Fast Numerical Program Analysis with Reinforcement Learning

Gagandeep Singh, Markus Püschel, and Martin Vechev

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

Abstract. We show how to leverage reinforcement learning (RL) in order to speed up static program analysis. The key insight is to establish a correspondence between concepts in RL and those in analysis: a state in RL maps to an abstract program state in analysis, an action maps to an abstract transformer, and at every state, we have a set of sound transformers (actions) that represent different trade-offs between precision and performance. At each iteration, the agent (analysis) uses a policy learned offline by RL to decide on the transformer which minimizes loss of precision at fixpoint while improving analysis performance. Our approach leverages the idea of online decomposition (applicable to popular numerical abstract domains) to define a space of new approximate transformers with varying degrees of precision and performance. Using a suitably designed set of features that capture key properties of abstract program states and available actions, we then apply Q-learning with linear function approximation to compute an optimized context-sensitive policy that chooses transformers during analysis. We implemented our approach for the notoriously expensive Polyhedra domain and evaluated it on a set of Linux device drivers that are expensive to analyze. The results show that our approach can yield massive speedups of up to two orders of magnitude while maintaining precision at fixpoint.

1 Introduction

Static analyzers that scale to real-world programs yet maintain high precision are difficult to design. Recent approaches to attacking this problem have focused on two complementary methods. On one hand is work that designs clever algorithms that exploits the special structure of particular abstract domains to speed up analysis [20, 21, 5, 10, 16, 15]. These works tackle specific types of analyses but the gains in performance can be substantial. On the other hand are approaches that introduce creative mechanisms to trade off precision loss for gains in speed [12, 19, 18, 9]. While promising, these methods typically do not take into account the particular abstract states arising during analysis which determine the precision of abstract transformers (e.g., join), resulting in suboptimal analysis precision or performance. A key challenge then is coming up with effective and general approaches that can decide where and how to lose precision *during analysis* for best tradeoff between performance and precision.

Our Work. We address the above challenge by offering a new approach for dynamically losing precision based on reinforcement learning (RL) [24]. The key idea is to learn a policy that determines when and how the analyzer should lose the least precision at an abstract state to achieve best performance gains. Towards that, we establish a correspondence between concepts in static analysis and RL, which demonstrates that RL is a viable approach for handling choices in the inner workings of a static analyzer.

To illustrate the basic idea, imagine that a static analyzer has at each program state two available abstract transformers: the precise but slow T_p and the fast but less precise T_f. Ideally, the analyzer would decide adaptively at each step on the best choice that maximizes speed while producing a final result of sufficient precision. Such a policy is difficult to craft by hand and hence we propose to leverage RL to discover the policy automatically.

To explain the connection with RL intuitively, we think of abstract states and transformers as analogous to states of a Go board and moves made by the Go player, respectively. In Go, the goal is to learn a policy that at each state decides on the next player action (transformer to use) which maximizes the chances of eventually winning the game (obtaining a precise fixpoint while improving performance in our case). Note that the reward to be maximized in Go is long-term and not an immediate gain in position, which is similar to iterative static analysis. To learn the policy with RL, one typically extracts a set of features ϕ from a given state and action, and uses those features to define a so-called Q-function, which is then learned, determining the desired policy.

In the example above, a learned policy would determine at each step whether to choose action T_p or T_f. To do that, for a given state and action, the analyzer computes the value of the Q-function using the features ϕ. Querying the Q-function returns the suggested action from that state. Eventually, such a policy would ideally lead to a fixpoint of sufficient precision but be computed quicker.

While the overall connection between static analysis and reinforcement learning is conceptually clean, the details of making it work in practice pose significant challenges. The first is the design of suitable approximations to actually be able to gain performance when precision is lost. The second is the design of features ϕ that are cheap to compute yet expressive enough to capture key properties of abstract states. Finally, a suitable reward function combining both precision and performance is needed. We show how to solve these challenges for Polyhedra analysis.

Main contributions. Our main contributions are:

- A space of sound, approximate Polyhedra transformers spanning different precision/performance trade-offs. The new transformers combine online decomposition with different constraint removal and merge strategies for approximations (Section 3).
- A set of feature functions which capture key properties of abstract states and transformers, yet are efficient to extract (Section 4).

– A complete instantiation of RL for Polyhedra analysis based on Q-learning with linear function approximation (i.e., actions, reward function, Q-function).
– An end-to-end implementation and evaluation of our approach. Given a training dataset of programs, we first learn a policy (based on the Q-function) over analysis runs of these programs. We then use the resulting policy during analysis of new, unseen programs. The experimental results on a set of realistic programs (e.g., Linux device drivers) show that our RL-based Polyhedra analysis achieves substantial speed-ups (up to 515x) over a heavily optimized state-of-the-art Polyhedra library.

We believe the reinforcement learning based approach outlined in this work can be applied to speed up other program analyzers (beyond Polyhedra).

2 Reinforcement Learning for Static Analysis

In this section we first introduce the general framework of reinforcement learning and then discuss its instantiation for static analysis.

2.1 Reinforcement Learning

Reinforcement learning (RL) [24] involves an *agent* learning to achieve a goal by interacting with its *environment*. The agent starts from an initial representation of its environment in the form of an initial state $s_0 \in \mathcal{S}$ where $\mathcal{S}$ is the set of possible states. Then, at each time step $t = 0, 1, 2, \dots$, the agent performs an action $a_t \in \mathcal{A}$ in state s_t ($\mathcal{A}$ is the set of possible actions) and moves to the next state s_{t+1}. The agent receives a numerical reward $r(s_t, a_t, s_{t+1}) \in \mathbb{R}$ for moving from the state s_t to s_{t+1} through action a_t. The agent repeats this process until it reaches a final state. Each sequence of states and actions from an initial state to the final state is called an *episode*.

In RL, state transitions typically satisfy the Markov property: the next state s_{t+1} depends only on the current state s_t and the action a_t taken from s_t. A *policy* $p\colon \mathcal{S} \to \mathcal{A}$ is a mapping from states to actions: it specifies the action $a_t = p(s_t)$ that the agent will take when in state s_t. The agent's goal is to learn a policy that maximizes not an immediate but a cumulative reward for its actions in the long term. The agent does this by selecting the action with the highest expected long-term reward in a given state. The quality function (Q-function) $Q\colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ specifies the long term cumulative reward associated with choosing an action a_t in state s_t. Learning this function, which is not available a priori, is essential for determining the best policy and is explained next.

