Structurally Defined Conditional Data-Flow Static Analysis

Elena Sherman[1(✉)] and Matthew B. Dwyer[2]

[1] Boise State University, Boise, ID 83706, USA
elenasherman@boisestate.edu
[2] University of Nebraska - Lincoln, Lincoln, NE 68588, USA
matthewbdwyer@unl.edu

Abstract. Data flow analysis (DFA) is an important verification technique that computes the effect of data values propagating over program paths. While more precise than flow-insensitive analyses, such an analysis is time-consuming.

This paper investigates the acceleration of DFA by structural decomposition of the underlying control flow graph. Specifically, we explore the cost and effectiveness of dividing program paths into subsets by partitioning path suffixes at conditional statements, applying a DFA on each subset, and then combining the resulting invariants. This yields a family of independent DFA problems that are solved in parallel and where the partial results of each problem represent safe program invariants.

Empirical evaluations reveal that depending on the DFA type and its conditional implementation the invariants for a large fraction of program points can be computed in less time than traditional DFA. This work suggests a strategy for an "anytime DFA" algorithm: computing safe program invariants as the analysis proceeds.

1 Introduction

Software developers use static analyses as a supplement to traditional dynamic testing approaches. Tools such as AbsInt Astrée [1], Facebook Infer [2], and MathWorks Polyspace[1] are becoming standard parts of development workflows. Advances in program analysis and theorem proving have helped static program analysis become more feasible for verification of general-purpose software.

The power of static analysis to consider all program behaviors follows from its ability to safely over-approximate program behaviors by abstracting the concrete domain of program variables and the programming language semantics. But at the same time its over-approximating nature causes static analysis to identify some property violations as uncertain. The reason for this uncertainty is that a static analysis cannot tell if a violation happens on a feasible or an infeasible, i.e., strictly over-approximating, program behavior. This inconclusiveness is

---

[1] http://www.mathworks.com/products/polyspace.html.

unacceptable since each potential violation must be examined further. An automatic solution to the elimination of false positive violations is to increase the precision of a static analysis, i.e., improve the analysis so it considers fewer infeasible behaviors.

However, improving analysis precision generally increases analysis cost in terms of running time and memory consumption. A common approach to address this problem is to decompose the program's state space into several subspaces and perform analysis on each separately. What distinguishes those techniques are the underlying decomposition methods.

One approach focuses on making a precise static analysis scalable by decomposing a large program into modules like procedures and classes, and allowing the analysis to examine each partition independently. Next, the analyzed information of each module is composed together to obtain the result of the whole program analysis. In the literature [3,4] this method is referred to as *partial static analysis*.

Another approach aims to improve the scalability of precise analysis by permitting the analysis to explore only those program states for which it is adequately precise, i.e., able to provide definitive result. In the literature [5–7] this approach is called *conditional static analysis* (CSA) since the permitted states are described by a condition θ expressed as a logical formula. In such a framework an analysis verifies a program under some assumptions, i.e., there are no null pointer exceptions or a pre-condition on input values is assumed to hold. Next, another analysis attempts to prove these assumptions by showing that the states, which do not satisfy θ are either not reachable or do not lead to property violations. In prior work the condition θ is either determined from the analysis design [5,6], where θ is applicable to all program states, or determined during program analysis execution [7], where θ is composed of the conditions assumed to hold for a certain set of states.

While previous work on CSA focuses on finding values of θ that ensure an increase in analysis precision, in this paper we explore the decomposition of the program's state space in order to improve the efficiency of the analysis. We decompose the program's state space based on the program's control flow graph (CFG), i.e., on the program's structural information. Each partition corresponds to a set of paths expressed as a set of CFG branches π. This permits a path, or π, defined CSA to compute invariants for each π independently and in parallel. While one can use a logical formula θ as a precondition to restrict program input values to those that follow a particular path, we conjecture two primary advantages of structural decomposition. First, π is expressed directly as a subset of CFG branches and computing an equivalent θ, expressing constraints on input values, would require complex value propagating analyses. Second, because π is structural its effect on the analysis is independent of the abstract domain, whereas even an equivalent θ may not be effective in preventing values from flowing along a branch due to over-approximation by the abstract domain.

The contributions of this paper are presentation of:

1. A formalization of the path-define CSA as a data-flow framework.
2. Two algorithms for implementing CSA in existing analysis frameworks.
3. An approach to efficiently partition CFG paths for path-defined CSA.

In the next section, we provide an overview of the structural CSA approach and pose our research questions. After that we formalize CSA in Sect. 3 and demonstrate in Sect. 4 two different ways of implementing CSA in an existing program analysis framework. In Sect. 5 we present our approach to partitioning a CFG. Then we present our experiments and discuss related work.

2 Overview

We begin with an example of a traditional data-flow analysis. Data-flow analysis calculates some information for each point in a program based on the program structure and the language semantics. The calculated facts, i.e., program invariants, are then later used to reason about program properties, usually safety properties, which must hold on all feasible program executions. Data-flow analyses that compute invariants that are satisfied by all paths are called *must* analyses. In our example we show how a data-flow analysis computes invariants for each program statement.

Consider a program and its corresponding CFG in Fig. 1(a). In this example x is an integer variable. The edges of the CFG are labeled with T for *true* branches and F for *false* branches of the conditional statement.

