Path Sensitive MFP Solutions in Presence of Intersecting Infeasible Control Flow Path Segments

Komal Pathade
komal.pathade@tcs.com
TCS Research/ IIT Bombay
India

Uday P. Khedker
uday@cse.iitb.ac.in
IIT Bombay
India

ABSTRACT

Data flow analysis computes Maximum Fix Point (MFP) solution which represents an over approximation of the data reaching a program point along all control flow paths (CFPs). Some of these CFPs may be infeasible; meaning, the necessary pre-condition for execution of CFP is not satisfiable in any run of the program. Approximations that do not discern data along infeasible CFPs may lead to imprecision, because they include spurious information.

Recent methods progressively separate the data along feasible and infeasible prefixes of infeasible CFPs to ignore data corresponding to prefix that is infeasible. A criteria called *minimal infeasible path segment* is used to identify the cluster of infeasible CFPs which can be considered equivalent for maintaining separate data. Clustering is useful because it avoids the possibly exponential cost of keeping the data along each infeasible CFP separate. The recent clustering approach is imprecise in presence of shared edges between CFPs from two different clusters.

In this work, we formalize the interaction between clusters and provide a more general and effective criteria for clustering the infeasible CFPs. Our experiments indicate an average 300% increase in the precision over previous approach, with 100% increase in average analysis time . This is possible because our empirical observation indicates that on average 70% clusters overlap with other clusters.

CCS CONCEPTS

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

KEYWORDS

Program Analysis, Data Flow Analysis, Compilers, Static Analysis, Infeasible Control Flow Paths

ACM Reference Format:
Komal Pathade and Uday P. Khedker. 2019. Path Sensitive MFP Solutions in Presence of Intersecting Infeasible Control Flow Path Segments. In *Proceedings of the 28th International Conference on Compiler Construction (CC '19), February 16–17, 2019, Washington, DC, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3302516.3307349

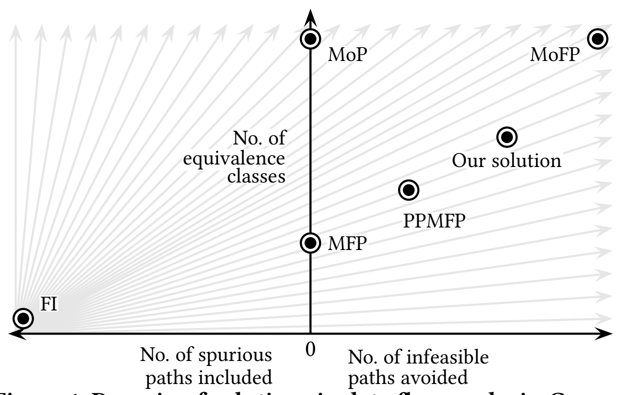

MoP
MoFP
No. of equivalence classes
Our solution
PPMFP
MFP
FI
No. of spurious paths included
0
No. of infeasible paths avoided

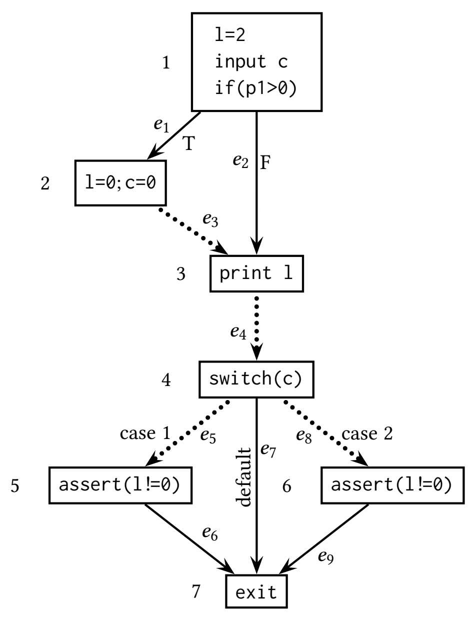
Figure 1: Domain of solutions in data flow analysis. Gray arrows indicate increased precision and decreased scalability.

1 INTRODUCTION

Data flow analysis traverses control flow graph (CFG) representation of programs to discover a sound over-approximation of the information corresponding to all possible runs of a program. This summary information is useful for program verification, debugging, code optimization etc. Since a program can have numerous possible runs, existing approaches partition information into various equivalence classes based on paths and nodes in the CFG and compute one data flow value that represents a sound over-approximation of all data flow values in a class. Thus an analysis could compute a single data flow value per procedure (FI or *Flow-Insensitive* solution), one data flow value per program point in a procedure (MFP or *Maximum Fixed-Point* solution), one data flow value per path reaching a program point in a procedure (MoP or *Meet over Paths* solution), etc.