Q-learning and approximating the Q-function. Q-learning [25] can be used to learn the Q-function over state-action pairs. Typically the size of the state space is so large that it is not feasible to explicitly compute the Q-function for each state-action pair and thus the function is approximated. In this paper, we consider a *linear* function approximation of the Q-function for three reasons: (i) *effectiveness*: the approach is efficient, can handle large state spaces, and works well in practice [6]; (ii) *it leverages our application domain*: in our setting, it is

Algorithm 1 Q-learning algorithm

```
1: function Q-LEARN($\mathcal{S}, \mathcal{A}, r, \gamma, \alpha, \phi$)
2:     Input:
3:         $\mathcal{S} \leftarrow$ set of states, $\mathcal{A} \leftarrow$ set of actions, $r \leftarrow$ reward function
4:         $\gamma \leftarrow$ discount factor, $\alpha \leftarrow$ learning rate
5:         $\phi \leftarrow$ set of feature functions over $\mathcal{S}$ and $\mathcal{A}$
6:     Output: parameters $\theta$
7:     $\theta$ = Initialize arbitrarily (which also initializes $Q$)
8:     for each episode do
9:         Start with an initial state $s_0 \in \mathcal{S}$
10:        for $t = 0, 1, 2, \dots, length(episode)$ do
11:            Take action $a_t$, observe next state $s_{t+1}$ and $r(s_t, a_t, s_{t+1})$
12:            $\theta := \theta + \alpha \cdot (r(s_t, a_t, s_{t+1}) + \gamma \cdot \mathtt{max}_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)) \cdot \phi(s_t, a_t)$
13:    return $\theta$
```

possible to choose meaningful features (e.g., approximation of volume and cost of transformer) that relate to precision and performance of the static analysis and thus it is not necessary to uncover them automatically (as done, e.g., by training a neural net); and (iii) *interpretability of policy*: once the Q-function and associated policy are learned they can be inspected and interpreted.

The Q-function is described as a linear combination of ℓ basis functions $\phi_i \colon \mathcal{S} \times \mathcal{A} \to \mathbb{R}$, $i = 1, \dots, \ell$. Each ϕ_i is a feature that assigns a value to a (state, action) pair and ℓ is the total number of chosen features. The choice of features is important and depends on the application domain. We collect the feature functions into a vector $\phi(s,a) = (\phi_1(s,a), \phi_2(s,a), \dots, \phi_\ell(s,a))$; doing so, the Q-function has the form:

$$Q(s,a) = \sum_{j=1}^{\ell} \theta_j \cdot \phi_j(s,a) = \phi(s,a) \cdot \theta^T, \tag{1}$$

where $\theta = (\theta_1, \theta_2, \dots, \theta_\ell)$ is the parameter vector. The goal of Q-learning with linear function approximation is thus to estimate (learn) θ.

Algorithm 1 shows the Q-learning procedure. In the algorithm, $0 \leq \gamma < 1$ is the *discount factor* which represents the difference in importance between immediate and future rewards. $\gamma = 0$ makes the agent only consider immediate rewards while $\gamma \approx 1$ gives more importance to future rewards. The parameter $0 < \alpha \leq 1$ is the *learning rate* that determines the extent to which the newly acquired information overrides the old information. The algorithm first initializes θ randomly. Then, for each step t in an episode, the agent takes an action a_t, moves to the next state s_{t+1} and receives a reward $r(s_t, a_t, s_{t+1})$. Line 12 in the algorithm shows the equation for updating the parameters θ. Notice that Q-learning is an off-policy learning algorithm as the update in the equation assumes that the agent follows a greedy policy (from state s_{t+1}) while the action (a_t) taken by the agent (in s_t) need not be greedy.

Table 1. Mapping of RL concepts to Static analysis concepts.

RL concept	Static analysis concept
Agent	Static analyzer
State $s \in \mathcal{S}$	Features of abstract state
Action $a \in \mathcal{A}$	Abstract transformer
Reward function r	Transformer precision and runtime
Feature	Value associated with abstract state features and transformer

Once the Q-function is learned, a policy p^* for maximizing the agent's cumulative reward is obtained as:

$$p^*(s) = \texttt{argmax}_{a \in \mathcal{A}} Q(s, a). \tag{2}$$

In the application, p^* is computed on the fly at each stage s by computing Q for each action a and choosing the one with maximal $Q(s, a)$. Since the number of actions is typically small, this incurs little overhead.

2.2 Instantiation of RL to Static Analysis

We now discuss a general recipe for instantiating the RL framework described above to the domain of static analysis. The precise formal instantiation to the specific numerical (Polyhedra) analysis is provided later.

In Table 1, we show a mapping between RL and program analysis concepts. Here, the analyzer is the agent that observes its environment, which is the abstract program state (e.g., polyhedron) arising at every iteration of the analysis. In general, the number of possible abstract states can be very large (or infinite) and thus, to enable RL in this setting, we abstract the state through a set of features (Table 2). An example of a feature could be the number of bounded program variables or the volume of a polyhedron. The challenge is to define the features to be fast to evaluate, yet sufficiently representative so the policy derived through learning generalizes well to unseen abstract program states.

Further, at every abstract state, the analyzer should have the choice between different actions corresponding to different abstract transformers. The transformers should range from expensive and precise to cheap and approximate. The reward function r is thus composed of a measure of precision and speed and should encourage approximations that are both precise and fast.

The goal of our agent is to then learn an approximation policy that at each step selects an action that tries to minimize the loss of analysis precision at fixpoint, while gaining overall performance. Learning such a policy is typically done offline using a given dataset $\mathcal{D}$ of programs (discussed in evaluation). However, this is computationally challenging because the dataset $\mathcal{D}$ can contain many programs and each program will need to be analyzed many times over during training: even a single run of the analysis can contain many (e.g., thousands) calls to abstract transformers. Thus, a good heuristic may be a complicated function of the chosen features. Hence, to improve the efficiency of learning in practice, one would typically exercise the choice for multiple transformers/actions only at certain program points. A good choice, and one we employ, are join points, where the most expensive transformer in numerical domains usually occurs.

Another key challenge lies in defining a suitable space of transformers. As we will see later, we accomplish this by leveraging recent advances in online decomposition for numerical domains [20–22]. We show how to do that for the notoriously expensive Polyhedra analysis; however, the approach is easily extendable to other popular numerical domains, which all benefit from decomposition.

3 Polyhedra Analysis and Approximate Transformers

In this section we first provide brief background on polyhedra analysis and online decomposition, a recent technique to speed up analysis *without losing precision* and applicable to all popular numerical domains [22]. Then we leverage online decomposition to define a flexible approximation framework that *loses precision* in a way that directly translates into performance gains. This framework forms the basis for our RL approach discussed in Section 4.

