Computing Partially Path-Sensitive MFP Solutions in Data Flow Analyses

Komal Pathade
Tata Consultancy Services /
Indian Institute of Technology Bombay
India
komal.pathade@tcs.com

Uday P. Khedker
Indian Institute of Technology Bombay
India
uday@cse.iitb.ac.in

Abstract

Data flow analysis traverses paths in a control flow graph (CFG) representation of programs to compute useful information. Many of these paths are infeasible, i.e. they cannot arise in any possible execution. The information computed along these paths adds imprecision to the conventional Maximal Fixed Point (MFP) solution of a data flow analysis. Existing approaches for removing this imprecision are either specific to a data flow problem or involve control flow graph restructuring which has exponential complexity.

We introduce partial path-sensitivity to the MFP solution by identifying clusters of minimal infeasible path segments to distinguish between the data flowing along feasible and infeasible control flow paths. This allows us to lift any data flow analysis to an analysis over $k+1$ tuples where k is the number of clusters. Our flow function for a $k+1$ tuple shifts the values of the underlying analysis from an element in the tuple to other element(s) at the start and end of a cluster as appropriate. This allows us to maintain the distinctions where they are beneficial. Since k is linear in the number of conditional edges in the CFG, the effort is multiplied by a factor that is linear in the number of conditional edges (and is not exponential, unlike conventional approaches of achieving path sensitivity.)

We have implemented our method of computing partially path sensitive MFP for reaching definitions analysis and value range analysis of variables. Our measurements on benchmark programs show up to 9% reduction in the number of reaching definitions and up to 14% cases where the value range of a variable is smaller.

CC'18, February 24–25, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5644-2/18/02…$15.00
https://doi.org/10.1145/3178372.3179497

CCS Concepts • **Theory of computation** → *Program analysis*; • **Software and its engineering** → *Compilers*;

Keywords Data Flow Analysis, Infeasible Control Flow Path , Static Program Analysis, Maximum Fix Point Solution, Path Sensitivity

ACM Reference Format:
Komal Pathade and Uday P. Khedker. 2018. Computing Partially Path-Sensitive MFP Solutions in Data Flow Analyses. In *Proceedings of 27th International Conference on Compiler Construction (CC'18).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3178372.3179497

1 Introduction

A data flow analysis captures the runtime behaviour of the computer programs in terms of the possible information at each program point over all executions of the program. We say the analysis is *precise*, if the captured information is same as the actual possible runtime information. Such precise data flow information leads to more opportunities for code optimization and program verification.

A data flow analysis traverses *control flow paths* (*cfp*s) which are paths in the *control flow graph* (CFG) representation of programs. Hence the precision and computability of a data flow analysis depends on the nature of *cfp*s. The *cfp*s that do not represent any possible execution of the program are *infeasible*; others are *feasible*. Computing information along infeasible paths makes the solutions of data flow analysis imprecise. Apart from infeasible *cfp*s, a data flow analysis could also traverse *spurious cfp*s which are paths that result from an over-approximation of *cfp*s—they do not appear in the CFG but are effectively traversed by a specific method of data flow analysis as explained below.

Table 1 lists the conventional solutions of a data flow analysis and mentions their computability and (relative) precision by describing the nature of paths traversed in computing the solutions.

- Meet Over Feasible Paths (MoFP) captures the information reaching a program point along *feasible cfp*s only and does not merge information across *cfp*s. Since a program could have infinitely many *cfp*s, computing this solution is *undecidable*. The information

Table 1. Solutions of a data flow analysis. As we go down the rows, more entries become "Yes" leading to increased efficiency and decreased precision.

Solution	Traverses *cfps*			Merges information		Decidable	Precision
	Infeasible	Feasible	Spurious	Across *cfps*	Across program points		
MoFP	No	Yes	No	No	No	No	Most
MoP	Yes	Yes	No	No	No	No	⇓
MFP	Yes	Yes	No	Yes	No	Yes	⇓
FI	Yes	Yes	Yes	Yes	Yes	Yes	Least

defined by MoFP precisely represents the possible runtime information at each program point.

- Meet Over All Paths (MoP) captures the information reaching a program point along *all cfps*. Although the information is not merged across *cfps*, since the information reaching along infeasible *cfps* is also considered, MoP is less precise compared to MoFP. Computing MoP is also *undecidable*.
- Maximal Fix Point (MFP) differs from MoP in that the information is merged in the shared segments of *cfps*. Since only one piece of information is stored at a program point regardless of the number of paths passing through it, computing MFP is *decidable*. MFP is less precise than MoP because information across paths is merged and this over-approximation cannot distinguish between feasible and infeasible path.
- Flow Insensitive (FI) merges information across all program points by ignoring the control flow. Effectively, it over-approximates the set of *cfps*.

Empirical evidence on the Linux kernel code shows that 9-40% of the conditional statements in programs show statically detectable correlation with generating at least one infeasible *cfp* [3]. The greater the number of conditional statements in a path, the greater the probability of the path being infeasible [18]. Data along such paths causes imprecision in static analysis results. The state of the art approaches to remove this imprecision [2, 3, 10, 23, 24] are either analysis-specific [3, 10, 24] or involve CFG restructuring [2, 23] which has exponential worst case complexity in terms of number of nodes in CFG.

Our work can be seen in the mould of trace partitioning [19] which tries to maintain some disjunction based on control criteria instead of merging values indiscriminately across all paths to compute an MFP solution. However, instead of recording control flow, we lift partitioning from within an analysis to the infeasibility of control flow paths and devise an automatic approach to implement a practical trace partitioning. The main challenge in this is that it is difficult to eliminate the effect of an infeasible path when data flow values are merged which happens in MFP computation. We meet this challenge by identifying *minimal infeasible path segments* (*mips*) (computed using existing approaches [3]) and creating equivalence classes (called *clusters*) containing *cfps* that share a suffix of some prefix of a set of *mips* sharing some control flow edges. This allows us to lift any data flow analysis to a data flow analysis over $k+1$ tuples where k is the number of clusters. A careful design of edge flow function that shifts the values from an element in the tuple to other element(s) in the tuple at the start and end of a cluster allows us to maintain the distinctions where they are beneficial. Since k is linear in the number of edges in the CFG, the effort has a linear multiplication factor and is not exponential, unlike conventional approaches of achieving path sensitivity. The resulting solution is more precise than MFP (and sometimes more precise than MoP like in Figure 1).

Figure 1 shows an example of different solutions for an analysis that determines the values of variables. For MoP and MoFP, each data flow value has a constraint that represents the *cfps* along which the value reaches the program point. Since MFP and FI solutions merge the values, they compute a range of values in terms of lower and upper bound . PPMFP solution is more precise than MoP and MFP because it excludes the infeasible path reaching node n_6 (shown by double lines).

