Improving the Static Analysis of Loops by Dynamic Partitioning Techniques

Matthieu Martel
CEA - Recherche Technologique
LIST-DTSI-SLA
CEA F91191 Gif-Sur-Yvette Cedex, France
e-mail : mmartel@cea.fr

April 25, 2003

Abstract — **Static analyses aim at assigning to each control point of a program an invariant property that characterizes any state of a trace corresponding to this point. The choice of the set of control points determines the states of an execution trace for which a common property must be found. In this article, we focus on sufficient conditions to substitute one control flow graph to another during an analysis. Next, we introduce a dynamic partitioning algorithm that improves the precision of the calculated invariants by deciding dynamically how to map the states of the traces to the control points, depending on the properties resulting from the first steps of the analysis. In particular, this algorithm enables to unfold the loops only if this improves the precision of the final invariant. Its correctness stems from the fact that it uses legal graph substitutions.**

I. Introduction

Static analyses aim at assigning to each control point of a program an invariant property that characterizes any state of a trace corresponding to the control point [6]. An analysis relying on a static partitioning strategy takes as input the set S of control points of the program to be analyzed and calculates the related invariant properties, without modifying S. This approach leads to unnecessary approximations in the results of many static analyses because the set of control points freezes a priori the states of an execution trace that must be merged during the analysis, that is, the states for which a common property must be found. On the converse, a static analysis based on a dynamic partitioning strategy attempts to improve the precision of the properties by deciding dynamically how to map the states of the traces to the control points.

This article firstly introduces a formal framework to prove the correctness of dynamic partitioning based static analyses and, secondly, proposes an analysis independent dynamic partitioning algorithm. The algorithm starts the analysis using a very precise control flow graph, e.g. in which any loop is unfolded n times, and coarsens it at some stages, depending on the computed abstract values. In the graph, the instances of the statements are identified by timestamps [8, 15]. The graph is then coarsened by merging consistently timestamps. As a result, this algorithm unfolds the loops just enough to find a precise property but no more (no unfolding is done that do not lead to a better analysis result).

Dynamic partitioning techniques have been proposed that enable to separately analyze the calls to a function, depending on the context, whenever more precise properties may be obtained [1, 2]. These techniques also enable to merge close abstract contexts in order to keep the analysis time acceptable. Dynamic partitioning techniques for declarative synchronous languages also have been proposed in ref. [12]. They improve the precision of an analysis by distinguishing some execution branches corresponding to different values of boolean variables. Related techniques also have been proposed in ref. [11], in which the authors propose a method to refine the set of control point for which a common property is calculated. This method uses the notion of reduced cardinal power of two domains introduced in ref. [6].

Our dynamic partitioning algorithm was first designed to unfold the loops occurring in imperative programs, when this may improve the precision of the resulting invariants. It may also improve the analysis of functions in some cases. This work was initially motivated by the design of a static analyzer for numerical codes written in assembler, in order to validate embedded critical applications. Static analyses based on complex domains like those for numerical precision [9, 13] makes the operations between abstract values particularly time and memory consuming, and new algorithms must be defined to cope with industrial-size problems. In particular, the analyses for numerical precision require that most loops be unfolded a finite but a priori unknown number of times to prove the stability of the computations (i.e. that the errors due to roundoffs decrease at each iterations). Obviously, the analyzer should avoid useless unrollings to be efficient. Next, the loops are not always precisely identified in assembler since branchings can refer to dynamic addresses. The algorithm introduced in this article is well-suited for the analysis of numerical assembly codes, yet being general. Because of the complexity of the semantics of floating-point numbers with errors [13], and for the sake of generality, the examples given in this article are high-level codes manipulating integers.

Motivating examples are introduced in Section II. Section III introduces some formal notations. Mostly, the valid control flow graphs of a program are compared using a partial order relation, and we discuss the relations between analyses based on comparable graphs. Section IV introduces the sufficient conditions that allow to substitute one control flow graph to another during the analysis. Finally, in Section V, we introduce a dynamic partitioning algorithm whose correctness relies on the criteria introduced in Section IV. This algorithm was implemented and we illustrate its behavior on the example programs of Section II.

II. Motivating Examples

With a static partitioning strategy, the control graph freezes a priori the states of an execution trace that must be merged during the analysis and this choice indeed determines the properties that can be established. For instance, let us consider the program p below, in which the numbers $|\,1\,|$, $|\,2\,|$, etc. indicate the control points.

```
|1| x = 1 ;
|2| y = 0 ;
|3| for(int i=0;i<n;i++) {
      |4| x = x * -1;
      |5| y = x + y;
    } |6|
```

An abstract interpretation of this program, using as abstract domain the domain of intervals is realized below. The values of x and y correspond to the result of the execution of Point $|\,5\,|$. x_i and y_i denote the abstract values of the variables x and y after i iterations.

$$\begin{array}{rl} x_1 = [-1,-1] & y_1 = [-1,-1] \\ x_2 \leftarrow x_1 \cup [1,1] = [-1,1] & y_2 \leftarrow y_1 \cup [0,0] = [-1,0] \\ x_3 \leftarrow x_2 \cup [-1,1] = [-1,1] & y_3 \leftarrow y_2 \cup [-2,1] = [-2,1] \\ x \leftarrow x_2 \nabla x_3 = [-1,1] & y \leftarrow y_2 \nabla y_3 =]-\infty,+\infty[\end{array}$$

At the second iteration, the analysis calculates that x is set to $[-1,-1] \times -1 = [1,1]$. Then the values of the first two iterations are joined, yielding $x_2 = [-1,1]$. Next, $[-1,1]$ is added to y at each step and, by widening the analysis terminates by stating that y belongs to $]-\infty,+\infty[$.

