A Progress Bar for Static Analyzers *

Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi

Seoul National University

Abstract. We present a technique for devising a progress indicator of static analyzers. Progress indicator is a useful user interface that shows how close a static analysis has progressed so far to its completion. Because static analysis' progress depends on the semantic complexity, not on the code size, of the target software, devising an accurate progress-indicator is not obvious. Our technique first combines a semantic-based pre-analysis and a statistical method to approximate how a main analysis progresses in terms of lattice height of the abstract domain. Then, we use this information during the main analysis and estimate the analysis' current progress. We apply the technique to three existing analyses (interval, octagon, and pointer analyses) for C and show the technique estimates the actual analysis progress for various benchmarks.

1 Introduction

We aim to develop a progress bar for static analyzers. Realistic semantic-based static analyzers usually take a long time to analyze real-world software. For instance, Sparrow [1], our static analyzer for full C, takes more than 4 hours to analyze one million lines of C code [14]. Astrée [2] has also been reported to take over 20 hours to analyze programs of size over 500KLOC [5]. Nonetheless, such static analyzers are silent during their operation and users cannot but wait several hours without any progress information.

Estimating static analysis progress at real-time is challenging in general. Static analyzers take most of their time in fixpoint computation, but estimating the progress of fixpoint algorithms has been unknown. One challenge is that the analysis time is generally not proportional to the size of the program to analyze. For instance, Sparrow [14] takes 4 hours in analyzing one million lines but require 10 hours to analyze programs of sizes around 400KLOC. Similar observations have been made for Astrée as well: Astrée takes 1.5 hours for 70KLOC but takes 40 minutes for 120KLOC [5].

In this paper, we present an idea for estimating static analysis progress. Our basic approach is to measure the progress by calculating lattice heights of intermediate analysis results and comparing them with the height of the final analysis result. To this end, we employ a semantic-based pre-analysis and a

* This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Science, ICT & Future Planning(MSIP) / National Research Foundation of Korea(NRF) (Grant NRF-2008-0062609).

statistical regression technique. First, we use the pre-analysis to approximate the height of the fixpoint. This estimated height is then fine-tuned with the statistical method. Second, because this height progress usually does not indicate the actual progress (speed), we normalize the progress using the pre-analysis.

We show that our technique effectively estimates static analysis progress in a realistic setting. We have implemented our idea on top of Sparrow [1]. In our experiments with various open-source benchmarks, the proposed technique is found to be useful to estimate the progress of interval, octagon, and pointer analyses. The pre-analysis overheads are 3.8%, 7.3%, and 36.6% on average in interval, pointer, and octagon analysis, respectively.

Contributions This paper makes the following contributions:

- We present a technique for estimating static analysis progress. To our knowledge, our work is the first attempt to estimate static analysis progress.
- We show its applicability for numerical analyses (with intervals and octagons) and a pointer analysis on a suite of real C benchmarks.

Related Work Though progress estimation techniques have been extensively studied in other fields [12, 7, 4, 8, 10, 11], there have been no research for static analyzers. For instance, a varaiety of progress estimation techniques have been proposed for long-running software systems such as databases [7, 4, 8] and parallel data processing systems [11, 10]. Static analyzers are also a long-running software system but there are no progress estimation techniques for them. Furthermore, our method is different from existing techniques. Existing progress estimators [10, 8, 11, 4] and algorithm runtime prediction [6] are based solely on statistics or machine learning. By contrast, we propose a technique that combines a semantics-based pre-analysis with machine learning.

Outline Section 2 describes the overall approach to our progress estimation and the remaining sections fill the details. Section 3 defines a class of non-relational static analyses and Section 4 gives the details on how we develop a progress bar for these analyses. Section 5 experimentally evaluates the proposed technique. Section 6 discusses the application to relational analyses. Section 7 concludes.

2 Overall Approach to Progress Estimation

In this section, we describe the high-level idea of our progress estimation technique. In Section 4, we give details that we used in our experiments.

2.1 Static Analysis

We consider a static analysis designed by abstract interpretation. In abstract interpretation, a static analysis is specified with an abstract domain $\mathbb{D}$ and semantic function $F : \mathbb{D} \to \mathbb{D}$, where $\mathbb{D}$ is a cpo (complete partial order).The

analysis' job is to compute the following sequence until stabilized:

$$\bigsqcup_{i\in\mathbb{N}} F^i(\bot) = F^0(\bot) \sqcup F^1(\bot) \sqcup F^2(\bot) \sqcup \cdots \tag{1}$$

where $F^0(\bot) = \bot$ and $F^{i+1}(\bot) = F(F^i(\bot))$. When the chain is infinitely long, we can use a widening operator $\bigtriangledown : \mathbb{D} \times \mathbb{D} \to \mathbb{D}$ to accelerate the sequence.

2.2 Progress Estimation

We aim to develop a progress bar that proceeds at a linear rate. That is, the estimated progress directly indicates the amount of work that has been completed so far. Suppose that the sequence in (1) requires n iterations to stabilize, and assume that computing the abstract semantics $F(X)$ at each iteration takes a constant time regardless of the input X. Then, the *actual progress* of the analysis at ith iteration is defined by $\frac{i}{n}$. We aim at estimating this progress.

Basically, our method estimates the progress by calculating the lattice heights of intermediate analysis results. Suppose that we have a function $\mathsf{H} : \mathbb{D} \to \mathbb{N}$ that takes an abstract domain element $X \in \mathbb{D}$ and computes its height. The heights of domain elements need not be precisely defined, but we assume that H satisfies two conditions: 1) the height is initially zero. 2) H is monotone. The second condition is for building a progress bar that monotonically increases as the analysis makes progress.