We explain different possibilities of solutions by bringing out the finer nuances of *control flow paths* (CFPs) that are paths in a *control flow graph* (CFG) representation of a procedure. Some CFPs are *infeasible* although they are present in the CFG, they do not correspond to any execution instance of the program. On the other hand, some analyses over-approximate CFPs in a CFG thereby considering *spurious* CFPs that are not present in the CFG. The precision of solutions and the scalability of the methods to compute them depends on the nature of CFPs. We use them to illustrate the complete range of solutions of a data flow analysis in Figure 1. We explain specific solutions in this range below and illustrate them for value flow analysis for an example program shown in Figure 2.

Flow-insensitive (FI) solution creates single equivalence class for all program points—in Figure 2, FI solution has single value $l \mapsto [0, 2]$ at all points indicating that l can have any (integer) value



The edges in the CFG are numbered e_i. The CFPs involving the sub-paths $\mu_1 : e_3 \rightarrow e_4 \rightarrow e_5$ and $\mu_2 : e_3 \rightarrow e_4 \rightarrow e_8$ are infeasible.

	Values for variable l in different solutions				
	FI	MoFP	MoP	MFP	PPMFP
Out_1	$l \mapsto [0, 2]$	$l \mapsto \{2\}$	$l \mapsto \{2\}$	$l \mapsto [2, 2]$	$l \mapsto [2, 2]$
In_2	$l \mapsto [0, 2]$	$l \mapsto \{2\}$	$l \mapsto \{2\}$	$l \mapsto [2, 2]$	$l \mapsto [2, 2]$
Out_2	$l \mapsto [0, 2]$	$l \mapsto \{0\}$	$l \mapsto \{0\}$	$l \mapsto [0, 0]$	$l \mapsto [0, 0]$
In_3	$l \mapsto [0, 2]$	$l \mapsto \{0, 2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
Out_3	$l \mapsto [0, 2]$	$l \mapsto \{0, 2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
In_4	$l \mapsto [0, 2]$	$l \mapsto \{0, 2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
Out_4	$l \mapsto [0, 2]$	$l \mapsto \{0, 2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
In_5	$l \mapsto [0, 2]$	$l \mapsto \{2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
Out_5	$l \mapsto [0, 2]$	$l \mapsto \{2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
In_6	$l \mapsto [0, 2]$	$l \mapsto \{2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
Out_6	$l \mapsto [0, 2]$	$l \mapsto \{2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
In_7	$l \mapsto [0, 2]$	$l \mapsto \{0, 2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$
Out_7	$l \mapsto [0, 2]$	$l \mapsto \{0, 2\}$	$l \mapsto \{0, 2\}$	$l \mapsto [0, 2]$	$l \mapsto [0, 2]$

In_n/Out_n represent program points just before and after node n respectively in CFG on the left. Meet-over-Paths solutions contain sets of discrete values whereas other solutions contain value ranges. Only MoFP is precise at nodes 5,6 because it eliminates $l \mapsto 0$ reaching from infeasible CFPs.

Figure 2: Different solutions for value analysis over an example program.

in the interval from 0 to 2. Effectively, an FI solution over approximates CFPs in CFG so it may include data flow information along *spurious* as well as *infeasible* CFPs leading to lowest precision of the solution. The methods that compute FI solutions have relatively highest scalability.

Flow-sensitive solutions create at least one equivalence class per program point. They do not consider *spurious* CFPs so they achieve higher precision than FI solutions. These solutions are further classified into *path-sensitive* and *path-insensitive* categories as explained below:

- Path-sensitive solutions compute one data flow value per CFP per program point whereas path-insensitive solutions compute one data flow value across all CFPs reaching a program point. The methods that compute path-insensitive solutions are more scalable than those that compute path-sensitive solution. We describe the following variants of path-sensitive solutions.
  - A meet-over-feasible-paths (MoFP) solution discards data flow values corresponding to infeasible CFPs, so it represents the most precise data flow information. In Figure 2, the data flow value for nodes 5 and 6 excludes the value 0 for l in the MoFP solution because it reaches along *infeasible* CFPs. Theoretically, the computation of MoFP is undecidable. Practically, even in the limited cases when MoFP can be computed, the methods that compute it are not scalable.
  - A meet-over-paths (MoP) solution differs from MoFP in that it does not exclude the infeasible CFPs which makes it less precise than MoFP. In Figure 2 the MoP solution includes the value 0 also for variable l in nodes 5 and 6 unlike MoFP.
- Path-insensitive solutions do not distinguish information based on CFPs; for example a maximum fixed-point (MFP) solution in Figure 2 merges the information along all CFPs reaching node 3. Thus, a MFP solution creates one equivalence class per program point, effectively over-approximating the data flow values along all CFPs reaching a program point. Thus MFP is less precise than MoFP or MoP solutions. Computation of MFP is decidable and the methods that compute MFP solutions are more scalable than those that attempt to compute MoP or MoFP.
- Partially path-sensitive solutions explore the large middle ground between MFP solutions on the one hand and MoP (or MoFP) solutions on the other hand. The methods that compute partially path-sensitive solutions do so by partitioning CFPs using some criteria giving the name *trace-partitioning* to this broad approach [17, 19, 22].
  An important solution in this category is partially path-sensitive maximum fixed-point (PPMFP) solution [24] which manages to discard the data flow information along some infeasible CFPs but not all. This is achieved by clustering CFPs that contain infeasible segments ending with the same edge. It is equal or more precise than MFP solutions in all cases and also more precise than MoP solution in some cases. In Figure 2, the PPMFP solution is imprecise because the infeasible paths overlap which is limiting case of PPMFP.

We propose a novel and effective data flow solution called improved PPMFP (IPPMFP) that overcomes the limitations of the

PPMFP solution. We achieve this by further dividing the equivalence classes created by PPMFP solution by separating the overlapping infeasible CFPs. Our empirical observation indicates up to 3 times more precision compared to PPMFP solution because many infeasible CFPs overlap in our benchmarks. Since IPPMFP solution increases the number of equivalence classes, we found an average 100% increase in analysis time over the PPMFP computation.

The rest of the paper is organized as follows: Section 2 gives motivation of our work. Section 3 describes main idea of IPPMFP solution, Section 4 presents a method for computing IPPMFP solution, Section 5 proves soundness of IPPMFP solution, Section 6 explains the experimental setup and discusses the results, Section 7 reviews the related literature, and Section 8 concludes the paper.

2 MOTIVATION

A PPMFP solution creates equivalence classes of CFPs using a criteria called *minimal infeasible path segments* (MIPS) [5, 24]. A solution computed over these equivalence classes is imprecise in presence of intersecting MIPSs.

In this section, we state the concept of *minimal infeasible path segments* (MIPS) [5, 24], and explicate the limitations of PPMFP solution. We use following basic concepts related to infeasible paths from prior work [5, 24].

A CFP $\rho\colon n_1 \xrightarrow{e_1} n_2 \xrightarrow{e_2} n_3 \xrightarrow{e_3} \ldots \rightarrow n_k \xrightarrow{e_k} n_{k+1}$,
where $k \geq 1$, is an *infeasible control flow path* if n_1 is the start node of the CFG, and there is a 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 $\mu\colon n_j \xrightarrow{e_j} \ldots \rightarrow n_i \xrightarrow{e_i} n_{i+1}$ of CFP ρ above is a *minimal infeasible path segment* (MIPS) if it is not a subpath of any execution path but every proper subpath of μ is a subpath of some execution path. For example in Figure 3(a) (on next page), the path segment $n_1 \xrightarrow{e_1} n_3 \xrightarrow{e_3} n_4 \xrightarrow{e_4} n_5$ (shown with dotted edges) is a MIPS.

For convenience, we also denote a path segment μ above as a sequence of edges—$\mu\colon e_j \rightarrow \ldots \rightarrow e_i$. Edges $e_j,\ e_i$ are called the *start* and the *end* edge of MIPS μ, while other edges $e_k,\ j < k < i$ are called the *inner* edges of MIPS μ. Two MIPSs are equivalent (denoted $\simeq$) if they have the same *end* edge. We denote the start, inner, and end edges of a MIPS μ by *start*(μ), *inner*(μ), and *end*(μ); for convenience and brevity, we refer to these sets as edges. Note that *start*(μ) and *end*(μ) sets are singleton, and *end*(μ) edge is always a conditional edge.

A MIPS has following precision property that allows us to eliminate data flow values reaching along infeasible CFPs:

> *PresP* : The data flow values that reach the *start* edge of a MIPS μ do not reach the *end* edge of μ.