This analysis does not detect that $y \in [-1,0]$ and returns the upper approximation $y \in]-\infty,+\infty[$. This is because we implicitly chose to use the control-flow graph of Figure 1 a) that assigns a single control point to each statement in the body of the loop. The same analysis applied to the more precise control flow graph of Figure 1 b), in which a different control point is assigned to odd and even instances of the statements, yields the following results:

$$\begin{array}{rl} x_1 = [-1,-1] & y_1 = [-1,-1] \\ x'_1 = [1,1] & y'_1 = [0,0] \\ x_2 \leftarrow x'_1 \times -1 \cup x_1 = [-1,-1] & y_2 \leftarrow y'_1 + x_2 \cup y_1 = [-1,-1] \\ x'_2 \leftarrow x_2 \times -1 \cup x'_1 = [1,1] & y'_2 \leftarrow y_2 + x'_2 \cup y'_1 = [0,0] \\ x \leftarrow x_1 \nabla x_2 = [-1,-1] & y \leftarrow y_1 \nabla y_2 = [-1,-1] \\ x' \leftarrow x'_1 \nabla x'_2 = [1,1] & y' \leftarrow y'_1 \nabla y'_2 = [0,0] \end{array}$$

The results of this second analysis state that $y \in [-1,-1]$ or $y \in [0,0]$ depending on the parity of the iterations. Let us remark that, after the loop, the odd and even control points could be collapsed and the analysis of the rest of the code could be made assuming that, at point $|\,6\,|$, $x \in [-1,1]$ and $y \in [-1,0]$. Let us also remark that the control flow graph used for the second analysis corresponds to the usual one of the program p in which the body of the loop is unfolded once. For example, static yet parameterizable loop unfolding strategies,

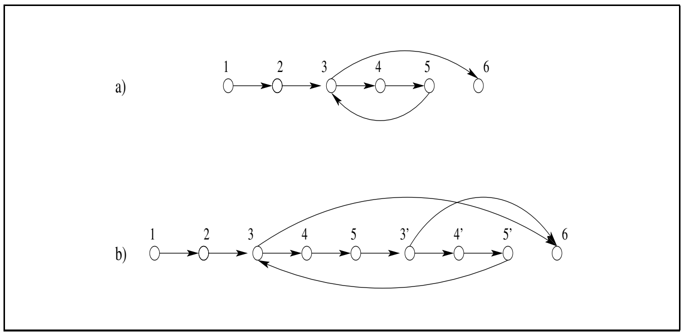

Figure 1: Two possible control-flow graph for the program of Section II.

similar to the one used in this example, are implementated in Fluctuat, a tool that analyzes the precision of floating-point calculations made in C programs [10]. For this example, the algorithm developed in Section V automatically detects that $y \in [-1,-1]$ or $y \in [0,0]$ depending on the parity of the iterations. In Section V, we also precisely show how our algorithm works on this example. Finally, let us also remark that, for this example, the results $y \in [-1,-1]$ or $y \in [0,0]$ can be obtained using the cardinal power of two domains [6], as described in ref. [11].

Our second example corresponds to the programs A, B and C below.

```
Program A;
if (i<=10) {
  j=0;
  while (i>-100) {
    i = i-1;
    j = j+i;
} }

Program B;
float x=1.0;
while (x>epsilon) {
  x=x*0.618;
}
```

```
Program C;
k=0;
while (k<10) {
  i=T[k];
  j=0; k=k+1;
  if (i<=k) {
    while (i>-100) {
      i = i-1;
      j = j+i;
} } }
```

Program A mimics what frequently happens when analyzing the numerical precision of a stable calculation, using the abstract semantics of ref. [9] and ref. [13]. In Program A, j increases while i is possibly positive and next decreases at each iteration, just like the error attached to the float variable x in Program B increases for a few iterations and next decreases, when x is small enough. In the loop of Program A, j$\in]-\infty,45]$. However the successive first values of this variable are bound by 9, 17, 24, etc. Hence, a static analyzer can output j$\in]-\infty,+\infty[$ or j$\in]-\infty,45]$, depending on how many iterations are unrolled before widening. For this example, the algorithm proposed in Section V unrolls the loop the minimal number of times required to determine that j$\in]-\infty,45]$. The same result is obtained if the loop of Program A is the inner loop of a nest, as in Program C.

III. Formalization

Let $p \in \mathcal{L}$ be a program written in a procedural language $\mathcal{L}$. $\mathcal{L}$ is equipped with a small step operational semantics whose transitions are denoted $s_1 \to s_2$ and each state s_k is a tuple $(\ell_k, \tau_k, \sigma_k)$ where:

- ℓ_k $\in$Lab is a unique label attached to each syntactic control point, i.e. node of the syntactic tree of the program,
- τ_k : Lab $\to \mathbb{N}$ is a timestamp mapping any control point ℓ $\in$Lab to an integer i. It indicates how many times any control point ℓ has been executed, before reaching the current state. Θ denotes the set of timestamps,
- σ_k is the value of the environment in the current state.

τ_k enables to count the instances of ℓ in a sequence of reductions. For example, if ℓ occurs in the inner block of a loop nest, the timestamp enables to determine how many instances of each loop has been executed. Timestamps are used in alias analyses to uniquely identify the objects created at a given point [8, 15]. The small-step operational semantics of a simple imperative language, in which timestamps equivalent to ours are associated to the states in given in ref. [15]. We order the set Θ of timestamps by the relation $\sqsubseteq_\Theta$:

$$\tau_1 \sqsubseteq_\Theta \tau_2 \iff \forall \ell \in \text{Lab},\ \tau_1(\ell) \leq \tau_2(\ell) \tag{1}$$

Let $G = (V, E)$ be a control-flow graph for the program p which vertices $v \in V$ are pairs (L, T) with L $\subseteq$Lab and $T \subseteq \Theta$. Intuitively, L, and T are used to merge some states $(\ell_k, \tau_k, \sigma_k)$ of an execution of p, for example the instances of a statement executed in some iterations of a loop or in some procedure calls. We use the following notations related to the pairs (L, T).