The rest of the paper is organized as follows: Section 2 describes our main idea, Section 3 presents a method for computing the PPMFP solution, Section 4 proves the soundness of PPMFP, Section 5 explains the experimental setup and discusses the results, Section 6 reviews the related literature, and Section 7 concludes the paper.

2 Our Key Idea

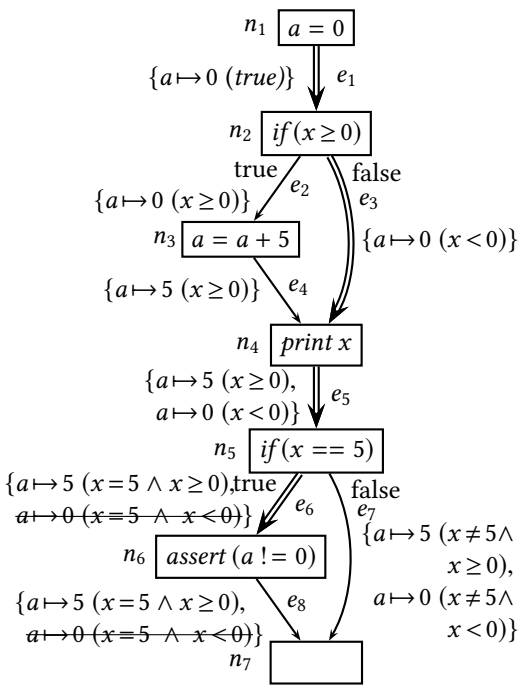
In this section, we define minimal infeasible path segments and show how they allow us to define the PPMFP solution.

2.1 Background: Infeasible Control Flow Paths

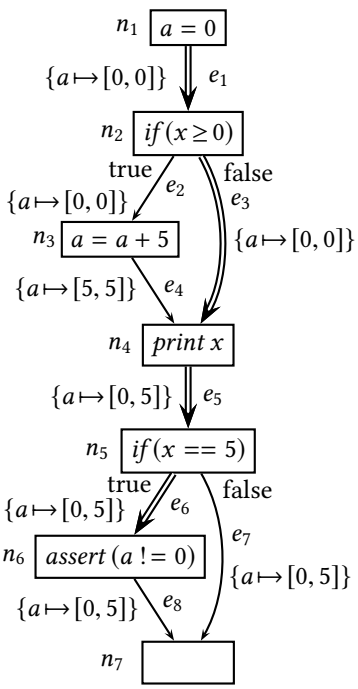
We use the following concepts related to infeasible paths [3].

A *cfp* $\rho : n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_2} n_3 \xrightarrow{e_3} \ldots \rightarrow n_k \xrightarrow{e_k} n_{k+1}$, $k \geq 1$, is an *infeasible control flow path* if n_1 is the start node of the CFG, and there is some conditional node n_i such that the subpath from n_1 to n_i of ρ is a prefix of some execution path but the subpath from n_1 to n_{i+1} is not a prefix of any execution path.[1] A path segment $\mu : n_j \xrightarrow{e_j} \ldots \rightarrow n_i \xrightarrow{e_i} n_{i+1}$ of *cfp* ρ above is a *minimal infeasible path segment* (*mips*) if it

[1]Note that the subscripts used in the nodes and edges in a path segment signify positions of the nodes and edges in the path segment and should not be taken as labels. By abuse of notation, we also use n_1, e_2 etc. as labels of nodes and edge in a control flow graph and then juxtapose them when we write path segments.



(a) The MoP solution. The MoFP solution discards the struck off values.



(b) The MFP solution. In PPMFP solution, edges e_6 and e_8 have the interval $[5, 5]$.

Figure 1. Different solutions for value analysis over an example program. The nodes in the CFG are numbered n_i whereas the edges are numbered e_i. The path reaching n_6 shown by double lines is infeasible. For simplicity we have shown the values for a alone. The FI solution merges values across all program point resulting a single interval $[0, 5]$.

The states reaching different edges are:

- $states(e_0, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq \infty\}$
- $states(e_1, \{a\}) = \{\{a \mapsto 0\}\}$
- $states(e_2, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq \infty\}$
- $states(e_3, \{a\}) = \{\{a \mapsto i\} \mid -\infty \leq i \leq \infty\}$
- $states(e_4, \{a\}) = \{\{a \mapsto i\} \mid 5 < i \leq \infty\}$

Path segment μ: $n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ is *mips* because e_4 cannot be reached from e_1 in any execution although it can be reached from e_3 in some execution.
$start(\mu) = e_1,\ inner(\mu) = e_3,\ end(\mu) = e_4$

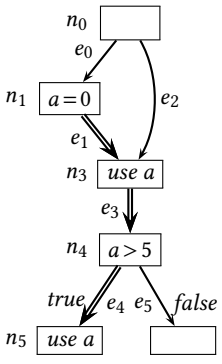


Figure 2. Illustrating minimal infeasible path segment (*mips*).

is not a subpath of any execution path but every subpath of μ is a subpath of some execution path.

We call the edges e_j, e_i as the *start* and the *end* edge of *mips* μ respectively, while all edges e_k, $j < k < i$ are called as the *inner* edges of the *mips* μ.

Figure 2 illustrates infeasible paths. Observe that the following *cfp* $\rho : n_0 \xrightarrow{e_0} n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ is infeasible but not minimal . However, its suffix μ is a *mips*

$\mu : n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ because μ is infeasible but no sub-segment of μ is infeasible. □

2.2 Defining Partially Path Sensitive MFP Solution

For a given *mips* μ, let $start(\mu)$, $inner(\mu)$, and $end(\mu)$ denote the set of *start*, *inner*, and *end* edges of μ. $start(\mu)$ and $end(\mu)$ are singleton sets. The edge in $end(\mu)$ blocks all data flow values that reach the edge in $start(\mu)$. Thus, if we define a data flow analysis that separates the data flow values that

are to be blocked, we can eliminate them thereby avoiding the effect of infeasible paths.

We define an equivalence relation $\simeq$ over the set of all *mips* which partitions the set of *mips* into *clusters* where each cluster ψ_i is the maximal set of *mips* that share the same end edge.

$$\mu \simeq \mu' \Leftrightarrow end(\mu) = end(\mu')$$

By abuse of notation, we denote the start, inner, and end edges of a cluster ψ by $start(\psi)$, $inner(\psi)$, and $end(\psi)$, respectively. The $end(\psi)$ is singleton.