The PPMFP solution proposed by the prior work [24] uses this property of MIPS to separate the data flow values for different MIPSs. The idea is to block the data flow values reaching *start*(μ) edge from reaching the *end*(μ) edge by separating it out from the other data flow values. For example in Figure 3(a), the data flow value $c \mapsto [0,0]$ at e_1 does not reach e_4 because path $e_1 \rightarrow e_3 \rightarrow e_4$ is a MIPS, a PPMFP solution (shown in Figure 3(b)) keeps the value $c \mapsto [0,0]$ separate (in $\delta(\mu)$) and blocks it at e_4 (set to $\top$). Unfortunately, the approach still suffers from imprecision because the separation achieved by different MIPSs is not sufficient to satisfy the property *PresP* . This happens when a MIPS μ' *intersects* with μ as defined below

> The *start*(μ') edge is contained in μ also, either as the start edge or as an inner edge.

Since μ and μ' are distinct MIPSs, their end edges are distinct. The data flow values of μ' also reach *end*(μ). Unfortunately, they also include the data flow values of *start*(μ) which are are blocked within μ but still reach *end*(μ) as the data flow values of μ'. This violates the precision property *PresP* .

Our motivating example in Figure 2 illustrates this case. Table 1 shows the computation of PPMFP solution for example in Figure 2 which has two MIPSs: $\mu_1 : e_3 \rightarrow e_4 \rightarrow e_5$ and $\mu_2 : e_3 \rightarrow e_4 \rightarrow e_8$ that have the same start edge. The data flow values associated with them are denoted by $\delta(\mu_1)$ and $\delta(\mu_2)$ respectively. By abuse of notation, the default information that is outside of any MIPS is dentoted by δ. The information discarded by $\delta(\mu_1)$ at edge e_5 is maintained by $\delta(\mu_2)$. Hence the PPMFP solution—computed using the meet of information in $\delta(\mu_1)$, $\delta(\mu_2)$, and δ spuriously includes information from e_3 at edges e_5 and e_8.

A PPMFP solution is imprecise in presence of intersecting MIPSs. Towards this end, we present a data flow solution that is precise even in the presence of intersecting MIPSs.

3 OUR KEY IDEA

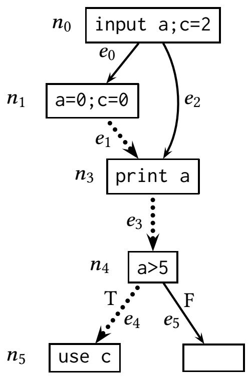
We now introduce the Improved PPMFP (IPPMFP) solution that uses the following key idea to handle intersecting MIPSs.

Continuing with the spirit of PPMFP solution, we separate the data flow values that reach a program point along CFPs that overlap with a MIPS, from the data flow values that do not overlap with any MIPS. However, unlike the PPMFP solution, we do not associate data flow values with individual MIPSs (identified by *end* edge, as shown in Table 1) but associate them with a set of MIPSs that intersect with each other (as shown in Figure 4). Thus our data flow values are partitioned in *parts* of the form $\langle d, m\rangle$ where d is a data flow value and m is a label representing a subset of the set of all MIPSs.

This retains the property of PPMFP that allows us to discard the data flow values associated with specific MIPSs at the *end* edges of the MIPSs. However, unlike PPMFP, it prevents multiple copies of one data flow value when MIPSs intersect. For example, in Figure 4 the data flow value $l \mapsto [0,0]$ at edge e_3 is associated with set of MIPSs $\{\mu_1, \mu_2\}$ and is eliminated at edges e_5 and e_8 because they are *end* edges of μ_1 and μ_2 respectively.

We achieve the improved PPMFP solution by partitioning data flow values present in original analysis in various parts, and associating each part with a set of MIPSs. A set of parts is computed at each program point such that the meet of data flow values over all parts gives atleast as precise information as original analysis,

[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_1 etc. as labels of nodes and edges in a control flow graph and then juxtapose them when we write path segments.



(a) 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.