3.1 Polyhedra analysis

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be the set of n (numerical) program variables where each variable $x_i \in \mathbb{Q}$ takes a rational value. An abstract element $P \subseteq \mathbb{Q}^n$ in the Polyhedra domain is a conjunction of linear constraints $\sum_{i=1}^{n} a_i x_i \leq c$ between the program variables where $a_i \in \mathbb{Z}, c \in \mathbb{Q}$. This is called the *constraint* representation of the polyhedron.

Constraints and generator representation. For efficiency, it is common to maintain besides the constraint representations also the *generator* representation, which encodes a polyhedron as the convex hull of a finite set of vertices, rays, and lines. Rays and lines are represented by their direction. Thus, by abuse of prior notation we write $P = (\mathcal{C}_P, \mathcal{G}_P)$ where $\mathcal{C}_P$ is the constraints representation (before just called P) and $\mathcal{G}_P$ is the generator representation.

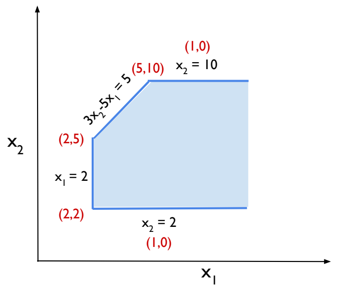

Fig. 1. Two representations of polyhedron P: As conjunction of 4 constraints $\mathcal{C}_P$, and as convex hull of 3 vertices and 2 rays $\mathcal{G}_P$.

Example 1 *Fig. 1 shows an example of the two representations of an abstract element P in the Polyhedra domain. $\mathcal{C}_P$ is the intersection of 4 linear constraints:*

$$\mathcal{C}_P = \{-x_1 \leq -2, -x_2 \leq -2, x_2 \leq 10, 3x_2 - 5x_1 \leq 5\}.$$

$\mathcal{G}_P$ is the convex hull of 3 vertices and 2 rays:

$$\mathcal{G}_P = \{\textit{vertices, rays, lines}\} = \{\{(2,2),(2,5),(5,10)\},\{(1,0),(1,0)\},\emptyset\}.$$

Notice that $\mathcal{G}_P$ contains two rays in the same direction $(1,0)$; thus one of them could be removed without changing the set of points in P.

During analysis, the abstract elements are manipulated with abstract transformers that model the effect of statements and control flow in the program such as assignment, conditional, join, and others. Upon termination of the analysis, each program statement has an associated subsequent P containing all possible variable values after this statement. The main bottleneck for the Polyhedra analysis is the join transformer ($\sqcup$), and thus it is the focus for our approximations.

Recently, Polyhedra domain analysis was sped up by orders of magnitude, without approximation, using the idea of online decomposition [21]. The basic idea is to dynamically decompose the occurring abstract elements into independent components (in essence abstract elements on smaller variable sets) based on the connectivity between variables in the constraints, and to maintain this (permanently changing) decomposition during analysis. The finer the decomposition, the faster the analysis.

Our approximation framework builds on online decomposition. The basic idea is simple: we approximate by dropping constraints to reduce connectivity among constraints and thus to yield finer decompositions of abstract elements. These directly translate into speedup. We consider various options of such approximation; reinforcement learning (in Section 4) will then learn a proper, context-sensitive strategy that stipulates when and which approximation option to apply.

Next, we provide brief background on the ingredients of online decomposition and explain our mechanisms for soundly approximating the join transformer.

3.2 Online Decomposition

Online decomposition is based on the observation that during analysis, the set of variables $\mathcal{X}$ in a given polyhedron P can be partitioned as $\pi_P = \{\mathcal{X}_1, \ldots, \mathcal{X}_r\}$ into *blocks* $\mathcal{X}_t$, such that constraints exist only between variables in the same block. Each unconstrained variable $x_i \in \mathcal{X}$ yields a singleton block $\{x_i\}$. Using this partition, P can be decomposed into a set of smaller Polyhedra $P(\mathcal{X}_t)$ called *factors*. As a consequence, the abstract transformer can now be applied only on the small subset of factors relevant to the program statement, which translates into better performance.

Example 2 *Consider the set $\mathcal{X} = \{x_1, x_2, x_3, x_4, x_5, x_6\}$ and the polyhedron:*

$$P = \{2x_1 - 3x_2 + x_3 + x_4 \le 0, x_5 = 0\}.$$

Here, $\pi_P = \{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$ is a possible partition of $\mathcal{X}$ with factors

$$P(\mathcal{X}_1) = \{2x_1 - 3x_2 + x_3 + x_4 \le 0\},\ P(\mathcal{X}_2) = \{x_5 = 0\},\ P(\mathcal{X}_3) = \emptyset.$$

The set of partitions of $\mathcal{X}$ forms a lattice with the ordering $\pi \sqsubseteq \pi'$ iff every block of π is a subset of a block of π'. Upper and lower bound of two partitions π_1, π_2, i.e., $\pi_1 \sqcup \pi_2$ and $\pi_1 \sqcap \pi_2$ are defined accordingly.

The optimal (finest) partition for an element P is denoted with π_P. Ideally, one would always determine and maintain this finest partition for each output Z of a transformer but it may be too expensive to compute. Thus, the online decomposition in [20, 21] often computes a (cheaply computable) *permissible* partition $\overline{\pi}_Z \sqsupseteq \pi_Z$. Note that making the output partition coarser (while keeping the same constraints) does not change the precision of the abstract transformer.

3.3 Approximating the Polyhedra Join

Let $\overline{\pi}_{\text{com}} = \overline{\pi}_{P_1} \sqcup \overline{\pi}_{P_2}$ be a common permissible partition for the inputs P_1, P_2 of the join transformer. Then, from [21], a permissible partition for the (not approximated) output is obtained by keeping all blocks $\mathcal{X}_t \in \overline{\pi}_{\text{com}}$ for which $P_1(\mathcal{X}_t) = P_2(\mathcal{X}_t)$ in the output partition $\overline{\pi}_Z$, and fusing all remaining blocks into one. Formally, $\overline{\pi}_Z = \{\mathcal{N}\} \cup \mathcal{U}$, where

$$\mathcal{N} = \bigcup \{\mathcal{X}_k \in \overline{\pi}_{\text{com}} : P_1(\mathcal{X}_k) \neq P_2(\mathcal{X}_k)\}, \quad \mathcal{U} = \{\mathcal{X}_k \in \overline{\pi}_{\text{com}} : P_1(\mathcal{X}_k) = P_2(\mathcal{X}_k)\}.$$

