge in online decomposition is to maintain partitions that keep the precision of a non-decomposed analysis. We note that static partitioning [7, 25, 35] is imprecise as the partition changes dynamically during analysis.

Decomposed join. Now we introduce the decomposed join transformer. Let $\mathcal{I}_P$ and $\mathcal{I}_Q$ be inputs to the join in the domain $\mathcal{D}$ with the associated partitions $\overline{\pi}_{\mathcal{I}_P}$ and $\overline{\pi}_{\mathcal{I}_Q}$ respectively. We compute the partition $\overline{\pi} = \overline{\pi}_{\mathcal{I}_P} \sqcup \overline{\pi}_{\mathcal{I}_Q}$ for the inputs where $\sqcup$ is the least upper bound operator in the partition lattice. Next, consider $\mathcal{N} = \bigcup\{\mathcal{A} \in \overline{\pi} \mid \mathcal{I}_P(\mathcal{A}) \neq \mathcal{I}_Q(\mathcal{A})\}$ as the union of all blocks $\mathcal{A} \in \overline{\pi}$ such that the corresponding factors in $\mathcal{I}_P$ and $\mathcal{I}_Q$ are not equal, and $\mathcal{U} = \{\mathcal{A} \in \overline{\pi} \mid \mathcal{I}_P(\mathcal{A}) = \mathcal{I}_Q(\mathcal{A})\}$ be the set of blocks for which the factors are equal. Then the decomposed output and an associated partition can be computed [44] as:

$$\begin{aligned} \overline{\pi}_{\mathcal{I}_P \sqcup \mathcal{I}_Q} &= \mathcal{N} \cup \mathcal{U} \\ \mathcal{I}_P \sqcup \mathcal{I}_Q &= \bigcup\{\mathcal{I}_P(\mathcal{A}) \sqcup \mathcal{I}_Q(\mathcal{A}) \mid \mathcal{A} \in \overline{\pi}_{\mathcal{I}_P \sqcup \mathcal{I}_Q}\}. \end{aligned} \tag{1}$$

The output is computed by refactoring $\mathcal{I}_P$ and $\mathcal{I}_Q$ so that the factors in both correspond to the partition $\overline{\pi}_{\mathcal{I}_P \sqcup \mathcal{I}_Q}$ and then applying the join transformer on the new factors separately.

3.3 Graph Neural Network Model

We now explain the graph neural network model used to represent the LAIT transformer discussed in Section 5.

Graph convolutional network. Consider a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is the set of nodes and $\mathcal{E}$ is the set of weighted edges. A Graph Convolutional Network (GCN) [31] transforms feature vectors associated with each node in $\mathcal{V}$

according to the graph structure. Formally, it consists of multiple graph convolutional layers. Each layer $C^i : \mathbb{R}^{|\mathcal{V}| \times d_i} \to \mathbb{R}^{|\mathcal{V}| \times d_{i+1}}$ performs the following computation:

$$C^i(\mathbf{H}) = \sigma(\mathbf{A}\mathbf{H}\mathbf{W}^i),$$

where $\mathbf{H} \in \mathbb{R}^{|\mathcal{V}| \times d_i}$ is the input feature matrix to the layer, $\mathbf{A} \in \mathbb{R}^{|\mathcal{V}| \times |\mathcal{V}|}$ is the adjacency matrix of $\mathcal{G}$, $\mathbf{W}^i \in \mathbb{R}^{d_i \times d_{i+1}}$ is the learned weight matrix, and σ is an activation function. Intuitively, at each layer, each node propagates its feature vector to its neighbors and updates it according to the received feature vectors. After multiple layers, a feature matrix associated with $\mathcal{V}$ is computed that takes the graph structure of $\mathcal{G}$ into account. Fully connected networks can be added after GCNs, enabling the network to perform classification.

4 Redundancy in Abstract Sequences

Our high-level goal is to speed up numerical analysis by exploiting the redundancy found in a *sequence* of abstract elements generated when computing the fixed point of the loop. The key observation is that it is possible to lose precision in some abstract elements in the sequence *without* affecting the final result computed by that sequence (Section 7). By losing the appropriate amount of precision, one can speed up the computation in the sequence as well as the overall analysis. Determining such abstract elements and how to lose precision is the challenging problem that we address in this paper. Concretely, we will focus on losing precision in the sequence of outputs produced by the join operations. We consider the join transformer as it is usually the most expensive transformer [43, 44] and can introduce many redundant constraints causing a performance bottleneck. To formalize our problem statement, we first introduce the abstract sequence of join outputs induced by analyzing the loop, define redundancy in these sequences, and finally state our objective for removing redundant constraints.

Abstract sequence. For a given loop, we capture the join outputs in each of its l iterations, $\mathcal{T} = [(\mathcal{I}_H^i, \mathcal{L}_i)]_{i=1}^l$. For each loop iteration i, $\mathcal{I}_H^i$ is the current abstract element at the head of the loop, and $\mathcal{L}_i = [(\mathcal{I}_P^{(i,j)}, \mathcal{I}_Q^{(i,j)}, \mathcal{I}_\sqcup^{(i,j)})]_{j=1}^s$ is a sequence of s join operations during the analysis of the loop body, with $\mathcal{I}_\sqcup^{(i,j)} = \mathcal{I}_P^{(i,j)} \sqcup \mathcal{I}_Q^{(i,j)}$. Loop analysis reaches a fixed point when $\mathcal{I}_H^{l+1} \sqsubseteq \mathcal{I}_H^l$, i.e., $\mathcal{I}_H^l$ is a loop invariant.

Redundancy in a sequence. A join transformer $\widehat{\sqcup}$ approximates $\sqcup$ when for any $\mathcal{I}_P$ and $\mathcal{I}_Q$, we have that $\mathcal{I}_P \sqcup \mathcal{I}_Q = \mathcal{I}_\sqcup \sqsubseteq \mathcal{I}_{\widehat{\sqcup}} = \mathcal{I}_P \mathbin{\widehat{\sqcup}} \mathcal{I}_Q$. For a given loop, analysis with $\widehat{\sqcup}$ generates the loop invariant $\mathcal{I}_{\widehat{H}}^l$ with iteration sequence $\widehat{\mathcal{T}} = [(\mathcal{I}_{\widehat{H}}^i, \widehat{\mathcal{L}}_i)]_{i=1}^l$, where $\widehat{\mathcal{L}}_i = [(\mathcal{I}_{\widehat{P}}^{(i,j)}, \mathcal{I}_{\widehat{Q}}^{(i,j)}, \mathcal{I}_{\widehat{\sqcup}}^{(i,j)})]_{j=1}^s$. We say that the sequence of join outputs $\mathcal{T}$ contains redundancy if $\widehat{\mathcal{T}}$ yields a loop invariant of equal or better precision, that is, $\mathcal{I}_{\widehat{H}}^l \sqsubseteq \mathcal{I}_H^l$. Note that better precision could happen as a less precise abstraction could lead to more precise analysis results [38]. An example of redundancy was shown in Section 2. We note that a simpler and more approximate $\mathcal{I}_{\widehat{\sqcup}}^{(i,j)}$ can significantly improve the performance of all abstract transformers applied downstream (but may compromise overall analysis precision). Note that, for simplicity in our formalization, we assume $\mathcal{T}$ and $\widehat{\mathcal{T}}$ have the same length l. If this is not the case, then we pad the shorter one.

Objective for redundancy removal. We assume the existence of an auxiliary function $\textsc{Diff}(\mathcal{I}, \mathcal{I}')$ which measures the degree of approximation an abstract element $\mathcal{I}'$ induces over an abstract element $\mathcal{I}$. Given a sequence $\mathcal{T}$, our objective is to find an approximate transformer $\widehat{\sqcup}$ that generates a sequence $\widehat{\mathcal{T}}$ satisfying the following objective:

$$\begin{aligned} &\operatorname*{arg\,max}_{\widehat{\sqcup}} \quad \sum_{i,j} \textsc{Diff}(\mathcal{I}_\sqcup^{(i,j)}, \mathcal{I}_{\widehat{\sqcup}}^{(i,j)}), \\ &\text{subject to:} \qquad \mathcal{I}_{\widehat{H}}^l \sqsubseteq \mathcal{I}_H^l. \end{aligned} \tag{2}$$

That is, we aim to maximize the degree of approximation induced by $\widehat{\sqcup}$ while preserving loop analysis precision.

5 Learning an Approximate Join

We now describe how we solve the optimization problem (2) to obtain an approximate join $\widehat{\sqcup}$. Our method is generic and can be applied to various numerical domains to benefit from the removal of the underlying redundancy. Before presenting the method, we first describe a hand-crafted approximate join transformer, which will serve as one of the baselines.