	δ	$\delta(\mu)$	PPMFP Solution $\delta \sqcap \delta(\mu)$
e_0	$c \mapsto [2,2]$	$\top$	$c \mapsto [2,2]$
e_2	$c \mapsto [2,2]$	$\top$	$c \mapsto [2,2]$
e_1	$\top$	$c \mapsto [0,0]$	$c \mapsto [0,0]$
e_3	$c \mapsto [2,2]$	$c \mapsto [0,0]$	$c \mapsto [0,2]$
e_4	$c \mapsto [2,2]$	$\top$	$c \mapsto [2,2]$
e_5	$c \mapsto [2,2]$	$c \mapsto [0,0]$	$c \mapsto [0,2]$

(b) In PPMFP computation, values reaching from CFPs which overlap with μ are kept separately in $\delta(\mu)$, Rest of the values are kept in δ. At e_4—*end* edge of μ —$\delta(\mu)$ eliminates $c \mapsto [0,0]$ reaching from e_1. $\top$ is $c \mapsto [+\infty, -\infty]$

Figure 3: Illustrating minimal infeasible path segment (MIPS) and the PPMFP solution.

Table 1: Computation of the PPMFP solution for example in Figure 2. $\top$ **represents the top value which is** $l \mapsto [+\infty, -\infty]$

	δ	$\delta(\mu_1)$	$\delta(\mu_2)$	PPMFP $\delta \sqcap \delta(\mu_1) \sqcap \delta(\mu_2)$
e_3	$\top$	$l \mapsto [0,0]$	$l \mapsto [0,0]$	$l \mapsto [0,0]$
e_4	$l \mapsto [2,2]$	$l \mapsto [0,0]$	$l \mapsto [0,0]$	$l \mapsto [0,2]$
e_5	$l \mapsto [2,2]$	$\top$	$l \mapsto [0,0]$	$l \mapsto [0,2]$
e_8	$l \mapsto [2,2]$	$l \mapsto [0,0]$	$\top$	$l \mapsto [0,2]$

and gives more precise information if a part contains information reaching from infeasible CFPs because such parts are discarded.

Let D_n be data flow value in original analysis at node n, and S_n be the set of parts computed in new analysis at node n then

$$D_n \sqsubseteq \bigsqcap_{\langle d,\, m\rangle \in S_n} d$$

Each part $\langle d, m\rangle \in S_n$ satisfies following properties that allows us to identify and discard data flow values reaching from infeasible CFPs:

- The MIPSs in set m interesect with each other.
- The data flow value d reaches the *start* edge of every MIPS in m.
- The data flow value d does not reach the *end* edge of any MIPS in m
- Neither d nor m appears in any other part in S_n.

We create, maintain, and delete parts in partition during data flow analysis using the following criteria: a part $\langle d, m\rangle$ is created at *start* edges of MIPSs in m, is propagated along the *inner* edges of MIPSs in m, and is discarded at *end* edges of MIPSs in m. Note that the part $\langle d, m\rangle$ is not maintained at edges that are not related to MIPSs in set m.

3.1 Defining IPPMFP solution

We now explain how these ideas are incorporated in a data flow analysis. Let L denote the lattice of data flow values of underlying data flow analysis. Then the MFP solution of data flow analysis for a program computes values in L using data flow variables In_n/Out_n for every node n of the CFG of the program. Assume we are given a set M containing MIPSs in the CFG of the program, then each data flow value in L could be associated with one of the $2^{|M|}$ subsets of M. We lift the analysis to a product lattice $\widehat{L} = L \times \mathcal{P}(M)$, where $\mathcal{P}(M)$ is the power set of M which is also a lattice.

We further lift the analysis to power set of $\widehat{L}$ i.e. $\mathcal{P}(\widehat{L})$ and compute $\overline{In_n}/\overline{Out_n}$ values, each of which is a is a set of pairs $\langle d, m\rangle$ where $d \in L$ and label m represents a subset of M. This collection of *parts* gives us the distinctions required to achieve precision by separating data flow values along infeasible paths from those along feasible paths.

For a data flow value $\overline{In_n}$

- *Part* $\langle d, m\rangle \in \overline{In_n}$, represents the data flow value d reaching node n along CFPs that overlap with the set of MIPSs in m. If no value along such a path reaches n, then $\langle d, m\rangle \notin \overline{In_n}$.

  In Figure 4, data flow value $l \mapsto [0,0]$ reaches e_4 from CFP $e_1 \to e_3 \to e_4$ that overlaps with MIPSs μ_1, μ_2 shown with dotted lines.
- *Part* $\langle d, \{\}\rangle \in \overline{In_n}$, represents the data flow value d reaching node n along CFPs that do not overlap with any MIPS. If no value along such a path reaches n, then $d = \top$.

  In Figure 4, data flow value $l \mapsto [2,2]$ reaches e_4 from CFP $e_2 \to e_4$ that does not overlap with any MIPS.

Formally, we say a CFP σ overlaps with MIPS μ to mean suffix of σ is non-empty prefix of μ.