In order to calculate invariants *static analysis* (SA) works with abstract values of x, which are composed of the elements of an abstract domain. For example, the *signs* abstract domain has three elements $\{+, 0, -\}$. 0 denotes the singleton set $\{0\}$ of concrete values, $+$ denotes positive values, and $-$ denotes negative values. If SA employs the *signs* abstract domain then the values of x are expressed as a set containing any of those three elements, including special cases $\{\} \equiv \bot$ for no values and $\{+, 0, -\} \equiv \top$ for all values

SA starts by assigning x to $\top$ at the CFG's entry point, since x can have any concrete value. Upon encountering the conditional statement SA computes invariants for x along the true branch, then along the false branch, and then merges these values before the return statement. The left CFG in Fig. 1 shows the result of the analysis where the CFG's edges are annotated with computed invariants for x. Clearly, the computations along these two branches are independent of each other and could be done simultaneously, thus reducing the computational time. This observation is the main idea behind our approach.

In other parallel SA approaches that we discuss in Sect. 7, the parallel computation is done inside a full SA. During the computation a parallel SA waits at the merge point, where the analysis combines the results of the two branches, on the completion of each branch before proceeding further thereby reducing parallelism.
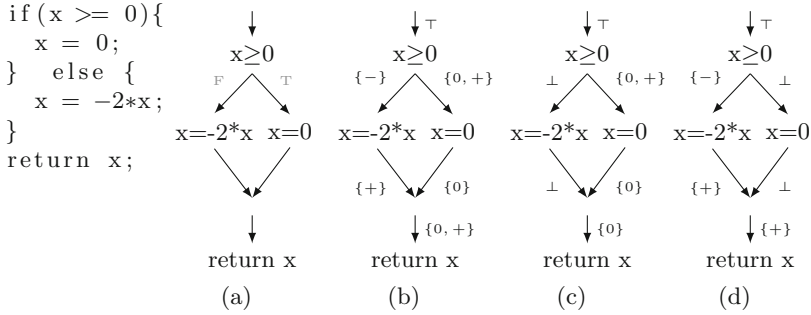
```
if (x >= 0){
    x = 0;
}  else {
    x = -2*x;
}
return x;
```


Fig. 1. Source code and its CFG (a); analysis examples: *signs* analysis result (b), CSA *sign* analysis result for set paths with *1t* prefix (c) and *1f* prefix (d)

Moreover, if we can analyze the true and the false branches independently then the invariants computed along the true branch could be accessed even sooner for a user to process. This observation is another inspiration for designing "anytime DFA", which provides a sound information about some program's invariants.

As mentioned, in general, it would be difficult to compute a precondition θ that restricts the input values of x to only those that would take the CSA computation to a particular set of branches. However, in path-defined CSA those branches can be stated explicitly. In our example we can have two set of paths: one defined by $\pi_1 = \{1t\}$, i.e., take the true branch of the first conditional statement only, and $\pi_2 = \{1f\}$, i.e., take the false branch of the first conditional statement. The results of these two path-defined CSA are in (c) and (d) in Fig. 1, respectively. We can see that the union of the abstract element sets for π_1 CSA and π_2 CSA on the corresponding edges results in the same invariants of the full analysis, that is CSA produces sound results. Section 3 formalizes the conditions under which soundness holds in CSA. Overall CSA can potentially provide two main benefits to a user: (1) the speedup of the analysis using parallelism *and* (2) delivering fast useful feedback to users.

One of the objectives of our work is to investigate the efficiency of two π-defined CSA implementations in an existing data-flow framework and its ability to compute sound invariants at intermediate points in the analysis.

To evaluate efficiency improvements we consider a traditional reaching definitions (RD) analysis and value-based data flow analysis (VB) for disjoint domains [8] similar to one used in the above example. Our approach automatically generates a set of π for each method based on heuristics discussed in Sect. 5. Then based on π it recombines CSA in the order of its completion and then compares the result of each combination step to the results of the full SA. Through our experiments we aim to answer the following research questions:

1. Does path-defined CSA compute sound invariants faster than SA?
2. At what rate does CSA compute sound invariants?
3. How efficient are the two implementations of CSA?

We answer these research questions through an extensive empirical evaluation on real-world programs.

3   Conditional Analysis

In this section we first present the traditional monotone framework for data flow analysis followed by the discussion of the necessary changes that extend it to a conditional data flow framework. This section also outlines the approach of composing the unconditional result from conditional ones.

  We use the data flow analysis framework similar to one presented in [9] for an analysis $\mathcal{A}$, only we extended it to express branch-sensitive analysis, where the outgoing flow of a statement $l \in CFG_P$ is defined for each of its outgoing edges $(l, l') \in CFG_P$. Thus, the following parameters define $\mathcal{A}$.

– The complete lattice $D_\mathcal{A}$ that describes the abstract domain of $\mathcal{A}$.
– CFG_P for a program P.
– A set of monotone transfer functions $\mathcal{F}_\mathcal{A}$ for each statement $(l, l') \in CFG_P$ that maps an element of $D_\mathcal{A}$ to itself, i.e., $f_{ll'} \in \mathcal{F}_\mathcal{A} : D_\mathcal{A} \mapsto D_\mathcal{A}$.
– Entry statements E in CFG_P.
– An initial value $\iota \in D_\mathcal{A}$ for statements in E.