5.1 Hand-crafted Approximate Join

As a first attempt, we propose a reasonable heuristic for approximating $\sqcup$ based on online decomposition described in Section 3.2. The decomposed join transformer in (1) merges all blocks $\mathcal{A} \in \overline{\pi}_{\mathcal{I}_P} \sqcup \overline{\pi}_{\mathcal{I}_Q}$ for which the corresponding input factors $\mathcal{I}_P(\mathcal{A})$ and $\mathcal{I}_Q(\mathcal{A})$ are not equal and applies join on the factor corresponding to the merged block $\mathcal{N}$. This creates constraints between variables not related by any constraint in the inputs. We observed that many of these new constraints are often redundant w.r.t. the abstract sequence.

Therefore, we design a hand-crafted approximate join transformer (HC) that avoids creating such constraints by not merging blocks into $\mathcal{N}$. Formally, given join inputs $\mathcal{I}_P$ and $\mathcal{I}_Q$, our hand-crafted approximate join transformer $\text{HC}(\mathcal{I}_P, \mathcal{I}_Q)$ computes the output as follows:

$$\begin{aligned} \overline{\pi}_{\text{HC}(\mathcal{I}_P,\mathcal{I}_Q)} &= \overline{\pi}_{\mathcal{I}_P} \sqcup \overline{\pi}_{\mathcal{I}_Q}, \\ \text{HC}(\mathcal{I}_P, \mathcal{I}_Q) &= \bigcup \{\mathcal{I}_P(\mathcal{A}) \sqcup \mathcal{I}_Q(\mathcal{A}) \mid \mathcal{A} \in \overline{\pi}_{\text{HC}(\mathcal{I}_P,\mathcal{I}_Q)}\}. \end{aligned}$$

Similar to $\overline{\pi}_{\mathcal{I}_P \sqcup \mathcal{I}_Q}$ in (1), the partition $\overline{\pi}_{\text{HC}(\mathcal{I}_P,\mathcal{I}_Q)}$ is dynamically computed based on the partitions of the join inputs. The difference is that $\overline{\pi}_{\text{HC}(\mathcal{I}_P,\mathcal{I}_Q)}$ is finer, which can speed up both the current join and the downstream analysis, but with potential loss of precision. HC is applicable to all numerical domain implementations based on online decomposition.

We evaluate HC in Section 7 showing that it can accelerate analysis for many benchmarks. However, it is based on a *fixed policy* that never creates constraints between variables not related by any constraint in the inputs. This can introduce significant imprecision as some of those constraints might be necessary for computing the invariant at the fixed point. Moreover, since it does not remove existing constraints, many redundant constraints can remain to cause the analysis to get stuck and time out. These observations motivate the design of an adaptive redundancy removal policy that takes the inputs into account.

5.2 LAIT: A Learned Approximate Join Transformer

We adopt a data-driven approach to learn an approximate abstract transformer whose output is adaptive w.r.t. the input constraints. Our proposed transformer LAIT is fairly generic as it approximates $\sqcup$ by dropping constraints from the output $\mathcal{I}_\sqcup = \mathcal{I}_P \sqcup \mathcal{I}_Q$ of the precise join $\sqcup$ ($\mathcal{I}_P$ and $\mathcal{I}_Q$ are join inputs).

The core part of LAIT is a learned constraint removal policy ψ, represented by a neural classifier which takes the structure of constraints into account when making decisions. In Algorithm 1, we show the procedure LAIT for computing the approximate join output $\mathcal{I}_{\widehat{\sqcup}} = \text{LAIT}(\mathcal{I}_P, \mathcal{I}_Q, \mathcal{I}_\sqcup)$. We explain it step by step:

- Line 2: Apply the feature extraction $\text{FEAT}(\mathcal{I}_P, \mathcal{I}_Q, \mathcal{I}_\sqcup)$ which outputs a feature matrix $\mathbf{F} \in \mathbb{Z}^{|\mathcal{I}_\sqcup| \times k}$. Each row is a k-dimensional vector for each constraint in $\mathcal{I}_\sqcup$.
- Line 3: Extract a structured relationship between the constraints in $\mathcal{I}_\sqcup$ as a set $\mathcal{E}$ of edges. Each edge $e = (c, c', w) \in \mathcal{E}$ connects constraints c and c' with weight w. We represent $\mathcal{E}$ by its adjacency matrix $\mathbf{A}$, which is obtained by calling $\text{EDGE}(\mathcal{I}_\sqcup)$.
- Line 4: Invoke $\psi(\mathbf{F}, \mathbf{A})$ to obtain a boolean vector $B \in \{0,1\}^{|\mathcal{I}_\sqcup|}$, where each value B_i indicates whether the constraint c_i should be removed ($B_i = 1$) or not ($B_i = 0$).
- Line 5: Drop constraints from $\mathcal{I}_\sqcup$ based on B to produce the output abstract element $\mathcal{I}_{\widehat{\sqcup}}$.

We note that $\mathcal{I}_{\widehat{\sqcup}}$ contains only a subset of $\mathcal{I}_\sqcup$'s constraints, hence LAIT is sound. We show how to instantiate the required functions FEAT and EDGE for both Polyhedra and Octagon numerical domains in Section 6.

Benefits of constraint removal. The benefits of constraint removal by LAIT are two-fold. First, the time complexity of the abstract transformers depends on the number of constraints [43, 44]. LAIT reduces the number of constraints in the join output which speeds up the downstream analysis. We show such a case in Section 7.2. Second, constraint removal is a general method for approximation which can potentially be applied to speed up any constraint-based numerical domain. Our experimental results demonstrate that LAIT is effective in practice (Section 7).

Algorithm 1: Applying LAIT with a learned ψ.

```
1 Procedure LAIT ($\mathcal{I}_P$, $\mathcal{I}_Q$, $\mathcal{I}_\sqcup$)
      Input  : $\mathcal{I}_\sqcup = \mathcal{I}_P \sqcup \mathcal{I}_Q$.
      Output : $\mathcal{I}_{\widehat{\sqcup}}$, the approximate join output.
2     F = FEAT($\mathcal{I}_P$, $\mathcal{I}_Q$, $\mathcal{I}_\sqcup$)          // feature extraction
3     A = EDGE($\mathcal{I}_\sqcup$)                   // edge extraction
4     $B = \psi(\mathbf{F}, \mathbf{A})$                 // invoke removal policy
5     $\mathcal{I}_{\widehat{\sqcup}}$ = DROPCONS($\mathcal{I}_\sqcup$, $B$)         // drop constraints
6     return $\mathcal{I}_{\widehat{\sqcup}}$
```

5.3 Learning a Constraint Removal Policy

The main challenge which LAIT must address is to decide on the set of constraints to remove so that the overall analysis precision is maintained, while the analysis speed is improved. The search space for an optimal removal policy is large as the abstract sequences can contain many joins and each join output in the sequence contains many candidate constraints for removal. Next, we describe an iterative training algorithm that aims to learn a policy ψ addressing this challenge.

Generate a dataset of precise abstract sequences. Given a set of programs used for training, we first generate a dataset of abstract sequences $\mathcal{S} = \{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_h\}$ containing potential redundancy. Each $\mathcal{T}_k \in \mathcal{S}$ in this dataset corresponds to one abstract sequence of a particular loop k in a program. The sequences are built by analyzing each program with a precise analysis. We will use $\mathcal{S}$ as the ground truth for measuring the precision of the (currently computed) approximate analysis during training.

Produce a classification dataset for training ψ. To learn the classification policy ψ, we first produce a classification dataset D. This is accomplished via procedure GENCLASSIFDATA, shown in Algorithm 2. The procedure shows how to process a single loop abstract sequence $\mathcal{T} \in \mathcal{S}$ in order to produce D. It takes as input the current policy ψ (initially set to not drop constraints), a precise training abstract sequence $\mathcal{T} \in \mathcal{S}$, and an exploration parameter ϵ (not shown in Algorithm 2 but assumed globally). We use $\mathcal{I}_H^k$ to denote the abstract element at the loop head at iteration k in the precise training abstract sequence $\mathcal{T}$ as well as $\widehat{\mathcal{I}}_H^k$ to denote the abstract element at the loop head of the approximate analysis at iteration k.

The procedure first sets the starting point of the approximate analysis to be the same as the precise analysis ($\widehat{\mathcal{I}}_H^1 = \mathcal{I}_H^1$ at Line 2). It then performs approximate analysis of the given loop (at Line 3) until a fixed point is reached (Line 11). For each of the s joins inside the loop, it invokes an approximate abstract transformer (Line 5). To simplify presentation we assume the same set of s joins taking place in each iteration, in both the precise and approximate analysis.