Section 4 defines the data flow equations to compute $\overline{In_n}/\overline{Out_n}$.

Our IPPMFP solution also computes values in L, and is computed in data flow variables $\overline{\overline{In_n}}/\overline{\overline{Out_n}}$ for every node n in the CFG of the program. They are computed from $\overline{In_n}/\overline{Out_n}$ using following

Paths and MIPSs:
- MIPS $\mu_1 : e_3 \to e_4 \to e_5$
- MIPS $\mu_2 : e_3 \to e_4 \to e_8$
- $start(\mu_1) = start(\mu_2) = e_3$
- $inner(\mu_1) = inner(\mu_2) = e_4$
- $end(\mu_1) = e_5, end(\mu_2) = e_8$

Lattices and data flow values:
- $L = \{l \mapsto [i, j] \mid -\infty \leq i \leq j \leq +\infty\}$
- $M = \{\mu_1, \mu_2\}$
- $\mathcal{P}(M) = \{m \mid m \subseteq M\}$
- $\widehat{L} = L \times \mathcal{P}(M)$
- $\top = l \mapsto [+\infty, -\infty]$
- $\overline{\top} = \{\langle l \mapsto [+\infty, -\infty], \{\}\rangle\}$
- $\overline{Out}_{e_5} = \{\langle l \mapsto [2, 2], \{\}\rangle\}$
- $\overline{\overline{Out}}_{e_5} = l \mapsto [2, 2]$
- $Out_{e_5} = l \mapsto [0, 2]$

(MFP, PPMFP solution in Figure 2)

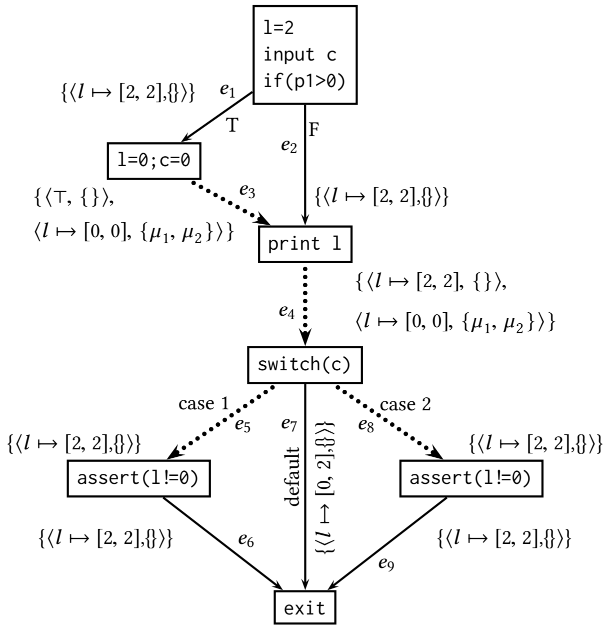

l=2
input c
if(p1>0)
$\{\langle l \mapsto [2, 2], \{\}\rangle\}$ e_1
T
F
e_2
l=0;c=0
$\{\langle \top, \{\}\rangle,$
$\langle l \mapsto [0, 0], \{\mu_1, \mu_2\}\rangle\}$
e_3
$\{\langle l \mapsto [2, 2], \{\}\rangle\}$
print l
$\{\langle l \mapsto [2, 2], \{\}\rangle,$
$\langle l \mapsto [0, 0], \{\mu_1, \mu_2\}\rangle\}$
e_4
switch(c)
case 1
case 2
e_5
e_7
e_8
$\{\langle l \mapsto [2, 2], \{\}\rangle\}$
$\{\langle l \mapsto [2, 2], \{\}\rangle\}$
default
$\{\langle l \mapsto [0, 2], \{\}\rangle\}$
assert(l!=0)
assert(l!=0)
$\{\langle l \mapsto [2, 2], \{\}\rangle\}$ e_6
$\{\langle l \mapsto [2, 2], \{\}\rangle\}$ e_9
exit

Figure 4: Computing IPPMFP solution for the example in Figure 2. We label the data flow values in L with set of MIPSs. This allows us to discard data flow value $l \mapsto [0, 0]$ at edges e_5, e_8. With slight abuse of notation we use In_e/Out_e to denote information at In and Out of edge e respectively.

operations

$$\overline{\overline{In}}_n = fold_{\sqcap}(\overline{In}_n) \tag{1}$$

$$\overline{\overline{Out}}_n = fold_{\sqcap}(\overline{Out}_n) \tag{2}$$

$$fold_{\sqcap}(s) = \bigsqcap_{\langle d,\, m\rangle \in s} d \tag{3}$$

The $\overline{\overline{In}}_n/\overline{\overline{Out}}_n$ values represent the data flow information reaching node n along all CFPs that are feasible. They exclude the information along infeasible CFPs.