  Then the set of equations for forward $\mathcal{A}$ is defined as follows on entry and exit of each statement $l \in CFG_P$:

$$\mathcal{A}_{in}(l) = \bigsqcup \{\mathcal{A}_{out}(l', l) \mid (l', l) \in CFG_P\} \sqcup \iota_E^l \qquad (1)$$

$$\text{where } \iota_E^l = \begin{cases} \iota & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases}$$

$$\mathcal{A}_{out}(l, l') = f_{ll'}(\mathcal{A}_{in}(l)), (l, l') \in CFG_P$$

where $\sqcup$ is the least upper bound operator, $\bot$ is the bottom element of $D_\mathcal{A}$ for which $\forall d \in D_\mathcal{A} : \bot \sqcup d = d$ and $\forall (l, l') \in CFG_P : f_{ll'}(\bot) = \bot$. For safety, $\bot$ corresponds to the empty set of concrete values and $\top$ to the set containing all concrete values. The value of ι is assigned to $\top$, i.e., the analysis considers all possible input values for a program. The solution of the above set of equations provides the result of the analysis for P.

  In our work we express a condition for DFA as a condition that identifies the set of paths to be analyzed π, which defines a CFG partition. We describe CSA as a special case of $\mathcal{A}$, which we denote as $\mathcal{A}^\pi$. Thus, a traditional data flow analysis $\mathcal{A} = \mathcal{A}^{(\emptyset)}$; unspecified branches in π are explored fully. For our formulation of CSA, the edges in π are not nested inside a loop.

  We have chosen π to be represented by the set of branch edges in CFG_P, at most one for each conditional statement l, which the analysis must include while excluding their counterparts. If l has l' and l'' as its true and false targets, respectively, then π can contain the edge (l, l'), or the edge (l, l''), or none of them. To capture the relation between the opposite branches of l we designate

$(l, l') = \neg(l, l'')$ and vice versa $(l, l'') = \neg(l, l')$. If $(l, l') \in \pi$ then the values of all variables x_i incoming to the target of its opposite edge l'', i.e., along edge $\neg(l, l')$, are set to $\bot$. For brevity, we denote such case, i.e., when $\forall i : x_i = \bot$, as $\bot$ state. Those $\bot$ values of the infeasible edges are propagated further to its children making them excluded from the analysis. The same principle applies when the opposite target $(l, l'') \in \pi$. When none of the edges are present in π then the analysis treats them in its usual manner, i.e., propagates the information through both branches.

With these path-based conditions we can now write the set of equations for conditional data flow framework for an analysis $\mathcal{A}^\pi$:

$$\mathcal{A}^\pi_{in}(l) = \bigsqcup \{\mathcal{A}^\pi_{out}(l', l) \mid (l', l) \in CFG_P\} \sqcup \iota^l_E \tag{2}$$

$$\text{where } \iota^l_E = \begin{cases} \top & \text{if } l \in E \\ \bot & \text{if } l \notin E \end{cases}$$

$$\mathcal{A}^\pi_{out}(l, l') = \begin{cases} f_{ll'}(\mathcal{A}^\pi_{in}(l)) & \text{if } (l, l') \in CFG_P \text{ and } \neg(l, l') \notin \pi \\ \bot & \text{if } (l, l') \in CFG_P \text{ and } \neg(l, l') \in \pi \end{cases}$$

Let Π be the set of path-based conditions for an analysis $\mathcal{A}$. Executing $\mathcal{A}$ with different conditions $\pi_j \in \Pi$ produces a set of conditional analysis $\mathcal{A}^{\pi_j}$. The solution for an $l \in CFG_P$ over Π can be expressed as the meet over all maximal fixed point computations (MFP) produced by each $\mathcal{A}^{\pi_j}$, which, when equal to the MFP for $\mathcal{A}$, means that SA and CSA produce the same results.

$$\bigsqcup_{\pi_j \in \Pi} MFP_{\mathcal{A}^{\pi_j}}(l) = MFP_{\mathcal{A}}(l) \tag{3}$$

Since SA performs the computation over all program execution paths then in order for CSA to be sound it must ensure the same. For example consider two conditions $\{(l, l')\}$ and $\{\neg(l, l')\}$. The conditional analysis $\mathcal{A}^{\{(l, l')\}}$ analyzes all possible input values for the set of paths containing the true branch of l while $\mathcal{A}^{\{\neg(l, l')\}}$ does it for the set of paths containing the false branch of l. Thus, together $\mathcal{A}^{\{(l, l')\}}$ and $\mathcal{A}^{\{\neg(l, l')\}}$ analyze all program paths. To formalize the soundness of CSA, we express π as a boolean function g_π as follows.

Each true edge in CFG_P is mapped to a boolean variable x_i and each false edge is mapped to $\neg x_i$. Then edges in π are mapped to a set of literals and g_π is expressed as a conjunction of those literals. In our example if (l, l') is mapped to x_1 then $g_{\{(l,l')\}} := x_1$ and $g_{\{\neg(l,l')\}} := \neg x_1$. The union of these two sets of paths is equivalent to the disjunction of $g_{(l,l')}$ and $g_{\neg(l,l')}$. Thus, the combination of arbitrary π_1 and π_2 is given as $g_{\pi_1} \vee g_{\pi_2} \equiv \pi_1 \cup \pi_2$.

Π yields a sound CSA if $\bigvee_{\pi_j \in \Pi} g_{\pi_j}$ is a tautology. To maximize efficiency of CSA π should be pairwise disjoint – thereby eliminating duplicate computation.

$$\forall \pi_i, \pi_j \in \Pi \text{ and } \pi_i \neq \pi_j : g_{\pi_i} \wedge g_{\pi_j} = false$$

Therefore in order for the analysis to be sound and efficient the partition algorithm should generate partitions of Π that satisfy these two constraints. We discuss our partitioning algorithm in Sect. 5.

