Resource-aware Program Analysis via Online Abstraction Coarsening

Kihong Heo
University of Pennsylvania
kheo@cis.upenn.edu

Hakjoo Oh
Korea University
hakjoo_oh@korea.ac.kr

Hongseok Yang
KAIST
hongseok.yang@kaist.ac.kr

***Abstract*—We present a new technique for developing a resource-aware program analysis. Such an analysis is aware of constraints on available physical resources, such as memory size, tracks its resource use, and adjusts its behaviors during fixpoint computation in order to meet the constraint and achieve high precision. Our resource-aware analysis adjusts behaviors by coarsening program abstraction, which usually makes the analysis consume less memory and time until completion. It does so multiple times during the analysis, under the direction of what we call a controller. The controller constantly intervenes in the fixpoint computation of the analysis and decides how much the analysis should coarsen the abstraction. We present an algorithm for learning a good controller automatically from benchmark programs. We applied our technique to a static analysis for C programs, where we control the degree of flow-sensitivity to meet a constraint on peak memory consumption. The experimental results with 18 real-world programs show that our algorithm can learn a good controller and the analysis with this controller meets the constraint and utilizes available memory effectively.**

***Index Terms*—static analysis; resource constraint; learning**

I. Introduction

When a static program analysis aims at reasoning about deep semantic properties, it typically requires a huge amount of resources such as memory [1]–[4]. The fixpoint computation of the analysis is usually neither compositional nor incremental. As a result, when the analysis runs out of memory or hits time limit, its intermediate results are simply discarded. Some researchers have suggested to make the analysis store intermediate results and then resume it with more resources [5]. There has also been work on estimating the amount of resource use, such as analysis time, before the analysis begins [6]. However, none of these existing techniques addresses the essence of the issue, which asks for a new type of program analysis that is aware of a constraint on available resources and constantly controls its resource use during fixpoint computation.

The amount of resource that a program analysis needs for a given analysis task highly depends on the degree of program abstraction employed by the analysis, such as flow-sensitivity and context-sensitivity. For example, a flow-sensitive analysis stores different abstract states for different program points, unlike its flow-insensitive counterpart that computes only one abstract state (which summarizes information for all program points). In our experiments, the flow-insensitive interval analysis on emacs-26.0.91 (503KLOC) requires 18GB of memory, but the flow-sensitive counterpart needs more than 128GB. Unfortunately, going beyond this general trend and predicting the maximum amount of resource use is hard. Syntactic characteristics of an analyzed program are not enough. For instance, while vim60 (227KLOC) is smaller than emacs-26.0.91, the flow-insensitive analysis consumes more memory (33GB more) on the former than on the latter.

In this paper, we present a resource-aware program analysis. This analysis is aware of a given resource constraint, tracks its resource use during fixpoint computation, and constantly adjusts its behavior via online abstraction coarsening. More concretely, instead of fixing a program abstraction prior to the main part of the analysis (i.e. fixpoint computation), our resource-aware analysis starts with the most expensive abstraction, and gradually coarsens the abstraction by observing the analysis behavior and its resource use. This coarsening is directed by a *controller*, which continually intervenes the fixpoint computation of the analysis, monitors resource use and other status of the analysis (such as workset size), and decides how much the analysis should coarsen abstraction. It is the quality of this controller that determines the success of our resource-aware analysis in terms of analysis precision and conformance to resource constraint. We present an algorithm for learning such a controller automatically from a given codebase. The algorithm repeatedly runs the analysis with a gradually improving controller on all training programs in the codebase, collects important parts of traces of these analysis runs, and uses them to improve the controller. We formalize our approach in a general setting, so that it becomes applicable to a wide range of static program analyses.

We instantiated our approach with a partially flow-sensitive analysis for C programs, and evaluated it with 8 large programs (130–503KLOC). This instantiated analysis adjusts the degree of flow-sensitivity online, under the direction of a controller that was learned from 10 other smaller programs (15-80KLOC). We compared this analysis with a baseline analysis [7] that does not control flow-sensitivity online but picks the degree of flow-sensitivity before the analysis begins. Both analyses control flow-sensitivity by choosing a subset of variables in the program to be analyzed flow-sensitively. When this baseline analysis was set to choose the 10% of variables for flow-sensitivity as in [7], it could not analyze 3 out of the 8 programs under the 128GB of memory budget. It turns out that to do so, the baseline should be set to choose less than 3%, a number difficult to find a priori. Meanwhile, our analysis did not require any such predefined parameter. Once the 128GB

limit was given, it adapted program abstraction appropriately, and completed all the 8 analysis tasks. The outcome was the same when we changed the budget to 64GB. Furthermore, our analysis was more precise than the baseline that was set to use the 3% of variables for flow-sensitivity. For null dereference, it reported 15% and 21% fewer alarms under 64GB and 128GB budgets, and for buffer overrun, 3% and 6% fewer alarms.

We summarize our contributions below:

- We present resource-aware program analysis, a new approach to meet a given resource requirement by constantly observing the use of physical resources and coarsening program abstraction online based on the observations.
- We propose an algorithm for learning a controller for abstraction coarsening. Our learning algorithm is inspired by batch mode reinforcement learning [8]–[10].
- We demonstrate the effectiveness of our approach with a real-world static analyzer and large C programs.

II. Overview

Our main research question is how to build a program analysis that would work well under a given resource constraint. In particular, we are interested in constraints on memory. In this section, we illustrate this question and our learning-based solution using partially flow-sensitive interval analysis.

A. Problem: Static Analysis under Resource Constraint

```
1 x = 0; y = 0; z = 1; v = input(); w = input();
2 x = z;
3 z = z + 1;
4 y = x;
5 assert(y > 0);   // Query 1 (hold)
6 assert(z > 0);   // Query 2 (hold)
7 assert(v == w); // Query 3 (may fail)
```

Consider the program above. The series of assignments until the line 4 set `x`, `y` to 1 and `z` to 2, while letting `v` and `w` keep the values selected by the user. As a result, Queries 1 and 2 hold, but Query 3 may fail.

Now consider the problem of developing a partially flow-sensitive interval analysis that can prove Queries 1 and 2 of our example, but does not keep more than 10 intervals during the analysis of the example. Here we do not regard $\top$ (meaning $[-\infty, \infty]$) as interval. More generally, the analysis takes a program P and a bound B on the maximum number of intervals that can be kept in memory during the analysis of P.[1] We want the analysis to prove as many queries in P as possible while respecting the constraint on memory. This problem is an instance of a general resource-constrained static-analysis problem to which we will return later.

The problem cannot be solved by the standard flow-sensitive or flow-insensitive analysis. The former does not meet the resource constraint. When it does its best in terms of accuracy, the flow-sensitive analysis computes the following result that associates an abstract state at each program point:

[1]For brevity, we assume that a resource bound is given as the maximum number of intervals. In our implementation, however, the analyzer takes the maximum amount of memory budget.

line	flow-sensitive abstract state
1	$\{x \mapsto [0,0], y \mapsto [0,0], z \mapsto [1,1], v \mapsto \top, w \mapsto \top\}$
2	$\{x \mapsto [1,1], y \mapsto [0,0], z \mapsto [1,1], v \mapsto \top, w \mapsto \top\}$
3	$\{x \mapsto [1,1], y \mapsto [0,0], z \mapsto [2,2], v \mapsto \top, w \mapsto \top\}$
4	$\{x \mapsto [1,1], y \mapsto [1,1], z \mapsto [2,2], v \mapsto \top, w \mapsto \top\}$

Note that the analysis uses 12 intervals, two more than allowed 10. The flow-insensitive analysis, on the other hand, is not accurate enough, although it meets the constraint. It computes a single memory state where variables `y` and `z` have the intervals $[0, +\infty]$ and $[1, +\infty]$, respectively, which are not strong enough to prove Query 1.

B. Solution: Online Abstraction Coarsening

Our solution is a resource-aware static analysis that coarsens program abstraction during analysis, adaptively based on a given resource constraint, the current resource usage and analysis states, and the properties of the program. This adaptive online coarsening is directed by two functions $\mathcal{M}$ and π.