The join transformer computes the generators $\mathcal{G}_Z$ for the output Z as $\mathcal{G}_Z = \mathcal{G}_{P_1(\mathcal{X} \setminus \mathcal{N})} \times (\mathcal{G}_{P_1(\mathcal{N})} \cup \mathcal{G}_{P_2(\mathcal{N})})$ where $\times$ is the Cartesian product. The constraint representation $\mathcal{C}_Z$ is computed as $\mathcal{C}_Z = \mathcal{C}_{P_1(\mathcal{X} \setminus \mathcal{N})} \cup \texttt{conversion}(\mathcal{G}_{P_1(\mathcal{N})} \cup \mathcal{G}_{P_2(\mathcal{N})})$. The conversion algorithm has worst-case exponential complexity and is the most expensive step of the join. Note that the decomposed join applies it only on the generators $\mathcal{G}_{P_1(\mathcal{N})} \cup \mathcal{G}_{P_2(\mathcal{N})}$ corresponding to the block $\mathcal{N}$.

The cost of the decomposed join transformer depends on the size of the block $\mathcal{N}$. Thus, it is desirable to bound this size by a *threshold* $\in \mathbb{N}$. Let $\mathcal{B} = \{\mathcal{X}_k \in \overline{\pi}_{\text{com}} : \mathcal{X}_k \cap \mathcal{N} \neq \emptyset\}$ be the set of blocks that merge into $\mathcal{N}$ in the output $\overline{\pi}_Z$ and $\mathcal{B}_t = \{\mathcal{X}_k \in \mathcal{B} : |\mathcal{X}_k| > \textit{threshold}\}$ be the set of blocks in $\mathcal{B}$ with size $>$ *threshold*.

Splitting of large blocks. For each block $\mathcal{X}_t \in \mathcal{B}_t$, we apply the join on the associated factors: $Z(\mathcal{X}_t) = P_1(\mathcal{X}_t) \sqcup P_2(\mathcal{X}_t)$. We then remove constraints from $Z(\mathcal{X}_t)$ until it decomposes into blocks of sizes $\leq$ *threshold*. Since we only remove constraints from $Z(\mathcal{X}_t)$, the resulting transformer remains sound. There are many choices for removing constraints as shown in the next example.

Example 3 *Consider the following polyhedron and threshold* $= 4$

$$\begin{aligned} \mathcal{X}_t &= \{x_1, x_2, x_3, x_4, x_5, x_6\}, \\ Z(\mathcal{X}_t) &= \{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 + x_4 \leq 0, x_2 + x_3 \leq 0, \\ &\qquad x_3 + x_4 \leq 0, x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}. \end{aligned}$$

We can remove $\mathcal{M} = \{x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$ *from* $Z(\mathcal{X}_t)$ *to obtain the constraint set* $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 + x_4 \leq 0, x_2 + x_3 \leq 0, x_3 + x_4 \leq 0\}$ *with partition* $\{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$*, which obeys the threshold.*

We could also remove $\mathcal{M}' = \{x_2 + x_3 + x_4 \leq 0, x_3 + x_4 \leq 0\}$ *from* $Z(\mathcal{X}_t)$ *to get the constraint set* $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 \leq 0, x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$ *with partition* $\{\{x_1, x_2, x_3\}, \{x_4, x_5, x_6\}\}$*, which also obeys the threshold.*

We next discuss our choices for the constraint removal algorithm.

Stoer-Wagner min-cut. The first basic idea is to remove a minimal number of constraints in $Z(\mathcal{X}_t)$ that decomposes the block $\mathcal{X}_t$ into two blocks. To do so, we associate with $Z(\mathcal{X}_t)$ a weighted undirected graph $G = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V} = \mathcal{X}_t$. Further, there is an edge between x_i and x_j, if there is a constraint containing both; its weight m_{ij} is the number of such constraints. We then apply the standard Stoer-Wagner min-cut algorithm [23] to obtain a partition of $\mathcal{X}_t$ into $\mathcal{X}_t'$ and $\mathcal{X}_t''$. $\mathcal{M}$ collects all constraints that need to be removed, i.e., those that contain at least one variable from both $\mathcal{X}_t'$ and $\mathcal{X}_t''$.

Example 4 *Fig. 2 shows the graph G for $Z(\mathcal{X}_t)$ in Example 3. Applying the Stoer-Wagner min-cut on G once will cut off x_5 or x_6 by removing the constraint $x_4 - x_5$ or $x_4 - x_6$, respectively. In either case a block of size 5 remains, exceeding the threshold of 4. After two applications, both constraints have been removed and the resulting block structure is given by $\{\{x_1, x_2, x_3, x_4\}, \{x_5\}, \{x_6\}\}$. The associated factors are $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 + x_4 \leq 0, x_2 + x_3 \leq 0, x_3 + x_4 \leq 0\}$ and x_5, x_6 become unconstrained.*

Weighted constraint removal. Our second approach for constraints removal does not associate weights with edges but with constraints. It then removes greedily edges with high weights. Specifically, we consider the following two choices of constraint weights, yielding two different constraint removal policies:

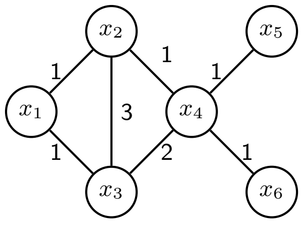

Fig. 2. Graph G for $Z(\mathcal{X}_t)$ in Example 3

- For each variable $x_i \in \mathcal{X}_t$, we first compute the number n_i of constraints containing x_i. The weight of a constraint is then the sum of the n_i over all variables occurring in the constraint.
- For each pair of variables $x_i, x_j \in \mathcal{X}_t$, we first compute the number n_{ij} of constraints containing both x_i and x_j. The weight of a constraint is then the sum of the n_{ij} over all pairs x_i, x_j occurring in the constraint.

Once the weights are computed, we remove the constraint with maximum weight. The intuition is that variables in this constraint most likely occur in other constraints in $Z(\mathcal{X}_t)$ and thus they do not become unconstrained upon constraint removal. This reduces the loss of information.