Algorithm 1. A branch-sensitive work-list algorithm for a CFG

1: $w \leftarrow \texttt{quasiTopOrder}(CFG)$
2: **while** $\neg w.isEmpty()$ **do**
3:     $l \leftarrow w.removeNext()$
4:     $in = \bot$
5:     **for** $p \in \text{pred}(l)$ **do**
6:         $in \leftarrow \texttt{merge}(in, out[p][l])$
7:     **end for**
8:     $outNew = \texttt{f}(in, l)$
9:     **for** $s \in \text{succ}(l)$ **do**
10:         **if** $outNew[s] \neq out[l][s]$ **then**
11:             $out[l][s] = outNew[s]$
12:             **if** $\neg w.contains(s)$ **then**
13:                 $w.insert(s)$
14:             **end if**
15:         **end if**
16:     **end for**
17: **end while**

Algorithm 2. A quasi-topological order for a CFG

1: $\texttt{quasiTopOrder}(CFG)$
2: $N \leftarrow |CFG|$
3: **for** $i \in (1, \ldots, N)$ **do**
4:     $marked[i] \leftarrow false$
5: **end for**
6: $indx \leftarrow 0$
7: $\texttt{DFS}(CFG.entry())$
8: **return** $ordered$

1: $\texttt{DFS}(l)$
2: **if** $\neg mark[i]$ **then**
3:     $mark[i] \leftarrow true$
4:     **for** $s \in \text{succ}(l)$ **do**
5:         $\texttt{DFS}(s)$
6:     **end for**
7:     $ordered[indx] \leftarrow l$
8:     $indx \leftarrow indx + 1$
9: **end if**

4   Implementations of Conditional Analysis

Static analysis developers commonly solve Eq. 1 using an iterative work-list algorithm that propagates the abstract values from the entry nodes $l \in E$, usually the single entry node of a program, to the rest of the nodes while computing $\mathcal{A}_{in}$ and $\mathcal{A}_{out}$ flow values. The algorithm terminates when for each node in the CFG its $\mathcal{A}_{in}$ and $\mathcal{A}_{out}$ are unchanged.

Algorithm 1 sketches a basic work-list algorithm for a branch-sensitive data-flow analysis where for brevity $\mathcal{A}_{in}$ and $\mathcal{A}_{out}$ are denoted as in and out, respectively. A work-list data structure w keeps track of CFG nodes for which in values are changed in the previous iteration and, thus, require recalculation. The computation reaches a fixed-point when no changes in in are detected which corresponds to w becoming empty. At each iteration a new node l is removed from work-list w, its incoming flows are calculated (lines 4 - 7), and its new outgoing flow is recalculated using the transfer function $\texttt{f}$ (line 8) for each of its successors. That is $outNew$ is an array where each element contains an outgoing flow to each of l's successors. For example, a conditional statement would have its first elements associated with the true branch and the second elements associated with the false branch. Lines 9 - 16 determine the changes in the outgoing flows for each of l's successors by comparing the new and old values of out and insert the affected successors back to w.

In order to further improve the efficiency of the work-list algorithm, an analysis framework takes into the consideration the ordering of nodes in the CFG. It ensures that the nodes in w appearing topologically before a given node are processed first. Since, the CFG can be a cyclic graph, the framework populates w

Algorithm 3. CFA_1 implementation of a CFA

```
1: f(in, l)
2: for s ∈ succ(l) do
3:    if in = ⊥ ∨ ¬(l, s) ∈ π then
4:       outNew[s] ← ⊥
5:    else
6:       outNew[s] ← f(in, l, s)
7:    end if
8: end for
9: return outNew
```

Algorithm 4. CFA_2 implementation of CFA

```
1: CDFS(l)
2: if ¬mark[i] then
3:    mark[i] ← true
4:    for s ∈ succ(l) do
5:       if ¬(l, s) ∉ π then
6:          DFS(s)
7:       end if
8:    end for
9:    ordered[indx] ← l
10:   indx ← indx + 1
11: end if
```

using a quasi-topological ordering algorithm similar to one presented in Algorithm 2. The node removal and insertion operations on w preserve the CFG's quasi-topological ordering.

A program analysis framework provides analysis developers with implementations of these work-list and ordering algorithms. The developers instantiate their analyses by providing implementations for merge and f functions, as well as an abstract domain and initial flow values. We present two approaches for implementing CSA in such analysis framework.

The first approach CSA_1 uses the transfer function f to set the outgoing flows to the infeasible branches and its successors to $\bot$. Algorithm 3 details that approach. Here π is a global variable which in line 3 determines whether the outgoing flow for a successor should be set to $\bot$, or computed using $f(in, l, s)$ of the full SA. Extending an analysis framework to implement CSA in f is straightforward and does not require analysis developers to further understand the framework's implementation. However, CSA_1 does perform extra computations along infeasible program paths.