The first function $\mathcal{M}$, called *model*, assigns a number between 0 and 1 (including boundary values) to each variable. The assigned number $\mathcal{M}(v)$ indicates how important it is to analyze the variable v flow-sensitively in terms of removing false alarms. We use $\mathcal{M}$ to rank variables. For instance, for our example program, we may have a model $\mathcal{M}$ that satisfies $\mathcal{M}(\mathtt{w}) < \mathcal{M}(\mathtt{v}) < \mathcal{M}(\mathtt{z}) < \mathcal{M}(\mathtt{y}) < \mathcal{M}(\mathtt{x})$. The model $\mathcal{M}$ induces the ranking $\mathtt{w} < \mathtt{v} < \mathtt{z} < \mathtt{y} < \mathtt{x}$, which suggests, among other things, that the top two beneficiaries of flow sensitivity are variables `x` and `y`. Note that the suggestion is good; for our goal of proving Queries 1 and 2, it suffices to treat only `x` and `y` flow-sensitively.

The second function π, called *controller*, takes information about the current status of the analysis, such as current memory usage, and decides the number of program variables that are currently treated flow-sensitively but should be treated otherwise. That is, it decides how to coarsen the current program abstraction. Note that analyzing fewer variables flow-sensitively entails keeping fewer intervals in memory, so that the analysis is more likely to meet the constraint (i.e. at most ten intervals in memory). The function π is at the heart of the analysis's effort for meeting the constraint.

Our analysis uses these functions $\mathcal{M}$ and π and analyzes our example program as follows. In the beginning, it treats all program variables flow-sensitively. But after a fixed amount of computation, the analysis pauses the usual fixpoint computation, gathers information about resource usage and abstract states, and asks the controller π how many more variables it should analyze flow-*in*sensitively. For instance, the analysis might pause at the line 2, and the controller π might instruct that 20% ($5 \times 0.2 = 1$) of variables should be analyzed flow-insensitively. Using $\mathcal{M}$, the analysis ranks variables, picks `w` at the bottom of the ranking as a variable to be analyzed flow-insensitively, and coarsens the analysis results as follows:

line	FS abstract state	FI abstract state
1	$\{x \mapsto [0,0], y \mapsto [0,0], z \mapsto [1,1], v \mapsto \top\}$	$\{w \mapsto \top\}$
2	$\{x \mapsto [1,1], y \mapsto [0,0], z \mapsto [1,1], v \mapsto \top\}$	

Note that the analysis no longer keeps the abstract value of w per program point, but it maintains just a single flow-insensitive memory state for it.

This controller-guided abstraction coarsening happens regularly. Suppose that the second one occurs when the analysis reaches line 3. The analysis result at this moment is:

line	FS abstract state	FI abstract state
1	$\{x \mapsto [0,0], y \mapsto [0,0], z \mapsto [1,1], v \mapsto \top\}$	$\{w \mapsto \top\}$
2	$\{x \mapsto [1,1], y \mapsto [0,0], z \mapsto [1,1], v \mapsto \top\}$	
3	$\{x \mapsto [1,1], y \mapsto [0,0], z \mapsto [2,2], v \mapsto \top\}$	

Note that the analysis already uses nine intervals and is close to the resource limit. The controller π may act more aggressively now and instruct the analysis to coarsen the abstraction of the half of the remaining variables, i.e., to analyze two more variables flow-*in*sensitively. If such an instruction is indeed given, the analysis chooses variables z and v based on their low $\mathcal{M}$-ranking, and changes the analysis result as follows:

line	FS abstract state	FI abstract state
1	$\{x \mapsto [0,0], y \mapsto [0,0]\}$	$\{z \mapsto [1,+\infty], v \mapsto \top, w \mapsto \top\}$
2	$\{x \mapsto [1,+\infty], y \mapsto [0,0]\}$	
3	$\{x \mapsto [1,+\infty], y \mapsto [0,0]\}$	

The variables z and v are now treated flow-*in*sensitively, and their abstract values are changed by flow-independent counterparts. Note that the abstract values of x at lines 2 and 3 are updated as well. These updates are due to the assignment $\mathtt{x} = \mathtt{z}$ at line 2, which propagates to x the change of z's abstract value from $[1,1]$ to $[1,\infty]$.

Finally, the analysis reaches the fixpoint with the abstract state $\{x \mapsto [1,+\infty], y \mapsto [1,+\infty]\}$ for the last line (i.e. lines 4–7). Note that the fixpoint is strong enough to prove Queries 1 and 2, and that the resource constraint is also met.

C. Learning a Controller π

Whether our analysis performs well or not depends on $\mathcal{M}$ and π. This naturally raises the question: how to come up with good $\mathcal{M}$ and π? Answering the question about $\mathcal{M}$ is relatively easy. There already exist techniques for automatically learning a scoring function like $\mathcal{M}$ from a codebase [7]. The situation about π is, however, not so simple. A controller π is an example of what reinforcement learning researchers call policy, and learning it from a codebase requires solving an optimization problem more difficult than those involved in learning $\mathcal{M}$. For this purpose, we have developed an algorithm by altering the batch version of a SARSA-style on-policy algorithm from reinforcement learning [8], [11].

For each program P, let $\mathbf{S}_P$ be a set of data structures that store all the runtime information of the analysis, such as current abstract states, memory usage and workset size. We call $s \in \mathbf{S}_P$ *analysis state*. Our algorithm is equipped with a feature map $\alpha_P : \mathbf{S}_P \to \mathbf{F}$ for each program P, where $\mathbf{F}$ is a subset of $\mathbb{R}^n$ for some n. The map α_P takes an analysis state $s \in \mathbf{S}_P$, which may refer to specific aspects of the program P being analyzed, and returns a real-valued vector $f \in \mathbf{F}$ that may not make such reference.

Given a collection $\mathbf{P}_{\mathrm{tr}}$ of programs, our algorithm constructs a function $\pi : \mathbf{F} \to \Pr(\mathbf{A})$, which takes information about the current status of the analysis in the feature-vector form, and returns a probability distribution on $\mathbf{A} = \{0, 1, \ldots, 99, 100\}$. Elements in $\mathbf{A}$ represent percentages of variables that have been treated flow-sensitively so far by the analysis but will be treated differently in the future. When analyzing a program P, our analysis uses the function π in conjunction with α_P. It first transforms $s \in \mathbf{S}_P$ to a feature-vector representation $\alpha_P(s)$, then computes a probability distribution $\pi(\alpha_P(s))$ over the percentages in $\mathbf{A}$, and finally returns $a \in \mathbf{A}$ with the highest probability.

In order to construct a controller, our algorithm builds a function $Q : \mathbf{F} \times \mathbf{A} \to [0,1]$, which assigns a score to every pair of feature vector f and percentage a. An ideal Q assigns a high score to (f, a) when the following property holds: when the current analysis state s is abstracted to f (i.e. $\alpha_P(s) = f$), treating $a\%$-more variables flow-insensitively gives the best analysis outcome in terms of both proving queries and meeting the resource constraint. Once the algorithm finishes building Q, it defines a controller π_Q by[2] $\pi_Q(f)(a) = \frac{Q(f,a)}{\sum_{a' \in \mathbf{A}} Q(f,a')}$, which becomes the result of the algorithm.

The function Q is built by an iterative process, which involves the invocation of the analysis on the programs in the codebase $\mathbf{P}_{\mathrm{tr}}$. Let us illustrate how this is done when our codebase is really simple: $\mathbf{P}_{\mathrm{tr}} = \{P_0\}$. Recall that when the analysis is run on P_0 under a controller π, it may invoke $\pi \circ \alpha_{P_0}$ multiple times with different analysis states.