- $(\ell, \tau) \in_\times (L, T) \iff \ell \in L$ and $\tau \in T$
- $(L, T) \subseteq_\times (L', T') \iff \forall (\ell, \tau) \in_\times (L, T) : (\ell, \tau) \in_\times (L', T')$

By construction, we assume that $G = (V, E)$ is sound for p, i.e. we assume that for any initial environment σ, the execution of p is correctly abstracted in G, as stated by the following definition, where λ_0 denotes the label attached to the entry point of p and τ_0 the initial timestamp such that $\forall \ell, \tau_0(\ell) = 0$.

Definition 1 *A graph $G = (V, E)$ is sound for p iff for all environment σ, if $(\ell_0, \tau_0, \sigma) \to^k (\ell_k, \tau_k, \sigma_k)$ and $(\ell_k, \tau_k, \sigma_k) \to (\ell_{k+1}, \tau_{k+1}, \sigma_{k+1})$ then there exists an edge $e = ((L, T), (L', T'))$ in E such that $(\ell_k, \tau_k) \in_\times (L, T)$ and $(\ell_{k+1}, \tau_{k+1}) \in_\times (L', T')$.*

We introduce now the abstract semantics based on the control flow graph G. Let $\mathcal{D}$ be the domain of the values of $\mathcal{L}$ and $\mathcal{D}^\sharp$ an abstraction of $\mathcal{D}$, i.e. there exists a Galois connection $(\wp(\mathcal{D}), \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathcal{D}^\sharp, \sqsubseteq_{\mathcal{D}^\sharp})$. $\wp(X)$ dentes the powerset of X. An abstract environment $\sigma^\sharp$ is defined with respect to a set $\iota = \{\sigma_1, \ldots, \sigma_n\}$ of concrete initial environments by $\sigma^\sharp(\mathrm{x}) = \alpha(\{\sigma_1, \ldots, \sigma_n\})$. The abstract interpretation of p based on G calculates :

$$[\![p]\!]^\sharp_G\ :\ \text{Env}^\sharp \to (V \to \text{Env}^\sharp) \tag{2}$$

The abstract interpretation of p in the abstract environment $\sigma^\sharp$ computes a function $[\![p]\!]^\sharp_G\, \sigma^\sharp$ which maps any control point $v = (L, T) \in V$ of G to an abstract environment $\sigma^\sharp_v$. If, for some $v \in V$ with $v = (L, T)$, $([\![p]\!]^\sharp_G\ \sigma^\sharp)(v) = \sigma^\sharp_v$ then $\sigma^\sharp_v$ abstracts any environment σ_k such that $(\ell_0, \tau_0, \sigma) \to^k (\ell_k, \tau_k, \sigma_k)$, $\sigma \in \gamma(\sigma^\sharp)$ and $(\ell_k, \tau_k) \in_\times (L, T)$.

Obviously, the most a control graph G distinguishes states of the execution, the most the analysis based on G is precise. From a formal point of view, we define a partial order on the set of control flow graphs and we show that an analysis based on a less precise graph is an abstraction of the analysis based on a more precise one. We partially order the set of sound control flow graphs for a program p as follows.

Definition 2 *Let G_1 and G_2 be two sound control flow graphs for p. The relation G_2 is coarser than G_1, denoted $G_1 \prec G_2$ is defined by*

$$\begin{gathered} G_1 \prec G_2 \\ \Leftrightarrow \\ \forall v_1 = (L_1, T_1) \in G_1,\ \exists v_2 = (L_2, T_2) \in G_2\ : \\ (L_1, T_1) \subseteq_\times (L_2, T_2) \end{gathered}$$

Intuitively, the instances of the statement labeled by the control point ℓ, are partitioned in a control flow graph and each vertex (L, T) collapses the states $(\ell_k, \tau_k, \sigma_k)$ of an execution such that $(\ell_k, \tau_k) \in_\times (L, T)$. $G_1 \prec G_2$ if the partition used for G_2 is coarser than the one used for G_1 or, differently stated, if some vertices of the finest graph have been collapsed in the coarser one. In addition, let us remark that the vertices (L, T) of G do not need to define a strict partition of the execution states. Instead, they may correspond to a covering, i.e. for each state $(\ell_k, \tau_k, \sigma_k)$ such that $(\ell_0, \tau_0, \sigma_0) \to^k (\ell_k, \tau_k, \sigma_k)$, there exists at least one node (L, T) in G such that $\ell_k \in L$ and $\tau_k \in T$. Indeed, the algorithm introduced in Section V uses coverings of the set of control points that are not strict partitions.

The partial order $\prec$ enables us to relate the semantics based on comparable graphs. Indeed, for two comparable control flow graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, with $G_1 \prec G_2$, the abstract semantics $[\![p]\!]^\sharp_{G_2} \sigma^\sharp$ is an abstraction of the abstract semantics $[\![p]\!]^\sharp_{G_1} \sigma^\sharp$ based on G_1.