3.2 Reducing the Number of Parts

The efficiency of computing IPPMFP solution can be increased by reducing the number of parts. The PPMFP solution [24] reduced the number of MIPSs to be considered by creating clusters of mipss that have the same end edge because they share the property that the data flow values associated with them must be blocked at the same edge. For IPPMFP, the intersection between two MIPSs is defined in terms of a start edge rather than an inner edge or an end edge (see Section 2). Thus it is possible for us reduce the number of parts by adapting the clustering idea from PPMFP as follows: We merge two parts $\langle d, m\rangle$ and $\langle d', m'\rangle$ into a single part $\langle d \sqcap d', m \cup m'\rangle$ *iff* the sets m and m' are end edge equivalent as defined below:

$$m \overset{end}{=} m' \Leftrightarrow \bigcup_{\mu \in m} end(\mu) = \bigcup_{\mu' \in m'} end(\mu') \tag{4}$$

They are merged because they contain the data flow values that are blocked at the same set of *end* edges.

4 COMPUTING IPPMFP SOLUTION

We now illustrate how the data flow equations for computing IPPMFP solution are derived from that of MFP specifications of a data flow analysis. Section 4.1 explains this lifting by defining the equations, node flow functions, and the meet operator. A key enabler for this lifting are the specially crafted edge flow functions (Section 4.2) that discard data flow values that reach a program point from infeasible CFPs.

4.1 IPPMFP Solution Specification

Let $In_n, Out_n \in L$ be the data flow values computed by a data flow analysis at node n, where L is a meet semi-lattice satisfying the descending chain condition. Additionally, it contains a $\top$ element—we add an artificial $\top$ value, if there is no natural $\top$.

The desired solution of a data flow analysis is the MFP solution of the data flow equations 5 and 6 given below which are 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 *glb* (greatest lower bound) of elements in L. Function *pred(n)* returns the predecessor nodes of node n in the CFG of p. The node flow function f_n and edge flow function $g_{m \to n}$ compute the effect of node n and edge $m \to n$ in the CFG respectively. Usually the edge flow functions are identity.

$$In_n = \begin{cases} BI & n = Start_p \\ \prod\limits_{m \in pred(n)} g_{m \to n}(Out_m) & otherwise \end{cases} \tag{5}$$

$$Out_n = f_n(In_n) \tag{6}$$

We lift Equations 5 and 6 to define a data flow analysis that computes data flow variables $\overline{In}_n, \overline{Out}_n$ each of which is a set of parts of the form $\langle d, m\rangle$. As explained in Section 3.1, these parts are values in $\mathcal{P}(\widehat{L})$ where $\widehat{L} = L \times \mathcal{P}(M)$ where M is the set of MIPS in the program and L is the lattice of values computed by Equations 5 and 6.

$$\overline{In}_n = \begin{cases} \{\langle BI, \{\}\rangle\} & n = Start_p \\ \overline{\prod\limits_{m \in pred(n)}} \overline{g}_{m \to n}(\overline{Out}_m) & otherwise \end{cases} \tag{7}$$

$$\overline{Out}_n = \overline{f}_n(\overline{In}_n) \tag{8}$$

The top value of $\widehat{L}$ used for initialization is $\langle \top, \{\}\rangle$. The node flow function $(\overline{f}_n)$ is pointwise application of f_n to the parts in the input set:

$$\overline{f}_n(s) = \{\langle f_n(d), m\rangle \mid \langle d, m\rangle \in s\} \tag{9}$$

We define the meet operator ($\overline{\sqcap}$) using the intuition explained in Section 3.2. It merges parts that have *end* edge equivalent labels and copies the remaining parts unchanged in the output.