2.2.1 Lifting a Data Flow Analysis to Handle Infeasible Paths

We use the following key idea to lift a data flow analysis to handle infeasible paths.

We separate the data flow values that reach a program point along *cfp*s that overlap with a *mips*, from the data flow values that do not overlap with any *mips*. This allows us to discard the values at the end of infeasible *cfp*s (because they are unreachable due to infeasibility of *cfp*s).

Assume that we have k clusters $\psi_1, \psi_2, \ldots, \psi_k$ of *mips* in the CFG. Let L denote the lattice of data flow values of our underlying data flow analysis. Then the MFP solution of our data flow analysis computes values in L using data flow variables In_n/Out_n. We lift the analysis to a product lattice $\overline{L} = L \times L \times \ldots \times L$ of arity $k+1$ and compute $\overline{In_n}/\overline{Out_n}$ values which are tuples of $k+1$ values each of which is in L. For a data flow value $\overline{In_n} = \langle d_1, d_2, \ldots, d_{k+1} \rangle$:

- Component d_i, $1 \leq i \leq k$ represents the data flow value reaching node n along *cfp*s that overlap with the *mips* cluster ψ_i. If no value along such a path reaches n, then d_i is $\top$.
- Component d_{k+1} represents the data flow values reaching node n from *cfp*s that do not overlap with any *mips*. If no value along such a path reaches n, then d_{k+1} is $\top$.

Component d_i, $1 \leq i \leq k$ is the meet of data flow values reaching node n along those *cfp*s whose suffixes are non-empty prefixes of some *mips* in cluster ψ_i. For example in Figure 3, we have a single *mips* $n_2 \xrightarrow{e_3} n_4 \xrightarrow{e_5} n_5 \xrightarrow{e6} n_6$ and hence a single cluster. At node n_4, *cfp* $\rho : n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_3} n_4$ has a suffix $\sigma : n_2 \xrightarrow{e_3} n_4$ which is a prefix of the *mips* $n_2 \xrightarrow{e_3} n_4 \xrightarrow{e_5} n_5 \xrightarrow{e6} n_6$. These paths are identified by $cpaths(n, \psi_i)$ defined below where (a) $\rho' \cdot \sigma$ denotes concatenation of ρ' and σ, (b) $paths(n)$ denotes the set of all *cfp*s reaching node n, and (c) $prefixes(\psi)$ denotes the set of non-empty prefixes of the *mips* in ψ.

$$cpaths(n, \psi) = \{\rho \mid \rho \in paths(n), \exists\, \sigma \in prefixes(\psi) \text{ such that } \rho \equiv \rho' \cdot \sigma\}$$

$cpaths(n, \psi_i)$ is non-empty when node n is contained in cluster ψ_i (i.e. an in-edge $m \rightarrow n$ is a part of some *mips* $\mu \in \psi_i$). If node n is not contained in cluster ψ_i (i.e. no in-edge $m \rightarrow n$ is a part of any *mips* $\mu \in \psi_i$) then $cpaths(n, \psi_i) = \emptyset$ and $d_i = \top$.

An important observation about d_i is that it remains confined to ψ_i because it is blocked by the edge in $end(\psi_i)$. The only way it can go out is through those successor edges of start or inner edges of ψ_i which do not belong to ψ_i, at such edges d_i is added d_{k+1} and set to $\top$ afterwards.

2.2.2 Defining PPMFP Solution

The PPMFP solution also consists of values in L and is computed in data flow variables $\overline{\overline{In_n}}/\overline{\overline{Out_n}}$. They are computed from $\overline{In_n}/\overline{Out_n}$ using the $fold_k$ operation defined on $k+1$ tuples:

$$\overline{\overline{In_n}} = fold_k(\overline{In_n}) \tag{1}$$

$$\overline{\overline{Out_n}} = fold_k(\overline{Out_n}) \tag{2}$$

$$fold_k(\overline{X}) = \prod_{i=1}^{k+1} d_i \tag{3}$$

$$fold_k(\overline{X}|_s) = \prod_{\psi_i \in s} d_i \quad \text{where} \quad \overline{X} \equiv \langle d_1, d_2, \ldots, d_{k+1} \rangle \tag{4}$$

$\overline{\overline{In_n}}/\overline{\overline{Out_n}}$ values represent the data flow information reaching node n along all feasible *cfp*s. They exclude the information along infeasible *cfp*s.

Figure 3 illustrates PPMFP solution for our motivating example of Figure 1. For node n_6, the lifted data flow analysis separates the data flow values reaching along paths that have *mips* μ: $n_2 \xrightarrow{e_3} n_4 \xrightarrow{e_5} n_5 \xrightarrow{e6} n_6$ as a suffix (data flow value $\top$ because the path is infeasible) from the data flow values reaching along paths that do not have μ as a suffix. Thus PPMFP has a more precise data flow value for n_6. We show how this can be achieved in the next section.

3 Computing Partially Path Sensitive MFP Solution

We first describe generic data flow equations and then customize them to our needs. Let $In_n, Out_n \in L$ be the data flow values computed by a data flow analysis where L is the meet semi-lattice containing a $\top$ element—if there is no natural $\top$, we add an artificial $\top$ value.

The desired solution of the analysis is the MFP solution of the set of data flow equations (5) and (6) below that computed over the CFG of a procedure, say p. For simplicity, we assume a forward analysis. *BI* represents the *boundary information* reaching procedure p from its callers. The meet operator $\sqcap$ computes the glb of elements in L. Node flow functions f_n and edge flow function $g_{m \rightarrow n}$ compute the effect of node n and edge $m \rightarrow n$ respectively. Usually, the

Paths, *mips*, and clusters:

- Single *mips* μ: $n_2 \xrightarrow{e_3} n_4 \xrightarrow{e_5} n_5 \xrightarrow{e6} n_6$
- Single cluster $\psi_1 = \{\mu\}$, k=1
- $start(\mu) = start(\psi_1) = \{e_3\}$
- $inner(\mu) = inner(\psi_1) = \{e_5\}$
- $end(\mu) = end(\psi_1) = \{e_6\}$

Lattices and data flow values

- $L = \big\{\{a \mapsto [i,j]\} \mid -\infty \le i \le j \le +\infty\big\}$
- $\top = \{a \mapsto [+\infty, -\infty]\}$
- $\overline{L} = L \times L$
- $\overline{In}_{n_6} = \langle \top, \{a \mapsto [5,5]\}\rangle$
- $\overline{\overline{In}}_{n_6} = \top \sqcap \{a \mapsto [5,5]\} = \{a \mapsto [5,5]\}$
- $In_{n_6} = \{a \mapsto [0,5]\}$ (Figure 1(b))