When our algorithm starts, it draws a probability on $\mathbf{A}$ from the uniform distribution on probabilities on $\mathbf{A}$. This becomes the initial value of π. Then, the algorithm runs the analysis under the ϵ-noise version of π for some small $\epsilon \in (0,1)$ [11]. It means that when the analysis calls the controller π with the current analysis state $s \in \mathbf{S}_{P_0}$, the controller picks $\mathrm{argmax}_a(\pi(\alpha_{P_0}(s))(a))$ with probability $1-\epsilon$, or with probability ϵ, it picks one of the rest $a' \in \mathbf{A}$ uniformly. Suppose that this run follows the trajectory and score:

$$\langle s_0, a_1, s_1, a_1, s_2, a_1, s_3 \rangle : 0.7$$

The algorithm then replaces each s_i in the trajectory by $\alpha_{P_0}(s_i)$, and generates labeled data as follows:

$$\mathcal{D}_1 = \{(\langle \alpha_{P_0}(s_i), a_1 \rangle, 0.7) \mid i \in \{0,1,2\}\}.$$

Next the algorithm uses $\mathcal{D}_1$ as a training set, and calls an off-the-shelf supervised learning algorithm on it. The result of this call becomes the intial Q.

Our algorithm then repeats the steps just described, but with the controller π_Q induced by Q instead of the randomly initialized one. It runs the analysis with π_Q. Suppose that this time the analysis goes through the following path:

$$\langle s_0, a_1, s_1, a_2, s_4 \rangle : 1.0$$

The algorithm generates labeled data from this pair of trajectory and score, and adds them to its training set:

$$\mathcal{D}_2 = \mathcal{D}_1 \cup \{(\langle \alpha_{P_0}(s_0), a_1 \rangle, 1.0),\ (\langle \alpha_{P_0}(s_1), a_2 \rangle, 1.0)\}.$$

Note that the algorithm reuses the training data from the previous iteration. This accumulation is important for the

[2]For this construction to work, we should have $\sum_{a' \in \mathbf{A}} Q(f,a') > 0$. The algorithm ensures this condition.

efficiency of our learning algorithm, and is a well-known technique in reinforcement learning where $\mathcal{D}_2$ is called replay buffer [12]. As before, the algorithm invokes the off-the-shelf supervised learning algorithm on $\mathcal{D}_2$, sets Q to the result of this invocation, and defines π_Q.

This process of learning Q is repeated until the algorithm reaches a fixpoint or hits upon its iteration limit. The result of our algorithm is the controller π_Q for the last Q.

III. Controllable Program Analysis

Now we formalize our approach. In this section, we define a class of program analyses equipped and guided by controllers.

Recall our notation $\mathbf{S}_P$ for each program P. It is the set of analysis states that may arise during the analysis of P. A state $s \in \mathbf{S}_P$ contains not only the usual logical information, such as abstract states, but also the information about the runtime status of the analysis, such as memory usage and workset size. We formalize a *controllable program analysis* as a collection of tuples (I_P, T_P, B_P, R_P) indexed by a program P, where the tuples have the following types: $I_P \in \mathbf{S}_P$, $T_P : \mathbf{S}_P \times \mathbf{A} \to \mathbf{S}_P$, $B_P : \mathbf{S}_P \to \mathbb{B}$, and $R_P : \mathbf{S}_P \to [0,1]$. Intuitively, I_P is the initial analysis state, T_P models a singe execution step of the analysis, and $B_P(s)$ is a predicate for testing whether the analysis is finished at the state s or not. The predicate $B_P(s)$ is true if the analysis reaches a fixpoint or runs out of a given resource budget. The last $R_P(s)$ reports the ratio between the queries proved in s and all queries. If the analysis stops at s because of a resource shortage, then $R_P(s) = 0$.

Note that in order to transform a state s to the next s', the analysis T_P should be given an additional parameter $a \in \mathbf{A}$, called *action*. We assume that the set $\mathbf{A}$ of actions is finite. The main responsibility of a controller π is to select a good action a using information stored in the current analysis state s. A *trajectory* $\tau \in \Phi_P$ for a program P is a finite alternating sequence of states and actions:

$$\tau = \langle s_0, a_0, s_1, a_1, s_2, \ldots, a_n, s_{n+1} \rangle$$

such that $T_P(s_k, a_k) = s_{k+1}$, $B_P(s_{n+1}) = \mathsf{true}$, and $B_P(s_k) = \mathsf{false}$ for all $0 \le k \le n$. We say that τ *starts at* s_0. If s_0 is the initial state of the analysis I_P, we say that τ is *complete*. We require that for every program P, there should exist an upper bound $N > 0$ on the lengths of trajectories $\tau \in \Phi_P$. This requirement means that for each program P, regardless of how a controller chooses an action in each analysis state, the analysis of P terminates in less than $N/2$ number of steps.

IV. Controller

A controller is our main addition to a static analysis, and is in charge of selecting an appropriate action before each analysis step. Formally, it is a tuple of

1) a subset $\mathbf{F} \subseteq \mathbb{R}^n$ that consists of feature vectors;
2) a *controller* $\pi : \mathbf{F} \to \Pr(\mathbf{A})$ from feature vectors to probability distributions on $\mathbf{A}$; and
3) a *feature map* $\alpha_P : \mathbf{S}_P \to \mathbf{F}$ for each program P, which abstracts analysis states to feature vectors.

A good way to understand this definition is to go through the steps through which the analysis uses it. Assume that we are given a program P to analyze. For brevity, we further assume that the controller is used at each analysis step. In the implementation, however, the controller is triggered when some particular events, such as new memory allocation, occur. The details will appear in Section VI.

The analysis uses the controller to define an action selector $A_P : \mathbf{S}_p \to \mathbf{A}$ to be $A_P(s) = \mathrm{argmax}_a \Big((\pi \circ \alpha_P)(s)(a) \Big)$. Given an analysis state, the selector picks an action in three steps. It first converts the state to a feature vector f, then looks up the f-th entry of π and gets a probability distribution $\pi(f)$ over actions, and finally picks an action that has the highest probability in $\pi(f)$. The analysis combines this selector and the T_P function in order to define the transfer function $F_P(s) = T_P(s, A_P(s))$. Then, it invokes the transfer function F_P to carry out the analysis of P.

In our instantiation of this framework, we defined α_P using manually crafted features. The details of these features will appear in Section V. For a controller π, however, we found it automatically using an algorithm. We considered a parameterized controller π_θ, and formulated an optimization problem over θ whose objective function encourages the analysis with π_θ to perform well on a given set of benchmark programs, i.e., when running on those programs under a given resource constraint, the analysis proves as many queries as possible and avoids violating the constraint as much as possible. In the rest of this section, we explain this learning algorithm in detail.

A. Q Function

All the controllers π_θ that we consider are defined in terms of functions $Q_\theta : \mathbf{F} \times \mathbf{A} \to [0,1]$. Intuitively, $Q_\theta(f, a)$ predicts how well the analysis would perform if it takes an action a at an analysis state f. It is an estimate of the R_P value of a program P that the analysis would produce when (i) its starting analysis state is represented by the feature vector f, (ii) it takes the action a at this starting point, and (iii) after the action a, the analysis chooses best actions for maximizing the R_P value. This function corresponds to the Q function in reinforcement learning. It gives rise to the following controller π_θ:

$$\pi_\theta(f)(a) = \begin{cases} \frac{1}{|\mathbf{A}|} & \text{if } Q_\theta(f,b) = 0 \text{ for all } b \in \mathbf{A} \\ \frac{Q_\theta(f,a)}{\sum_{b \in \mathbf{A}} Q_\theta(f,b)} & \text{otherwise.} \end{cases} \tag{1}$$

B. Learning a Parameter θ

We find θ automatically using a given set of training programs in $\mathbf{P}_{\mathrm{tr}}$. We set an optimization problem for θ, which asks for finding θ that induces the best estimate Q_θ for programs in $\mathbf{P}_{\mathrm{tr}}$ in a sense. Then, we solve the problem approximately. A solution is a parameter θ of some Q_θ, which is then turned into the controller π_θ by the recipe in (1).