Example 5 *Applying the first definition of weights in Example 3, we get $n_1 = 1, n_2 = 3, n_3 = 4, n_4 = 4, n_5 = 1, n_6 = 1$. The constraint $x_2 + x_3 + x_4 \leq 0$ has the maximum weight of $n_2 + n_3 + n_4 = 11$ and thus is chosen for removal. Removing this constraint from $Z(\mathcal{X}_t)$ does not yet yield a decomposition; thus we have to repeat. Doing so $\{x_3 + x_4 \leq 0\}$ is chosen. Now, $Z(\mathcal{X}_t) \setminus \mathcal{M} = \{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 \leq 0, x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$ which can be decomposed into two factors $\{x_1 - x_2 + x_3 \leq 0, x_2 + x_3 \leq 0\}$ and $\{x_4 - x_5 \leq 0, x_4 - x_6 \leq 0\}$ corresponding to blocks $\{x_1, x_2, x_3\}$ and $\{x_4, x_5, x_6\}$, respectively, each of size $\leq$ threshold.*

Algorithm 2 Approximation algorithm for Polyhedra join

```
1:  function APPROXIMATE_JOIN($(\overline{\pi}_{P_1}, P_1), (\overline{\pi}_{P_2}, P_2), threshold$)
2:      Input:
3:          $(\overline{\pi}_{P_1}, P_1), (\overline{\pi}_{P_2}, P_2) \leftarrow$ decomposed inputs to the join
4:          $threshold \leftarrow$ Upper bound on size of $\mathcal{N}$
5:      Output: decomposed output $(\overline{\pi}_Z, Z)$ of the join
6:      $Z := \bigcup\{P_1(\mathcal{X}_k) : P_1(\mathcal{X}_k) = P_2(\mathcal{X}_k)\}, \overline{\pi}_Z := \mathcal{U}$      ▷ initialize output
7:      $\mathcal{B} := \{\mathcal{X}_k \in \overline{\pi}_{P_1} \sqcup \overline{\pi}_{P_2} : \mathcal{X}_k \cap \mathcal{N} \neq \emptyset\}$, $\mathcal{B}_t := \{\mathcal{X}_t \in \mathcal{B} : |\mathcal{X}_t| > threshold\}$
    ▷ join factors for blocks in $\mathcal{B}_t$ and split the outputs via a split algorithm
8:      for $\mathcal{X}_t \in \mathcal{B}_t$ do
9:          $P' := P_1(\mathcal{X}_t) \sqcup P_2(\mathcal{X}_t)$
10:         $s\_algo :=$ split_alg$(\mathcal{X}_t, \mathcal{C}_{P'})$, $(\mathcal{C}, \overline{\pi}) :=$ split$(\mathcal{X}_t, \mathcal{C}_{P'}, threshold, s\_algo)$
11:         for $\mathcal{X}_{t'} \in \overline{\pi}$ do
12:             $\mathcal{G}(\mathcal{X}_{t'}) :=$ conversion$(\mathcal{C}(\mathcal{X}_{t'}))$, $Z := Z \cup (\mathcal{C}(\mathcal{X}_{t'}), \mathcal{G}(\mathcal{X}_{t'}))$
13:         $\overline{\pi}_Z := \overline{\pi}_Z \cup \overline{\pi}$
    ▷ merge blocks $\in \mathcal{B} \setminus \mathcal{B}_t$ via a merge algorithm and apply join
14:     $m\_algo :=$ merge_alg$(\mathcal{B} \setminus \mathcal{B}_t)$, $\mathcal{B}_m :=$ merge$(\mathcal{B} \setminus \mathcal{B}_t, threshold, m\_algo)$
15:     for $\mathcal{X}_m \in \mathcal{B}_m$ do
16:         $Z := Z \cup (P_1(\mathcal{X}_m) \sqcup P_2(\mathcal{X}_m))$, $\overline{\pi}_Z := \overline{\pi}_Z \cup \{\mathcal{X}_m\}$
        return $(\overline{\pi}_Z, Z)$
```

Merging blocks. The sizes of all blocks in $\mathcal{B} \setminus \mathcal{B}_t$ are $\leq$ *threshold* and we can apply merging to obtain larger blocks $\mathcal{X}_m \leq$ *threshold* to increase the precision of the subsequent join. The join is then applied on the factors $P_1(\mathcal{X}_m), P_2(\mathcal{X}_m)$ and the result is added to the output Z. We consider the following three merging strategies. To simplify the explanation, we assume that the blocks in $\mathcal{B} \setminus \mathcal{B}_t$ are ordered by ascending size:

1. *No merge:* None of the blocks are merged.
2. *Merge smallest first:* We start merging the smallest blocks as long as the size stays below the threshold. These blocks are then removed and the procedure is repeated on the remaining set.
3. *Merge large with small:* We start to merge the largest block with the smallest blocks as long as the size stays below the threshold. These blocks are then removed and the procedure is repeated on the remaining set.

Example 6 *Consider threshold* $= 5$ *and* $\mathcal{B} \setminus \mathcal{B}_t$ *with block sizes* $\{1, 1, 2, 2, 2, 2, 3, 5, 7, 10\}$. *Merging smallest first yields blocks* $1 + 1 + 2$, $2 + 2$, $2 + 3$ *leaving the rest unchanged. The resulting sizes are* $\{4, 4, 5, 5, 7, 10\}$. *Merging large with small leaves* $10, 7, 5$ *unchanged and merges* $3+1+1$, $2+2$, *and* $2+2$. *The resulting sizes are also* $\{4, 4, 5, 5, 7, 10\}$ *but the associated factors are different (since different blocks are merged), which will yield different results in following transformations.*

Need for RL. Algorithm 2 shows how to approximate the join transformer. Different choices of threshold, splitting, and merge strategies yield a range of transformers with different performance and precision depending on the inputs. All of the transformers are non-monotonic, however the analysis always converges to a fixpoint when combined with widening [2]. Determining the suitability of a given choice on an input is highly non-trivial and thus we use RL to learn it.

Table 2. Features for describing RL state s ($m \in \{1,2\}, 0 \leq j \leq 8, 0 \leq h \leq 3$).