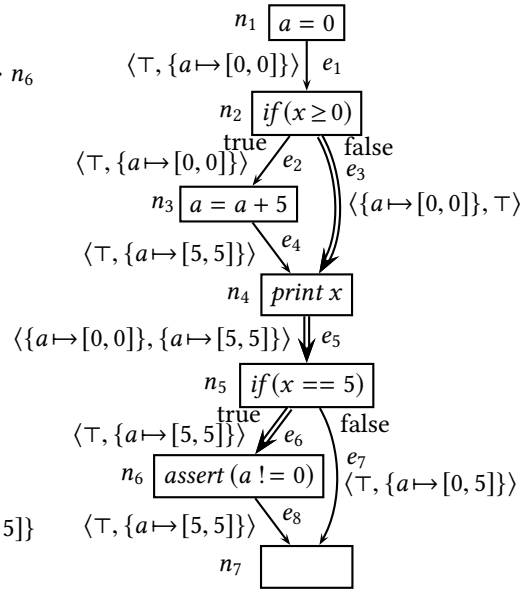


Figure 3. Computing PPMFP for the example of Figure 1. The lifted data flow analysis computes MFP solution $\overline{In}_n / \overline{Out}_n$ as pairs of values. The first element represents the data flow values reaching node n along *cfp*s whose suffix is a prefix of *mips* μ. The second element represents the data flow values reaching node n from all other feasible paths.

edge flow functions are identity.

$$In_n = \begin{cases} BI & n = Start_p \\ \bigsqcap_{m \in pred(n)} g_{m \to n}(Out_m) & \end{cases} \quad (5)$$

$$Out_n = f_n(In_n) \quad (6)$$

Assume that we are given k clusters $\psi_1, \psi_2, \ldots, \psi_k$ from the program. Apart from $start(\psi_i)$, $inner(\psi_i)$, and $end(\psi_i)$, we also define the concept of the *frontier* of a cluster to contain the edges that are not contained in the cluster but are successor edges of the start or an inner edge of the cluster:

$$frontier(\psi_i) = \big\{m \to n \mid m \to n \notin (start(\psi_i) \cup inner(\psi_i) \cup end(\psi_i)), \exists\, (l \to m) \in (start(\psi_i) \cup inner(\psi_i))\big\}$$

For a data flow analysis specified using equations 5 and 6, we compute $\overline{In}/\overline{Out}$ by solving the following equations where $\overline{BI}$ is a tuple of k+1 BI's.

$$\overline{In}_n = \begin{cases} \overline{BI} & n = Start_p \\ \bar{\bigsqcap}_{m \in pred(n)} \overline{g}_{m \to n}(\overline{Out}_m) & \text{otherwise} \end{cases}$$

$$\overline{Out}_n = \overline{f}_n(\overline{In}_n)$$

Let $\overline{X}$ denote a tuple $\langle d_1, d_2, \ldots, d_{k+1}\rangle$. Then the meet $\overline{\sqcap}$ is defined as follows

$$\overline{X}_a \,\overline{\sqcap}\, \overline{X}_b = \langle d_1^a \sqcap d_1^b, d_2^a \sqcap d_2^b, \ldots, d_{k+1}^a \sqcap d_{k+1}^b\rangle \quad (7)$$

The node flow function $\overline{f}_n$ for node n is defined as follows:

$$\overline{f}_n(\overline{X}) = \langle f_n(d_1), f_n(d_2), \ldots, f_n(d_{k+1})\rangle \quad (8)$$

The most ingenious part of this lifting is defining the edge flow functions $\overline{g}_{m \to n}$ to handle *mips* appearing in *cfp*s:

$$\overline{g}_{m \to n}(\overline{X}) = \bigsqcap_{i=1}^{k+1} adjustFlow(m \to n, i, \overline{X}) \quad (9)$$

This flow function adjusts the data flow across clusters depending upon whether the edge $m \to n$ is a start edge, an end edge, a frontier edge, an inner edge, or is outside of any cluster. Since there are multiple clusters, these situations are not mutually exclusive, i.e., an edge could be a start edge of some cluster ψ_i, an end edge of some other cluster ψ_j, and may be the frontier of a third cluster ψ_l.

We use the following notation to define *adjustFlow*. We write a tuple with three values separated by ellipses (...) and the middle value corresponds to the i^{th} component of the tuple. With a bit of abuse of notation, we define the sets of clusters involving an edge e in a specific role as follows:

$$C(start, e) = \big\{\psi_j \mid e \in start(\psi_j)\big\}$$

$$C(end, e) = \big\{\psi_j \mid e \in end(\psi_j)\big\}$$

$$C(\neg end, e) = \big\{\psi_j \mid e \notin end(\psi_j)\big\}$$

With the above notation, we define *adjustFlow* by considering the following cases which are mutually exclusive and exhaust all possibilities.

1. Edge e is a start edge some clusters (i.e.$C(start, e) \neq \emptyset$). In this case, if $\psi_i \in C(start, e)$ then data flow value in the i^{th} component is going to be blocked by the edge in $end(\psi_i)$, thus it must be separated from the common information and must remain confined to ψ_i (unless there is a *frontier* edge). Hence, at the start edge of the cluster, the data flow information in the i^{th} component includes all information reaching from outside of the cluster except for those clusters that end with e. This information is computed by applying the underlying edge flow function to the meet of appropriate data flow values as defined below. All other components are $\top$.

$$adjustFlow(e, i, \overline{X}) = \tag{10}$$

$$\begin{cases} \langle \top, \ldots, g_e(fold_k(\overline{X}\,|_Y)), \ldots, \top \rangle & \psi_i \in C(start, e) \\ \langle \top, \ldots, \top, \quad \ldots, \top \rangle & \text{otherwise} \end{cases}$$

where $Y = C(\neg end, e)$

2. Edge e is not a start edge of any cluster (i.e., $C(start, e) = \emptyset$). In this case, we examine whether e ends a cluster, is the frontier of a cluster, or is outside of all clusters. The three cases are defined below followed by their explanations:

$$adjustFlow(e, i, \overline{X}) = \tag{11}$$

$$\begin{cases} \langle \top, \ldots, \top, \quad \ldots, \top \quad \rangle & e \in end(\psi_i) \\ \langle \top, \ldots, \top, \quad \ldots, g_e(d_{k+1} \sqcap d_i) \rangle & e \in frontier(\psi_i) \\ \langle \top, \ldots, g_e(d_i), \ldots, g_e(d_{k+1}) \quad \rangle & \text{otherwise} \end{cases} \tag{12}$$