Let us go into the details of our algorithm for learning θ from $\mathbf{P}_{\mathrm{tr}}$. For a program P and an analysis state $s \in \mathbf{S}_P$, let $Q^*(P, s, a)$ be the real number in the interval $[0,1]$ defined by

$$Q^*(P,s,a) = \sup\big\{ \gamma^n \cdot R_P(\mathrm{last}(\tau)) \;\big|\; \tau \in \Phi_P \text{ starts with } (s,a) \text{ and has } n \text{ actions.} \big\}.$$

Algorithm 1 Controller Learning

Input: training set $\mathbf{P}_{\text{tr}}$, exploration parameter ϵ, discount factor γ
Output: controller π

1: **for** $f \in \mathbf{F}$ **do**
2: Set $\pi(f)$ to a uniform distribution on $\mathbf{A}$
3: **end for**
4: $\mathcal{D} \leftarrow \emptyset$
5: **while** timeout **do**
6: **for** $P \in \mathbf{P}_{\text{tr}}$ **do**
7: $\langle s_0, a_0, \ldots, s_n, a_n, s_{n+1} \rangle \leftarrow \mathsf{DoAnalysis}(P, \pi, \epsilon)$
where $a_i = \begin{cases} \text{random } a & \text{with prob. } \epsilon \cdot 0.99^i \\ \text{argmax}_a\, \pi(f_i)(a) & \text{otherwise} \end{cases}$
and $f_i = \alpha_P(s_i)$
8: $\mathcal{D}' \leftarrow \{\langle \langle f_j, a_j \rangle, r_j \rangle\}_{j=0}^{n}$ where $r_j = R_P(s_{n+1}) \cdot \gamma^{n-j}$
9: $\mathcal{D} \leftarrow \mathcal{D} \cup \mathcal{D}'$
10: **end for**
11: $Q \leftarrow \mathsf{DoSupervisedLearning}(\mathcal{D})$
12: Construct π from Q using (1)
13: **end while**
14: **return** π

Here $\gamma \in (0, 1)$ is a discount factor that penalizes long trajectories. Intuitively, $Q^*(P, s, a)$ represents the best score that the analysis of P can achieve by choosing actions carefully when it starts (or resumes) with taking the action a at the state s. Also, let $\mathbf{S}_P^r$ be the analysis states that may be reached:

$$\mathbf{S}_P^r = \left\{ s \in \mathbf{S}_P \;\middle|\; s \text{ appears in a complete trajectory } \tau \in \Phi_P \right\}.$$

Our algorithm tries to find Q_θ that approximates Q^* well. The following optimization problem formalizes what the algorithm aspires to achieve:

$$\theta^* = \underset{\theta}{\text{argmin}} \sum_{\substack{P \in \mathbf{P}_{\text{tr}} \\ (s,a) \in \mathbf{S}_P^r \times \mathbf{A}}} \Big(Q^*(P, s, a) - Q_\theta(\alpha_P(s), a) \Big)^2. \quad (2)$$

Intuitively, the optimization objective says that Q_θ is the best estimate of Q^* for all reachable analysis states. Unfortunately, this is an intractable optimization problem. Just evaluating the objective in the problem is difficult, because it involves Q^*, which is difficult to find, and the index set $\mathbf{S}_P^r \times \mathbf{A}$ in the summation is too large.

Our algorithm solves the problem in (2) approximately using heuristics from the reinforcement learning community. It is given in Algorithm 1. On the high level, the algorithm repeats the following two steps and improves the candidate θ. First, it runs the analysis on all the programs in the training set $\mathbf{P}_{\text{tr}}$ using a slightly randomized version of the controller π_θ. During this run, the algorithm collects information about analysis states encountered, actions taken at those states, and qualities of analysis results measured by the R_P functions. Second, the algorithm uses the collected information to improve θ. It uses the information not just from the analysis run in the first step, but also from all the prior iterations. Then, it invokes an off-the-shelf supervised learning algorithm with this information, and computes a better parameter θ.

Here is a more detailed explanation of our algorithm. It starts with a randomly-initialized controller π (lines 2). For each program P in $\mathbf{P}_{\text{tr}}$, the algorithm generates a complete trajectory by running the static analysis with a slightly-randomized variant of the controller π (line 7). At each analysis state $s \in \mathbf{S}_P$, this variant picks an action a that maximizes $\pi(\alpha_P(s))(a)$, with probability $1 - \epsilon \cdot 0.99^i$ (where i is the number of iteration of the algorithm). With the remaining $\epsilon \cdot 0.99^i$ probability, the variant draws an action from the uniform distribution on $\mathbf{A}$. This random-draw part is introduced here mainly to encourage the analysis to explore and try new actions. Note that the degree of exploration is determined by the parameter ϵ; in our experiment, we use $\epsilon = 0.1$. Our variant of π is an instance of the well-known ϵ-greedy exploration strategy in reinforcement learning [11].

Every state-action pair in the generated trajectory is stored in $\mathcal{D}$ together with a discounted R_P value of the trajectory, after its state part is abstracted by the feature map α_P (line 9). Thus, $\mathcal{D}$ ends up with containing $\langle \mathbf{x}, \mathbf{r} \rangle$ where $\mathbf{x}$ is a pair of feature vector and action, and $\mathbf{r}$ is a real number between 0 and 1. Note that the set $\mathcal{D}$ accumulates such $\langle \mathbf{x}, \mathbf{r} \rangle$ from all the iterations of the algorithm so far. This accumulation makes our algorithm more data-efficient. A similar technique called replay buffer [12] is often used by offline algorithms in reinforcement learning. We want to point out that $\{\mathbf{x} \mid \langle \mathbf{x}, \mathbf{r} \rangle \in \mathcal{D}\}$ can be understood as an approximation of the set $\mathbf{S}_P^r \times \mathbf{A}$ in (2), and the $\mathbf{r}$ part of each pair in $\mathcal{D}$ can be viewed as an estimate of the value of Q^* at (a concretization of) $\mathbf{x}$.

Once our algorithm runs the analysis for all programs in $\mathbf{P}_{\text{tr}}$, it invokes an off-the-shelf supervised learning algorithm (line 11) on $\mathcal{D}$. In this invocation, each $\langle \mathbf{x}, \mathbf{r} \rangle$ in $\mathcal{D}$ is treated as a data point $\mathbf{x}$ labeled with $\mathbf{r}$. The result of this supervised learning is then converted to the controller π by our recipe in (1), and our algorithm repeats what we have just described with this newly constructed π.

V. Application to Flow-Sensitive Analysis

We applied the method to a static analysis that checks buffer-overrun and null-dereference for C programs, where the goal is to learn a controller π that adjusts the degree of flow-sensitivity online and makes the analysis work under a given constraint on memory usage. We describe an existing partially flow-sensitive analysis [7] and explain how to learn a controller by following the recipe described in the previous sections.

The partially flow-sensitive analysis by [7] is defined with the semantic function $G_L : \mathbb{D} \to \mathbb{D}$, where $\mathbb{D} = \mathbb{C} \to \mathbb{S}$ denotes the abstract domain, i.e., maps from program points ($\mathbb{C}$) to abstract states ($\mathbb{S}$). An abstract state, $s \in \mathbb{S} = \mathbb{L} \to \mathbb{V}$, maps abstract locations in $\mathbb{L}$ to abstract values in $\mathbb{V}$. In our instance, $\mathbb{L}$ and $\mathbb{V}$ are the set of program variables and the lattice of intervals, respectively. Note that the semantic function is parameterized by a set $L \subseteq \mathbb{L}$ of abstract locations, which controls the degree of flow-sensitivity. The analysis becomes fully flow-sensitive when $L = \mathbb{L}$, and becomes completely flow-insensitive when $L = \emptyset$. The fixpoint of G_L can be computed by a standard workset algorithm.

Using the partially flow-sensitive analysis, we construct its controllable counterpart (I_P, T_P, B_P, R_P) in Section III for each program P. The analysis states in $\mathbf{S}_P$ are triples

TABLE I: Features for our learning methods.