Algorithm 2: Producing classification dataset.

```
1  Procedure GenClassifData($\psi$, $\mathcal{T}$)
       Input  : $\psi$, a classifier for constraints removal.
                $\mathcal{T}$, a precise abstract sequence.
       Output : $D$, a dataset for supervised learning.
2      $i = 0$;  $\widehat{\mathcal{I}}_H^1 = \mathcal{I}_H^1$
3      repeat                                        // approximate analysis
4          $i \leftarrow i + 1$;  $D_i \leftarrow \varnothing$
5          for $j \leftarrow 1$ to $s$ do           // $s$ joins per loop iteration
6              perform analysis, obtain join inputs $\mathcal{I}_{\widehat{P}}$, $\mathcal{I}_{\widehat{Q}}$
7              $\mathcal{I}_{\sqcup} = \mathcal{I}_{\widehat{P}} \sqcup \mathcal{I}_{\widehat{Q}}$
8              $(\mathcal{I}_{\widehat{\sqcup}}, \mathbf{F}, \mathbf{A}, B) = \text{LaitTrain}(\mathcal{I}_{\widehat{P}}, \mathcal{I}_{\widehat{Q}}, \mathcal{I}_{\sqcup})$  // with $\psi$
9              add $\{(\mathbf{F}_B, \mathbf{A}_B)\}$ to $D_i$
10         continue analysis and compute $\widehat{\mathcal{I}}_H^{i+1}$
11     until $\widehat{\mathcal{I}}_H^{i+1} \sqsubseteq \widehat{\mathcal{I}}_H^i$
12     $D \leftarrow \varnothing$;  $pos = \arg\max_k \widehat{\mathcal{I}}_H^k \sqsubseteq \mathcal{I}_H^k$
13     for $i \leftarrow 1$ to $pos - 1$ do          // positive labeling
14         for $(\mathbf{F}, \mathbf{A})$ in $D_i$ do
15             add $\{((\mathbf{F}, \mathbf{A}), \mathbf{1})\}$ to $D$
16     for $(\mathbf{F}, \mathbf{A})$ in $D_{pos}$ do  // negative labeling
17         add $\{((\mathbf{F}, \mathbf{A}), \mathbf{0})\}$ to $D$
18     return $D$
```

Importantly, the procedure invokes LaitTrain for approximating join outputs. LaitTrain is a variant of Algorithm 1, specifically used for training. It employs an exploration-exploitation mechanism to obtain the boolean vector B (different from Line 4 of Algorithm 1):

$$B = \begin{cases} \text{Explore}(\mathcal{I}_{\sqcup}) & \text{with probability } \epsilon, \\ \psi(\mathbf{F}, \mathbf{A}) & \text{otherwise.} \end{cases}$$

This means that with probability ϵ it calls an exploration policy Explore that randomly samples a portion γ of constraints to be removed from $\mathcal{I}_{\sqcup}$ (we set ϵ and γ both to 0.1 in our experiments); otherwise, it invokes ψ to make the decision. LaitTrain returns a tuple: the approximate join output $\mathcal{I}_{\widehat{\sqcup}}$, and intermediate variables $\mathbf{F}$, $\mathbf{A}$ and B (Line 8). Next, features $\mathbf{F}_B$ and edges $\mathbf{A}_B$ of the removed constraints (determined by B) are added to the set D_i (Line 9). After finishing all joins, the analysis proceeds to compute the loop head element $\widehat{\mathcal{I}}_H^{i+1}$ (Line 10), checks for convergence (Line 11), and enters the next loop iteration if it does not converge.

When the approximate analysis completes, we label the data points in D_i. For this we first calculate pos, the maximal loop iteration number where the precision is preserved (Line 12). We then assign positive labels to all data points

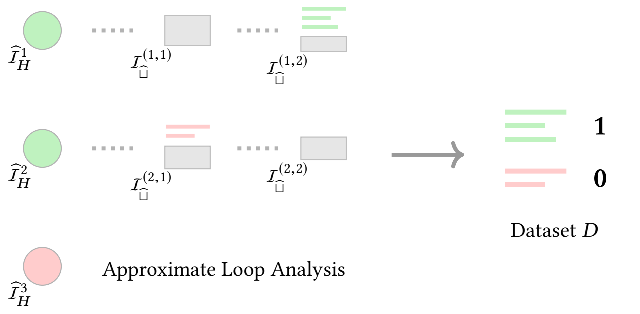

$\widehat{\mathcal{I}}_H^1$
$\mathcal{I}_{\widehat{\sqcup}}^{(1,1)}$
$\mathcal{I}_{\widehat{\sqcup}}^{(1,2)}$
$\widehat{\mathcal{I}}_H^2$
$\mathcal{I}_{\widehat{\sqcup}}^{(2,1)}$
$\mathcal{I}_{\widehat{\sqcup}}^{(2,2)}$
1
0
Dataset D
$\widehat{\mathcal{I}}_H^3$
Approximate Loop Analysis

Figure 2. Visualizing an example run of GenClassifData.

Algorithm 3: Iterative learning of a policy ψ.

```
1  Procedure Learn($\mathcal{S}$, $N$)
       Input  : $\mathcal{S} = \{\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_h\}$, the generated training
                abstract sequences.
                $N$, the number of training iterations.
       Output : $\psi$: a classifier for removing constraints.
2      $D \leftarrow \varnothing$;  Initialize($\psi$)
3      for $j \leftarrow 1$ to $N$ do
4          for $\mathcal{T}$ in $\mathcal{S}$ do
5              $D \leftarrow D \cup$ GenClassifData($\psi$, $\mathcal{T}$)
6          $\psi$ = SupervisedLearning($D$)
7      return $\psi$
```

before the loop iteration *pos* (Line 15). Intuitively, we do that because dropping constraints at these iterations did not affect the precision of the abstract element at the loop head at iteration *pos*. For data points at loop iteration *pos*, we assign negative labels (0) as these resulted in precision loss (Line 17). This labeling scheme drives ψ to drop as many constraints as possible while keeping precision (a proxy for our objective (2)). The data points after iteration *pos* are ignored since we cannot determine whether they keep precision or not.

Figure 2 visualizes an example run of GenClassifData. On the left, we run the approximate analysis with two loop iterations each containing two joins. The analysis decides to drop three constraints (depicted as green lines) to obtain $\mathcal{I}_{\widehat{\sqcup}}^{(1,2)}$ and two constraints (depicted as red lines) to obtain $\mathcal{I}_{\widehat{\sqcup}}^{(2,1)}$. This produces the precise result $\widehat{\mathcal{I}}_H^2 \sqsubseteq \mathcal{I}_H^2$ and the imprecise result $\mathcal{I}_H^3 \sqsubset \widehat{\mathcal{I}}_H^3$. The imprecision propagates downstream in the analysis (not shown in Figure 2). The dataset D is depicted on the right of Figure 2. The algorithm labels the three removed constraints at iteration one with **1** because the analysis precision is preserved even after removing them ($\widehat{\mathcal{I}}_H^2 \sqsubseteq \mathcal{I}_H^2$). In contrast, it labels the two removed constraints at iteration two with **0** because removing them results in imprecision ($\mathcal{I}_H^3 \sqsubset \widehat{\mathcal{I}}_H^3$).

Table 1. Features for approximating Polyhedra join.

#	Description	Calculation
1	Number of variables in the block $\mathcal{X}^c_\sqcup$	$\lvert\mathcal{X}^c_\sqcup\rvert$
2	Number of constraints in the factor $\mathcal{X}^c_\sqcup$	$\lvert\mathcal{I}_\sqcup(\mathcal{X}^c_\sqcup)\rvert$
3	Number of generators in the factor $\mathcal{X}^c_\sqcup$	$\lvert\mathcal{G}_\sqcup(\mathcal{X}^c_\sqcup)\rvert$
4	Number of loop head variables in c	$\lvert\mathcal{X}^c \cap \mathcal{X}_H\rvert$
5	Boolean, true if $\mathcal{X}^c$ is a subset of $\mathcal{X}_H$	$\mathcal{X}^c \subseteq \mathcal{X}_H$
6	Boolean, true if c is in $\mathcal{I}_H$	$c \in \mathcal{I}_H$
7	Number of variables in constraint c	$\lvert\mathcal{X}^c\rvert$.
8	Number of large coefficients in c	$\sum_i(\lvert a_i\rvert >= 100)$
9	Sum of scores for variables in c	$\sum_i \textsc{Score}(x_i)$.
10	Boolean, true if c is in join inputs	$c \in \mathcal{I}_P \wedge c \in \mathcal{I}_Q$
11	Boolean, true if c coarsens partition	$\textsc{Coarse}(\mathcal{X}^c_\sqcup, \overline{\pi}_P, \overline{\pi}_Q)$
12	Boolean, true if c is an equality	$\circ$ is $=$

Table 2. Features for approximating Octagon join.

#	Description	Calculation
1	Number of variables in the block $\mathcal{X}^c_\sqcup$	$\lvert\mathcal{X}^c_\sqcup\rvert$
2	Number of constraints in the factor for $\mathcal{X}^c_\sqcup$	$\lvert\mathcal{I}_\sqcup(\mathcal{X}^c_\sqcup)\rvert$
3	Number of loop head variables in c	$\lvert\mathcal{X}^c \cap \mathcal{X}_H\rvert$
4	Boolean, true if c is in $\mathcal{I}_H$	$c \in \mathcal{I}_H$
5	Score of variable x_i	$\textsc{Score}(x_i)$
6	Score of variable x_j	$\textsc{Score}(x_j)$
7	Number of finite bounds for variable x_i	See text
8	Number of finite bounds for variable x_j	See text
9	Absolute value of constraint upper bound b	$\lvert b\rvert$
10	Boolean, true if upper bound b' is ∞	See text
11	Number of constraints coarsening partition	See text
12	Loop iteration number	$iter$

Final iterative learning algorithm. Given the size of the search space, a large classification dataset is necessary for learning a suitable ψ. Moreover, since the dataset and the policy are mutually dependent, they should be improved iteratively. Towards that we adopt a Dagger-style learning scheme [37] which alternates dataset generation with policy learning, illustrated in Algorithm 3.