The second approach CSA_2 addresses this potential performance drawback by modifying the quasi-topological DFS search as shown in Algorithm 4. The algorithm does not traverse CFG down the paths of the excluded branches
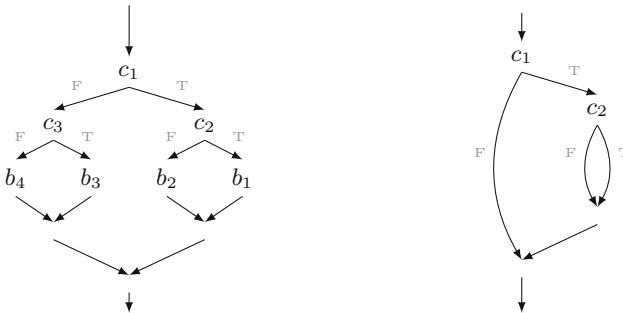

Fig. 2. Combining selected conditional statement c_2 and CFG (left) to produce an abstract graph (right) encoding $\Pi = \{c_1 f\}, \{c_1 t, c_2 f\}, \{c_1 t, c_2 t\}$

(line 5), thus assigning w only those nodes that are in π. When a node is inserted back to w (Algorithm 1 line 13) only the nodes in π are inserted in w at their proper positions. CSA_2 implementation requires that analysis developers an advanced understanding of the analysis framework, i.e., the algorithms and data-structures used in the quasi-topological ordering. However, this approach only iterates over the nodes that are defined in π. We have implemented two approaches and in Sect. 6 we empirically compare them. In the next section we present our approach on partitioning a CFG into a set of partitions Π.

5   Partitioning CFG

A program can have many branches and if we decide to use each of them to partition CFG then the size of Π could become prohibitively large, thus we need to determine which branches should be used to generate Π. The goal of our selection heuristic is to chose those branches that might reduce the computational time. We explore three main characteristics of a conditional statement: (a) whether it has non-empty blocks of code b_1 and b_2 on both *true* and *false* branches respectively, (b) the size of b_1 and b_2 in relation to the entire method and (c) the difference between the sizes of b_1 and b_2.

The first heuristic ensures that there is an opportunity for a parallel execution of two branches b_1 and b_2. The next two heuristics quantify that opportunity. Among b_1 and b_2, we select the one with the maximum block size and calculate its ratio to the number of statement in the method. We call this value r_t. Then we calculate another ratio r_d which is the ratio between the difference in block sizes to the number of statements in the method. If we use $|b_i|$ to denote the size of b_i block and $|m|$ the number of statements in method m, then

$$r_t = \frac{max(|b_1|, |b_2|)}{|m|}, r_d = \frac{abs(|b_1| - |b_2|)}{|m|}$$

The larger the r_t and the smaller the r_d, the higher the chances that CSA has better performance if those branches are used to partition CFG. After selecting a set of branches, we first ensure, for sound CSA analysis, that they do not appear inside loops. Next, we combine the selected conditional statements c_i with structural information about the CFG to generate an efficient set of Π.

For example, consider the CFG on the left of Fig. 2 where c_i are conditional statement and b_i are blocks of code. If the heuristic determines that the branches of c_2 are suitable for the CFG partition then simply expressing the set of partitions Π as $\{\{c_2f\}, \{c_2t\}\}$ would result in both CSA computing the invariants along c_1's false branch, that is performing the computation twice. In order to avoid this redundancy our partition algorithm traverses the CFG and finds all branches of the conditional statements through which the original conditional statements are reachable and store it as an "abstracted" graph similar to one shown on the left of Fig. 2. Next, using the abstracted graph we generate Π for CSA which in this case are $\{c_1f\}, \{c_1t, c_2f\}, \{c_1t, c_2t\}$.

Such post-processing also handles cases when both c_2 and c_3 are marked for partition. A simplistic approach is to create all possible combinations of their branches, but that results in identical partition that compute the same invariants, for example, $\{c_3t, c_4f\}$ and $\{c_3t, c_4f\}$ compute the true branch of c_3 both times. In contrast, our partition generation detects that c_3 and c_4 are independent. In our evaluation section we describe the threshold values we used for r_d and r_t parameters.

6   Evaluation

We evaluate our implementations of the path-defined conditional analysis using two distinct analyses: intra-procedural value-based analysis (VB) and an intra-procedural reaching definitions analysis (RD). For VB analysis we used implementation and abstract domains that we developed in our previous work [8]. For RD we used the implementation provided with Soot framework distribution. RD is a relatively fast analysis with an easily computable transfer function, while VA takes longer to complete due to its complex transfer function evaluations. For each of the analysis we performed experiments with their full versions SA, i.e., VB and RD, their CSA_1 versions implemented with Algorithm 3, which we name CVB_1 and CRD_1, and their CSA_2 versions implemented with Algorithm 4, which we name CVB_2 and CRD_2 respectively. The source code, program subjects and instructions on replicating the experiment are available on GitHub[2].

Program Subjects. In order to perform our evaluations we first analyzed 105 methods in 19 Java classes across 10 open-source projects that we used in our previous work [8] where we employed Boa [10] to mine methods of open-source programs from GitHub, count the number of operations in each method and then we randomly selected those methods that contain at least 180 of integer operations. Among those 105 methods we selected methods with conditional statements that meet the first requirement of our partitioning algorithm to have a non-trivial conditional statement where both true and false branches have non-empty blocks of code. This step reduced the number of methods to 68. Among them 53 methods have at least one non-trivial condition statement outside of loops, which allows for computing sound CSA. Those methods have on average 177 statements and 19 simple conditional statements.