Feature ψ_i	Extraction complexity	Typical range	n_i	Buckets for feature ψ_i
$\lvert\mathcal{B}\rvert$	$O(1)$	1–10	10	$\{[j+1, j+1]\} \cup \{[10, \infty)\}$
$\texttt{min}(\lvert\mathcal{X}_k\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–100	10	$\{[10 \cdot j+1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\texttt{max}(\lvert\mathcal{X}_k\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–100	10	$\{[10 \cdot j+1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\texttt{avg}(\lvert\mathcal{X}_k\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–100	10	$\{[10 \cdot j+1, 10 \cdot (j+1)]\} \cup \{[91, \infty)\}$
$\texttt{min}(\lvert\bigcup \mathcal{G}_{P_m(\mathcal{X}_k)}\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–1000	10	$\{[100 \cdot j+1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\texttt{max}(\lvert\bigcup \mathcal{G}_{P_m(\mathcal{X}_k)}\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–1000	10	$\{[100 \cdot j+1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\texttt{avg}(\lvert\bigcup \mathcal{G}_{P_m(\mathcal{X}_k)}\rvert : \mathcal{X}_k \in \mathcal{B})$	$O(\lvert\mathcal{B}\rvert)$	1–1000	10	$\{[100 \cdot j+1, 100 \cdot (j+1)]\} \cup \{[901, \infty)\}$
$\lvert\{x_i \in \mathcal{X} : x_i \in [l_m, u_m] \text{ in } P_m\}\rvert$	$O(ng)$	1–25	5	$\{[5 \cdot h+1, 5 \cdot (h+1)]\} \cup \{[21, \infty)\}$
$\lvert\{x_i \in \mathcal{X} : x_i \in [l_m, \infty) \text{ in } P_m\}\rvert +$ $\lvert\{x_i \in \mathcal{X} : x_i \in (-\infty, u_m] \text{ in } P_m\}\rvert$	$O(ng)$	1–25	5	$\{[5 \cdot h+1, 5 \cdot (h+1)]\} \cup \{[21, \infty)\}$

4 Reinforcement Learning for Polyhedra Analysis

We now describe how to instantiate reinforcement learning for approximating Polyhedra domain analysis. The instantiation consists of the following steps:

- Extracting the RL state s from the abstract program state numerically using a set of features.
- Defining actions a as the choices among the threshold, merge and split methods defined in the previous section.
- Defining a reward function r favoring both high precision and fast execution.
- Defining the feature functions $\phi(s, a)$ to enable Q-learning.

States. We consider nine features for defining a state s for RL. The features ψ_i, their extraction complexity and their typical range on our benchmarks are shown in Table 2. The first seven features capture the asymptotic complexity of the join [21] on the input polyhedra P_1 and P_2. These are the number of blocks, the distribution (using maximum, minimum and average) of their sizes, and the number of generators. The precision of the inputs is captured by considering the number of variables $x_i \in \mathcal{X}$ with finite upper and lower bound, and the number of those with only a finite upper or lower bound in both P_1 and P_2.

As shown in Table 2, each state feature ψ_i returns a natural number, however, its range can be rather large, resulting in a massive state space. To ensure scalability and generalization of learning, we use bucketing to reduce the state space size by clustering states with similar precision and expected join cost. The number n_i of buckets for each ψ_i and their definition are shown in the last two columns of Table 2. Using bucketing, the RL state s is then a 9-tuple consisting of the indices of buckets where each index indicates the bucket that ψ_i's return value falls into.

Actions. An action a is a 3-tuple (*th, r_algo, m_algo*) consisting of:

- *th* $\in \{1, 2, 3, 4\}$ depending on *threshold* $\in [5, 9]$, $[10, 14]$, $[15, 19]$, or $[20, \infty)$.
- *r_algo* $\in \{1, 2, 3\}$: the choice of a constraint removal, i.e., splitting method.
- *m_algo* $\in \{1, 2, 3\}$: the choice of merge algorithm.

All three of these have been discussed in detail in Section 3. The *threshold* values were chosen based on performance characterization on our benchmarks. With the above, we have 36 possible actions per state.

Reward. After applying the (approximated join transformer) according to action a_t in state s_t, we compute the precision of the output polyhedron $P_1 \sqcup P_2$ by first computing the smallest (often unbounded) box[1] covering $P_1 \sqcup P_2$ which has complexity $O(ng)$. We then compute the following quantities from this box:

- n_s: number of variables x_i with singleton interval, i.e., $x_i \in [l, u], l = u$.
- n_b: number of variables x_i with finite upper and lower bounds, i.e., $x_i \in [l, u], l \neq u$.
- n_{hb}: number of variables x_i with either finite upper or finite lower bounds, i.e., $x_i \in (-\infty, u]$ or $x_i \in [l, \infty)$.

Further, we measure the runtime in CPU cycles *cyc* for the approximate join transformer. The reward is then defined by

$$r(s_t, a_t, s_{t+1}) = 3 \cdot n_s + 2n_b + n_{hb} - \log_{10}(cyc). \qquad (3)$$

As the order of precision for different types of intervals is: singleton > bounded > half bounded interval, the reward function in (3) weighs their numbers by $3, 2, 1$. The reward function in (3) favors both high performance and precision. It also ensures that the precision part $(3 \cdot n_s + 2n_b + n_{hb})$ has a similar magnitude range as the performance part $(\log_{10}(cyc))$[2].

Q-function. As mentioned before, we approximate the Q-function by a linear function (1). We define binary feature functions ϕ_{ijk} for each (state, action) pair. $\phi_{ijk}(s, a) = 1$ if the tuple $s(i)$ lies in j-th bucket and action $a = a_k$

$$\phi_{ijk}(s, a) = 1 \iff s(i) = j \text{ and } a = a_k \qquad (4)$$

The Q-function is a linear combination of state action features ϕ_{ijk}

$$Q(s, a) = \sum_{i=1}^{9} \sum_{j=1}^{n_i} \sum_{k=1}^{36} \theta_{ijk} \cdot \phi_{ijk}(s, a). \qquad (5)$$

Q-learning. During the training phase, we are given a dataset of programs $\mathcal{D}$ and we use `Q-LEARN` from Algorithm 1 on each program in $\mathcal{D}$ to perform Q-learning. Q-learning is performed with input parameters instantiated as explained above and summarized in Table 3. Each episode consists of a run of Polyhedra analysis on a benchmark in $\mathcal{D}$. We run the analysis multiple times on each program in $\mathcal{D}$ and update the Q-function after each join by calling `Q-LEARN`.

A Q-function is typically learned using an ϵ-greedy policy [24] where the agent takes greedy actions by exploiting the current Q-estimates while also exploring randomly. The policy requires initial random exploration to learn good

[1] A natural measure of precision is the volume of $P_1 \sqcup P_2$. However, calculating it is very expensive and $P_1 \sqcup P_2$ is often unbounded.

[2] The log is used since the join has exponential complexity.

Table 3. Instantiation of Q-learning to Polyhedra static analysis.

RL concept	Polyhedra Analysis Instantiation
Agent	Polyhedra analysis
State $s \in \mathcal{S}$	As described in Table 2
Action $a \in \mathcal{A}$	Tuple (*th, r_algo, m_algo*)
Reward function r	Shown in (3)
Feature ϕ	Defined in (4)
Q-function	Q-function from (5)

Q-estimates that can be later exploited. This is infeasible for the Polyhedra analysis as a typical episode contains thousands of join calls. Therefore, we generate actions for Q-learning by exploiting the optimal policy for precision (which always selects the precise join) and explore performance by choosing a random approximate join: both with a probability of 0.5[3].