The algorithm first initializes the dataset D to $\varnothing$ and ψ to a policy which does not remove constraints (Line 2). It then performs N training iterations. Each iteration calls the procedure GenClassifData on a precise abstract sequence $\mathcal{T} \in \mathcal{S}$ (Line 5) returning a newly constructed dataset, which is then added to D. An improved policy ψ is trained on D through a supervised learning classification algorithm (Line 6), which depends on the used classifier (in our case, the classifier is a GCN followed by a fully connected network as instantiated in Section 6.2). The procedure can repeat a number of times, gradually improving the quality of the learned policy ψ.

6 Instantiation of Lait

We now describe our instantiation of the Lait approximate join transformer for the Polyhedra and Octagon domains. Our instantiation is based on a precise analysis with online decomposition [45]. We do not approximate other transformers used in the analysis. Next, we first define our features (i.e., Feat in Section 5) and edges (i.e., Edge in Section 5) for representing the set of constraints as a graph and then discuss our instantiation of the constraint removal policy ψ.

6.1 Features and Edges for Constraints

As discussed in Section 5.2, we extract features for each constraint $c \in \mathcal{I}_\sqcup$ by calling $\textsc{Feat}(\mathcal{I}_P, \mathcal{I}_Q, \mathcal{I}_\sqcup)$ where $\mathcal{I}_\sqcup = \mathcal{I}_P \sqcup \mathcal{I}_Q$. The features distill domain-specific information critical for identifying redundant constraints. These include features that are (a) specific to the constraint, (b) dependent on both the constraint and the abstract state, and (c) dependent on both the constraint and the context information about the loop. We denote the set of variables in the constraint c by $\mathcal{X}^c$ and the block containing the variables in c in the partition of $\mathcal{I}_\sqcup$ by $\mathcal{X}^c_\sqcup$. Note that $\mathcal{X}^c \subseteq \mathcal{X}^c_\sqcup$. $\mathcal{I}_\sqcup(\mathcal{X}^c_\sqcup)$ is the subset of constraints over the variables in $\mathcal{X}^c_\sqcup$. We define this information analogously for $\mathcal{I}_P$ and $\mathcal{I}_Q$. Moreover, we record global information about the loop by tracking the current loop head element $\mathcal{I}_H$, the set of variables $\mathcal{X}_H$ which appears in at least one constraint in $\mathcal{I}_H$, and the iteration number $iter$.

Features for Polyhedra constraints. The extracted features for constraints in the Polyhedra domain are shown in Table 1. The implementation of the Polyhedra domain used in our instantiation keeps both the generator and the constraint representation [44]. The cost of the join in the factor containing the constraint c is characterized by the number of variables in the block $\mathcal{X}^c_\sqcup$ and the number of generators and constraints in the factor [44]. The first three features in Table 1 record this information.

Features 4–6 record information for predicting whether the constraint c affects the abstract element at the loop head $\mathcal{I}_H$. Feature 4 records the number of variables that are both in c and $\mathcal{I}_H$. Boolean features 5 and 6, respectively, record whether all variables in c are in $\mathcal{X}_H$ and whether $\mathcal{I}_H$ has constraint c. The constraints that link with $\mathcal{I}_H$ are likely to be non-redundant.

Features 7 and 8 record the number of variables and the number of large coefficients (with absolute value ≥ 100) in c. We use a function $\textsc{Score}(x)$ that computes the number of constraints containing the variables x in $\mathcal{I}_\sqcup$. Feature 9 then records the sum of $\textsc{Score}(x)$ for all $x \in \mathcal{X}^c$. A constraint with large values for these three features is likely to be redundant.

Boolean features 10–11 characterize the relationship of c w.r.t. the two join inputs $\mathcal{I}_P$ and $\mathcal{I}_Q$. Feature 10 checks whether c appears in both inputs. If yes, then c should likely be kept. Feature 11 checks whether c coarsens the partition, i.e., $\forall \mathcal{X}_j \in \overline{\pi}_P, \mathcal{X}^c \subsetneq \mathcal{X}_j \wedge \forall \mathcal{X}_k \in \overline{\pi}_Q, \mathcal{X}^c \subsetneq \mathcal{X}_k$, by calling the function Coarse. Such constraints are likely to be redundant. The last feature checks whether c is an equality constraint.

Features for Octagon constraints. Table 2 shows features extracted from an Octagon constraint $c := a_i x_i + a_j x_j \leq b$. The first four coincide with features 1, 2, 4 and 6 for Polyhedra in Table 1. Features 5 and 6 calculate the Score of the two variables x_i and x_j in c, respectively. A large value of these features indicates that c has more interaction with other constraints. Features 7 and 8 compute the number of finite bounds for the variables x_i and x_j, respectively. For $x_i \in [l_i, u_i]$, we assign to the 7th feature the value 2 if both l_i and u_i are finite, 1 if exactly one of them is finite, and 0 otherwise. Feature 8 for x_j is analogous. Constraints with bounded variables should likely be kept to transitively bound other variables. Feature 9 records the upper bound b of the constraint. Feature 10 tests if the bound b' in the constraint $c' := -a_i x_i - a_j x_j \leq b'$ is finite.

Feature 11 calculates the number of constraints in the octagon element that coarsens the partition (computed by the function Coarse as described earlier). If the value of this feature is large, there may be more redundancy so more constraints should be dropped. The last feature is the current loop iteration *iter* to capture loop progress.

Edges for relating constraints. The extraction function Edge($\mathcal{I}_\sqcup$) extracts weighted edges connecting the constraints in $\mathcal{I}_\sqcup$ (associated with the feature vectors). Edge inspects all pairs of constraints c and c' in $\mathcal{I}_\sqcup$, and adds an edge (c, c', w) iff $|\mathcal{X}^c \cap \mathcal{X}^{c'}| > 0$, i.e., if c and c' share variables. For Polyhedra, the edge weight $w = |\mathcal{X}^c \cap \mathcal{X}^{c'}|$ is the number of variables that c and c' share. We chose this metric since it characterizes the relationship between two constraints. For the Octagon domain where each constraint involves at most two variables, we always set $w = 1$. The obtained (weighted) edges, represented by the adjacency matrix $\mathbf{A}$, enable structured prediction with GCNs.

6.2 Instantiating Constraint Removal

We represent ψ by a 3-layer GCN followed by a 4-layer fully connected network with all hidden dimensions set to 128. The features $\mathbf{F}$ and the adjacency matrix $\mathbf{A}$ are first fed to the GCN to obtain structured feature embeddings. Then, the feature embeddings are fed to the fully connected network to produce the output boolean vector B.

Pruning the search space. As described earlier, the search space for finding the right constraints to drop is large since both the number of joins in the abstract sequences and the number of constraints in the join outputs can be large. We incorporate domain-specific knowledge to prune the search space. For Polyhedra, we only consider removing a constraint c when $|\mathcal{X}^c| > 2$ and $|\mathcal{I}_\sqcup(\mathcal{X}^c_\sqcup)| > 20$. Such constraints usually cause a performance bottleneck. For Octagon, we consider a constraint c as a candidate for removal when it coarsens the partition (i.e., Coarse($\mathcal{X}^c_\sqcup, \overline{\pi}_P, \overline{\pi}_Q$) returns 1). Removing such constraints helps in refining the partition which, in turn, boosts speed.

7 Experimental Evaluation

We now present our extensive evaluation of Lait, focusing on the following two questions:

- **Effectiveness**: How effective is Lait for redundancy removal in terms of both speed and precision?
- **Interpretability**: How to interpret Lait? What types of constraints are most likely to be removed by Lait?

We discuss our results for the Polyhedra analysis in Sections 7.1 and 7.2 and for the Octagon analysis in Section 7.3.

Analyzer and platforms. We used the crab-llvm analyzer, which is part of the Seahorn verification framework [22]. The analysis is intra-procedural with function inlining. By default, widening is applied after the second loop iteration. The analysis runs to obtain abstract sequences for training and testing were performed on a machine with four 2.13 GHz Intel Xeon E7-4830 CPUs and 256 GB RAM. All analysis libraries were compiled with gcc 5.4.0 using the flags -O3 -m64 -march=native.