The first job in our progress estimation is to approximate the height of the final analysis result. Let H_{final} be the height of the final analysis result, i.e., $H_{final} = \mathsf{H}(\bigsqcup_{i\in\mathbb{N}} F^i(\bot))$. In Section 4.3, we describe a method for precisely estimating H_{final} with the aid of statistical regression. This height estimation method is orthogonal to the rest part of our progress estimation technique. In this overview, let $H^\sharp_{final}$ be the estimated final height and assume, for simplicity, that $H^\sharp_{final} = H_{final}$.

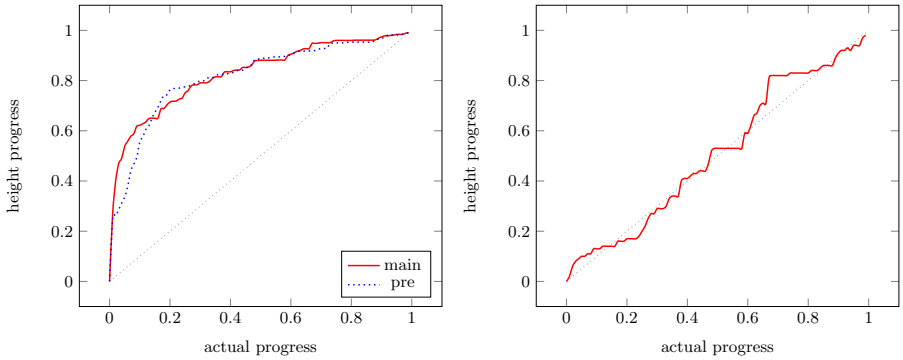
A Naive Approach Given H and $H^\sharp_{final}$, a simple progress bar could be developed as follows. At each iteration i, we first compute the height of the current analysis result:

$$H_i = \mathsf{H}(F^i(\bot)).$$

Then, we show to the users the following *height progress* of the analysis :

$$P_i = \frac{H_i}{H^\sharp_{final}}$$

Note that we can use P_i as a progress estimation: P_i is initially 0, monotonically increases as the analysis makes progress, and has 1 when the analysis is completed.



(a) original height-progress (b) normalized height-progress

Fig. 1. The height progress of a main analysis can be normalized using a pre-analysis. In this program (`sendmail-8.14.6`), the pre-analysis takes only 6.6% of the main analysis time.

Problem of the Naive Approach We noticed that this simple method for progress estimation is, however, unsatisfactory in practice. The main problem is that the height progress does not necessarily indicate the amount of computation that has been completed. For instance, the solid line in Figure 1(a) depicts how the height progress increases during our interval analysis of program `sendmail-8.14.6` (The dotted diagonal line represents the ideal progress bar). As the figure shows, the height progress rapidly increases during the early stage of the analysis and after that slowly converges. We found that this progress bar is not much useful to infer the actual progress nor to predict the remaining time of the analysis.

Our Approach We overcome this problem by normalizing the height progress using the relationship between the actual progress and the height progress. Suppose at the moment that we are given a function $\mathsf{normalize} : [0,1] \to [0,1]$ that maps the height progress into the corresponding actual progress. Indeed, $\mathsf{normalize}$ represents the inverse of the graph (the solid line) shown in Figure 1(a). Given such $\mathsf{normalize}$, the normalized height progress is defined as follows:

$$\bar{P}_i = \mathsf{normalize}(P_i) = \mathsf{normalize}\left(\frac{H_i}{H^{\sharp}_{final}}\right) \tag{2}$$

Note that, unlike the original height progress P_i, the normalized progress $\bar{P}_i$ would represent the actual progress, increasing at a linear rate. However, note also that we cannot compute $\mathsf{normalize}$ unless we run the main analysis.

The key insight of our method is that we can predict the $\mathsf{normalize}$ function by using a less precise, but cheaper pre-analysis than the main analysis. Our hypothesis is that if the pre-analysis is semantically related with the main analysis, it is likely that the pre-analysis' height-progress behavior is similar to that of

the main analysis. In this article, we show that this hypothesis is experimentally true and allows to estimate sufficiently precise normalization functions.

We first design a pre-analysis as a further abstraction of the main analysis. Let $\mathbb{D}^\sharp$ and $F^\sharp : \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ be such abstract domain and semantic function of the pre-analysis, respectively. In Section 4.2, we give the exact definition of the pre-analysis design we used. Next, we run this pre-analysis, computing the following sequence until stabilized:

$$\bigsqcup_{i\in\mathbb{N}} {F^\sharp}^i(\bot^\sharp) = {F^\sharp}^0(\bot^\sharp) \sqcup {F^\sharp}^1(\bot^\sharp) \sqcup {F^\sharp}^2(\bot^\sharp) \sqcup \cdots$$

Suppose that the pre-analysis stabilizes in m steps (m is often much smaller than n, the number of iterations for the main analysis to stabilize). Then, we collect the following data during the course of the pre-analysis:

$$(\frac{H^\sharp_0}{H^\sharp_m}, \frac{0}{m}), \quad (\frac{H^\sharp_1}{H^\sharp_m}, \frac{1}{m}), \quad \cdots, \quad (\frac{H^\sharp_i}{H^\sharp_m}, \frac{i}{m}), \quad \cdots, \quad (\frac{H^\sharp_m}{H^\sharp_m}, \frac{m}{m})$$

where $H^\sharp_i = \mathsf{H}(\gamma({F^\sharp}^i(\bot^\sharp)))$. The second component $\frac{i}{m}$ of each pair represents the actual progress of the pre-analysis at the ith iteration, and the first represents the corresponding height progress. Generalizing the data (using a linear interpolation method), we obtain a normalization function $\mathsf{normalize}^\sharp : [0,1] \to [0,1]$ for the pre-analysis.