Formally, the action $a_t := p(s_t)$ selected in state s_t during learning is given by $a_t = (th,\ r\_algo,\ m\_algo)$ where

$$th = \begin{cases} \texttt{rand() \% 4+1} \text{ with probability } 0.5 \\ \texttt{min}(4, (\sum_{i=1}^{|\mathcal{B}|} |\mathcal{X}_k|)/5) \text{ with probability } 0.5 \end{cases}, \tag{6}$$
$$r\_algo = \texttt{rand}()\ \%\ 3 + 1, m\_algo = \texttt{rand}()\ \%\ 3 + 1.$$

Obtaining the learned policy. After learning over the dataset $\mathcal{D}$, the learned approximating join transformer in state s_t chooses an action according to (2) by selecting the maximal value over all actions. The value of $th = 1, 2, 3, 4$ is decoded as $threshold = 5, 10, 15, 20$ respectively.

5 Experimental Evaluation

We implemented our approach in the form of a C-library for Polyhedra analysis, called Poly-RL. We compare the performance and precision of Poly-RL against the state-of-the-art ELINA [1], which uses online decomposition for Polyhedra analysis without losing precision. In addition, we implemented two Polyhedra analysis approximations (baselines) based on the following heuristics:

- Poly-Fixed: uses a *fixed* strategy based on the results of Q-learning. Namely, we selected the threshold, split and merge algorithm most frequently chosen by our (adaptive) learned policy during testing.
- Poly-Init: uses an approximate join with probability 0.5 based on (6).

All Polyhedra implementations use 64-bit integers to encode rational numbers. In the case of overflow, the corresponding polyhedron is set to top.

[3] We also tried exploitation probabilities of 0.7 and 0.9, however the resulting policies had suboptimal performance during testing due to limited exploration.

Experimental setup. All our experiments including learning the parameters θ for the Q-function and the evaluation of the learned policy on unseen benchmarks were carried out on a 2.13 GHz Intel Xeon E7- 4830 Haswell CPU with 24 MB L3 cache and 256 GB memory. All Polyhedra implementations were compiled with gcc 5.4.0 using the flags `-O3 -m64 -march=native`.

Analyzer. For both learning and evaluation, we used the *crab-llvm* analyzer for C-programs, part of the larger SeaHorn [7] verification framework. The analyzer performs intra-procedural analysis of llvm-bitcode to generate Polyhedra invariants which can be used for verifying assertions using an SMT solver [11].

Benchmarks. SVCOMP [3] contains thousands of challenging benchmarks in different categories suited for different kinds of analysis. We chose the Linux Device Drivers (LD) category, known to be challenging for Polyhedra analysis [21] as to prove properties in these programs one requires Polyhedra invariants (and not say Octagon invariants which are weaker).

Training Dataset. We chose 70 large benchmarks for Q-learning. We ran each benchmark a thousand times over a period of three days to generate sample traces of Polyhedra analysis containing thousands of calls to the join transformer. We set a timeout of 5 minutes per run and discarded incomplete traces in case of a timeout. In total, we performed Q-learning over 110811 traces.

Evaluation Method. For evaluating the effectiveness of our learned policy, we then chose benchmarks based on the following criteria:

- No overfitting: the benchmark was not used for learning the policy.
- Challenging: ELINA takes ≥ 5 seconds on the benchmark.
- Fair: there is no integer overflow in the expensive functions in the benchmark. Because in the case of an overflow, the polyhedron is set to top resulting in a trivial fixpoint at no cost and thus in a speedup that is due to overflow.

Based on these criteria, we found 11 benchmarks on which we present our results. We used a timeout of 1 hour and memory limit of 100 GB for our experiments.

Inspecting the learned policy. Our learned policy chooses in the majority of cases *threshold*=20, the binary weighted constraint removal algorithm for splitting, and the merge smallest first algorithm for merging. Poly-Fixed always uses these values for defining an approximate transformer, i.e., it follows a fixed strategy. Our experimental results show that following this fixed strategy results in suboptimal performance compared to our learned policy that makes adaptive, context-sensitive decisions to improve performance.

Results. We measure the precision as a fraction of program points at which the Polyhedra invariants generated by approximate analysis are semantically the same or stronger than the ones generated by ELINA. This is a less biased and more challenging measure than the number of discharged assertions [4, 18, 19] where one can write weak assertions that even a weaker domain can prove.

Table 4 shows the number of program points[4], timings (in seconds), and the precision (in %) of Poly-RL, Poly-Fixed, and Poly-Init w.r.t. ELINA on all 11 benchmarks. In the table, the entry `TO` (`MO`) means that the analysis did not

[4] The benchmarks contain up to 50K LOC but SeaHorn encodes each basic block as one program point, thus the number of points in Table 4 is significantly reduced.

Table 4. Timings (seconds) and precision of approximations (%) w.r.t. ELINA.

Benchmark	#Program	ELINA	Poly-RL		Poly-Fixed		Poly-Init	
	Points	time	time	precision	time	precision	time	precision
`wireless_airo`	2372	877	6.6	100	6.7	100	5.2	74
`net_ppp`	680	2220	9.1	87	`TO`	34	7.7	55
`mfd_sm501`	369	1596	3.1	97	1421	97	2	64
`ideapad_laptop`	461	172	2.9	100	157	100	`MO`	41
`pata_legacy`	262	41	2.8	41	2.5	41	`MO`	27
`usb_ohci`	1520	22	2.9	100	34	100	`MO`	50
`usb_gadget`	1843	66	37	60	35	60	`TO`	40
`wireless_b43`	3226	19	13	66	`TO`	28	83	34
`lustre_llite`	211	5.7	4.9	98	5.4	98	6.1	54
`usb_cx231xx`	4752	7.3	3.9	$\approx$100	3.7	$\approx$100	3.9	94
`netfilter_ipvs`	5238	20	17	$\approx$100	9.8	$\approx$100	11	94

finish within 1 hour (exceeded the memory limit). For an incomplete analysis, we compute the precision by comparing program points for which the incomplete analysis can produce invariants.

Poly-RL vs ELINA. In Table 4, Poly-RL obtains $>$ 7x speed-up over ELINA on 6 of the 11 benchmarks with a maximum of 515x speedup for the `mfd_sm501` benchmark. It also obtains the same or stronger invariants on $\geq$ 87% of program points on 8 benchmarks. Note that Poly-RL obtains both large speedups and the same invariants at all program points on 3 benchmarks.