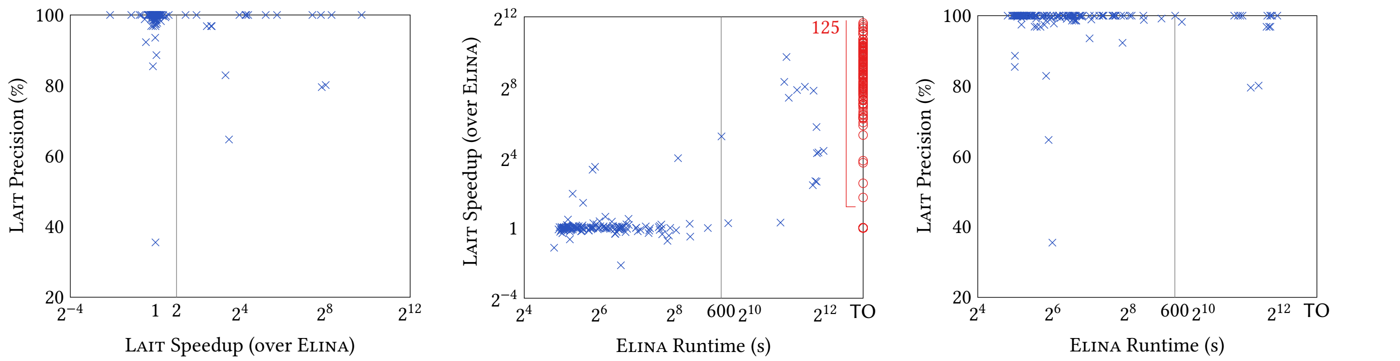
The neural networks were implemented in PyTorch [2] and trained with the data from analysis runs (i.e., SupervisedLearning in Algorithm 3) on a machine with one 3.30 GHz Intel Core i9-7900X CPU, three GTX 1080 GPUs, and 128 GB RAM. We used only one GPU for training.

Baselines. We consider the following three baselines:

- Elina: a state-of-the-art library supporting multiple domains including Polyhedra and Octagon [45]. It leverages online decomposition and we use its invariants as ground-truth for measuring precision. We use the version released when we write the paper.
- HC: the hand-crafted fixed policy discussed in Section 5.1. It applies to both Polyhedra and Octagon.
- Poly-Rl[2]: an approximate Polyhedra analysis based on reinforcement learning [42]. It learns a policy to select among a set of hand-crafted approximate join transformers during analysis based on the join inputs.

As discussed in Section 5.3, our learning algorithm requires an analysis to generate abstract loop sequences for training. In our work, we use the analysis with Elina, i.e., our goal is to remove redundancy from abstract elements obtained via the analysis with Elina. However, our approach can be used with other numerical analysis libraries, e.g., APRON [27] or PPL [3]. We use Elina because it is significantly faster on the considered domains [43–45].

[2]We used the transformers and the learned models from the paper artifact in https://www.sri.inf.ethz.ch/cav2018-paper259.



(a) LAIT: precision vs. speedup **(b)** LAIT speedup vs. ELINA runtime **(c)** LAIT precision vs. ELINA runtime

Figure 3. Precision and speedup of LAIT with the Polyhedra analysis.

Benchmarks. The benchmarks were taken from the popular software verification competition sv-comp [6]. It contains multiple categories for different analyses, e.g., pointer and numerical. We mainly focus on the Linux device drivers (LDD) categories known for being suitable and challenging for numerical analysis [16, 42, 45]. We report training and testing programs for Polyhedra and Octagon in Sections 7.1 and 7.3, respectively. For testing, we set a time limit of 2h and a memory limit of 50 GB.

Metric. We measure analysis speed by the runtime in seconds and speedup over ELINA. We provide a lower bound on the speedup when ELINA timed out (i.e., we divide the time limit 2h by the runtime of the analysis).

The precision is measured as the fraction of program points at which the invariants generated by the approximate analysis are semantically the same or stronger than the ones generated by ELINA. If the approximate analysis does not finish and thus does not generate invariants at certain program points, we set the invariants at those program points to $\top$. This precision metric is used in [42] and is more challenging than the number of verified assertions [25, 36] as the latter is sensitive to the choice of assertions. If the assertions are already provable with the weaker Interval analysis then one may get the same precision with this metric by simply switching to the faster Interval transformers. Further, it has been shown that program analysis is a harder problem than verification [12]. In contrast, our metric measures whether the approximate analysis can prove at least all the assertions that the precise analysis can.

We also report mean and median value of precision and speed. We refer mean to arithmetic mean in the paper.

7.1 Effectiveness of LAIT on Polyhedra Analysis

We evaluate the precision and speed of LAIT. The results are shown in Figure 3 and Figure 4, addressing two questions:

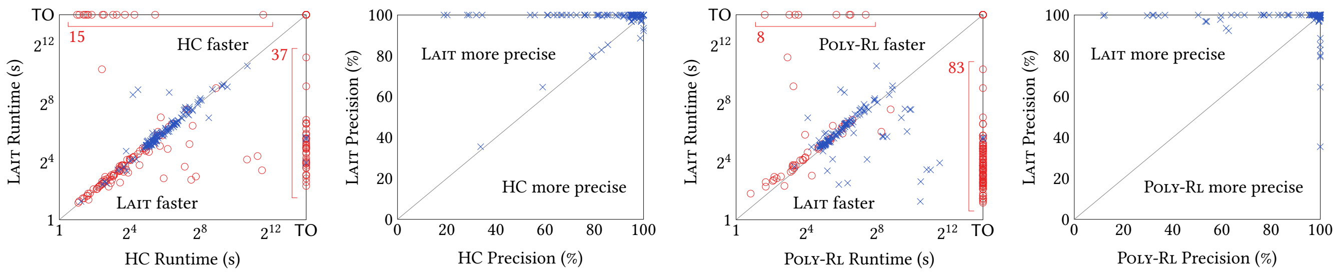
- Can LAIT effectively remove redundancy? i.e., can LAIT achieve speedup over ELINA while preserving precision?
- How does LAIT compare to HC and POLY-RL [42] in terms of both precision and speed?

Training and testing. We chose 30 LDD benchmarks for training. The training procedure ran for about 20 iterations (Algorithm 3), which took about 8 hours. For testing, we selected from the LDD categories all remaining 354 benchmarks for which the analysis with ELINA took > 30s. Of these ELINA finished 147 (Test set 1) and timed out at 2h on 207 (Test set 2). For Test set 1 (blue x marker), we report the speed and the precision of LAIT, HC, and POLY-RL. For Test set 2 (red o marker), we report only speed since precision cannot be measured as ELINA did not finish.

LAIT vs. ELINA. Figure 3(a) plots LAIT's precision vs. its speedup over ELINA on Test set 1. LAIT achieves > 2x speedup on 19 benchmarks and 100% precision on 12 of these with a mean precision of 95%. The remaining 128 benchmarks where LAIT achieved ≤ 2x speedup usually have little redundancy in the abstract sequences and LAIT maintains a median precision of 100% (mean 99%).

In Figure 3(b) we show the speedup of LAIT vs. the runtime of ELINA on Test set 1 and the 125/207 benchmarks from Test set 2 for which ELINA timed out but LAIT finished. The red circles thus show lower bounds for the speedup. We see that LAIT gains most when ELINA is slow. For example, for the 16 benchmarks where ELINA finished in > 10m (in Test set 1), LAIT achieved a median of 29x speedup (mean 138x). For Test set 2, the median speedup is at least 554x (mean at least 722x). The overall median speedup on all benchmarks (Test set 1 and Test set 2) is at least 20x (mean at least 118x). The large speedups for time consuming ELINA analyses in Test set 1 tend to incur little loss in precision, as shown in Figure 3(c).

Based on these results, we conclude that LAIT is effective in redundancy removal: it achieves significant speedups and preserves ≈ 100% precision on benchmarks where the analysis with ELINA becomes time-consuming or even unpractical due to the complexity in the abstract sequences.



(a) Lait vs. HC: runtime **(b)** Lait vs. HC: precision **(c)** Lait vs. Poly-Rl: runtime **(d)** Lait vs. Poly-Rl: precision

Figure 4. Comparison of Lait with HC and Poly-Rl on the Polyhedra analysis.

Table 3. Speed and precision for different Polyhedra analyses on 10 benchmarks.

Benchmark	Elina Time	HC		Poly-Rl [42]		Lait	
		Speed	Prec.	Speed	Prec.	Speed	Prec.
qlogic_qlge	3474	49x	99	4.2x	83	53x	100
peak_usb	1919	325x	81	1.3x	100	315x	100
stv090x	3401	4.6x	95	MO	54	6.3x	97
acenic	3290	TO	65	1.1x	100	223x	100
qla3xxx	2085	163x	95	210x	79	169x	100
cx25840	56	8.8x	83	0.7x	100	9.9x	83
mlx4_en	46	1.2x	91	1.2x	32	1.0x	100
advansys	109	1.7x	92	Crash	30	1.4x	99.7
i7300_edac	36	2.6x	83	1.2x	100	1.4x	99
oss_sound	2428	245x	80	1.2x	100	229x	80

Table 4. Statistics on $m_{\max}$ for different Polyhedra analyses on 10 benchmarks.