   a. $e \in end(\psi_i)$. At the end edge of the cluster ψ_i, no cluster specific information reaches node n so all components are $\top$.
   In Figure 3,
   $\overline{g}_{n_5 \to n_6}(\langle\{a \mapsto [0,0]\}, \{a \mapsto [5,5]\}\rangle) = \langle \top, \{a \mapsto [5,5]\}\rangle$
   b. $e \in frontier(\psi_i)$. In this case, $end(\psi_i)$ is not reached and hence all *cfp*s are feasible *cfp*s. Thus , the data flow information from the i^{th} component is merged with $k+1$ component
   In Figure 3,
   $\overline{g}_{n_5 \to n_7}(\langle\{a \mapsto [0,0]\}, \{a \mapsto [5,5]\}\rangle) = \langle \top, \{a \mapsto [0,5]\}\rangle$
   c. In all other cases (whether e is an *inner* edge or whether e is outside of the cluster), the data flow information in different clusters remains distinct. For ψ_i, the components i and $k+1$ are passed along by applying the underlying edge flow function; all other components are $\top$.
   - An example of an *inner* edge, in Figure 3,
     $\overline{g}_{n_4 \to n_5}(\langle\{a \mapsto [0,0]\}, \{a \mapsto [5,5]\}\rangle)$
     $= \langle\{a \mapsto [0,0]\}, \{a \mapsto [5,5]\}\rangle$.
   - An example of an edge outside a cluster, in Figure 3,
     $\overline{g}_{n_1 \to n_2}(\langle \top, \{a \mapsto [0,0]\}\rangle) = \langle \top, \{a \mapsto [0,0]\}\rangle$

4 Proof of Soundness

We prove that edge flow function $\overline{g}_e$ computes sound information at all program points. Before that, we prove some important properties of fold and cases under which the properties hold.

Let $Y \equiv C(\neg end, e)$

$\overline{X} \equiv \langle d_1, d_2, \ldots, d_{k+1}\rangle$

Lemma 4.1. *If e is not end edge of any cluster (i.e.,$C(end, e) = \phi$) then* $fold_k(\overline{X}\,|_Y) = fold_k(\overline{X})$

Proof.

$$Given : C(end, e) = \phi\ ,\ C(\neg end, e) = \{\psi_1, \psi_2, .., \psi_{k+1}\} \tag{13}$$

$$LHS : fold_k(\overline{X}\,|_Y) = \bigsqcap_{\psi_i \in Y} d_i \quad \ldots \text{ from } 4 \tag{14}$$

$$= \bigsqcap_{i=1}^{k+1} d_i \quad \ldots \text{ from } 13, 14$$

$$= fold_k(\overline{X}) \quad \ldots \text{from } 3 \tag{15}$$

□

Lemma 4.2. *If e is start edge of some clusters (i.e.,$C(start, e) \neq \phi$) then* $fold_k(\overline{g}_e(\overline{X})) = g_e(fold_k(\overline{X}\,|_Y))$

Proof.

$$\begin{aligned} LHS &= fold_k(\overline{g}_e(\overline{X})) \\ &= fold_k(\bigsqcap_{i=1}^{k+1} adjustFlow(e, i, \overline{X})) \quad \ldots \text{from } 9 \\ &= fold_k(\bigsqcap_{\psi_i \in C(start,\, e)} adjustFlow(e, i, \overline{X})) \quad \ldots \text{from } 10 \\ &= g_e(fold_k(\overline{X}\,|_Y)) \quad \ldots from \text{ reflexivity of } \sqcap \text{ and} 10 \end{aligned}$$

□

Lemma 4.3. *If e is neither start nor end edge of any cluster (i.e.,$C(start, e) = \phi$,$C(end, e) = \phi$) then* $fold_k(\overline{g}_e(\overline{X})) \sqsupseteq g_e(fold_k(\overline{X}))$

Proof.

$$\begin{aligned} LHS &= fold_k(\overline{g}_e(\overline{X})) \\ &= fold_k(\bigsqcap_{i=1}^{k+1} adjustFlow(e, i, \overline{X})) \\ &\sqsupseteq fold_k(g_e(\bigsqcap_{i=1}^{k+1} d_i)) \end{aligned}$$

from monotonicity ; $g_e(a \sqcap b) \sqsubseteq g_e(a), g_e(b)$; and substituting $C(end, e) = \phi$ in 11

$\sqsupseteq g_e(fold_k(\overline{X}))$ reflexivity of $\sqcap$

□

In following lemma, we prove that edge flow function $\overline{g}_e$ computes sound information at all edges that are not end edge of any cluster.

Lemma 4.4. *If e is not end edge of any cluster (i.e.,*$C(end, e) = \phi$*) then*
$X \sqsubseteq fold_k(\overline{X}) \implies g_e(X) \sqsubseteq fold_k(\overline{g}_e(\overline{X}))$

Proof. We divide the proof in following two cases:
Case1: e is start edge of some clusters
from lemma 4.1 and lemma 4.2 we get
$fold_k(\overline{g}_e(\overline{X})) = g_e(fold_k(\overline{X}))$
Case2: e is not start edge of any cluster
from lemma 4.1 and lemma 4.3 we get
$fold_k(\overline{g}_e(\overline{X})) \sqsupseteq g_e(fold_k(\overline{X}))$

$LHS : X \sqsubseteq fold_k(\overline{X})$
$\implies g_e(X) \sqsubseteq g_e(fold_k(\overline{X}))$ monotonicity property
$\implies g_e(X) \sqsubseteq fold_k(\overline{g}_e(\overline{X}))$
from case 1 and 2 and transitivity of $\sqsubseteq$
$\implies RHS$

□

If e is end edge of cluster i Then ψ_i includes only that information reaching along mips ending at edge e because of following considerations: 1) Information is added to ψ_i only when a start edge of ψ_i is encountered. 2) information is kept separate at inner edges of ψ_i and is blocked only at end edge 3)if there is a frontier edge between start and end then information is merged with d_{k+1} and ψ_i is set to $\top$.

In the particular case of overlapping clusters, when the start edge of *mips* of one cluster overlaps with the start or an inner edge of *mips* of another cluster then our approach is not be able to remove the information arrived at from infeasible paths associated with one of the *mips*. However the result is still sound.

Complexity Analysis: For a program with N nodes in its control flow graph representation, the complexity of computation of the MFP solution of a data flow analysis is N^2 [15]. If the number of clusters in program is k then the time and space complexity of computing the PPMFP solution will be equivalent to that of doing $k+1$ simultaneous data flow analyses which is $O(N^2 * (k + 1))$.