Abstract Domain Subjects for VB Analysis. VB analysis uses atomic elements of its abstract domain to express the computed program invariants. To determine whether the size of the disjoint abstract domain influences the efficiency of VB analysis we used three disjoint abstract domains of small (8 atomic elements), medium (10 atomic elements) and large (12 atomic elements) sizes. We randomly chose those abstract domains among available disjoint domains with the same number of atomic elements. Our preliminary experiments have shown that there is no difference in the evaluation data between the domain sizes, so we present the data only for the medium size domain.
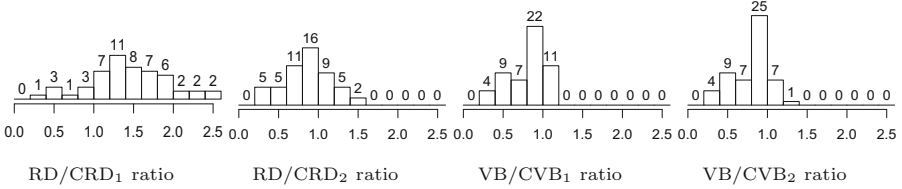
---

[2] https://github.com/BoiseState/Conditional-DFA.

Fig. 3. Histograms of ratios between runtimes of full and conditional analyses.

6.1   Experiment Description

First we analyze 53 methods using full SA, recording its run time and computed invariants after each statement. The CSA evaluation consists of three main steps: (1) generating a set of partitions Π for each method, (2) running CSA_1 and CSA_2 analyses on the partitions and recoding run time and invariants, and (3) aggregating the computed invariants for partitions of the same method. We run experiments on a 2.9 GHz Intel Core i5 processor with 8 GB of memory running OS X operating system with the analysis running on Java RE 1.8.

Step 1. We implemented the partition algorithm from Sect. 5 in the Soot Java Optimization framework to take advantages Soot's CFG and other related data structures. The partition algorithm takes as input a class and its method to be partitioned, and parameters $r_{\bar{t}}$ that determine the minimum value for r_t, and $r_{\underline{d}}$ that determines the maximum value for r_d. In our evaluations we set $r_{\bar{t}} = 3\%$ and $r_{\underline{d}} = 60\%$ for the majority of the methods and increased $r_{\bar{t}}$ and decreased $r_{\underline{d}}$ values when the number of partitions became greater than 45. This resulted in the increase of $r_{\bar{t}}$ to 15% for two methods and the following $(r_{\bar{t}}, r_{\underline{d}})$ values for three methods: $(15\%, 30\%)$, $(20\%, 15\%)$ and $(20\%, 30\%)$.

This step produced the total of 472 partitions for 53 methods, with the minimum of two partitions and maximum of 32 partitions per method. A partition π is encoded as a set of branches that CSA should take defined by the conditional statement id and the branch's outcomes: either true of false. As defined in our CSA framework, if a conditional statement is not present in π then CSA explores both of its branches.

Step 2. We implemented VB, CVB_1 and CVB_2 in the Soot Java Optimization framework and used Z3 version 4.3.2 as the constraint solver. CVB takes the following input parameters: a class name and its method to be analyzed, an abstract domain and a partition π. We executed VB_1 and VB_2, for each partition π and the full VB analysis. We implement RD, CRD_1 and CRD_2 also in the Soot framework. CRD takes three input parameters: a class name and its method to be analyzed and a partition π.

We recorded two sets of data that CSA produces: the running time of the analysis and the computed invariants for the corresponding analysis: set of reaching definition elements for CRD and abstract values for variables expressed as SMT constraints for CVB. We execute each experiment three times and use their

Table 1. CRD Cost vs. Precision

t, ratio of RD	% sound invariants of RD					
	0	0–25	25–50	50–75	75–100	100
CRD_1 *analysis*						
≤0.2	45	8	0	0	0	0
≤0.4	43	6	1	2	0	1
≤0.6	32	11	3	1	0	6
≤0.8	26	11	5	1	0	10
≤1.0	19	9	7	1	3	14
CRD_2 *analysis*						
≤0.2	31	18	1	2	0	1
≤0.4	21	16	4	4	2	6
≤0.6	13	10	9	3	5	13
≤0.8	10	8	5	2	5	23
≤1.0	1	2	2	2	6	40

Table 2. CVB Cost vs. Precision

t, ratio of VA	% sound invariants of VB					
	0	0–25	25–50	50–75	75–100	100
CVB_1 *analysis*						
≤0.2	23	21	5	3	1	0
≤0.4	15	18	7	5	4	4
≤0.6	13	11	7	6	3	13
≤0.8	7	7	5	5	8	21
≤1.0	1	0	2	0	5	45
CVB_2 *analysis*						
≤0.2	23	20	6	3	1	0
≤0.4	16	17	7	5	4	4
≤0.6	13	11	7	6	3	13
≤0.8	7	7	5	5	8	21
≤1.0	1	0	1	1	3	47

average to assess CSAs performances. We do not report the time for partitioning since the partitioning is performed once and its running time is negligible compared to the analysis time. For the same reason we do not report the time for combining the analysis described in the next step.