$f_1 = ([\![p]\!]^\sharp_{G_1}\ \sigma^\sharp)$ and $f_2 = ([\![p]\!]^\sharp_{G_2}\ \sigma^\sharp)$ are functions whose domains are $V_1 \to \text{Env}^\sharp$ and $V_2 \times \text{Env}^\sharp$ respectively. The set $V \to \text{Env}^\sharp$ is ordered by $f \sqsubseteq_V f' \Leftrightarrow \forall v \in V,\ f(v) \sqsubseteq_{\mathcal{D}^\sharp} f'(v)$. If $G_1 \prec G_2$ then we have the Galois connection:

$$\left(\wp(V_1 \to \text{Env}^\sharp), \subseteq\right) \underset{\alpha_{G_1,G_2}}{\overset{\gamma_{G_2,G_1}}{\leftrightarrows}} \left(V_2 \to \text{Env}^\sharp, \sqsubseteq_{V_2}\right) \tag{3}$$

If $F = \{f_i, 1 \leq i \leq n\}$ is a set of analysis of p based on G_1, $f_i : V_1 \to \text{Env}^\sharp$, then the abstraction $\alpha_{G_1,G_2}(F)$ is a new function f whose domain is $V_2 \to \text{Env}^\sharp$. f maps any node $v \in V_2$ to $\bigvee_{v' \in V_1, v' \subseteq_\times v} f_1(v')$. The abstraction consists of assigning to a vertex in the coarser graph the supremum of the abstract values which were distinguished in the finer graph.

$$\alpha_{G_1,G_2}(F) =$$

$$(L_2, T_2) \mapsto \bigvee_{\substack{f \in F \\ (L_1,T_1) \in V_1 \\ (L_1,T_1) \subseteq_\times (L_2,T_2)}} f(L_1, T_1) \tag{4}$$

The concretization assigns to the vertices of the finer graph any value greater or equal to the one of the related coarser state in the less precise semantics. So, it is a set of functions $f : V_1 \to \text{Env}^\sharp$ defined by:

$$\gamma_{G_2,G_1}(f_2) =$$
$$\left\{ \begin{array}{lllll} f & : & V_1 & \to & \text{Env}^\sharp \\ & & (L_1,T_1) & \mapsto & \sigma'^\sharp \sqsupseteq_E \bigvee_{\substack{(L_2,T_2) \in V_2 \\ (L_1,T_1) \subseteq_\times (L_2,T_2)}} [\![p]\!]^\sharp_{G_2}\, \sigma^\sharp(L_2,T_2) \end{array} \right\} \tag{5}$$

An interesting property is that, for any chain of comparable control-flow graphs, we have a corresponding chain of comparable semantics.

Proposition 3 *Let $G_0 \prec G_1 \prec \ldots \prec G_n$ be a chain of sound comparable graphs for p. Then the analyses based on $G_0, G_1, \ldots, G$ also are comparable:*

$$\begin{array}{ccccccc} G_0 & \prec & G_1 & \prec & \ldots & \prec & G \\ | & & | & & | & & | \\ [\![p]\!]^\sharp_{G_0}\,\sigma^\sharp & \leftrightarrows & [\![p]\!]^\sharp_{G_1}\,\sigma^\sharp & \leftrightarrows & \ldots & \leftrightarrows & [\![p]\!]^\sharp_{G}\,\sigma^\sharp \end{array} \tag{6}$$

The finest graph for a program p maps any state of an execution to a control point (fully unfolding the loops and distinguishing all the procedure calls) and the coarsest graph collapses all the states of the traces into a single control point.

IV. Static Versus Dynamic Partitioning

As illustrated by Equation (6), many partitions of the states of a program at some stage of an execution can be chosen to build the control-flow graph used by a static analysis. Classical choices include simple analyses in which a control-flow graph assigns one vertex per static control point, analyses that unfold m times the body of the loops and polyvariant analyses. Obviously, an analysis may use a mix of the above partitioning strategies. However, as stated in Section I, the choice of a control-flow graph freezes how the states occurring during an execution are merged to compute a common property. Any choice made a priori corresponds to a static partitioning strategy of the control point that cannot be appropriated for all programs and for any execution of a program.

A dynamic partitioning strategy consists of starting the analysis of the program with a control-flow graph and modifying it at certain stages of the analysis, depending on the abstract values occurring at this stage. Differently stated, this consists of using an analysis $[\![.]\!]^\sharp_{G_1}$ for one part of a program and continuing with another analysis $[\![.]\!]^\sharp_{G_2}$. G_2 can be chosen dynamically, depending on the results already computed. From Equation (6), we can establish general conditions to ensure the soundness of this change.

First of all, recall that, from Section III, the abstract semantics $[\![p]\!]^\sharp_G\, \sigma^\sharp$ of a program p is a function mapping the nodes of $G = (V, E)$ to environments:

$$[\![p]\!]^\sharp_G\, \sigma^\sharp \ : \ V \to \text{Env}^\sharp$$

In order to detail how $[\![p]\!]^\sharp_G\, \sigma^\sharp$ is calculated, we introduce the function Φ_G which executes one small step in the abstract semantics:

$$\begin{array}{rcccc} \Phi_G & : & (V \to \text{Env}^\sharp) & \to & (V \to \text{Env}^\sharp) \\ & & A & \mapsto & A' \end{array} \tag{7}$$

For the sake of simplicity, we assume that, during the analysis, Φ_G knows the set of nodes W that remain to be visited. Initially, W is the singleton containing the entry point of the program and next, anytime the environment related to a node is modified, its successors in G are added to W. For example, W can be implemented by a list accessed by means of a global variable, and updated at each stage of the analysis. W could also be added to the parameters of Φ_G but Φ_G would not be monotonic in W. In Section V, we show that W becomes empty after a finite number of steps for our particular iteration strategy ($\Phi_G(A) = A$ if $W = \emptyset$).