Since end edges of *mips*'s are always labeled edges [3], it can be seen that $k \leq E_l$, where E_l is total number of labeled edges in the program, in the worst case each labeled edge of the program will be the end edge of one cluster leading to a total of E_l clusters in the program. So the complexity becomes $O(N^2 * E_l)$.

5 Experiments and Results

This section describes our experiments and empirical observations.

5.1 Setup, Benchmarks, and Experiments

We performed our experiments on an Intel core i7, 64 bit machine, with 8GB RAM running Windows 7, and having clock speed of 2.4 GHz.

We used 10 C programs of up to 73kLoC from 7 SPEC CPU-2006 benchmarks and 3 industry code bases to cover a broad spectrum of coding patterns as shown in Table 2. The industry programs are real life embedded applications: Battery Control Module (BCM), Embedded Industry Code (EIC), and Car Navigation Module (CNM). The 10 selected benchmark programs contain up to 2,900 function calls. The number of clusters of minimal infeasible path segments (clusters) range from 0 for *specrand* to 1.5K for *sjeng* benchmark.

A key feature of the PPMFP solution is that its computation is orthogonal to data flow analysis. To illustrate the same, we performed experiments on two different data flow analyses namely reaching definitions analysis [15] and value range analysis of variables [16, 20].

Further, we evaluate the impact of our technique on these analyses in two ways: 1) first we measure the improvement at all the program points in the reaching definitions and range analyses, then, 2) for the reaching definitions analysis we check the reduction in the definitions of each variable only at its use points i.e. reduction in def-use pairs, while, for range analysis we check the improvement in the results of static program analysis tool *TCS Embedded Code Analyzer (TCS-ECA)* [17], which, verifies program properties like *zero division, array index out of bounds, dead code* using the range analysis.

5.2 Precision Gain in Reaching Definitions Analysis

Figure 4 depicts the increase in the precision of the reaching definitions analysis after applying our idea. For each program point p, we count the number of definitions reaching p and then add them up. The precision gain is in terms of reduction in this overall number. The actual numbers are presented in Table 2.

The *specrand* benchmark does not contain any infeasible path (clusters) so no improvement was observed in the results. While on the rest of the benchmarks, 0.1 to 9% definitions were found to be reaching along the infeasible paths (i.e. reaching the end edges of clusters of *mips*). Our method removed these definitions. This reduction in the reaching definitions helps all analyses that use the result of reaching definitions analysis.

We found 0-3% less def-use pairs in the PPMFP solution of reaching definitions analysis. In software testing it is important to cover each def-use pair by at least one test case, so 3% fewer def-use pairs means we have fewer test cases to

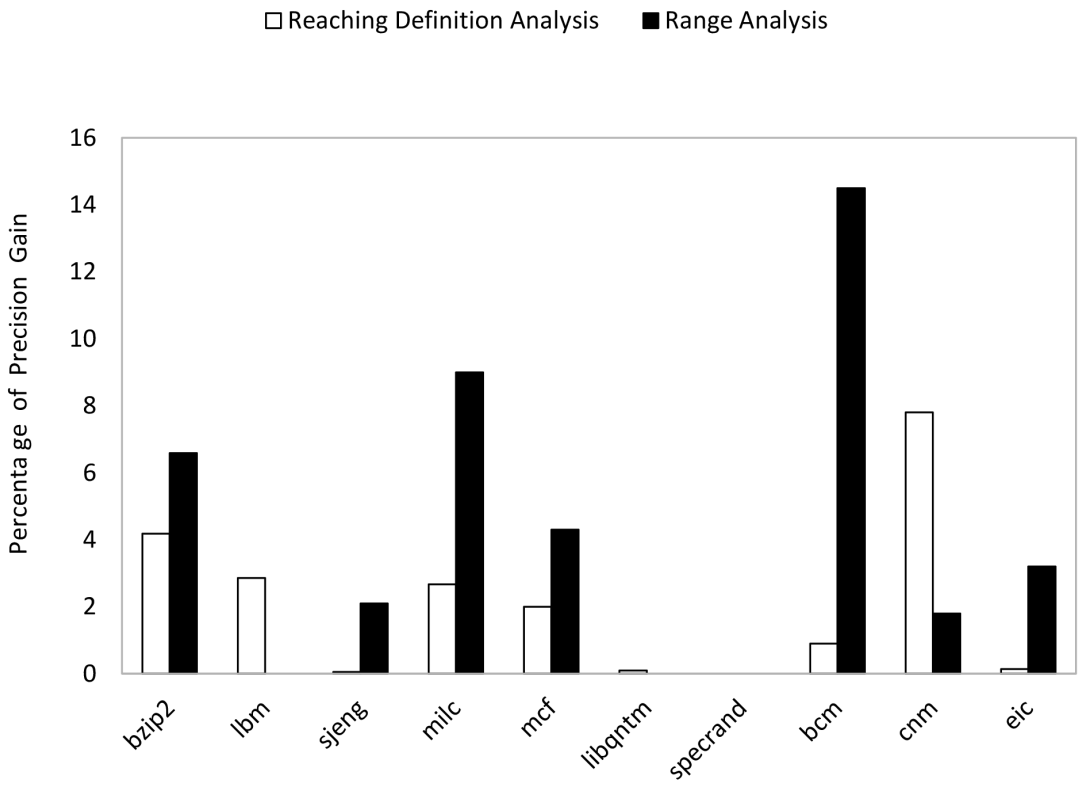


Figure 4. Precision gain in PPMFP over MFP. The precision is affected by type of data flow analysis, number and structure of clusters. **Note**: If some of the bars appear empty in a print out, please print using Acroread. A bug in Evince printing affects the pictures some times.

generate and test, which is a significant saving in terms of effort.

5.3 Precision Gain in Range Analysis

Figure 4 also depicts the increase in the precision of the range analysis. We used k path-sensitive range analysis proposed by Shrawan et.al [16] as a baseline for comparison, with $k = 1$. For each program point p, we count the number of variables which had shorter value range compared to the MFP solution of range analysis at p and then add them up. The actual numbers are presented in Table 3.

We found that the range of 0-14% variables was reduced at various program points. We found the following benefits of using the PPMFP solution of range analysis in program verification tool TCS-ECA: 1) Additional, 14, 124 and 216 program statements were identified as *dead code* on *sjeng, milc, bzip2* benchmarks respectively. This eliminates the effort required for checking correctness of these statements. 2) Two array accesses which earlier reported as possible *array index out of bound* on *milc* benchmark, were reported safe afterwords. 3) no improvement was found in *zero division* property verification.

5.4 Performance Measurements

Table 4 shows the time and memory requirement of MFP and PPMFP solutions. Preprocessing indicates time required for detecting minimal infeasible path segments from the program. This is one time effort for one benchmark. The theoretical complexity of preprocessing is quadratic in the number of nodes in the control flow graph[3].