Step 3. In the last step we combine invariants of CSA in a way that allows us to answer our research questions. First we order the method partitions based on their average execution time. Then in order to determine all invariants computed at the point when a CSA completes, we combine all invariants from previously completed CSA with the current one. The result is aggregated invariants ordered based on the execution time of the partitions - from fastest to slowest. To compare SA and CSA invariants we use the logical equivalence relation for two invariants. To compare RD and CRD we compared their sets of reaching definition at each program location. To compare VB and CVB we evaluate implication relations between their SMT formulas, i.e, $(CVB \implies VB) \land (VB \implies CVB)$ at each program point. If the formula evaluates to true then we count it as a sound invariant for CVB. If the formula evaluates to false and the first implication evaluates to true, then CSA under-approximates the invariant of SA. All other evaluation of the formula to false indicate either a conceptual mistake in our CSA approach or a bug in our implementations. In all our experiments, we have not observed such cases.

6.2   Results

Performance. We used the ratio between runtimes of the slowest CSA partition and the full SA for each method to compare CSA and SA performances. Fig. 3 shows the histograms the ratios for each analysis implementation. The x-axes show the ratio values and the labels on top of the bars are the counts for that bar interval.

The histograms show that CRD_1 performed the worst since it has many executions with higher runtimes than RD. However, their average runtimes across 53 methods are comparable: CRD_1 is 148 ms and RD is 143 ms. This is because CRD_1 performed much better on larger methods than on smaller ones.

Even though CRD_2 has 16 method with ratios greater than 1, its average runtime is 108 ms, which makes this implementation 24% faster than RD.

Both CVB_1 and CVB_2 have few methods with ratios greater than 1.0, however those value are very close to 1.0. Among the 11 CVB_1 methods that underperformed, 6 have ratios of 1.01 and the rest have rations no greater than 1.05. For CVB_2's 8 underperforming methods, 5 of them have the ratios of 1.01, 2 have the ratios no greater than 1.05 and one has 1.28 ratio. The average runtimes across 53 methods are 6989 ms for CVB_1 and 7035 ms for CVB_2, which is 20% faster than VB's 8689 ms. Even though CVB_1 and CVB_2 have comparable performances, CVB_2 was able to compute more programs faster.

Invariants. The results for sound invariants computation are presented in Table 1 for CRD and in Table 2 for CVB. The column headers describe the two points "0", "100" and four ranges "(0,25)", "[25, 50)", "[50, 75)" and "[75, 100)" of the percentage of sound invariants of a full SA that CSA is able to compute. The row header shows the same ratios of running time of CSA to a full SA running time. The cell values represent the count of methods for which CSA is able to compute sound invariants within the given invariant range and within the given time interval. For example in Table 2 the first data row and the second data column contains value 21, which can be interpreted as such: for 21 methods CVB_1 is able to produce up to 25% of the sound invariants computed by a full VB in 20% of time of the full VB. The data in the second data row and in the last column tells us that within 40% of the full VB computational time CVB_1 is able to compute all invariants for 4 methods.

The data show that CSA can produce sound invariants faster for several methods and compute partial sound invariants for a majority of them. For example CVB computes all invariants for 21 methods within 80% of VB runtime and can produce partial sound invariants within 20% of VB runtime. Note that the histogram counts and the values in the last column might not equal. This is because CVB was able to produce the same invariant values as VB after computing only a few partitions, thus the rest of partitions compute redundant information.

The data shows that the efficiency of the CSA_1 and CSA_2 implementations depend on the analysis type. Thus, for CRD its CRD_2 performs better than CRD_1. However, for CVB analysis both implementation produce close results with CVB_2 performing slightly better than CVB_1. CRD is more sensitive to the implementation because it is a relatively fast analysis - it runs in a fraction of a second while CVB requires several minutes to complete. Overall, the second implementation of CSA that require modification of the underlying topological order algorithm is a better implementation choice.

6.3   Discussion

The results indicate that CSA allows for faster analysis, while requiring minimal modification in SA frameworks. However, the main contribution of CSA is its ability to provide partial invariants in a fraction of a time of SA. While a user

waits for a completion of all partitions to complete she can use the invariants provided earlier to check the safety properties of the program. If such property does hold, then the user has more confidence about the program correctness. However, if the property does not hold for the computed invariants then she can start investigate the cause of it. Moreover, the partition information could accelerate this task since it narrows down the set of paths that causes property violation.

7   Related Work

Besides related work on conditional analysis described in the introduction our work relates the body of research that improve the performance of SA algorithms and the accuracy of SA using program's structural information. The body of work on designing parallel SA algorithms through partitioning the program's state space started back 1990's with the work of Lee at el., [11] that partitioned program CFG into strongly connected components applying fixed point computation inside those components and then using elimination algorithm [12] to combine the data from the external nodes of those components. Albarghouthi at el., [13] investigated parallel C interprocedural analysis, where based on the reachability in the call-graph multiple method analyzed intraprocedurally in parallel. Dewey at el., [14] explores parallel analysis of JavaScript by partitioning the state space of the program into regions that can be computed in parallel and those that require synchronizations of the parallel computations, i.e., merging points of the analysis.

Another body of work identifies partitions of CFG to improve the precision of the analysis by delaying the merge of abstract values from controls flows or adding new abstract elements that exactly describe the join of two abstract elements, i.e., computing disjunctive completion of the partially ordered set. However, disjunctive completion can lead to excessively large representation of abstract values, and at some point, at least some values should be joined in order for the computation to reach its fixed point. Prior research has explored what abstract values should be joined; computational traces [15] or some other heuristic based on the CFG, such as a trace partitioning domain method [16], can provide a basis for these determinations.