The normalization function $\mathsf{normalize}^\sharp$ of such a pre-analysis can be a good estimation of the normalization function $\mathsf{normalize}$ of the main analysis. For instance, the dotted curve in Figure 1(a) shows the height progress of our pre-analysis (defined in Section 4.2), which has a clear resemblance with the height progress (the solid line) of the main analysis. Thanks to this similarity, it is acceptable in practice to use the normalization function $\mathsf{normalize}^\sharp$ for the pre-analysis instead of $\mathsf{normalize}$ in our progress estimation. Thus, we revise (2) as follows:

$$\bar{P}^\sharp_i = \mathsf{normalize}^\sharp(\frac{H_i}{H_{final}}) \tag{3}$$

That is, at each iteration i of the main analysis, we show the estimated normalized progress $\bar{P}^\sharp_i$ to the users. Figure 1(b) depicts $\bar{P}^\sharp_i$ for `sendmail-8.14.6` (on the assumption that $H^\sharp_{final} = H_{final}$). Note that, unlike the original progress bar (the solid line in Figure 1(a)), the normalized progress bar progresses at an almost linear rate.

3 Setting

In this section, we define a class of static analyses on top of which we develop our progress estimation technique. For presentation brevity, we consider non-relational analyses. However, our overall approach to progress estimation is also applicable to relational analyses. In Section 6, we discuss the application to a relational analysis with the octagon domain.

Static Analysis A program is a tuple $\langle \mathbb{C}, \hookrightarrow \rangle$ where $\mathbb{C}$ is a set of program points, $(\hookrightarrow) \subseteq \mathbb{C} \times \mathbb{C}$ is a relation that denotes control flows: $c \hookrightarrow c'$ indicates that c' is a next program point of c. Each program point is associated with a command: $\mathsf{cmd}(c)$ denotes the command associated with program point c.

We consider a class of static analyses whose abstract domain maps program points to abstract states:

$$\mathbb{D} = \mathbb{C} \to \mathbb{S}$$

where the abstract state is a map from abstract locations to abstract values:

$$\mathbb{S} = \mathbb{L} \to \mathbb{V}$$

We assume that the set of abstract locations is finite and $\mathbb{V}$ is a complete lattice. The abstract semantics of the program is characterized by the least fixpoint of abstract semantic function $F \in (\mathbb{C} \to \mathbb{S}) \to (\mathbb{C} \to \mathbb{S})$ defined as,

$$F(X) = \lambda c \in \mathbb{C}. f_c(\bigsqcup_{c' \hookrightarrow c} X(c')) \tag{4}$$

where $f_c \in \mathbb{S} \to \mathbb{S}$ is the transfer function for control point c.

Example 1 (Interval Analysis). Consider the following imperative language.:

$$x := e \mid \mathsf{assume}(x < n) \quad \text{where} \quad e \to n \mid x \mid e \texttt{ + } e$$

All basic commands are assignments or assume commands. An expression may be a constant integer (n), a binary operation $(e \texttt{ + } e)$, a variable expression (x). Let *Var* be the set of all program variables. We define the abstract state as a map from program variables to the lattice of intervals:

$$\mathbb{L} = \mathit{Var} \qquad \mathbb{V} = \{[l, u] \mid l, u \in \mathbb{Z} \cup \{-\infty, +\infty\} \ \wedge \ l \leq u\} \cup \{\bot\} \tag{5}$$

The transfer function $f_c : \mathbb{S} \to \mathbb{S}$ is defines as follows:

$$f_c(s) = \begin{cases} s[x \mapsto \mathcal{V}(e)(s)] & \mathsf{cmd}(c) = x := e \\ s[x \mapsto s(x) \sqcap [-\infty, n-1]] & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases}$$

where auxiliary function $\mathcal{V}(e) \in \mathbb{S} \to \mathbb{V}$ computes the abstract value for e under s:

$$\mathcal{V}(n)(s) = [n, n], \quad \mathcal{V}(e_1 \texttt{ + } e_2)(s) = \mathcal{V}(e_1)(s) \oplus \mathcal{V}(e_2)(s), \quad \mathcal{V}(x)(s) = s(x)$$

where $\oplus$ denotes the abstract binary operator for the interval domain.

Example 2 (Pointer Analysis). Consider the following imperative language:

$$x := e \mid *x := e \quad \text{where} \quad e \to x \mid \&x \mid *x$$

We design a (flow-sensitive) pointer analysis as follows. The abstract state is a map from program variables to its points-to set, i.e.,

$$\mathbb{L} = \mathit{Var} \qquad \mathbb{V} = \mathcal{P}(\mathit{Var}) \tag{6}$$

The transfer function $f_c : \mathbb{S} \to \mathbb{S}$ is defines as follows:

$$f_c(s) = \begin{cases} s[x \mapsto \mathcal{V}(e)(s)] & \mathsf{cmd}(c) = x := e \\ s[l_1 \mapsto s(l_1) \cup \mathcal{V}(e)(s)] \cdots [l_n \mapsto s(l_n) \cup \mathcal{V}(e)(s)] & \mathsf{cmd}(c) = *x := e \end{cases}$$

where $s(x) = \{l_1, \ldots, l_n\}$. For simplicity, we do not consider strong updates. In this case, $\mathcal{V}(e)(s)$ is defined as follows:

$$\mathcal{V}(x)(s) = s(x), \quad \mathcal{V}(\&x)(s) = \{x\}, \quad \mathcal{V}(*x)(s) = \bigcup_{l \in s(x)} s(l)$$

Fixpoint Computation with Widening When the domain of abstract values ($\mathbb{V}$) has infinite height, we need a widening operator $\bigtriangledown : \mathbb{V} \times \mathbb{V} \to \mathbb{V}$ to approximate the least fixpoint of F. In practice, the widening operator is applied at only headers of flow cycles [3]. Let $\mathbb{W} \subseteq \mathbb{C}$ be the set of widening points (all loop headers in the program) in the program.