Benchmark	$m_{\max}^{\text{Elina}}$		$m_{\max}^{\text{HC}}$		$m_{\max}^{\text{Poly-Rl}}$		$m_{\max}^{\text{Lait}}$	
	max	avg	max	avg	max	avg	max	avg
qlogic_qlge	267	6	19	4	205	5	33	4
peak_usb	48	7	17	5	48	7	24	7
stv090x	74	12	32	14	-	-	35	13
acenic	98	9	-	-	98	8	28	5
qla3xxx	284	17	30	9	218	15	19	8
cx25840	26	10	17	7	26	9	17	8
mlx4_en	56	4	53	4	54	4	56	4
advansys	38	9	37	9	-	-	38	8
i7300_edac	41	14	20	9	41	14	28	11
oss_sound	47	9	38	7	47	8	23	7

Lait vs. HC. We next compare the runtime and precision of Lait and HC. Figure 4(a) shows that Lait and HC achieve overall a similar runtime on Test set 1 (except that HC timed out on 3 benchmarks). However, on Test set 2, Lait does better than HC: it finished 22 more benchmarks and tends to be faster on those where both finished.

In precision, Lait clearly outperforms HC as shown in Figure 4(b) on Test set 1. Lait achieves 100% precision in most cases as already mentioned above, whereas HC often loses precision and is more precise than Lait on only 11/147 benchmarks.

Lait vs. Poly-Rl. Figure 4(c) shows that Lait gains significantly over Poly-Rl in speed: for Test set 1, Lait is faster than Poly-Rl on a significant portion of the benchmarks and finished 75 more benchmarks from Test set 2 than Poly-Rl.

Lait gains in precision overall compared to Poly-Rl on most benchmarks in Test set 1 as shown in Figure 4(d). On the majority of benchmarks, the precision of Lait is ≈ 100%. Lait loses on 7 benchmarks where it has < 95% precision but gains on 24 benchmarks where HC achieves < 95% precision.

Finally, we note that Poly-Rl ran out of memory or crashed on 6 benchmarks because of too many constraints. These are not shown in Figure 4.

Testing benchmarks from other categories. To investigate how Lait generalizes across different types of benchmarks, we also tested benchmarks from two other categories in sv-comp, focusing on the subset for which Elina timed out at 2h. For the seq-mthreaded category, this subset contained 39 benchmarks of which Lait could finish 20 within 2h, with a median of at least 36x speedup (mean at least 174x). HC and Poly-Rl could finish none. For the product-lines category, Elina timed out on 65 of which Lait could handle 64 within 2h, with a median of at least 2229x speedup (mean at least 2155x). HC finished none and Poly-Rl finished 9. This shows that the constraint removal policy learned by Lait is able to speedup Polyhedra analysis for benchmark types not present in the training set. We note that even though we cannot report the precision as Elina timed out, the invariants produced by Lait are not expressible in weaker domains

7.2 Interpretability of Lait on Polyhedra Analysis

To better understand how Lait is able to achieve significant speedup and maintain precision, we analyze 10 benchmarks from Test set 1. They are collected in Table 3 together with the speedup and precision achieved by HC, Poly-Rl, and Laitw.r.t. Elina. In the table, Poly-Rl did not finish on

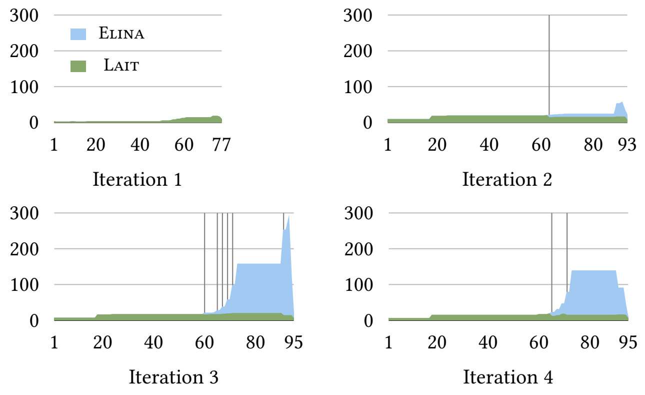


Figure 5. Join traces for one loop of the qla3xxx benchmark which is the performance bottleneck in the analysis. Shown is $m_{\max}$ (y-axis) vs. join index (x-axis).

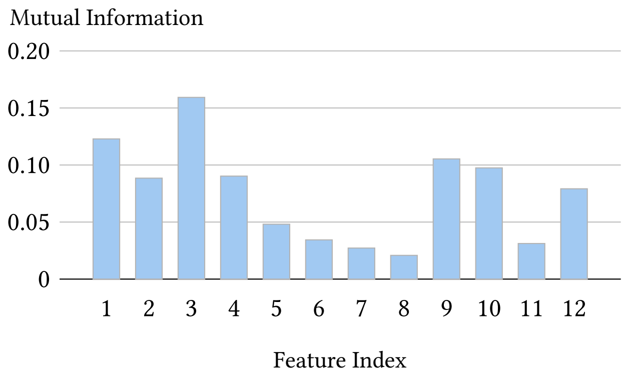


Figure 6. Dependency (measured by mutual information) between the features listed in Table 1 and LAIT's decisions for approximating Polyhedra analysis.

two benchmarks whereas HC timed out on one. Program i7300_edac has 309 program points and the rest each has > 1k program points.

Statistics. The time complexity of the Polyhedra analysis with online decomposition is exponential in the number of constraints in the largest block $m_{\max}$ [44]. We gathered statistics on $m_{\max}$ after each join in the loop and report its maximum and average value in Table 4. As can be seen, the speedups in the table are strongly correlated with the reduction in $m_{\max}$. For example, for the benchmark qlogic_qlge where LAIT is 53x faster, the maximum of $m_{\max}^{\text{LAIT}}$ is $\approx$ 8x smaller than the maximum of $m_{\max}^{\text{ELINA}}$. POLY-RL is not much faster than ELINA in many cases because the maximum of $m_{\max}^{\text{POLY-RL}}$ stays close to the maximum of $m_{\max}^{\text{ELINA}}$ for most of the benchmarks shown.

LAIT analysis trace visualization. To investigate the previous reasoning in greater detail, we visualize in Figure 5 $m_{\max}^{\text{LAIT}}$ for joins when analyzing the bottleneck loop of benchmark qla3xxx (ELINA spent 99% of its analysis time analyzing this loop). The loop consists of four analysis iterations and, for each iteration, we show the value of $m_{\max}^{\text{LAIT}}$ and $m_{\max}^{\text{ELINA}}$ (y-axis) of each join (indexed by the x-axis). In iterations 2–4, the value of $m_{\max}^{\text{ELINA}}$ starts exploding from join index 60 and drops after join index 90. Indeed, ELINA spent most of the analysis time in this portion of the program. LAIT, in contrast, can keep $m_{\max}^{\text{LAIT}}$ always reasonably small by occasionally removing constraints (vertical lines denote joins where LAIT removes constraints). Despite the constraint removal, the precision at the loop heads after iteration 2–4 are the same for both LAIT and ELINA in this case.

Feature dependency of LAIT's decisions. To identify important features, we compute mutual information scores, over all constraints, between the distribution of LAIT's decisions (keep or discard) on Test set 1 and the distribution of each feature listed in Table 1. We plot the scores in Figure 6. We found that LAIT's predictions have the highest dependency with features 1 and 3. Higher values of these features lead to a higher likelihood of constraint removal as it means that the corresponding block has more variables and generators and thus dominates the cost of the overall join. The same holds for feature 2. Further, features 9 and 10 also appear important. A higher value for feature 9 or feature 10 being false for a constraint indicates potential redundancy. Feature 4 and feature 12 are also closely related to LAIT's decisions. The former indicates whether the constraint is likely to be involved in the fixed point computation. As for the latter, LAIT tends to remove inequalities more often.

From Figure 6, we can see that LAIT made its decision based on various features. A possible direction for more refined interpretability of the constraint removal policy learned by LAIT is to jointly consider features and edges. This is an ongoing research area [47].

7.3 Results of LAIT on Octagon Analysis

We present results on applying LAIT for the Octagon domain.

Training and testing. We selected 10 benchmarks (8 from the LDD categories and 2 from the product-lines category) for training and another 10 LDD benchmarks where ELINA took > 10s for finishing the analysis for testing. The training took about 4h for around 20 iterations. We note that for Octagon, ELINA is a harder baseline as it is already very fast on most of the LDD benchmarks. Further, the complexity is cubic instead of exponential and thus inherently less amenable for dramatic improvements. We also compare LAIT against HC for the Octagon domain.

LAIT vs. ELINA. The results are presented in Table 5. We first compare LAIT to ELINA. LAIT can speed up analysis (median 1.32x) without losing much precision (median 99.4%).

Table 5. Speed and precision for different Octagon analyses on 10 benchmarks.