Another approach is to delay the join operation by conducting incremental analysis as guided analysis [17]. In this approach, each iteration of the fixed point computation is applied to an incrementally augmented subgraph of P's CFG. For instance on the first iteration, i.e., propagating abstract values through CFG, the analysis considers one true branch of a conditional statement, and on the second iteration it would add the false branch. This approach limits the loss of precision resulting from widening operators for numerical domains, such as polyhedra that have infinite ascending chains. This incremental approach also includes a disjunctive extension when the analysis first performs fixed point computation before extending the part of the CFG's to be analyzed, i.e., successively computing invariants. An orthogonal approach is the path focusing technique [18],

which computes invariants separately for each path between two loop-free points in the CFG. Thus, each part of the CFG between entrance and exit of a loop is expanded into a set of paths. After the computation is done, then results of each path are joined.

The latest development has been in combining guided analysis and path focusing techniques [19]. Using this approach, analysis continues to evaluate paths between loop-free points encoded separately with the SMT formula. This approach allows the analysis to explore only those paths that have the potential to improve the precision of the invariants.

Our approach is complimentary to the above techniques, since a CSA for a single partition could use a parallel algorithm for computing its propagation to further improve CSA efficiency.

8   Conclusion and Future Work

In this work we introduce structurally defined conditional static analysis, formalize it in terms of standard data-flow frameworks, provide algorithms for CSA, and two distinct implementations. We evaluate the efficiency and precision of these techniques through extensive empirical study on real-world programs. The key insight is that CSA partitions a program's CFG into a subset of graphs at the conditional statements. These partitions induce a series of independent CSA executions that can run in parallel. The empirical evaluation suggest that CSA provides improvements over the full SA for a significant fraction of a program. In particular depending on the analysis around 24% of methods completed their analysis within 60% of run time required by the full SA. Moreover, CSA is able to produce partial safe invariant computations for a majority of the programs.

In the future we plan to further improve the efficiency of CSA and the confidence of the partial information that it produces. Currently CSA that follow the same path prefix compute identical information for the prefix, we plan to investigate an approach where only one analysis computes the prefix information and communicates to the rest of CSA with the common prefixes. In addition, we would like to qualify CSA's partially computed invariants into safe or underapproximating based on the partition that CSA analyzes. Thus, when a CSA computes an invariant that is marked as safe, the user should use it with the same amount of confidence as she would for the full SA.

References

1. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTREÉ analyzer. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 21–30. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-31987-0_3

2. Calcagno, C., Distefano, D.: Infer: an automatic program verifier for memory safety of C programs. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 459–465. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-20398-5_33

3. Cousot, P., Cousot, R.: Modular static program analysis. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 159–179. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45937-5_13

4. Ballabriga, C., Cass, H., Sainrat, P.: WCET computation on software components by partial static analysis. In: Junior Researcher Workshop on Real-Time Computing, pp. 15–18 (2007)

5. Naik, M., Aiken, A.: Conditional must not aliasing for static race detection. In: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, pp. 327–338. ACM, New York (2007). https://doi.org/10.1145/1190216.1190265

6. Conway, C.L., Dams, D., Namjoshi, K.S., Barrett, C.: Pointer analysis, conditional soundness, and proving the absence of errors. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 62–77. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-69166-2_5

7. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE 2012, pp. 57:1–57:11. ACM, New York (2012). https://doi.org/10.1145/2393596.2393664

8. Sherman, E., Dwyer, M.B.: Exploiting domain and program structure to synthesize efficient and precise data flow analyses (T). In: 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, 9–13 November 2015, pp. 608–618 (2015). https://doi.org/10.1109/ASE.2015.41

9. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer, Heidelberg (1999). https://doi.org/10.1007/978-3-662-03811-6

10. Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N.: Boa: a language and infrastructure for analyzing ultra-large-scale software repositories. In: Proceedings of the 35th International Conference on Software Engineering, ICSE 2013, pp. 422–431, May 2013

11. Lee, Y.-F., Marlowe, T.J., Ryder, B.G.: Performing data flow analysis in parallel. In: Proceedings of the 1990 ACM/IEEE Conference on Supercomputing, Supercomputing 1990, pp. 942–951. IEEE Computer Society Press, Los Alamitos (1990). http://dl.acm.org/citation.cfm?id=110382.110625

12. Ryder, B.G., Paull, M.C.: Elimination algorithms for data flow analysis. ACM Comput. Surv. **18**(3), 277–316 (1986). https://doi.org/10.1145/27632.27649

13. Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K.: Parallelizing top-down interprocedural analyses. In: Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2012, pp. 217–228. ACM, New York (2012). https://doi.org/10.1145/2254064.2254091

14. Dewey, K., Kashyap, V., Hardekopf, B.: A parallel abstract interpreter for javascript. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, pp. 34–45. IEEE Computer Society, Washington, DC (2015). http://dl.acm.org/citation.cfm?id=2738600.2738606

15. Holley, L.H., Rosen, B.K.: Qualified data flow problems. IEEE Trans. Softw. Eng. **7**(1), 60–78 (1981)

16. Rival, X., Mauborgne, L.: The trace partitioning abstract domain. ACM Trans. Program. Lang. Syst. **29**(5), 26 (2007)

17. Gopan, D., Reps, T.: Guided static analysis. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 349–365. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-74061-2_22

18. Monniaux, D., Gonnord, L.: Using bounded model checking to focus fixpoint iterations. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 369–385. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-23702-7_27

19. Henry, J., Monniaux, D., Moy, M.: Succinct representations for abstract interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 283–299. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33125-1_20