Example 3. We use the following widening operator in our interval analysis:

$$[l, u] \bigtriangledown [l', u'] = [\mathit{if}\ (l' < l)\ \mathit{then} - \infty\ \mathit{else}\ l, \mathit{if}\ (u' > u)\ \mathit{then}\ + \infty\ \mathit{else}\ u].$$

4 Details on Our Progress Estimation

As described in Section 2, our progress estimation is done in two steps: (1) we first run a pre-analysis to obtain an estimated normalization function $\mathsf{normalize}^\sharp$ and an estimated final height $H^\sharp_{final}$; (2) using them, at each iteration of the main analysis, we measure the height progress, convert it to the estimated actual progress, and show it to users. However, Section 2 has left out a number of details. In this section, we give the details that we tried:

- In Section 4.1, we define our height function H.
- In Section 4.2, we describe our pre-analysis design.
- In Section 4.3, we present techniques for precise estimation of the final height.

4.1 The Height Function

We first define height function $\mathsf{H} : (\mathbb{C} \to \mathbb{S}) \to \mathbb{N}$ that takes an abstract domain element and computes its height. Since our analysis is non-relational, we assume that the height of an abstract domain element is computed point-wise as follows:

$$\mathsf{H}(X) = \sum_{c \in \mathbb{C}} \sum_{l \in \mathbb{L}} \mathsf{h}(X(c)(l)) \tag{7}$$

where $\mathsf{h} : \mathbb{V} \to \mathbb{N}$ is the height function for the abstract value domain ($\mathbb{V}$).

Example 4. For the interval domain $\mathbb{V}$ in (5), we use the following height function:

$$\mathsf{h}(\bot) = 0$$
$$\mathsf{h}([a,b]) = \begin{cases} 1 & a = b \ \wedge \ a,b \in \mathbb{Z} \\ 2 & a < b \ \wedge \ a,b \in \mathbb{Z} \\ 3 & a \in \mathbb{Z} \ \wedge \ b = +\infty \\ 3 & a = -\infty \ \wedge \ b \in \mathbb{Z} \\ 4 & a = -\infty \ \wedge \ b = +\infty \end{cases}$$

We defined this height function based on the actual workings of our interval analysis. Constant intervals (the first case) have height 1 since they are usually immediately generated from program texts. The finite intervals (the second case) are often introduced by joining two constant intervals. Intervals with one infinite bound (the third and fourth cases) are due to the widening operator. Note that our widening operator (Example 3) immediately assigns $\pm\infty$ to unstable bounds. $[-\infty, +\infty]$ is generated with the widening is applied to both bounds.

Example 5. For the pointer domain $\mathbb{V}$ in (6), we use the following height function:

$$\mathsf{h}(S) = \begin{cases} 4 & \| \ S \ \| \geq 4 \\ \| \ S \ \| & \text{otherwise} \end{cases}$$

This definition is based on our observation that, in flow-sensitive pointer analysis of C programs, most of the points-to sets have sizes less than 4.

4.2 Pre-analysis via Partial Flow-Sensitivity

A key component of our method is the pre-analysis that is used to estimate both the height-progress behavior and the maximum height of the main analysis. One natural method for further abstracting static analyses in Section 3 is to approximate the level of flow-sensitivity. In this subsection, we design a pre-analysis that was found to be useful in progress estimation.

We consider a class of pre-analyses that is partially flow-sensitive version of the main analysis. While the main analysis is fully flow-sensitive (i.e., the orders of program statements are fully respected), our pre-analysis only respects the orders of some selected program points and regards other program points flow-insensitively.

In particular, we are interested in a pre-analysis that only distinguishes program points around headers of flow cycles. In static analysis, the most interesting things usually happen in flow cycles. For instance, because of widening and join, significant changes in abstract states occur at flow cycle headers. Thus, it is reasonable to pay particular attention to height increases occurred at widening points ($\mathbb{W}$). To control the level of flow-sensitivity, we also distinguish some preceding points of widening points.

Formally, the set of distinguished program points is defined as follows. Suppose that a parameter *depth* is given, which indicates how many preceding points

of flow cycle headers are separated in our pre-analysis. Then, we decide to distinguish the following set $\Phi \subseteq \mathbb{C}$ of program points:

$$\Phi = \{c \in \mathbb{C} \mid w \in \mathbb{W} \ \wedge \ c \hookrightarrow^{depth} w\}$$

where $c \hookrightarrow^i c'$ means that c' is reachable from c within i steps of $\hookrightarrow$.