We found the time required to compute the PPMFP solution was 1-3 times that of the time required to compute the MFP solution. Memory requirement was 1-4 times that of the MFP solution.

Table 4. Performance PPMFP vs MFP

Benchmarks	Time (sec)					Memory (MB)				
	Prepro-cessing	Reaching Definition Analysis		Range Analysis		Prepro-cessing	Reaching Definition Analysis		Range Analysis	
		MFP	PPMFP	MFP	PPMFP		MFP	PPMFP	MFP	PPMFP
bzip2	24	15	43	15	31	85	46	146	37	114
lbm	2	1	2	2	2	17	15	18	17	17
sjeng	164	100	127	44	79	187	46	357	63	285
milc	25	6	12	15	18	75	36	94	52	98
mcf	5	2	3	3	5	18	15	21	17	21
libqntm	10	3	6	6	8	28	18	32	23	32
specrnd	1	0.6	1	0.7	1	10	10	10	12	10
BCM	8	2	3	5	7	48	40	50	64	51
CNM	14	3	5	9	11	45	22	51	31	53
EIC	35	8	15	48	92	154	174	206	271	457

Table 2. Increased Precision in Reaching Definition Analysis: 1) MFP and PPMFP refer to size of set of all reaching definitions computed using ($In_n \cup Out_n$) and ($\overline{\overline{In_n}} \cup \overline{\overline{Out_n}}$) over all program nodes n, respectively 2) Benchmarks suffixed by * are the industry code bases, whereas all other are the SPEC benchmarks.

Benchmark Properties				Reaching Definition Analysis		
				#definitions		
Name	KLOC	#callsites	#clusters	MFP	PPMFP	#reduction (%)
bzip2	16	662	712	1469090	1407649	61441(4.2)
lbm	2.3	70	60	3734	3657	107(2.9)
sjeng	27	1354	1504	922089	917172	4917(0.5)
milc	30	1575	685	218565	205834	12731(5)
mcf	5.4	80	26	28278	28213	65(0.2)
libqntm	8.7	521	98	42551	38417	4134(9)
specrnd	0.1	11	0	74	74	0(0)
BCM*	74	318	65	8641	8126	515(5)
CNM*	52	588	181	36596	33706	2890(8)
EIC*	13	2946	479	4116468	4110452	6016(0.1)

Table 3. Increased Precision in Range Analysis

Benchmark	#variables		
	with reduced range	Total	%reduction
bzip2	18	273	6.59
lbm	0	71	0
sjeng	7	325	2.1
milc	16	177	9
mcf	2	46	4.3
libqntm	0	112	0
specrnd	0	14	0
BCM	14	96	14.5
CNM	4	220	1.8
EIC	6	187	3.2

5.5 A Summary of Empirical Observations

Our empirical observations confirm that PPMFP solution is a sweet spot between two extremes: efficient but imprecise MFP solution on the one hand and extremely inefficient but very precise path-sensitive solution. We get up to 14% precision improvement over MFP solution with upto 3x performance degradation in the analysis time.

6 Related Work

Various attempts have been made for path-sensitive analysis in literature [6, 8–11, 24]. A completely path sensitive analysis has been tried where each path information is separately tracked along with path constraints. If the path constraint is

found unsatisfiable, the path information is not propagated further. This approach does not scale to most practical programs due to presence of loops and recursion leading to an unbounded number of program paths combined with possibly infinite data flow lattices.

Selective path sensitive analyses exist in literature [5, 6, 8–11, 23, 24]. They use selective path constraints governed by various heuristics for deciding which path information should be kept separately and which path information can be approximated at join points. At join points, the data flow information with the same path constraints is merged.

Das et.a, proposed a property specific path sensitive analysis [6] in which the information separation is maintained only for those paths along which property related behavior changes. Dhurjati et.al proposed an iterative path sensitive analysis [8] in which the precision is improved by finding and adding more effective predicates to the data flow analysis in successive iterations. Hampapuram et.al performed a symbolic path simulation approach [11] in which they executed each path symbolically and propagated the information along these paths. If a path is found infeasible in simulation its information is not propagated. A lightweight decision procedure is used to check the feasibility of a path during path simulation.

Xie et.al proposed an approach to detect memory access errors [24]. They use precise path sensitive analysis only for constraints and variables which directly or indirectly affect some memory accesses. Dillig et.al introduced the concept of observable and unobservable variables to improve the scalability of path sensitive analysis [9]. Dor et.al proposed a precise path sensitive value analysis approach using value bit vectors [10].

Infeasible control flow paths is a property of programs instead of any particular data flow analysis. We use this observation to first detect infeasible paths in programs and propose an approach to improve precision of any data flow analysis, unlike above approaches which remove the impact of infeasible paths from a particular data flow analysis.

Presence of the infeasible paths in the programs is well known, Hedley et al. presented a detailed analysis of the causes and effects of the infeasible paths [13] in programs. Malevris et al. observed that the greater the number of conditional statements contained in a path, the greater the probability of the path being infeasible [18]. Bodik et al. found that around 9%-40% of the conditional statements in programs show statically detectable correlation with infeasible control flow paths [3]

Many approaches detect infeasible program paths [1–5, 7, 18, 21, 22, 25, 26]. For most of these approaches, the essence is to detect infeasible control flow paths by analyzing the correlations of different conditional statements or between a conditional and an assignment statement.

In our work, we used the algorithms proposed by Bodik et al. for detecting infeasible paths [3] with some modifications. Our work differs from their work in the way we use the information after infeasible paths are detected. They have applied their work for improving the precision of traditional def-use analysis.

In other work, Bodik et al. propose an interprocedural version of infeasible path detection [2] and its application to improve the data flow precision. However, they use control flow graph restructuring which is not done in our approach. The graph restructuring is undesirable as its complexity is exponential.

Chen et al. propose an algorithm for detecting infeasible paths incorporating the branch correlations analysis [22] into the data flow analysis technique, which also improve the precision of traditional data flow analysis. However, their technique cannot be generically applied to all data flow analyses.

Aditya et al. proposed an approach to improve data flow precision by eliminating destructive merges from control flow graph [23] through control flow graph restructuring. However, they did not address the imprecision added by infeasible control flow paths.