$\Phi_G(A)$ takes a node $v_1 \in W$ such that $A(v_1) = \sigma^\sharp_{v_1}$ and executes, in the abstract semantics, the statement related to v_1 in the environment $\sigma^\sharp_{v_1}$, yielding a new environment $\sigma^\sharp$. Finally, A is updated, yielding the new function A' defined by:

$$\Phi_G(A) = A' \text{ such that } \left\{ \begin{array}{l} A'(v_1) = \sigma^\sharp \cup \sigma^\sharp_{v_1} \\ A'(v) = A(v) \cup \sigma^\sharp \text{ if } (v_1, v) \in E \\ A'(v) = A(v) \text{ otherwise} \end{array} \right.$$

Using Φ_G, the final result $[\![p]\!]^\sharp_G\, \sigma^\sharp$ of the analysis is obtained by:

$$[\![p]\!]^\sharp_G\, \sigma^\sharp_0 = \text{Fix } \Phi_G(A_0) \tag{8}$$

where A_0 is defined by

$$\left\{ \begin{array}{ll} A_0(v) = \sigma^\sharp_0 & \text{if } v \text{ is the entry point of the program} \\ A_0(v) = \bot & \text{otherwise} \end{array} \right. \tag{9}$$

Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two valid control flow graphs for $[\![p]\!]_{col}\ \iota$ and let $\Phi^{(n)}_G$ denote the n^{th} iterate of Φ_G. The analysis of p based on G_1 yields, after n steps of the abstract semantics, $A_n = \Phi^{(n)}_{G_1}(A_0)$. Substituting $[\![.]\!]^\sharp_{G_2}$ to $[\![.]\!]^\sharp_{G_1}$ then consists of analyzing the rest of the program with Φ_{G_2}. This is possible provided that $G_1 \prec G_2$, as stated by the following proposition.

Proposition 4 *Let G_1 and G_2 be two sound control flow graphs for $[\![p]\!]_{col}\ \iota$. Then if $G_1 \prec G_2$, we have:*

$$[\![p]\!]^\sharp_{G_1} \sigma^\sharp_0 \in \gamma_{G_2,G_1}\left(\textit{Fix } \Phi_{G_2}\left(\alpha_{G_1,G_2}(\Phi^{(n)}_{G_1}(A_0)) \right) \right) \tag{10}$$

The proof stems from Equation (6). Let G_1 and G_2 be two valid control flow graphs with respect to the collecting semantics $[\![p]\!]_{col}\ \iota$. Proposition 4 states that G_2 can be substituted to G_1 during the analysis without reanalyzing parts of the program provided that $G_1 \prec G_2$. The result of this calculation is less precise than $[\![p]\!]^\sharp_{G_1} \sigma^\sharp_0$.

The substitution of G_2 to G_1 made in Proposition 4 can be repeated many times to obtain an approximation of $[\![p]\!]^\sharp_{G_1} \sigma^\sharp_0$. Let $\mathcal{G}$ be the set of valid control flow graphs for $[\![p]\!]_{col}\ \iota$. $\mathcal{G}$ is partially ordered by $\prec$ and let G_0 be the most precise graph with respect to which we wish analyze the program. For example, G_0 may unfold the first n iterates of the loops or may implement a polyvariant analysis. An approximation of $[\![p]\!]^\sharp_{G_0}\sigma^\sharp_0$ is obtained by calculating the fixed point of a monotonic function:

$$[\![p]\!]^\sharp\ \sigma^\sharp_0 = \text{Fix}\ \Phi\big(A_0, G_0\big) \tag{11}$$

where A_0 is defined as in Equation (9) and where

$$\Phi\ :\ \big((V \to \text{Env}) \times \mathcal{G}\big) \to \big((V \to \text{Env}) \times \mathcal{G}\big) \tag{12}$$

Φ is a generalization of the function Φ_G of Equation (7). Φ maps any pair (A, G) made of a function $A\ :\ V \to \text{Env}$ and of a control flow graph G to a new pair (A', G'). We require that Φ be monotonic in A and G. So, at each iteration, it may either augment A or coarsen G (or both). More precisely, we require that Φ satisfies the following property:

$$\begin{array}{c}\forall (A,G) \in (V \to \text{Env}) \times \mathcal{G},\ \Phi(A,G) = (A', G') \Rightarrow \\ G \prec G' \text{ and } A \in \gamma_{G',G}(A')\end{array} \tag{13}$$

A dynamic partitioning algorithm starts to analyze the (user defined) most precise graph G_0 and then coarsen it at some stages, depending on the computed abstract values. By Proposition 4, the final result is an approximation of $[\![p]\!]^\sharp_{G_0}\ \sigma^\sharp_0$.

Proposition 5 *If Φ satisfies Equation (13) and if Fix $\Phi(A_0, G_0) = (A, G)$ then* $[\![p]\!]^\sharp_{G_0}\ \sigma^\sharp_0 \in \gamma_{G,G_0}(\Phi(A_0, G_0))$.

Proposition 5 is a direct consequence of Proposition 4. This proposition is used in Section V to prove the correctness of a dynamic partitioning algorithm.

V. A Dynamic Partitioning Algorithm

In this section, we introduce a dynamic partitioning algorithm that modifies the set of control points for which a common property is calculated during the analysis. This algorithm is one of the possible implementations of the function Φ used in Section IV. It is written as a function Ψ and its correctness stems from Proposition 5. More precisely, we show that Ψ satisfies the correctness criterium of Equation (13).