We define the pre-analysis that is flow-sensitive only for Φ as a special instance of the trace partitioning [16]. The set of partitioning indicies Δ is defined by $\Delta = \Phi \cup \{\bullet\}$, where $\bullet$ represents all the other program points not included in Φ. That is, we use the following partitioning function $\delta : \mathbb{C} \to \Delta$:

$$\delta(c) = \begin{cases} c & c \in \Phi \\ \bullet & c \notin \Phi \end{cases}$$

With δ, we define the abstract domain ($\mathbb{D}^\sharp$) and semantic function ($F^\sharp$) of the pre-analysis as follows:

$$\mathbb{C} \to \mathbb{S} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \Delta \to \mathbb{S}$$

where

$$\gamma(X) = \lambda c.\ X(\delta(c)).$$

The semantic function $F^\sharp : (\Delta \to \mathbb{S}) \to (\Delta \to \mathbb{S})$ is defined as,

$$F^\sharp(X) = \lambda i \in \Delta.\ (\bigsqcup_{c \in \delta^{-1}(i)} f_c(\bigsqcup_{c' \hookrightarrow c} X(\delta(c')))) \tag{8}$$

where $\delta^{-1}(i) = \{c \in \mathbb{C} \mid \delta(c) = i\}$.

Note that, in our pre-analysis, we can control the granularity of flow-sensitivity by adjusting the parameter *depth* $\in [0, \infty]$. A larger *depth* value yields a more precise pre-analysis. In our experiments (Section 5), we use 1 for the default value of *depth* and show that how the progress estimation quality improves with higher *depth* values. It is easy to check that our pre-analysis is sound with respect to the main analysis regardless of parameter *depth*.

4.3 Precise Estimation of the Final Height

The last component in our approach is to estimate H_{final}, the height value of the final analysis result. Note that H_{final} cannot be computed unless we actually run the main analysis. Instead, we compute $H^\sharp_{final}$, an estimation of H_{final}. We replace the H_{final} in (3) by $H^\sharp_{final}$ as follows:

$$\bar{P}^\sharp_i = \mathsf{normalize}^\sharp(\frac{H_i}{H^\sharp_{final}}) \tag{9}$$

Our goal is to compute $H^\sharp_{final}$ such that $|H^\sharp_{final} - H_{final}|$ is as smaller as possible, for which we use the pre-analysis and a statistical method. First, we compute H_{pre}, the final height of the pre-analysis result, i.e.,

$$H_{pre} = \mathsf{H}(\gamma(\mathbf{lfp}F^\sharp))$$

Next, we statistically refine H_{pre} into $H^\sharp_{final}$ such that $|H^\sharp_{final} - H_{final}|$ is likely smaller than $|H_{pre} - H_{final}|$. The job of the statistical method is to predict $\alpha = \frac{H_{final}}{H_{pre}}$ $(0 \leq \alpha \leq 1)$ for a given program. With α, $H^\sharp_{final}$ is defined as follows:

$$H^\sharp_{final} = \alpha \cdot H_{pre}$$

We assume that α is defined as a linear combination of a set of program features in Table 1. We used eight syntactic features and six semantic features. The features are selected among over 30 features by feature selection for the purpose of removing redundant or irrelevant ones for better accuracy. We used L1 based recursive feature elimination to find optimal subset of features using 254 benchmark programs.

The feature values are normalized to real numbers between 0 and 1. The Post-fixpoint features are about the post-fixpoint property. Since the pre-analysis result is a post fixpoint of the semantic function F, i.e., $\gamma(\mathbf{lfp}F^\sharp) \in \{x \in \mathbb{D} \mid x \sqsupseteq F(x)\}$, we can refine the result by iteratively applying F to the pre-analysis result. Instead of doing refinement, we designed simple indicators that show possibility of the refinement to avoid extra cost. For every traning example, a feature vector is created with a negligible overhead.

We used the ridge linear regression as the learning algorithm. The ridge linear regression algorithm is known as a quick and effective technique for numerical prediction.

Table 1. The feature vector used by linear regression to construct prediction models

Category	Feature
Inter-procedural (syntactic)	# function calls in the program
	# functions in recursive call cycles
	# undefined library function calls
Loop-related (syntactic)	the maximum loop size
	the average loop sizes
	the standard deviation of loop sizes
	the standard deviation of depths of loops
	# loopheads
Numerical analysis (semantic)	# bounded intervals in the pre-analysis result
	# unbounded intervals in the pre-analysis result
Pointer analysis (semantic)	# points-to sets of cardinality over 4 in the pre-analysis result
	# points-to sets of cardinality under 4 in the pre-analysis result
Post-fixpoint (semantic)	# program points where applying the transfer function once improves the precision
	height decrease when transfer function is applied once

In a way orthogonal to the statistical method, we further reduce $|H^\sharp_{final} - H_{final}|$ by tuning the height function. We reduce $|H^\sharp_{final} - H_{final}|$ by considering only subsets of program points and abstract locations. However, it is not the

best way to choose the smallest subsets of them when computing heights. For example, we may simply set both of them to be an empty set. Then, $|H^{\sharp}_{final} - H_{final}|$ will be zero, but both H_{final} and $H^{\sharp}_{final}$ will be also zero. Undoubtedly, that results in a useless progress bar as estimated progress is always zero in that case.

Our goal is to choose program points and abstract locations as small as possible, while maintaining the progress estimation quality. To this end, we used the following two heuristics:

- We focus only on abstract locations that contribute to increases of heights during the main analysis. Let $\mathsf{D}(c)$ an over-approximation of the set of such abstract locations at program point c:

$$\mathsf{D}(c) \supseteq \{l \in \mathbb{L} \mid \exists i \in \{1 \dots n\}.\mathsf{h}(X_i(c)(l)) - \mathsf{h}(X_{i-1}(c)(l)) > 0\}$$

  Note that since we cannot obtain the set a priori, we use an over-approximation.
- We consider only on flow cycle headers in the height calculation. This is because cycle headers are places where significant operations (join and widening) happen.