Feature	Description
MemBudget	The inverse of memory budget
MemConsum	Current memory consumption divided by the total budget
LatticeHeight	Current lattice position divided by the lattice height
WorksetSize	Current workset size divided by the total workset size

$(L, X, Y) \in 2^{\mathbb{L}} \times \mathbb{D} \times \mathbb{RT}$. The first component L of a triple is a set of locations to be analyzed flow-sensitively, the next X is an element in the abstract domain $\mathbb{D}$, and the last Y is a table in $\mathbb{RT}$ recording various information about analysis runtime, such as memory usage and the size of the current workset. The initial state is $I_P = (\mathbb{L}, \bot_{\mathbb{D}}, Y_0)$, where the table Y_0 stores information at the start of the analysis.

We define the set $\mathbf{A}$ of actions to be natural numbers between 0 and 100. Intuitively, an action a instructs the analysis to treat $a\%$ of program variables in L flow-insensitively.

A single execution step of the analysis is modeled by T_P:

$$T_P((L, X, Y), a) = (\mathsf{reduce}(L, a), X \sqcup G_L(X), Y')$$

First, it computes a subset $\mathsf{reduce}(L, a)$ of L, which determines how much to coarsen the flow-sensitivity. It calculates the subset by ranking the locations in L using $\mathcal{M}_P$ and picking only the $(100-a)\%$ of them from the top of the ranking. Here $\mathcal{M}_P$ is a map from abstract locations $\mathbb{L}$ to $[0, 1]$, and estimates, for each $l \in \mathbb{L}$, the impact of analyzing l flow-sensitively on proving queries. We use this map constructed by [7], which is available in the open-source distribution of [7]. Second, T_P computes the next abstract domain element by joining X and $G_L(X)$ (we use widening if $\mathbb{D}$ has an infinite height). Note that computing $G_L(X)$ involves running one fixpoint iteration. Third, after this iteration is finished, T_P collects the current runtime information about the analysis and stores it in Y'.

The predicate B_P checks whether the analysis terminates or not. The analysis terminates if the abstract domain element is a fixpoint of the semantic function (i.e., $G_L(X) \sqsubseteq X$) or the given resource is exhausted. The last R_P takes an analysis state $(L, X, Y) \in \mathbf{S}_P$, examines the program P using the current abstract domain element X, and returns the ratio of proved queries over the total number of queries. In our experiments, we used buffer-overrun and null-dereference queries.

To use our framework, we need to define a feature map $\alpha_P : \mathbf{S}_P \to \mathbf{F}$ that converts analysis states to feature vectors. In this instance analysis, we use four features in Table I, which are all related to the memory usage of the analysis. Each feature denotes a real number between 0 and 1. The first MemoryBudget records information about a given memory bound. It is one divided by the total memory budget given to the analysis. Here we take the multiplicative inverse for normalization. The next MemConsum is the ratio of the current memory usage to the total memory budget expressed in fraction. The third feature is LatticeHeight, which denotes the relative height of the current abstract domain element against the estimated total height of the lattice. For finite abstract domains, we directly compute both the height of the element and the total height, and compute their ratio.

TABLE II: Characteristics of benchmark programs for training and testing. **LOC** reports lines of code before pre-processing. **Var** reports the number of variables in the program (precisely, abstract location).

Training			Testing		
Program	**LOC**	**Var**	**Program**	**LOC**	**Var**
mp3c-0.29	15K	5K	sendmail-8.13.6	129K	26K
less-382	23K	3K	redis-4.0.8	148K	63K
make-3.76.1	27K	4K	nethack-3.3.0	209K	59K
fpgatools-201212	30K	12K	git-2.12.1	238K	69K
exifprobe-2.0.1	40K	13K	vim60	226K	58K
screen-4.0.2	41K	10K	python-2.3.6	332K	42K
clif-0.93	42K	12K	R-3.4.3	376K	80K
urjtag-0.10	63K	14K	emacs-26.0.91	503K	129K
gawk-4.1.0	78K	28K			
uucp-1.07	80K	15K			

For infinite abstract domains, we use approximate heights. In our experiments with interval analysis, we approximated the heights of elements in the lattice of intervals using the method described in [6]. Then, for an abstract-domain element $X \in \mathbb{D}$ in this analysis, we summed these estimated heights of the intervals in X, and used their sum as our approximate height of X. The last feature is WorksetSize, and denotes the normalized size of the workset.

The last part needed to instantiate our framework is a controller π. We can generate it automatically by applying Algorithm 1 to a training set $\mathbf{P}_{\mathrm{tr}}$ of programs.

VI. Experimental Evaluation

We designed and carried out experiments that aim at answering the following research questions:

- **Analysis Precision:** How precise is our analysis that coarsens its program abstraction online with a learned controller compared with the existing offline approach [7]?
- **Memory Utilization:** How well do learned controllers utilize given memory budgets?
- **Learning Algorithm:** How effective is our algorithm for learning a controller? How does the controller behave?
- **Runtime Overhead:** How much runtime overhead is incurred by coarsening program abstraction during analysis?

We implemented our technique on top of Sparrow, an open-source state-of-the-art static analyzer for C programs [13] implemented in OCaml. The analyzer is (partially) context-sensitive and field-sensitive, and tracks both numeric and pointer values using the interval domain and allocation-site-based heap abstraction. We used the implementations of sparse analysis [14], partially flow-sensitive analysis [7], and a pre-trained model $\mathcal{M}$ for ranking locations [7], which are available in the open-source distribution of Sparrow.

When measuring analysis precision, we counted the numbers of buffer-overrun alarms and null-dereference alarms. Instead of invoking a controller and coarsening abstraction in every analysis step, we triggered abstraction coarsening only when the OCaml runtime allocates memory in the major heap [15]. The decision tree algorithm in the scikit-learn package [16] was our choice of the supervised learner in Algorithm 1. We used the default hyper parameters of the decision tree library. The exploration rate ϵ and the discount

TABLE III: Effectiveness of our partially flow-sensitive analysis with online abstraction coarsening. **Baseline** reports the performance of the baseline analysis with $k = 10$. **Time** reports the execution time of each analysis in minutes. **BO** and **ND** report the number of buffer-overrun alarms and that of null-dereference alarms, respectively. **Mem** reports memory consumption in gigabytes. The numbers in parentheses represent memory utilization (memory consumption divided by budget). Dashes (-) mean that the analysis runs out of memory.

	Baseline (Offline)				Online (64GB)				Online (128GB)			
Program	**Time**	**BO**	**ND**	**Mem**	**Time**	**BO**	**ND**	**Mem**	**Time**	**BO**	**ND**	**Mem**
sendmail-8.13.6	19	1,796	890	4.3	960	1,754	749	38 (60%)	623	1,743	703	44 (34%)
redis-4.0.8	43	977	728	14.8	487	953	720	35 (54%)	948	791	510	50 (39%)
nethack-3.3.0	200	1,154	237	52.3	855	1,141	237	58 (91%)	684	1,140	236	81 (63%)
git-2.12.1	62	863	879	10.8	1,923	859	873	53 (83%)	2,539	798	807	58 (45%)
vim60	-	-	-	-	780	950	812	63 (99%)	864	949	801	111 (87%)
python-2.3.6	456	185	264	51.9	331	197	274	48 (75%)	1,056	185	264	80 (63%)
R-3.4.3	-	-	-	-	257	1,605	1,510	53 (83%)	534	1,591	1,483	93 (73%)
emacs-26.0.91	-	-	-	-	515	456	344	57 (89%)	1,933	453	341	104 (81%)

factor γ in Algorithm 1 were set to 0.1 and 0.7. We learned the controller by running the algorithm for 100 iterations.

We used the 18 programs in Table II. The ten small programs ($<$ 100KLOC) were used for training a controller π, and the remaining eight large programs ($>$ 100KLOC) for evaluating the learned controller. Since the training programs are small, even the flow-sensitive analysis of them does not exhaust 64GB and 128GB memory budgets. Thus, in the training phase, we set the memory budget for each program based on information that we collected by running the partially flow-sensitive analysis [7] in the SPARROW distribution, whose degree of flow-sensitivity had to be set manually. More concretely, for each training program $P \in \mathbf{P}_{\text{tr}}$, we first ran SPARROW's partial flow-sensitive analysis four times with different degrees of flow-sensitivity: we made it track the 60, 70, 80 and 90% of variables flow-sensitively in those runs. The peak memory consumption of these analysis runs was then used as our memory budget for the program in the training phase. In the testing phase, we set the memory budget to 64GB or 128GB, and measured the performance of the learned controller. Note that we did not use SPARROW's partial flow-sensitive analysis at all here.