Following the principles of Section IV, the algorithm takes as inputs an initial graph G_0 and the related function A_0 defined in Equation (9), and calculates $(A, G) = \text{Fix}\ \Psi\big(A_0, G_0\big)$ for some G such that $G_0 \prec G$. As a result, we have $[\![p]\!]_{G_0}\ \sigma^\sharp_0 \in \gamma_{G,G_0}(A)$.

First of all, Ψ consider a covering of the set of the control points instead of a strict partition, as discussed in Section III. Next, Ψ works on a subset $\mathcal{G}_I \subseteq \mathcal{G}$ of the acceptable control flow graphs: recall, from Section III, that any node v of $G \in \mathcal{G}$ is a pair (L, T), where L is a set of syntactic program points and $T \subseteq \Theta$ is a set of timestamps. A graph G belongs to $\mathcal{G}_I$ iff any set L is a singleton and iff any set T of timestamps used in G belongs to the subset $\Theta_I \subseteq \Theta$ of timestamps mapping any point ℓ to an interval of integers:

$$T \in \Theta_I \iff \forall \ell \in \text{Lab},\ \{\tau(\ell)\ :\ \tau \in T\} \text{ is an interval} \tag{14}$$

$G \in \mathcal{G}_I$ if for any node $v = (L, T)$ of G, $L = \{\ell\}$ and $T \in \Theta_I$. In other words, in $\mathcal{G}_I$, a set T of timestamps associates to any program point ℓ an interval $I \in \mathcal{I}$ indicating how many times ℓ was executed. In the following, the nodes of G are denoted by pairs (ℓ, θ) instead of $(\{\ell\}, T)$, with $\theta\ :\ \text{Lab} \to \mathcal{I}$.

Ψ realizes two kinds of widenings, on the timestamps and on the environments. Widenings of the timestamps occur in the following case. Let $s_1 = (\ell, \tau_1, \sigma_1)$ and $s_2 = (\ell, \tau_2, \sigma_2)$ be two states of an execution trace $[\![p]\!]\ \sigma$ of a program p, as defined in Section III. s_1 and s_2 correspond to two instances of the same statement ℓ. If s_2 follows s_1 in a trace then there is a path $v_1 \to^* v_2$ in any sound control graph G of $[\![p]\!]_{col}\ \iota$, with $\sigma \in \iota$, $v_1 = (\ell, \theta_1)$, $v_2 = (\ell, \theta_2)$, $\tau_1 \in \theta_1$ and $\tau_2 \in \theta_2$. When Ψ analyzes v_2 the following configuration may arise: If $\sigma^\sharp_2 \sqsubseteq \sigma^\sharp_1$ then the control flow visits for the second time a statement ℓ, with a smaller environment. For example, this happens when a loop is unfolded, and when the abstract values computed by the analysis decrease for each new instance of a statement. In this case, we wish to no longer continue to analyze the instances of ℓ separately, as required by the current control flow graph G. So, Ψ decides to coarsen G, by merging v_2 to the nodes corresponding to the next instances of ℓ. This yields a new, coarser, graph G' which is used for the rest of the analysis. More precisely, G' is the graph G in which the new node $v = (\ell, \theta_1 \nabla_\Theta \theta_2)$ is substituted to v_2 and to any other node $v' \subseteq_\times v$. θ_1 and θ_2 belonging to Θ_I, ∇_Θ denotes the following widening operator on timestamps:

$$\theta_1 \nabla_\Theta \theta_2 = \ell \mapsto \theta_1(\ell)\ \nabla_I\ \theta_2(\ell) \tag{15}$$

where ∇_I is an usual widening operator on intervals.

The second kind of widening concerns the environments. When a node v is analyzed twice, yielding two abstract environments $\sigma^\sharp_1$ and $\sigma^\sharp_2$ such that $\sigma^\sharp_1 \sqsubseteq \sigma^\sharp_2$, then we widen this result by assigning to v the abstract environment $\sigma^\sharp_1\ \nabla_E\ \sigma^\sharp_2$.

The function Ψ, given in Figure 2, takes as arguments a graph $G \in \mathcal{G}_I$, with $G = (V, E)$, and the map A_G defined above. Firstly, Ψ analyzes the statement related to one node v that remains to be visited in G. This is done by calling the function Ψ_G described below and whose result defines a new map A'_G. If $A'_G \sqsubseteq_A A_G$ then the fixed point is reached and A_G is returned. Otherwise, Ψ examines whether G must be widened and, next, whether A_G must be widened. The relation $\sqsubseteq_A$ is defined by:

$$A_G \sqsubseteq_A A'_G \iff \forall \ell \in V,\ A_G(\ell) \sqsubseteq_E A'_G(\ell)$$

To decide whether G must be widened, Ψ calls the auxiliary function *coarsen* which merges some nodes in G if the conditions explained previously in this section are satisfied. A point ℓ is reached for the second time with a smaller environment if $v = (\ell, \theta) \in V$ and $v' = (\ell, \theta') \in V$ with $\theta \prec \theta'$ and $A_G(v') \sqsubseteq_E A_G(v)$. This means that the same statement is executed twice with no additional information. In order to avoid this useless computation to be repeated later, the algorithm merges all the future instances of ℓ in G into a new node $(\ell, \theta \nabla_\Theta \theta')$. Finally, the function $A_{G'}$ related to the new graph G' is calculated by the function $\alpha_{G,G'}$ of Equation (4).