Thus, we revise the height function $\mathsf{H} : \mathbb{D} \to \mathbb{N}$ in (7) as follows:

$$\mathsf{H}(X) = \sum_{c \in \mathbb{W}} \sum_{l \in \mathsf{D}(c)} \mathsf{h}(X(c)(l)) \tag{10}$$

Because $\mathbb{W} \subseteq \mathbb{C}$ and $\forall c.\ \mathsf{D}(c) \subseteq \mathbb{L}$, the height approximation error for the new H is smaller than that of the original H in (7).

We performed 3-fold cross validation using 254 benchmarks including GNU softwares and linux packages. For interval analysis, we obtained 0.06 as a mean absolute error of α, and 0.05 for pointer analysis.

5 Experiments

In this section, we evaluate our progress estimation technique described so far. We show that our technique effectively estimates the progress of an interval domain–based static analyzer, and a pointer analyzer for C programs.

5.1 Setting

We evaluate our progress estimation technique with Sparrow [1], a realistic C static analyzer that detects memory errors such as buffer-overruns and null dereferences. Sparrow basically performs a flow-sensitive and context-insensitive analysis with the interval abstract domain. The abstract state is a map from abstract locations (including program variables, allocation-sites, and structure fields) to abstract values (including intervals, points-to sets, array and structure

blocks). Details on Sparrow's abstract semantics is available at [13]. Sparrow performs a sparse analysis [14] and the analysis has two phases: data dependency generation and fixpoint computation. Our technique aims to estimate the progress of the fixpoint computation step and, in this paper, we mean by analysis time the fixpoint computation time.

We have implemented our technique as described in Section 2 and 4. We used the height function defined in Example 4 and 5. To estimate numerical, and pointer analysis progresses, we split the Sparrow into two analyzers so that each of them may analyze only numeric or pointer-related property respectively. The pre-analysis is based on the partial flow-sensitivity defined in Section 4.2, where we set the parameter *depth* as 1 by default. That is, the pre-analysis is flow-sensitive only for flow cycle headers and their immediate preceding points.

All our experiments were performed on a machine with a 3.07 GHz Intel Core i7 processor and 24 GB of memory. For statistical estimation of the final height, we used the `scikit-learn` machine learning library [15].

5.2 Results

We tested our progress estimation techniques on 8 GNU software packages for each of analyses. Table 2 and 3 show our results.

Table 2. Progress estimation results (interval analysis). **LOC** shows the lines of code before pre-processing. **Main** reports the main analysis time. **Pre** reports the time spent by our pre-analysis. **Linearity** indicates the quality of progress estimation (best : 1). **Height-Approx.** denotes the precision of our height approximation (best : 1). **Err** denotes mean of absolute difference between **Height-Approx.** and 1 (best : 0).

		Time(s)				
Program	**LOC**	**Main**	**Pre**	**Linearity**	**Overhead**	**Height-Approx.**
bison-1.875	38841	3.66	0.91	0.73	24.86%	1.03
screen-4.0.2	44745	40.04	2.37	0.86	5.92%	0.96
lighttpd-1.4.25	56518	27.30	1.21	0.89	4.43%	0.92
a2ps-4.14	64590	32.05	11.26	0.51	35.13%	1.06
gnu-cobol-1.1	67404	413.54	99.33	0.54	24.02%	0.91
gnugo	87575	1541.35	7.35	0.89	0.48%	1.12
bash-2.05	102406	16.55	2.26	0.80	13.66%	0.93
sendmail-8.14.6	136146	1348.97	5.81	0.69	0.43%	0.93
TOTAL	686380	3423.46	130.5	**0.74**	**3.81%**	**Err : 0.07**

The **Linearity** column in Table 2, and 3 quantifies the "linearity", which we define as follows:

$$1 - \frac{\sum_{1 \le i \le n} (\frac{i}{n} - \bar{P}_i^\sharp)^2}{\sum_{1 \le i \le n} (\frac{i}{n} - \frac{n+1}{2n})^2}$$

where n is the number of iterations required for the analysis to stabilize and $\bar{P}_i^\sharp$ is the estimated progress at ith iteration of the analysis. This metric is

Table 3. Progress estimation results (pointer analysis).

Program	LOC	Time(s) Main	Time(s) Pre	Linearity	Overhead	Height-Approx.
screen-4.0.2	44745	15.89	1.56	0.90	9.82%	0.98
lighttpd	56518	11.54	0.87	0.76	7.54%	1.03
a2ps-4.14	64590	10.06	3.48	0.65	34.59%	1.04
gnu-cobol-1.1	67404	32.27	12.22	0.91	37.87%	1.03
gnugo	87575	217.77	3.88	0.64	1.78%	0.97
bash-2.05	102406	3.68	0.78	0.56	21.20%	1.04
proftpd-1.3.2	126996	74.64	11.14	0.82	14.92%	1.03
sendmail-8.14.6	136146	145.62	3.15	0.58	2.16%	0.98
TOTAL	686380	511.47	37.08	**0.73**	**7.25%**	**Err : 0.03**