Using the programs in our testing set, we compared our technique against the partially flow-sensitive analysis [7] in the SPARROW distribution. As we already mentioned, this baseline analysis selects program abstraction offline, and the percentage k of variables to be treated flow-sensitively should be set manually before the analysis begins. We used $k = 10\%$ as in the original work [7]. For detailed comparison, we also used different k values with $0 \leq k < 10$.

A. Analysis Precision

We experimentally compared the precision of our analysis with online abstraction coarsening, with that of the baseline analysis with offline abstraction coarsening. We measured the precision by counting the number of buffer-overrun alarms and that of null-dereference alarms (lower is better).

The experimental results appear in Table III, with further details shown in Figure 1. For redis-4.0.8, our analysis with online abstraction coarsening reported 953 and 791 buffer-overrun alarms under 64GB and 128GB memory budgets, respectively. Meanwhile, the baseline analysis reported 977 alarms. For null-dereference alarms, our analysis reported 720 and 510 under 64GB and 128GB memory budgets, respectively, while the baseline reported 728 alarms. The baseline analysis with a higher k value may be able to report fewer alarms but it is difficult to manually pick the best k for each memory budget. For example, when analyzing vim60, even the analysis with $k > 3\%$ ran out of memory. Even worse, memory consumption is not proportional to program size; analyzing larger programs such as python-2.3.6 and R-3.4.3 may consume less memory than analyzing smaller ones. On the other hand, by coarsening program abstraction online under the guidance of the learned controller, our analysis was able to automatically meet the given memory budget and successfully analyzed those programs (vim60, R-3.4.3, and emacs-26.0.91) that made the baseline analysis run out of memory.

B. Memory Utilization

We now report what we found about the memory utilization of the learned controller. These findings are based on our measurements on the peak memory consumption of our analysis and that of the baseline analysis.

The results of our experiments are given in Figure 2. The baseline analysis with ten different flow-sensitivity settings (i.e., $0, 1, \ldots, 10$ % of variables) could not utilize the memory budget effectively. It abstracted flow-sensitivity too much when it started and under-utilized a given memory budget. Or it abstracted flow-sensitivity too little, and exceeded the budget. Meanwhile, for every test program, our analysis met a given budget and never ran out of memory. We want to make two further important points. First, when our analysis was applied to sendmail-8.13.6, redis-4.0.8 and git-2.12.1 under the 128GB budget, it used less than 64GB, but this does not mean that our analysis under-utilized its budget 128GB. In all of these cases, the analysis reported as few alarms as the one run under no abstraction coarsening at all. Second, the learned controller guided the online coarsening of our analysis such that the given memory budget was never exceeded but well utilized. When run on vim60 under 64GB and 128GB budgets, our analysis used 63GB (99%) and 111GB (87%), respectively. The experiments on other larger programs showed the similar high utilization of a given memory budget. For the programs R-3.4.3 and emacs-26.0.91, our analysis utilized 53GB (83%)

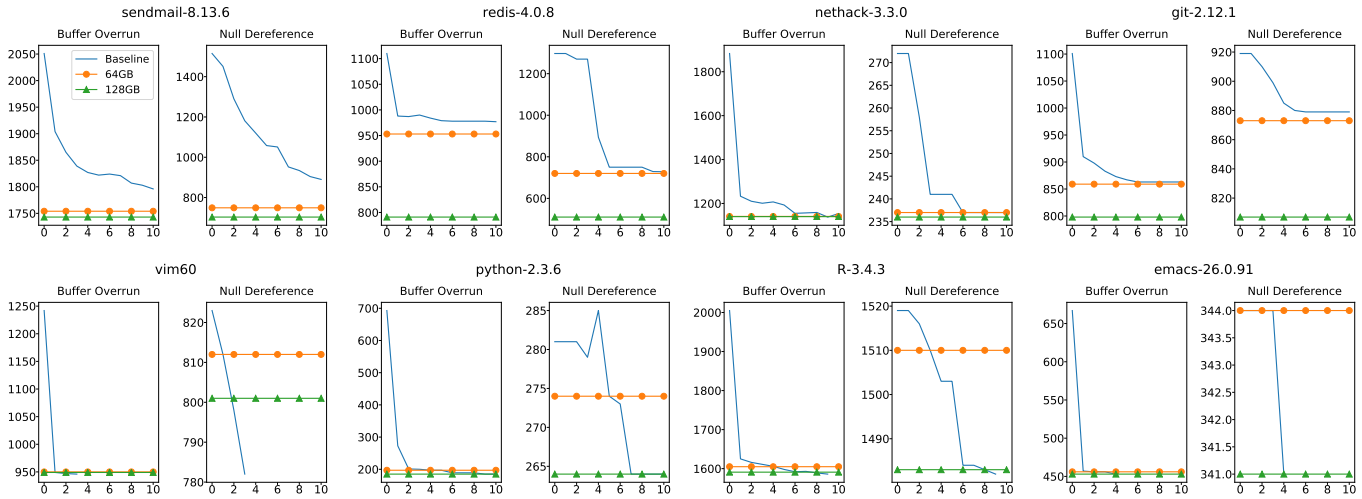

Fig. 1: Precision of our partially flow-sensitive analysis with online abstraction coarsening. The x-axis represents the percentage of variables for flow-sensitivity. The y-axis reports the number of alarms. The plain line shows the number of alarms by the baseline analysis. The line with ● and ▲ represent the numbers of alarms by our analysis under 64GB and 128GB memory budgets, respectively

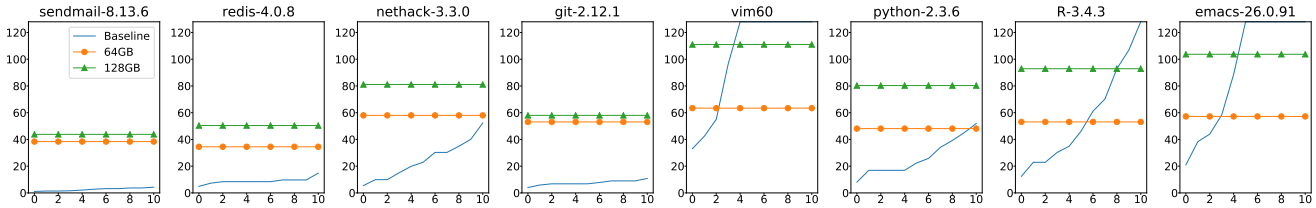

Fig. 2: Memory utilization of our partially flow-sensitive analysis with online abstraction coarsening. The x-axis is the % of variables for flow-sensitivity. The y-axis reports the peak memory consumption.

and 57GB (89%) when run under the 64GB budget, and 92GB (73%) and 104 (81%) when run under the 128GB budget.

C. Learning Algorithm

Next we report on our evaluation of Algorithm 1, which learns a controller from given training programs.

We compared our learning algorithm with the naive algorithm based on random sampling. The latter works as follows. It fixes a controller that chooses the percentage of variables to coarsen from the uniform distribution on $\{0, 1, \ldots, 100\}$. The algorithm then instantiates our analysis with this controller, analyzes all programs in the training set multiple times, and collects the trajectories of these runs. The collected trajectories are converted to a labeled dataset by the methodology described at the end of Section IV-B. Finally, the naive learning algorithm calls an off-the-shelf supervised learner with the dataset, and builds a controller from the output of the learner.

Figure 3 shows the learning curve of our algorithm for the first 100 iterations. It reports the numbers of alarms in a normalized form. The number of alarms is bounded. The upper bound is the number of alarms by the flow-insensitive analysis. The lower bound is the total number of alarms by the partially flow-sensitive analysis whose degree is manually set to 60, 70, 80, or 90 as described in Section VI. For each number of alarms, our normalization subtracts this lower bound from it, and divides the result by the difference between the upper bound and the lower bound. After this normalization, all the numbers fall in the range $[0, 1]$. The figure also shows how many alarms are raised by the analysis with a random controller used in the naive learning algorithm.