Benchmark	#Points	Elina Time	HC		Lait	
			Speed	Prec.	Speed	Prec.
advansys	3408	34	1.22x	99.4	1.15x	98.8
net_unix	2037	13	TO	52.5	1.45x	95.1
vmwgfx	7065	45	1.08x	100	1.24x	100
phoenix	644	26	1.55x	96.9	1.31x	100
mwl8k	4206	27	1.05x	64.2	1.55x	99.8
saa7164	6565	117	1.00x	57.8	1.54x	97.9
md_mod	8222	1309	TO	68.1	28x	98.9
block_rsxx	2426	14	1.11x	73.9	1.26x	98.8
ath_ath9k	3771	26	1.07x	65.7	1.33x	99.8
synclink_gt	2324	44	1.28x	100	1.23x	100

Table 6. Statistics on $m_{\max}$ for different Octagon analyses on 10 benchmarks.

Benchmark	$m_{\max}^{\text{Elina}}$		$m_{\max}^{\text{HC}}$		$m_{\max}^{\text{Lait}}$	
	max	avg	max	avg	max	avg
advansys	968	58	882	47	968	48
net_unix	118	32	-	-	58	10
vmwgfx	1736	385	1568	358	420	44
phoenix	933	175	310	100	648	131
mwl8k	1862	567	1800	527	72	22
saa7164	1486	264	1352	244	128	41
md_mod	314	149	-	-	128	10
block_rsxx	790	106	648	93	243	6
ath_ath9k	2382	1145	2312	889	800	218
synclink_gt	98	7	98	7	98	6

The highest speedup is 28x on the md_mod benchmark because approximation makes the analysis converge faster in fewer loop iterations.

Lait vs. HC. Lait outperforms HC in terms of both speed and precision. Lait was faster than HC for 8 benchmarks (HC has 2 TOs), and is equally or more precise than HC on 9 benchmarks. Specifically, the precision of HC can drop to $< 60\%$ while the precision of Lait stays $> 95\%$ on all benchmarks.

Statistics. In Table 6, we show $m_{\max}^{\text{Elina}}$, $m_{\max}^{\text{HC}}$ and $m_{\max}^{\text{Lait}}$ for the Octagon domain. $m_{\max}^{\text{Lait}}$ is significantly lower than $m_{\max}^{\text{Elina}}$, which speeds up the transformers in the Octagon domain implemented using sparse algorithms [43]. HC has a smaller speedup as $m_{\max}^{\text{HC}}$ remains large.

Feature dependency of Lait's decisions. As for the Polyhedra domain, we computed mutual information scores between Lait's decisions and the features for the Octagon domain in Table 2. Our results show that features 1, 2 and 11 are most important with scores of 0.32, 0.38 and 0.35, respectively. These features capture the complexity of the join output. Feature 5 and 6 are also closely correlated with Lait's predictions, with scores of 0.19 and 0.18. A high score for these features means that the constraint relates variables that are related to many other variables and thus the constraint is needed for maintaining precision.

7.4 Discussion

Balancing the tradeoff between precision and speed for numerical analysis with highly expressive numerical domains is known to be a challenging problem. For the Polyhedra analysis, we show in Section 7.1 that HC is often fast but can be imprecise whereas Poly-Rl is more precise than HC but only gets modest speedups over Elina. In contrast, Lait effectively identifies and removes redundancy in abstract sequences and achieves significant speedup, of sometimes orders of magnitude, while maintaining precision. Importantly, it accomplishes this with a generic strategy of removing constraints, rather than the heavily domain-specific approach used in Poly-Rl. Therefore, Lait also performs well on the Octagon analysis, as demonstrated in Section 7.3.

Different choices of the classifier. To investigate the necessity of structured prediction in Lait, we replace GCN with a decision tree (DT), or a 4-layer fully connected network (FCN) with hidden dimension 128, resulting in two additional baselines Lait_{DT} and Lait_{FCN}. We re-trained Lait_{DT} and Lait_{FCN} on the same training set as in the previous experiments for both Polyhedra and Octagon domains. The testing results for the benchmarks in Table 3 and Table 5 are shown in Table 7 and Table 8, respectively.

As shown in Table 7 and Table 8, Lait_{DT}, Lait_{FCN}, and Lait have similar precision and speed on most benchmarks. This indicates that our constraint removal framework is effective in reducing redundancy regardless of the choice of the classifier. For six benchmarks (marked with a red font in Table 7 and Table 8), Lait has significantly better speed or precision (or both) than Lait_{DT} or Lait_{FCN}. Lait_{DT} and Lait_{FCN} never exhibit significantly better results than Lait in neither precision nor speed. This suggests that structured prediction with a GCN is necessary to achieve best results.

8 Related Work

We discuss research on using machine learning to improve various aspects of program analysis. Related work in other directions was already discussed in the previous sections.

Numerical static analysis. The authors of [25, 35] learn a static partition strategy for the Octagon domain aiming to boost analysis speed while aiming to prove most of the assertions that the original analysis proves. Lait, however,

Table 7. Speed and precision of using different classifiers for the Polyhedra analysis on 10 benchmarks.

Benchmark	ELINA Time	LAIT$_{DT}$ Speed	LAIT$_{DT}$ Prec.	LAIT$_{FCN}$ Speed	LAIT$_{FCN}$ Prec.	LAIT Speed	LAIT Prec.
qlogic_qlge	3474	56x	100	59x	100	53x	100
peak_usb	1919	333x	100	325x	100	315x	100
stv090x	3401	5.6x	97	5.3x	96	6.3x	97
acenic	3290	227x	100	0.5x	65	223x	100
qla3xxx	2085	191x	100	189x	100	169x	100
cx25840	56	10x	83	8.7x	83	9.9x	83
mlx4_en	46	1.0x	100	1.1x	98	1.0x	100
advansys	109	0.2x	99.4	1.3x	99.3	1.4x	99.7
i7300_edac	36	2.0x	61	1.7x	99	1.4x	99
oss_sound	2428	247x	80	260x	80	229x	80

Table 8. Speed and precision of using different classifiers for the Octagon analysis on 10 benchmarks.

Benchmark	ELINA Time	LAIT$_{DT}$ Speed	LAIT$_{DT}$ Prec.	LAIT$_{FCN}$ Speed	LAIT$_{FCN}$ Prec.	LAIT Speed	LAIT Prec.
advansys	34	1.20x	98.9	1.13x	99.4	1.15x	98.8
net_unix	13	0.54x	78.2	1.41x	78.2	1.45x	95.1
vmwgfx	45	1.18x	100	1.19x	100	1.24x	100
phoenix	26	1.23x	100	1.58x	96.9	1.31x	100
mwl8k	27	1.50x	64.4	1.30x	64.4	1.55x	99.8
saa7164	117	1.35x	97.9	1.38x	99.4	1.54x	97.9
md_mod	1309	28x	99.3	28x	98.9	28x	98.9
block_rsxx	14	1.28x	98.8	1.25x	98.8	1.26x	98.8
ath_ath9k	26	1.21x	67.6	1.25x	99.7	1.33x	99.8
synclink_gt	44	1.25x	100	1.30x	100	1.23x	100

is based on dynamic online decomposition, which is significantly more precise than static partitioning [43, 44]. Another direction is to determine with Bayesian optimization [36] or reinforcement learning [26], a subset of variables to be tracked in a flow sensitive manner to speed up analysis [36] or increase utilization of the memory budget [26], with little loss of precision. These works focus on the non-relational Interval domain while our work targets the relational Polyhedra and Octagon domains. POLY-RL [42] designs a set of hand-crafted approximate Polyhedra join transformers and learns an adaptive policy via reinforcement learning to selectively apply the designed join transformers to balance precision and speed. Our experimental results show that LAIT outperforms POLY-RL. Moreover, our work is generic and does not require hand-crafted join approximations.

Numerical solvers. Machine learning methods have been extensively applied for optimizing different solvers. The work of [4, 29] learns branching rules via empirical risk minimization for solving mixed integer linear programming problems. The work of [30] learns to solve combinatorial optimization problems over graphs via reinforcement learning. fastSMT [5] learns a policy to apply appropriate tactics to speed up numerical SMT solving.

Inferring invariants. Recent research has investigated the problem of inferring numerical program invariants with machine learning. The works of [17, 40, 49] use machine learning for inferring inductive loop invariants for program verification. The learning algorithms in these works require specifications in the form of pre/post conditions. Our work speeds up numerical abstract interpretation which can be used to infer program invariants without pre/post conditions.

Testing. In recent years, there has been emerging interest in learning to produce test inputs for finding program bugs or vulnerabilities. AFLFast [8] models program branching behavior with Markov Chain, which guides the input generation. Several works train neural networks to generated new test inputs, where the training set can be obtained from existing test corpus [15, 21], inputs generated earlier in the testing process [39], or inputs generated by symbolic execution [24].