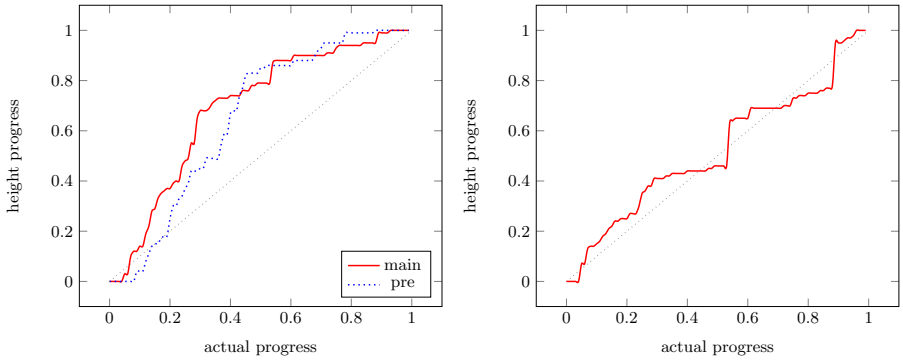
just a simple application of the coefficient of determination in statistics, i.e., R^2, which presents how well $\bar{P}^\sharp$ fits the actual progress rate $\frac{i}{n}$. The closer to 1 linearity is, the more similar to the ideal progress bar $\bar{P}^\sharp_i$ is. Figure 3 in appendix presents the resulting progress bars for each of benchmark programs providing graphical descriptions of the linearity. In particular, the progress bar proceeds almost linearly for programs of the linearity close to 0.9 (`lighttpd-1.4.25`, `gnugo-3.8` in interval analysis, `gnu-cobol-1.1`, `bash-2.05` in pointer analysis). For some programs of relatively low linearity (`gnu-cobol-1.1`, `bash-2.05` in interval analysis, `gnugo-3.8`, `proftpd-1.3.2` in pointer analysis), the progress estimation is comparatively rough but still useful.

The **Height-Approx.** column stands for the accuracy of final height approximation $\frac{H_{final}}{H^\sharp_{final}}$ where $H^\sharp_{final}$ is estimated final height via the statistical technique described in section 4.3. **Err** denotes an average of absolute errors |**Height-Approx.** -1|. To prove our statistical method avoids overfitting problem, we performed 3-fold cross validation using 254 benchmarks including GNU softwares and linux packages. For interval analysis, we obtained 0.063 **Err** with 0.007 standard deviation. For pointer analysis, 0.053 **Err** with 0.001 standard deviation. These results show our method avoids overfitting, evenly yielding precise estimations at the same time.

The **Overhead** column shows the total overhead of our method, which includes the pre-analysis running time (Section 4.2). The average performance overheads of our method are 3.8% in interval analysis, and 7.3% in pointer analysis respectively.

5.3 Discussion

Linearity vs. Overhead In our progress estimation method, we can make tradeoffs between the linearity and overhead. Table 2, 3 show our progress estimations when we use the default parameter value ($depth = 1$) in the pre-analysis. By using a higher *depth* value, we can improve the precision of the pre-analysis and hence the quality of the resulting progress estimation at the cost of extra



(a) original height-progress (b) normalized height-progress

Fig. 2. Our method is also applicable to octagon domain–based static analyses.

overhead. For two programs, the following table shows the changes in linearity and overhead when we change *depth* from 1 to 3:

Program	**Linearity change**	**Overhead change**
bash-2.05 (pointer)	0.56 → 0.70	21.2% → 37.5%
sendmail-8.14.6 (interval)	0.69 → 0.95	0.4% → 18.4%

Height Approximation Error In our experiments, we noticed that our progress estimation method is sensitive to the height approximation error ($H^{\sharp}_{final} - H_{final}$). Although we precisely estimate heights of the fixpoints, there are cases where even small error sometimes leads to unsatisfactory results. For instance, the reason why the progress for `gnu-cobol-1.1` is under-estimated is the height approximation error(0.09).

We believe enhancing the precision will be achieved by increasing training examples and relevant features.

6 Application to Relational Analyses

The overall approach of our progress estimation technique may adapt easily to relational analyses as well. In this section, we check the possibility of applying our technique to the octagon domain–based static analysis [9].

We have implemented a prototype progress estimator for the octagon analysis as follows. For pre-analysis, we used the same partial flow-sensitive abstraction described in Section 4.2 with $depth = 1$. Regarding the height function H, we also used that of the interval analysis. Note that, since an octagon domain element is a collection of intervals denoting ranges of program variables such as x and y, their sum $x + y$, and their difference $x - y$, we can use the same height function in Example 4. In this prototype implementation, we assumed that we are given heights of the final analysis results.

Figure 2 shows that our technique effectively normalizes the height progress of the octagon analysis. The solid lines in Figure 2(a) depicts the height progress of the main octagon analysis of program `wget-1.9` and the dotted line shows that of the pre-analysis. By normalizing the main analysis' progress behavior, we obtain the progress bar depicted in Figure 2(b), which is almost linear.

Figure 3 depicts the resulting progress bar for other benchmark programs, and the following table reports detailed experimental results.

		Time(s)			
Program	**LOC**	**Main**	**Pre**	**Linearity**	**Overhead**
httptunnel-3.3	6174	49.5	8.2	0.91	16.6%
combine-0.3.3	11472	478.2	16	0.89	3.4%
bc-1.06	14288	63.9	43.8	0.96	68.6%
tar-1.17	18336	977.0	73.1	0.82	7.5%
parser	18923	190.1	104.8	0.97	55.1%
wget-1.9	35018	3895.36	1823.15	0.92	46.8%
TOTAL	69193	5654.0	2069.49	**0.91**	**36.6%**

Even though we completely reused the pre-analysis design and height function for the interval analysis, the resulting progress bars are almost linear. This preliminary results suggest that our method could be applicable to relational analyses.

7 Conclusion

We have proposed a technique for estimating static analysis progress. Our technique is based on the observation that semantically related analyses would have similar progress behaviors, so that the progress of the main analysis can be estimated by a pre-analysis. We implemented our technique on top of a realistic C static analyzer and show our technique effectively estimates its progress.

Acknowledgment The authors would like to thank the anonymous referees for their comments in improving this work.