The trace partitioning approaches [12, 14, 19] partition program traces where values of some variable or conditional expression differ in two selected traces. The variables or expressions are selected based on the alarms to be analyzed or some heuristic. It is similar to our approach because it keeps data flow values separately for each partition. However, our approach does not rely on alarms, does not need the designer of an analysis to divine a suitable heuristic, does not record control flow, and is not restricted to a particular analysis. We lift partitioning from within an analysis to the infeasibility of control flow paths which is a fundamental property of control flow paths independently of an analysis. This allow us to devise an automatic approach to implement a practical trace partitioning. An interesting aspect of our approach is that although it is oblivious to any analysis, it can be seen as dynamic partitioning that uses abstract values being computed by an analysis.

7 Conclusions

We have defined the concept of a partially path sensitive MFP solution that is more precise than the MFP solution in the presence of infeasible *cfp*s. In some cases, it can also be more precise than the MoP solution. This precision is achieved by identifying the minimal number of distinctions that need to be made between data flow values reaching a program along different *cfp*s. Towards, this end, we use the concept of minimal infeasible path segments that tell us which information is blocked by an infeasible *cfp* and where. This allows us to lift any data flow analysis to a data flow analysis over $k + 1$ tuples (or $k + 1$ parallel, but interacting

data flow analyses). A careful design of edge flow function that shifts the values from an element in the tuple to other element(s) in the tuple at the start and end of a *mips* allows us to maintain the distinctions where they are beneficial. Since the number of clusters of *mips* is linear in the total number of conditional edges in the CFG, the effort has a linear multiplication factor and is not exponential, unlike conventional approaches of achieving path sensitivity. Our experiments on reaching definition analysis and range analysis show a precision gain up to 9%, and 14%, respectively with up to 3x performance degradation in terms of the analysis time. Thus, PPMFP solution is a sweet spot between two extremes: efficient but imprecise MFP solution on the one hand and extremely inefficient but very precise path-sensitive solution.

Infeasible path is a property of a program instead of a data flow analysis. However, we cannot simply optimize program to remove infeasible paths because of its exponential complexity. We found a middle path where, a data flow analysis is lifted using the infeasible path information from program to achieve better results in additional time linearly proportional to number of conditional edges in worst case. In future, more such program properties can be identified and used to improve data flow precision.

Acknowledgments

We wish to thank our shepherd and anonymous referees for their suggestions for improvements.

References

[1] Antonia Bertolino and Martina Marré. 1994. Automatic generation of path covers based on the control flow analysis of computer programs. *IEEE Transactions on Software Engineering* 20, 12 (1994), 885–899.

[2] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. 1997. Interprocedural conditional branch elimination. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 146–158.

[3] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. 1997. Refining data flow information using infeasible paths. In *Software EngineeringâĂŤESEC/FSE'97*. Springer, 361–377.

[4] Paulo Marcos Siqueira Bueno and Mario Jino. 2000. Identification of potentially infeasible program paths by monitoring the search for test data. In *Automated Software Engineering, 2000. Proceedings ASE 2000. The Fifteenth IEEE International Conference on*. IEEE, 209–218.

[5] Ting Chen, Tulika Mitra, Abhik Roychoudhury, and Vivy Suhendra. 2007. Exploiting branch constraints without exhaustive path enumeration. In *OASIcs-OpenAccess Series in Informatics*, Vol. 1. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik.

[6] Manuvir Das, Sorin Lerner, and Mark Seigle. 2002. ESP: Path-sensitive program verification in polynomial time. In *ACM Sigplan Notices*, Vol. 37. ACM, 57–68.

[7] Mickaël Delahaye, Bernard Botella, and Arnaud Gotlieb. 2010. Explanation-based generalization of infeasible path. In *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 215–224.

[8] Dinakar Dhurjati, Manuvir Das, and Yue Yang. 2006. Path-sensitive dataflow analysis with iterative refinement. In *International Static Analysis Symposium*. Springer, 425–442.

[9] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 270–280.

[10] Nurit Dor, Stephen Adams, Manuvir Das, and Zhe Yang. 2004. Software validation via scalable path-sensitive value flow analysis. In *ACM SIGSOFT Software Engineering Notes*, Vol. 29. ACM, 12–22.

[11] Hari Hampapuram, Yue Yang, and Manuvir Das. 2005. Symbolic path simulation in path-sensitive dataflow analysis. In *ACM SIGSOFT Software Engineering Notes*, Vol. 31. ACM, 52–58.

[12] Maria Handjieva and Stanislav Tzolovski. 1998. Refining static analyses by trace-based partitioning using control flow. In *SAS*, Vol. 98. Springer, 200–214.

[13] David Hedley and Michael A Hennell. 1985. The causes and effects of infeasible paths in computer programs. In *Proceedings of the 8th international conference on Software engineering*. IEEE Computer Society Press, 259–266.

[14] L Howard Holley and Barry K Rosen. 1981. Qualified data flow problems. *IEEE Transactions on Software Engineering* 1 (1981), 60–78.

[15] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data flow analysis: theory and practice*. CRC Press.

[16] Shrawan Kumar, Bharti Chimdyalwar, and Ulka Shrotri. 2013. Precise range analysis on large industry code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 675–678.

[17] Tata Consultancy Services Ltd. 2017. TCS Embedded Code Analyzer (TCS ECA). (2017).

[18] N Malevris, DF Yates, and A Veevers. 1990. Predictive metric for likely feasibility of program paths. *Information and Software Technology* 32, 2 (1990), 115–118.

[19] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*. Springer, 5–20.

[20] Steven S Muchnick and Neil D Jones. 1981. *Program flow analysis: Theory and applications*. Vol. 196. Prentice-Hall Englewood Cliffs, New Jersey.

[21] Minh Ngoc Ngo and Hee Beng Kuan Tan. 2007. Detecting large number of infeasible paths through recognizing their patterns. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 215–224.

[22] Chen Rui. 2006. *Infeasible path identification and its application in structural test*. Ph.D. Dissertation. Beijing: Institute of Computing Technology of Chinese Academy of Sciences.

[23] Aditya Thakur and R Govindarajan. 2008. Comprehensive path-sensitive data-flow analysis. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 55–63.

[24] Yichen Xie, Andy Chou, and Dawson Engler. 2003. Archer: using symbolic, path-sensitive analysis to detect memory access errors. *ACM SIGSOFT Software Engineering Notes* 28, 5 (2003), 327–336.

[25] Rui Yang, Zhenyu Chen, Baowen Xu, W Eric Wong, and Jie Zhang. 2011. Improve the effectiveness of test case generation on EFSM via automatic path feasibility analysis. In *High-Assurance Systems Engineering (HASE), 2011 IEEE 13th International Symposium on*. IEEE, 17–24.

[26] Xiaotong Zhuang, Tao Zhang, and Santosh Pande. 2006. Using branch correlation to identify infeasible paths for anomaly detection. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 113–122.