The widening on the environments corresponds to the statement $A'_{G'} \leftarrow \alpha_{G,G'}(A, G)\ \nabla_A\ A'_{G'}$ in the function Ψ. $A'_{G'}$ is the map resulting from the call to *coarsen* and corresponds to the new graph G'. Note that if *coarsen* does not modify

$\Psi(A_G, G)$ {
  $A'_G \leftarrow \Psi_G(A_G)$
  if $A'_G \sqsubseteq_A A_G$ then {
    return (A_G, G)
  } else {
    $(A'_{G'}, G') \leftarrow$ coarsen(A'_G, G, ℓ, θ)
    $A'_{G'} \leftarrow \alpha_{G,G'}(A, G)\ \nabla_A\ A'_{G'}$
    return $(A'_{G'}, G')$
} }

coarsen (A_G, G, v) {
  let $(\ell, \theta) = v$
  if $\exists v' = (\ell, \theta')\ :\ \theta' \sqsubseteq_\Theta \theta \wedge A_G(v) \sqsubseteq_E A_G(v')$
  then {
    $\sigma^\sharp \leftarrow A_G(v)$
    $G' =$ replace in G v and v' by $(\ell, \theta \nabla_\Theta \theta')$
    $A' \leftarrow \alpha_{G,G'}(A_G)$
    return (A', G')
  } else {
    return (A_G, G)
} }

Figure 2: The dynamic partitioning algorithm.

$\Psi_G(A_G)$ {
  if $W = \emptyset$ then return A_G
  else {
    let $v = (L, T) \in W$
    let $\sigma^\sharp = A_G(v)$
    $W \leftarrow W \setminus \{v\}$
    $\sigma'^\sharp \leftarrow [\![v]\!]\ \sigma^\sharp$
    if $\exists v' = (\ell, \theta') \in V\ :\ \theta \sqsubseteq_\Theta \theta' \wedge \sigma'^\sharp \sqsubseteq_E A_G(v')$
    then $\Psi_G(A_G)$
    else
      $A'_G \leftarrow A_G$
      $A'_G(v) \leftarrow \sigma^\sharp \cup \sigma'^\sharp$
      for $v' \in$ succ(v) {
        $A'_G(v') \leftarrow A_G(v') \cup \sigma'^\sharp$
      }
      $W \leftarrow W \cup$ succ(v)
      return A'_G
} } }

Figure 3: Implementation of the auxiliary function Ψ_G.

G then $A'_{G'} = A_G$. The first argument of ∇_A is the map A_G whose domain is first transformed by the function $\alpha_{G,G'}$ of Equation (4). ∇_A is defined by:

$$A_G \nabla_A A'_G = \ell \mapsto A_G(\ell) \nabla_E A'_G(\ell) \qquad (16)$$

Finally, the function Ψ_G implements the general function Φ_G of Section IV. It analyzes one relevant statement corresponding to a node of G, modifies A_G and updates the set W of nodes that remain to be visited. Ψ_G is given in Figure 3. This function first analyzes a statement, yielding a new environment $\sigma'^\sharp$. Next, Ψ_G either keeps $\sigma'^\sharp$ or discards it and recursively examines another node. Ψ_G discards $\sigma'^\sharp$ if it does not introduce additional information. Formally, if the current node is $v = (\ell, \theta)$ and if there exists a node $v' = (\ell, \theta')$ such that $\theta \sqsubseteq_\Theta \theta'$ then the instances of ℓ defined by θ' include these defined by θ' in the covering of the control points chosen by the algorithm. So, if $\sigma'^\sharp \sqsubseteq A_G(v')$ then the state defined by v and $\sigma'^\sharp$ has already been treated. In particular, if $\theta = \theta'$ and $\sigma'^\sharp \sqsubseteq_E \sigma^\sharp$ then the same node is analyzed for the second time with a smaller environment and no information need to be propagated in G. Once the environment $\sigma'^\sharp$ is kept, A_G is modified and W is updated. This corresponds to the else block of the inner conditional.

This algorithm was implemented and we conclude this section by illustrating its behavior on the examples of Section II. As in Section II, we start by examining the abstract values resulting from the execution of Point $|5|$ in the program of Page 2. We assume that the initial control flow graph G unfolds the loop $n = 6$ times, but the same calculation is made for any $n \geq 4$. The values of G and A_G at each step of the algorithm are displayed in Figure 4. For the sake of simplicity, we only associate one node to the body of the loop. We display the values of `y` after execution of Point $|5|$. After two iterations of Ψ, we have the graph of Figure 4 a). The nodes $(|5|, [1,1])$ and $(|5|, [2,2])$ correspond to the first two instances of $|5|$. We have:

$$A_G(|5|, [1,1]) = \{x = [-1,-1]; y = [-1,-1]\}$$

and

$$A_G(|5|, [2,2]) = \{x = [1,1]; y = [0,0]\}$$

During its third iteration, Ψ calls Ψ_G which returns

$$A_G(|5|, [3,3]) = \{x = [-1,-1]; y = [-1,1]\}$$

The values of G and A_G at this time are shown in Figure 4 b). Next Ψ calls *coarsen*. Since $(|5|, [1,1]) \sqsubseteq_\Theta (|5|, [3,3])$ and $A_G(|5|, [3,3]) \sqsubseteq_E A_G(|5|, [1,1])$, the nodes are merged, yielding the new graph of Figure 4 c). At the next iteration of Ψ, the new active node is $(|5|, [4,4])$ for which Ψ_G computes

$$A_G(|5|, [4,4]) = \{x = [1,1]; y = [0,0]\}$$

The new graph is displayed in Figure 4 d). Again, this node is merged to $(|5|, [2,2])$, yielding the graph of Figure 4 e). Finally, at the fifth iteration of Ψ, the auxiliary function Ψ_G has $v = (|5|, [5,5])$ and computes

$$\sigma'^\sharp = \{x = [-1,-1]; y = [-1,-1]\}$$

There exists a node $v' = (|5|, [1, +\infty])$ satisfying the condition of the inner `if` statement. Ψ_G is called recursively with $W = \emptyset$, and the fixed point of Ψ is reached. It corresponds to the graph of Figure 4 e). As a conclusion, the abstract value of y inside the loop is $\bot \cup [0,0] \cup [-1,-1] = [-1,0]$.