9 Conclusion

We proposed a data-driven approach for boosting the speed of existing numerical program analysis without significant precision loss. Our key insight is to leverage structured prediction for identifying and reducing redundancy in the sequence of abstract states produced by a static analyzer. Concretely, we showed how to learn an adaptive neural policy for redundancy removal that makes decisions based on the abstract states created during analysis.

Our experimental results on challenging real-world benchmarks demonstrate that our approach is indeed effective in enabling fast and precise numerical analysis: it achieves orders of magnitude speedups over state-of-the-art methods with only a small loss of precision.

References

[1] 2019. ELINA. http://elina.ethz.ch/
[2] 2019. Pytorch. https://pytorch.org/
[3] Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. 2008. 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 (2008), 3–21. https://doi.org/10.1016/j.scico.2007.08.001
[4] Maria-Florina Balcan, Travis Dick, Tuomas Sandholm, and Ellen Vitercik. 2018. Learning to Branch. In *ICML 2018*. http://proceedings.mlr.press/v80/balcan18a.html
[5] Mislav Balunovic, Pavol Bielik, and Martin T. Vechev. 2018. Learning to Solve SMT Formulas. In *NeurIPS 2018*. http://papers.nips.cc/paper/8233-learning-to-solve-smt-formulas

[6] Dirk Beyer. 2016. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2016). In *TACAS 2016*. https://doi.org/10.1007/978-3-662-49674-9_55

[7] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2003. A Static Analyzer for Large Safety-Critical Software. In *PLDI 2003*. https://doi.org/10.1145/781131.781153

[8] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *CCS 2016*. https://doi.org/10.1145/2976749.2978428

[9] Aziem Chawdhary and Andy King. 2018. Closing the Performance Gap Between Doubles and Rationals for Octagons. In *SAS 2018*. https://doi.org/10.1007/978-3-319-99725-4_13

[10] Aziem Chawdhary, Edward Robbins, and Andy King. 2019. Incrementally Closing Octagons. *Formal Methods Syst. Des.* 54, 2 (2019), 232–277. https://doi.org/10.1007/s10703-017-0314-7

[11] Robert Clarisó and Jordi Cortadella. 2004. The Octahedron Abstract Domain. In *SAS 2004*. https://doi.org/10.1007/978-3-540-27864-1_23

[12] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2018. Program Analysis Is Harder Than Verification: A Computability Perspective. In *CAV 2018*. https://doi.org/10.1007/978-3-319-96142-2_8

[13] Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. 2019. A^2I: Abstract^2 Interpretation. *PACMPL* 3, POPL (2019), 42:1–42:31. https://doi.org/10.1145/3290355

[14] Patrick Cousot and Nicolas Halbwachs. 1978. Automatic Discovery of Linear Restraints Among Variables of a Program. In *POPL 1978*. https://doi.org/10.1145/512760.512770

[15] Chris Cummins, Pavlos Petoumenos, Alastair Murray, and Hugh Leather. 2018. Compiler Fuzzing through Deep Learning. In *ISSTA 2018*. https://doi.org/10.1145/3213846.3213848

[16] Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2016. Exploiting Sparsity in Difference-Bound Matrices. In *SAS 2016*. https://doi.org/10.1007/978-3-662-53413-7_10

[17] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. 2016. Learning Invariants using Decision Trees and Implication Counterexamples. In *POPL 2016*. https://doi.org/10.1145/2837614.2837664

[18] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin T. Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *S&P 2018*. https://doi.org/10.1109/SP.2018.00058

[19] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *PLDI 2019*. https://doi.org/10.1145/3314221.3314590

[20] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. 2009. The Zonotope Abstract Domain Taylor1+. In *CAV 2009*. https://doi.org/10.1007/978-3-642-02658-4_47

[21] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. In *ASE 2017*. https://doi.org/10.1109/ASE.2017.8115618

[22] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. 2015. The SeaHorn Verification Framework. In *CAV 2015*. https://doi.org/10.1007/978-3-319-21690-4_20

[23] Nicolas Halbwachs, David Merchat, and Catherine Parent-Vigouroux. 2003. Cartesian Factoring of Polyhedra in Linear Relation Analysis, Vol. 2694. https://doi.org/10.1007/3-540-44898-5_20

[24] Jingxuan He, Mislav Balunovic, Nodar Ambroladze, Petar Tsankov, and Martin T. Vechev. 2019. Learning to Fuzz from Symbolic Execution with Application to Smart Contracts. In *CCS 2019*. https://doi.org/10.1145/3319535.3363230

[25] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2016. Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis. In *SAS 2016*. https://doi.org/10.1007/978-3-662-53413-7_12

[26] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware Program Analysis via Online Abstraction Coarsening. In *ICSE 2019*. https://doi.org/10.1109/ICSE.2019.00027

[27] Bertrand Jeannet and Antoine Miné. 2009. Apron: A Library of Numerical Abstract Domains for Static Analysis. In *CAV 2009*. https://doi.org/10.1007/978-3-642-02658-4_52

[28] Jacques-Henri Jourdan. 2017. Sparsity Preserving Algorithms for Octagons. *Electr. Notes Theor. Comput. Sci.* 331 (2017), 57–70. https://doi.org/10.1016/j.entcs.2017.02.004

[29] Elias Boutros Khalil, Pierre Le Bodic, Le Song, George L. Nemhauser, and Bistra Dilkina. 2016. Learning to Branch in Mixed Integer Programming. In *AAAI 2016*. http://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/view/12514

[30] Elias B. Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning Combinatorial Optimization Algorithms over Graphs. In *NIPS 2017*. http://papers.nips.cc/paper/7214-learning-combinatorial-optimization-algorithms-over-graphs

[31] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR 2017*. https://openreview.net/forum?id=SJU4ayYgl

[32] Alexandre Maréchal, David Monniaux, and Michaël Périn. 2017. Scalable Minimizing-Operators on Polyhedra via Parametric Linear Programming. In *SAS 2017*. https://doi.org/10.1007/978-3-319-66706-5_11

[33] Antoine Miné. 2002. A Few Graph-Based Relational Numerical Abstract Domains. In *SAS 2002*. https://doi.org/10.1007/3-540-45789-5_11

[34] Antoine Miné. 2006. The octagon abstract domain. *Higher-Order and Symbolic Computation* 19, 1 (2006). https://doi.org/10.1007/s10990-006-8609-1

[35] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. 2014. Selective Context-Sensitivity Guided by Impact Pre-Analysis. In *PLDI 2014*. https://doi.org/10.1145/2594291.2594318

[36] Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. 2015. Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation. In *OOPSLA 2015*. https://doi.org/10.1145/2814270.2814309

[37] Stéphane Ross, Geoffrey J. Gordon, and Drew Bagnell. 2011. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. In *AISTATS 2011*. http://proceedings.mlr.press/v15/ross11a/ross11a.pdf

[38] Rahul Sharma, Aditya V. Nori, and Alex Aiken. [n. d.]. Bias-variance Tradeoffs in Program Analysis. In *POPL 2014*. https://doi.org/10.1145/2535838.2535853

[39] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. NEUZZ: Efficient Fuzzing with Neural Program Smoothing. In *S&P 2019*. https://doi.org/10.1109/SP.2019.00052

[40] Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *NeurIPS 2018*. http://papers.nips.cc/paper/8001-learning-loop-invariants-for-program-verification

[41] Axel Simon and Andy King. 2010. The Two Variable per Inequality Abstract Domain. *Higher-Order and Symbolic Computation* 23, 1 (2010), 87–143. https://doi.org/10.1007/s10990-010-9062-8

[42] Gagandeep Singh, Markus Püschel, and Martin Vechev. 2018. Fast Numerical Program Analysis with Reinforcement Learning. In *CAV 2018*. https://doi.org/10.1007/978-3-319-96145-3_12

[43] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2015. Making Numerical Program Analysis Fast. In *PLDI 2015*. https://doi.org/10.1145/2737924.2738000

[44] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2017. Fast Polyhedra Abstract Domain. In *POPL 2017*. http://dl.acm.org/citation.cfm?id=3009885

[45] Gagandeep Singh, Markus Püschel, and Martin T. Vechev. 2018. A Practical Construction for Decomposing Numerical Abstract Domains. *PACMPL* 2, POPL (2018), 55:1–55:28. https://doi.org/10.1145/3158143

[46] Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. 2018. Evaluating Design Tradeoffs in Numeric Static Analysis for Java. In *ESOP 2018*. https://doi.org/10.1007/978-3-319-89884-1_23

[47] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *NeurIPS 2019*. http://papers.nips.cc/paper/9123-gnnexplainer-generating-explanations-for-graph-neural-networks

[48] Hang Yu and David Monniaux. 2019. An Efficient Parametric Linear Programming Solver and Application to Polyhedral Projection. In *SAS 2019*. https://doi.org/10.1007/978-3-030-32304-2_11

[49] He Zhu, Stephen Magill, and Suresh Jagannathan. 2018. A Data-Driven CHC Solver. In *PLDI 2018*. https://doi.org/10.1145/3192366.3192416