According to Figure 3, our learning algorithm improves a controller continuously with some noise (due to ϵ-random exploration), and eventually ends up with a controller that removes up to the 94% of all the removable alarms. The graph in the figure shows a sudden change in the learning curve at iteration 30. Here is what happens. At nearly every iteration until the 29th, our algorithm changes the controller such that the controller coarsens fewer variables and leads to the decrease in the number of alarms. But at the 30th iteration, the algorithm finds that, under the current controller, most of the analysis runs with training programs exhaust memory. This failure produces trajectories with very low scores, and makes the off-the-shelf supervised learning algorithm fix the issues with the controller. After around 70 iterations, the algorithm gets stabilized; the fluctuations afterwards are mainly due to the ϵ-random exploration. The figure also shows the score (52%) of the naive learning algorithm with the random controller. Furthermore, the random controller does not perform well in the test programs. Among the eight test programs, the analyzer ran out of memory for four under the 64GB budget, and for five under the 128GB budget. This indicates that the

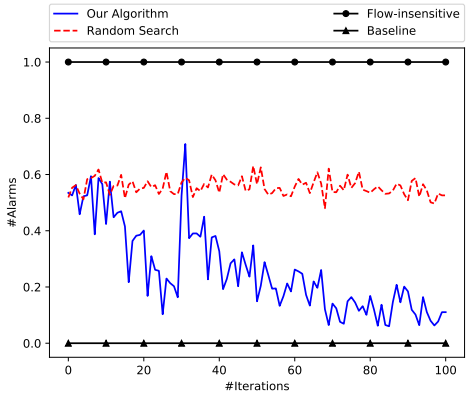

Fig. 3: Learning curve of Algorithm 1 up to 100 iterations.

random controller used by the naive algorithm does not try promising or informative trajectories often, a situation different from the one experienced by our learning algorithm.

Finally, we found that the learned policy is interpretable and its reasoning can be read off as reasonable human-readable rules. This is one of the benefits of using the decision tree algorithm. For example, the learned policy uses the following rules discovered by the algorithm:

- If the budget is enough (MemBudget $\leq 1.0^{-3}$), the memory consumption is low (MemConsum ≤ 0.52), and the workset is small (WorksetSize ≤ 0.06), coarsening a large percentage of variables ($\leq 58\%$) would result in more alarms.
- Even with a small budget MemBudget $\leq 1.5^{-3}$, if the lattice is high (LatticeHeight > 0.37) and the workset is small (WorksetSize ≤ 0.06), coarsening a small percentage of variables ($\leq 3\%$) would result in fewer alarms.

D. Runtime Overhead

We measured the runtime overhead of our approach. The overhead includes extracting features from the current analysis states, finding the most promising action using the controller, and coarsening flow-sensitivity. Overall, the abstraction coarsening process took up to 5% of the running time for all test programs. However, applying abstraction coarsening more aggressively did not always reduce the running time. For example, analyzing sendmail-8.13.6 under the 64GB memory budget took longer than the same analysis with the 128GB memory budget, even though the former is less precise. Such circumstances are not uncommon in program analysis, because a less precise analysis may explore more spurious behaviors of a target program than the precise version.

E. Limitations and Threats to Validity

Although our approach of developing a program analysis with online abstraction coarsening and learning its controller from a codebase can be applied to a wide range of program analysis problems in principle, we admit that in practice, such an application commonly requires further nontrivial engineering. What we have shown in our experiments is limited to a particular analysis problem. We list the specific aspects of this problem that may be absent in different analysis problems.

- **Type of Resource**: Our instance analysis and controller are only concerned with memory consumption. Hence, our experimental results and findings might not represent what would happen for constraints on different analysis resources such as time and hard disk usage.
- **Type of Abstraction**: Our instance analysis only controls flow-sensitivity. If it is asked to control other abstraction techniques, such as context-sensitivity and relational analysis, in addition, then we might have to alter our approach, especially, our algorithm for learning a controller, in order to get meaningful experimental results.
- **Type of Features**: To abstract analysis states, we have used a set of features carefully designed for our constraint on memory consumption. A new set of features will have to be designed for other types of analysis resources.

VII. Related Work

Our work is related or influenced by several lines of prior research in program analysis and machine learning.

A. Automatic Abstraction Finding

One high-level idea used by our work is to adapt program abstraction automatically to a given analysis task. There are several realizations of this idea, which let us automatically find good program abstraction that balances the precision and the cost of analysis. Well-known examples include counterexample-guided abstraction refinement (CEGAR) [17], [18], parametric program analysis based on pre-analysis [19], [20] or dynamic analysis [21], [22], and program analysis tuned by machine learning algorithms [2], [7], [23]–[26].

Although these techniques have been successful, they are not designed with resource constraints in mind and cannot cope with them well. For example, the recent techniques for controlling flow- and context-sensitivity [2], [7], [19], [20], [24] would abort, without producing any results, if there were no more memory available. Our aim is to address this resource-constraint problem in the context of automatic abstraction finding or adjustment. We equip a static analysis with the ability for coarsening abstraction online, and let a controller direct this coarsening, while learning a good controller automatically from a set of benchmark programs. All of these make our analysis regard resource constraints on a par with analysis precision and cost when it adapts program abstraction.

B. Resource-bounded Program Analysis

There have been a few studies on physical resources used by a program analyzer [5], [6]. The technique by [6] estimates the analysis time, one form of physical resource, before the analysis begins. However, it does not attempt to control the analysis so that it meets a constraint on resources, while such a control is the focus of our work. The bounded abstract interpretation [5] provides one way of coping with a resource constraint. It prepares the analysis for the case that all available resources are gone, by making the analysis store intermediate results. The analysis may resume from those stored results when more resources become available. Note that our goal is

different from that of this bounded abstract interpretation; we want to prevent this resource exhaustion from happening.

C. Data-driven Program Analysis

Our work is data-driven in that it learns a controller automatically from a set of benchmark programs. Such a data-driven approach has been pursued actively by program analysis researchers, to learn program-analysis heuristics [2], [7], [23], [24], [26], [27], also to predict program properties [28], [29], and sometimes to synthesize abstract semantics [30] as well.

Our work falls into the first heuristic-learning category, but addresses a more difficult learning problem than those considered in the existing work. Most of those works are concerned with analysis heuristics that are used once only at the beginning of the analysis. Thus, it is feasible to solve optimization problems related to learning directly by Bayesian optimization [7], boolean formula learning [7], [24], decision tree learning [2], and one-class SVM [23]. However in our case, we have to learn a controller that will make a series of decisions and receive a feedback for not a single but a group of decisions. The search space for learning in our case is much larger than that in the existing work. The situation is similar to the difference between supervised and reinforcement learning in the difficulties of problems tackled by them. In fact, most of the techniques used in the existing work come from supervised learning, while ours comes from reinforcement learning. Grigore et al.'s work in [27] is one of the few exceptions, where they presented a method to learn a model that guides abstraction refinement during analysis. However, their learning algorithm is restricted to Datalog, and so it cannot be applied to general non-Datalog program analyses.

D. Reinforcement Learning and Transfer Learning

Our approach can be understood as a variant of the batch-mode reinforcement learning algorithm [8]–[10]. A controller in our work corresponds to a so called policy in reinforcement learning [11], [31]. Our algorithm for learning a controller uses reinforcement-learning techniques for learning a good policy. The idea of repeatedly executing a policy and improving it based on execution trajectories [11], that of accumulating all the trajectories and using them for learning [8]–[10], [12], and the idea of approximating the ideal Q function and defining a policy based on the approximation, all of these are common techniques from reinforcement learning. However, our approach attempts to learn a controller from a collection of *small* programs, and to apply the controller to different *large* programs. Typically, the objectives of reinforcement learning algorithms are to learn a good policy for a given problem, not to learn a policy from one problem that generalizes well to other similar but different problems.