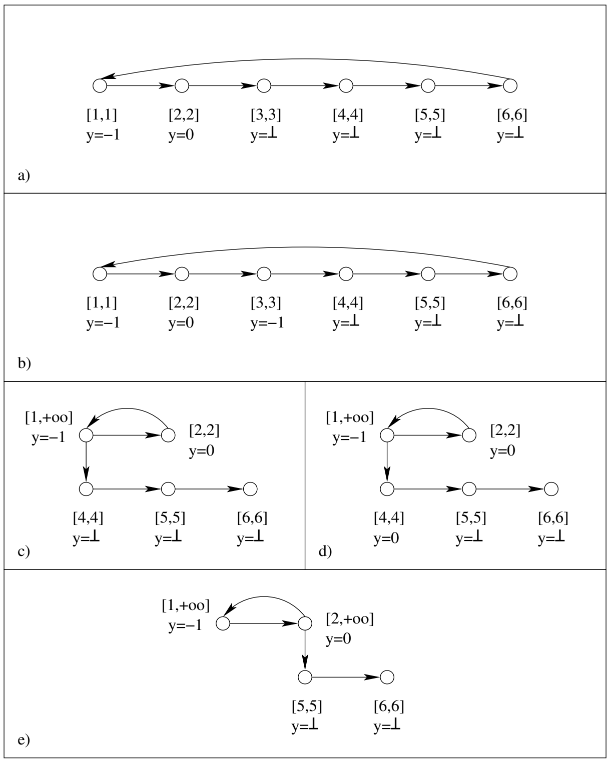

Figure 4: Successive steps of the algorithm for the program of Page 2.

For the program `A` of Section II, the algorithm works as follows. The upper bound of the interval associated to `j` increases for the first nine iterations and next decreases. If, in the initial graph, the loop is unfolded at least ten times then the abstract value of `j` becomes the constant $[-\infty, 45]$ after nine iterations. The conditions making the function $coarsen$ active are satisfied. A new node v corresponding to the instances 9 to infinity of the loop is inserted whose related environment maps `j` to $[-\infty, 45]$. At the next iteration of Φ_G, the fixed point is reached and the analysis terminates, independently of the precision of the initial control flow graph (i.e. independently of how many iterations are unfolded in the initial graph). Finally, the same precision is obtained for the program `C` of Section II in which the loop of the preceding example is the body of another loop. This is due to the fact that the widening ∇_Θ on the timestamps applied to the inner loop, do not merge instances of the outer loop during the first iterations. The inner loop is unfolded the right number of times for each instance of the outer loop.

VI. Conclusion

The dynamic partitioning algorithm introduced in this article relies on the calculation of the fixed point of a function Ψ of two variables. At each iteration, a coarser graph is chosen or a greater value is assigned to a node in the lattice of the abstract values. When the fixed point is reached, the algorithm outputs a control flow graph coarser than the initial one as well as the abstract values attached to its nodes. As illustrated in Section V, this enables to stop unrolling a loop when this does not improve the precision of the final result. This algorithm was implemented and we plan to use it in our analyzer of numerical assembly codes.

In Section IV, we introduced sufficient conditions to ensure the correctness of the analysis based on the calculation of the fixed point of a function Φ for which Ψ is one of the possible implementations. Ψ is well-suited for loop unrolling but we plan to define other algorithms, more general and possibly more accurate. An issue would be to substitute control words [3] to the timestamps used in this article.

References

[1] F. Bourdoncle. Abstract interpretation by dynamic partitioning. *Journal of Functional Programming*, 2(4):407–435, 1992.

[2] F. Bourdoncle. *Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite*. PhD thesis, Ecole Polytechnique, Paris, 1992.

[3] A. Cohen and J.-F. Collard. Instance-wise reaching definition analysis for recursive programs using context-free transductions. In *Parallel Architectures and Compilation Techniques*. IEEE Computer Society Press, 1998.

[4] P. Cousot. Semantics foundations of program analysis. In N.D. Jones and S.S. Muchnick, editors, *Program Flow Analysis: Theory and Applications*, chapter 10. Prentice-Hall, 1981.

[5] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL'77*, pages 238–252, 1977.

[6] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'79*. ACM Press, 1979.

[7] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Symbolic Computation*, 2(4):511–547, 1992.

[8] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Proceedings of the ACM-SIGPLAN Symposium on Principles of Programming Languages, POPL'90*, pages 157–168, 1990.

[9] E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS'01*, number 2126 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[10] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *European Symposium on Programming, ESOP'02*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[11] M. Handjieva and S. Tzolovski. Refining static analyses by trace-based partitioning using control flow. In *Static Analysis Symposium, SAS'98*, number 1503 in Lecture Notes in Computer Science. Springer-Verlag, 1998.

[12] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, number 1694 in Lecture Notes in Computer Science. Springer-Verlag, 1999.

[13] M. Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *European Symposium on Programming, ESOP'02*, number 2305 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[14] M. Martel. Static analysis of the numerical stability of loops. In *Static Analysis Symposium, SAS'02*, number 2477 in Lecture Notes in Computer Science. Springer-Verlag, 2002.

[15] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *Static Analysis Symposium, SAS'02*, number 2477 in Lecture Notes in Computer Science. Springer-Verlag, 2002.