The widening transformer removes many constraints produced by the precise join transformer from ELINA which allows Poly-RL to obtain the same invariants as ELINA despite the loss of precision during join in most cases. Poly-RL produces large number of non-comparable fixpoints on 3 benchmarks in Table 4 due to non-monotonic join transformers.

We also tested Poly-RL on 17 benchmarks from the product lines category. ELINA did not finish within an hour on any of these benchmarks whereas Poly-RL finished within 1 second. Poly-RL had 100% precision on the subset of program points at which ELINA produces invariants. With Poly-RL, SeaHorn successfully discharged the assertions. We did not include these results in Table 4 as the precision w.r.t. ELINA cannot be completely compared.

Poly-RL vs Poly-Fixed. Poly-Fixed is never significantly more precise than Poly-RL in Table 4. Poly-Fixed is faster than Poly-RL on 4 benchmarks, however the speedups are small. Poly-Fixed is slower than ELINA on 3 benchmarks and times out on 2 of these. This is due to the overhead of the binary weight constraints removal algorithm and the exponential number of generators in the output.

Poly-RL vs Poly-Init. From (6), Poly-Init takes random actions and thus the quality of its result varies depending on the run. Table 4 shows the results on a sample run. Poly-RL is more precise than Poly-Init on all benchmarks in Table 4. Poly-Init also does not finish on 4 benchmarks.

6 Related Work

Our work can be seen as part of the general research direction on parametric program analysis [9, 18, 14, 19, 4], where one tunes the precision and cost of the analysis by adapting it to the analyzed program. The main difference is that prior approaches fix the learning parameters for a given program while our method is adaptive and can select parameters dynamically based on the abstract states encountered during analysis, yielding better cost/precision tradeoffs. Further, prior work measures precision by the number of assertions proved whereas we target the stronger notion of fixpoint equivalence.

The work of [20] and [21] improve the performance of Octagon and Polyhedra domain analysis respectively based on online decomposition without losing precision. We compared against [21] in this paper. As our results suggest, the performance of Polyhedra analysis can be significantly improved with RL. We believe that our approach can be easily extended to the Octagon domain for achieving speedups over the work of [20] as the idea of online decomposition applies to all sub-polyhedra domains [22].

Reinforcement learning based on linear function approximation of the Q-function has been applied to learn branching rules for SAT solvers in [13]. The learned policies achieve performance similar to those of the best branching rules. We believe that more powerful techniques for RL such as deep Q-networks (DQN) [17] or double Q-learning [8] can be investigated to potentially improve the quality of results produced by our approach.

7 Conclusion

Polyhedra analysis is notoriously expensive and has worst-case exponential complexity. We showed how to gain significant speedups by adaptively trading precision for performance during analysis, using an automatically learned policy. Two key insights underlie our approach. First, we identify reinforcement learning as a conceptual match to the learning problem at hand: deciding which transformers to select at each analysis step so to achieve the eventual goal of high precision and fast convergence to fixpoint. Second, we build on the concept of online decomposition, and offer an effective method to directly translate precision loss into significant speed-ups. Our work focused on polyhedra analysis for which we provide a complete implementation and evaluation. We believe the approach can be instantiated to other forms of static analysis in future work.

Acknowledgments

We would like to thank Afra Amini for her help in implementing the approximate transformers. We would also 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, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Proc. Static Analysis Symposium (SAS)*, pages 337–354, 2003.
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. K. Chae, H. Oh, K. Heo, and H. Yang. Automatically generating features for learning program analysis heuristics for c-like languages. *Proc. ACM Program. Lang.*, 1(OOPSLA):101:1–101:25, 2017.
5. G. Gange, J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey. Exploiting sparsity in difference-bound matrices. In *Proc. Static Analysis Symposium (SAS)*, pages 189–211, 2016.
6. A. Geramifard, T. J. Walsh, and S. Tellex. *A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning.* Now Publishers Inc., Hanover, MA, USA, 2013.
7. A. Gurfinkel, T. Kahsai, A. Komuravelli, and J. A. Navas. The seahorn verification framework. In *Proc. Computer Aided Verification (CAV)*, pages 343–361, 2015.
8. H. V. Hasselt. Double q-learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Neural Information Processing Systems (NIPS)*, pages 2613–2621. 2010.
9. 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.
10. J.-H. Jourdan. Sparsity preserving algorithms for octagons. *Electronic Notes in Theoretical Computer Science*, 331:57 – 70, 2017. Workshop on Numerical and Symbolic Abstract Domains (NSAD).
11. A. Komuravelli, A. Gurfinkel, and S. Chaki. Smt-based model checking for recursive programs. In *Proc. Computer Aided Verification (CAV)*, pages 17–34, 2014.
12. S. Kulkarni, R. Mangal, X. Zhang, and M. Naik. Accelerating program analyses by cross-program training. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 359–377, 2016.
13. M. G. Lagoudakis and M. L. Littman. Learning to select branching rules in the dpll procedure for satisfiability. *Electronic Notes in Discrete Mathematics*, 9:344 – 359, 2001.
14. P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 31–42, 2011.
15. A. Maréchal, D. Monniaux, and M. Périn. Scalable minimizing-operators on polyhedra via parametric linear programming. In *Proc. Static Analysis Symposium (SAS)*, pages 212–231, 2017.
16. A. Maréchal and M. Périn. Efficient elimination of redundancies in polyhedra by raytracing. In *Proc. Verification, Model Checking, and Abstract Interpretation, (VMCAI)*, pages 367–385, 2017.
17. V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015.

18. H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi. Selective context-sensitivity guided by impact pre-analysis. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 475–484, 2014.
19. H. Oh, H. Yang, and K. Yi. Learning a strategy for adapting a program analysis via bayesian optimisation. In *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 572–588, 2015.
20. G. Singh, M. Püschel, and M. Vechev. Making numerical program analysis fast. In *Proc. Programming Language Design and Implementation (PLDI)*, pages 303–313, 2015.
21. G. Singh, M. Püschel, and M. Vechev. Fast polyhedra abstract domain. In *Proc. Principles of Programming Languages (POPL)*, pages 46–59, 2017.
22. G. Singh, M. Püschel, and M. Vechev. A practical construction for decomposing numerical abstract domains. *Proc. ACM Program. Lang.*, 2(POPL):55:1–55:28, 2017.
23. M. Stoer and F. Wagner. A simple min-cut algorithm. *J. ACM*, 44(4):585–591, 1997.
24. R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning.* MIT Press, Cambridge, MA, USA, 1st edition, 1998.
25. C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3):279–292, 1992.