The first point that we have just made is closely related to what so called transfer learning attempts to achieve. Transfer learning aims at developing techniques for automatically adapting a learned classifier or a model from one dataset or a problem domain to a different dataset or a new problem domain. A wide variety of techniques have been developed in the contexts of text classification [32]–[34], natural language processing [35], [36], and software defect prediction [37], just to name a few. Our work can be understood as an addition to this line of research on transfer learning. In our case, we encourage our controller-learning algorithm to come up with a transferable controller by making it use a program-independent carefully-designed common feature space.

Recently, Singh et al. [26] used reinforcement learning to select a series of cost-effective abstract transformers during polyhedra analysis. We tackle a different problem of analyzing programs under resource constraints, which requires different instantiation and adaption of reinforcement learning techniques, such as our on-policy controller-learning algorithm instead of (approximate) Q-learning algorithm used by them.

VIII. Conclusion

We have presented a methodology for building a static program analysis that should work well under a resource constraint, such as a limit on peak memory consumption. Our methodology suggests to design an analysis that continually adjusts its program abstraction online (i.e. during analysis), in such a way to meet a given resource constraint and also to prove as many queries as possible. The key component behind this online adjustment is a controller that takes runtime information about the current status of the analysis, such as memory usage and workset size, and instructs the analysis about how it has to change program abstraction. The controller itself can be learned automatically from a collection of programs using our algorithm presented in the paper. We illustrated our methodology with a partially flow-sensitive analysis, a constraint on peak memory consumption, and two types of program queries, and through experiments with this instantiation, we showed the promise of our methodology.

Let us finish this paper with one rather speculative remark. A standard formalism for static program analysis does not pay enough attention on how the analysis uses its physical resources, such as memory and time. Managing such resources well is typically regarded as a low-level issue. We think that it is worth revisiting this practice, and we hope that our work encouraged a reader to do so. If we extend the scope of high-level analysis specification, include resource management as its part, and allow the part to interact with the rest of the analysis closely, then we may be able to open up new ways of improving program analysis systematically with formal tools.

Acknowledgment

We thank Mayur Naik and Kee-Eung Kim for helpful comments. Oh was supported by Samsung Research Funding & Incubation Center of Samsung Electronics under Project Number SRFC-IT1701-09. Yang was supported by the Engineering Research Center Program through the National Research Foundation of Korea (NRF) funded by the Korean Government MSIT (NRF-2018R1A5A1059921), and also by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT (2017M3C4A7068177).

REFERENCES

[1] H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi, "Design and Implementation of Sparse Global Analyses for C-like Languages," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'12)*, 2012.

[2] K. Heo, H. Oh, and H. Yang, "Learning a Variable-Clustering Strategy for Octagon from Labeled Data Generated by a Static Analysis," in *23rd International Static Analysis Symposium (SAS'16)*, 2016.

[3] M. Naik, "Chord: A Program Analysis Platform for Java," 2006.

[4] M. Bravenboer and Y. Smaragdakis, "Strictly Declarative Specification of Sophisticated Points-to Analyses," in *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09)*, 2009.

[5] M. Christakis and V. Wüstholz, "Bounded abstract interpretation," in *Static Analysis Symposium*, 2016.

[6] W. Lee, H. Oh, and K. Yi, "A progress bar for static analyzers," in *Static Analysis Symposium*, 2014.

[7] H. Oh, H. Yang, and K. Yi, "Learning a Strategy for Adapting a Program Analysis via Bayesian Optimisation," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*, 2015.

[8] D. Ernst, P. Geurts, and L. Wehenkel, "Tree-Based Batch Mode Reinforcement Learning," *The Journal of Machine Learning Research*, vol. 6, pp. 503–556, 2005.

[9] M. G. Lagoudakis and R. Parr, "Least-Squares Policy Iteration ," *The Journal of Machine Learning Research*, vol. 4, pp. 1107–1149, 2003.

[10] D. Ormoneit and S. Śen, "Kernel-Based Reinforcement Learning," *Machine Learning*, vol. 49, no. 2-3, pp. 161–178, 2002.

[11] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[12] L.-J. Lin, "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching," *Machine Learning*, vol. 8, no. 3-4, pp. 293–321, 1992.

[13] "Sparrow." [Online]. Available: https://github.com/ropas/sparrow

[14] H. Oh, K. Heo, W. Lee, W. Lee, D. Park, J. Kang, and K. Yi, "Global Sparse Analysis Framework," *ACM Transactions on Programming Languages and Systems*, vol. 36, no. 3, pp. 1–44, 2014.

[15] "Ocaml garbage collector." [Online]. Available: https://caml.inria.fr/pub/docs/manual-ocaml/libref/Gc.html

[16] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[17] T. Ball and S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," in *Proceedings of the th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'02)*, 2002.

[18] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided Abstraction Refinement for Symbolic Model Checking," *Journal of the ACM (JACM)*, vol. 50, no. 5, pp. 752–794, 2003.

[19] H. Oh, W. Lee, K. Heo, H. Yang, and K. Yi, "Selective Context-sensitivity Guided by Impact Pre-analysis," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[20] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, "Introspective Analysis: Context-sensitivity, Across the Board," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, 2014.

[21] M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv, "Abstractions from Tests," in *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'12)*, 2012.

[22] A. Gupta, R. Majumdar, and A. Rybalchenko, "From Tests to Proofs," in *Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'09)*, 2009.

[23] K. Heo, H. Oh, and K. Yi, "Machine-Learning-Guided Selectively Unsound Static Analysis," in *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*, 2017.

[24] S. Jeong, M. Jeon, S. Cha, and H. Oh, "Data-driven Context-sensitivity for Points-to Analysis," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 100:1–100:28, 2017.

[25] K. Chae, H. Oh, K. Heo, and H. Yang, "Automatically generating features for learning program analysis heuristics for C-like languages." *PACMPL*, vol. 1, no. OOPSLA, pp. 1–25, 2017.

[26] G. Singh, M. Püschel, and M. T. Vechev, "Fast Numerical Program Analysis with Reinforcement Learning," in *30th International Conference on Computer Aided Verification (CAV'18)*, 2018.

[27] R. Grigore and H. Yang, "Abstraction Refinement Guided by a Learnt Probabilistic Model," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*, 2016.

[28] V. Raychev, M. Vechev, and A. Krause, "Predicting program properties from "big code"," in *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'15)*, 2015.

[29] O. Katz, R. El-Yaniv, and E. Yahav, "Estimating types in binaries using predictive modeling," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'16)*, 2016.

[30] P. Bielik, V. Raychev, and M. T. Vechev, "Learning a static analyzer from data," in *29th International Conference on Computer Aided Verification (CAV'17)*, 2017.

[31] D. P. Bertsekas and J. Tsitsiklis, *Neuro-Dynamic Programming*. Athena Scientific, 1996.

[32] G.-R. Xue, W. Dai, Q. Yang, and Y. Yu, "Topic-bridged PLSA for cross-domain text classification," in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'08)*, 2008.

[33] R. Raina, A. Y. Ng, and D. Koller, "Constructing Informative Priors Using Transfer Learning," in *Proceedings of the Twenty-Third International Conference on Machine Learning (ICML'06)*, 2006.

[34] W. Dai, G.-R. Xue, Q. Yang, and Y. Yu, "Transferring Naive Bayes Classifiers for Text Classification," in *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence (AAAI'07)*, 2007.

[35] J. Jiang and C. Zhai, "Instance Weighting for Domain Adaptation in NLP," in *Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics (ACL'07)*, 2007.

[36] S. J. Pan, X. Ni, J.-T. Sun, Q. Yang, and Z. Chen, "Cross-domain Sentiment Classification via Spectral Feature Alignment," in *Proceedings of the 19th International Conference on World Wide Web (WWW'10)*, 2010.

[37] J. Nam, S. J. Pan, and S. Kim, "Transfer Defect Learning," in *Proceedings of 35th International Conference on Software Engineering (ICSE'13)*, 2013.