References

1. Sparrow. http://ropas.snu.ac.kr/sparrow.
2. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 196–207, 2003.
3. Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993.
4. Surajit Chaudhuri, Vivek Narasayya, and Ravishankar Ramamurthy. Estimating progress of execution for sql queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 803–814, New York, NY, USA, 2004. ACM.

5. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, December 2009.
6. Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: The state of the art. *CoRR*, abs/1211.0906, 2012.
7. Arnd Christian König, Bolin Ding, Surajit Chaudhuri, and Vivek Narasayya. A statistical approach towards robust progress estimation. *Proc. VLDB Endow.*, 5(4):382–393, December 2011.
8. Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 791–802, New York, NY, USA, 2004. ACM.
9. A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.
10. K. Morton, A. Friesen, M. Balazinska, and D. Grossman. Estimating the progress of MapReduce pipelines. In *Proc. of ICDE*, pages 681–684. IEEE, 2010.
11. Kristi Morton, Magdalena Balazinska, and Dan Grossman. Paratimer: a progress indicator for mapreduce dags. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, pages 507–518, New York, NY, USA, 2010. ACM.
12. Brad A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '85, pages 11–17, New York, NY, USA, 1985. ACM.
13. Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. Access analysis-based tight localization of abstract memories. In *VMCAI 2011: 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2011.
14. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *Proceedings of The ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2012.
15. 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*, 12:2825–2830, 2011.
16. Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans on Programming Languages and System*, 29(5):26–51, 2007.

A Progress Graphs

In this appendix, progress graphs are presented. Figure 3 presents the resulting interval, pointer, and octagon analysis progress bars respectively. Dotted diagonal line denotes the ideal progress bar.

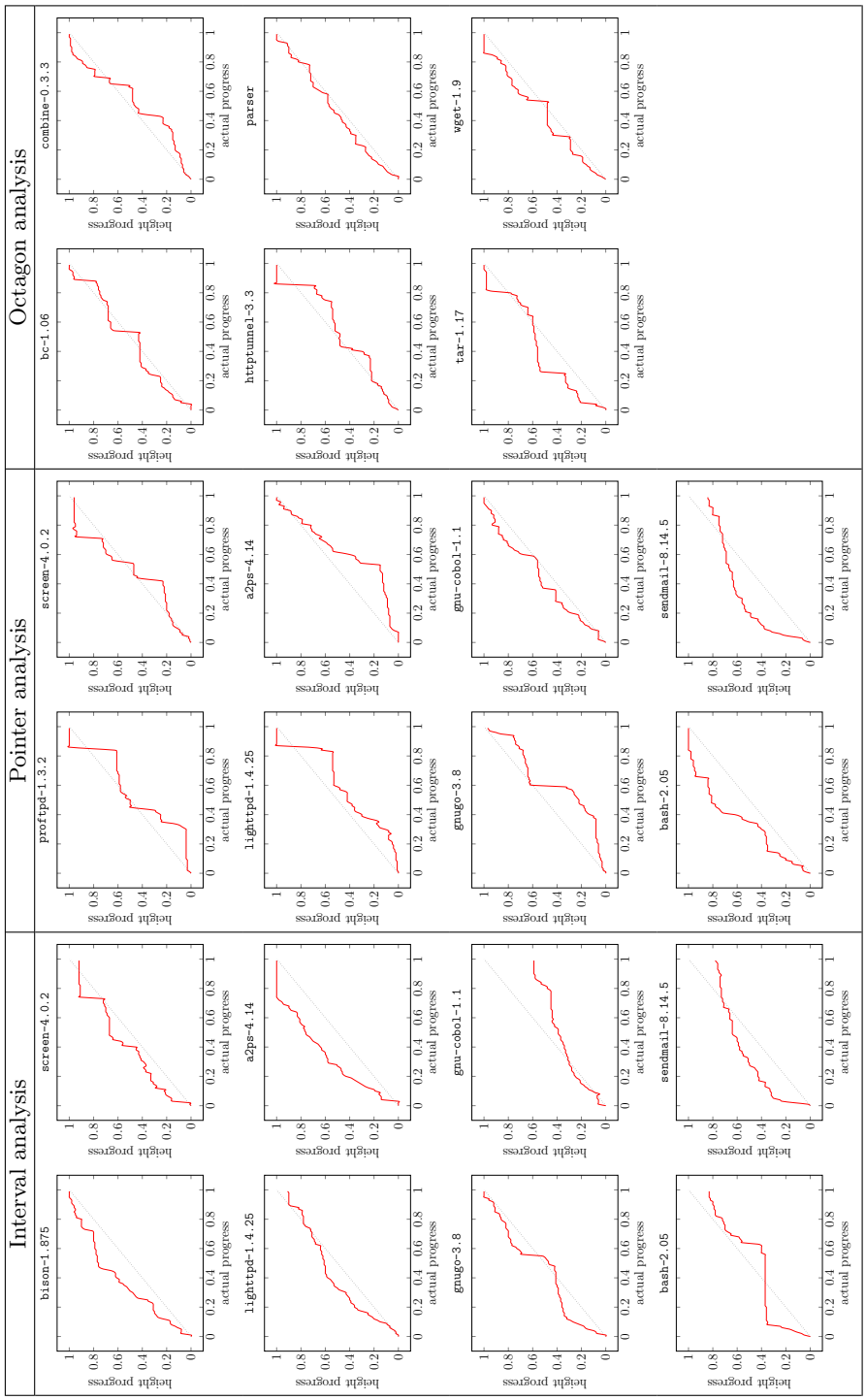


Fig. 3. Our progress estimation when $depth = 1$.