$$\begin{aligned} s' \,\overline{\sqcap}\, s'' = \{\langle d, m\rangle \mid\; & C \Rightarrow (d = d' \sqcap d'', m = m' \cup m''), \\ & C' \Rightarrow (d = d', m = m'), \\ & C'' \Rightarrow (d = d'', m = m'')\} \end{aligned} \tag{10}$$

where conditions C, C', and C'' (defined below) govern whether parts are merged or not by examining the end edge equivalence (Equation 4). They have been defined separately only for convenience and should be read inlined in Equation (10).[2]

$$C \Leftrightarrow \exists\langle d', m'\rangle \in s' \wedge \exists\langle d'', m''\rangle \in s'' \; s.t. \; m' \overset{end}{=} m''$$

$$C' \Leftrightarrow \exists\langle d', m'\rangle \in s' \wedge \nexists\langle d'', m''\rangle \in s'' \; s.t. \; m' \overset{end}{=} m''$$

$$C'' \Leftrightarrow \nexists\langle d', m'\rangle \in s' \wedge \exists\langle d'', m''\rangle \in s'' \; s.t. \; m' \overset{end}{=} m''$$

4.2 Edge Flow Functions

An edge flow function $\overline{g}_e$ associated with edge e, creates, deletes, or merges parts reaching edge e, depending on whether e is a *start*, an *inner*, or an *end* edge of the MIPSs in the labels of the parts. These operations allow us to discard data flow values reaching a program point from infeasible CFPs. Below, we define the edge flow function $\overline{g}_e$ with respect to the operations it performs. We illustrate each operation using example in Figure 4.

The argument of edge flow function is a set s of *parts* so multiple operations may be performed by each edge flow function. For simplicity of exposition, we ignore the sequence of operations when multiple conditions are satisfied at the same time—we consider these cases when proving soundness of the solution in Section 5.

[2]The free variables d', d'', m', and m'' occurring in Equation (10) are bound to those in the conditions.

For convenience, we refer to a part with empty label as *unlabeled* part, for example $\langle d, \{\}\rangle$ is an *unlabeled* part s.t. $d \in L$.

Operation 1 - Creation: A part is created when e is *start* edge of a set of MIPSs m: in this case, we add m as label to all existing *parts* in s, indicating that the data in these *parts* should not reach *end* edges of MIPSs in m. Further, we add an *unlabeled* part $\langle \top, \{\}\rangle$ to output.

$$\overline{g}_e(s) = \begin{cases} attach_s(m) \cup \{\langle \top, \{\}\rangle\} \\ \quad \ldots \text{for all } \mu \in m, e \text{ is } \textit{start} \text{ edge of } \mu \\ \\ s \quad \ldots e \text{ is not start edge of any MIPS} \end{cases} \tag{11}$$

The $attach_s(m)$ adds MIPSs in m to *parts* in s.

$$attach_s(m) = \{\langle d_1, m_1 \cup m\rangle \mid \langle d_1, m_1\rangle \in s\} \tag{12}$$

For example in Figure 4, e_3 is the *start* edge of MIPSs μ_1, μ_2 so the data flow value $l \mapsto [0, 0]$ is associated with $\{\mu_1, \mu_2\}$ and an *unlabeled* part is added at e_3.

Operation 2 - Flow/Discard: part $\langle d, m\rangle, m \neq \{\}$ is propagated or discarded through e depending on the following criteria

- e is *inner* edge for all MIPSs in m, then $\langle d, m\rangle$ is kept as it is in output.
- e is *end* edge for some MIPS in m, then $\langle d, m\rangle$ is discarded.

$$\overline{g}_e(s) = \begin{cases} s \quad \ldots e \text{ is } \textit{inner} \text{ edge for all MIPS in } m \\ \\ s \setminus \{\langle d, m\rangle\} \\ \quad \ldots e \text{ is } \textit{end} \text{ edge for some MIPS in } m \end{cases} \tag{13}$$

In Figure 4, e_4 is *inner* edge for MIPSs μ_1, μ_2 so the part $\langle l \mapsto [0, 0], \{\mu_1, \mu_2\}\rangle$ is kept as is. Whereas, at the *end* edges e_5, e_8 of MIPSs μ_1, μ_2, the part is discarded.

Operation 3 - Merging: Let *live*(e) denote the set of MIPSs that have e as *start* or *inner* edge i.e.,

live(e) $= \{\mu \mid e \in start(\mu) \cup inner(\mu),\ \mu \in M\}$, where M is set of all MIPSs in program. A part $\langle d, m\rangle$ is merged with other part, if atleast one of the MIPSs in m is not *live* at e i.e.*live*(e)$\cap m \neq m$; in this case we merge the part $\langle d, m\rangle$ with the part corresponding to label *live*(e)$\cap m$. If there is no such part, then a new part $\langle d$, *live*(e) $\cap\, m\rangle$ is created and added; while the original *part* $\langle d, m\rangle$ is discarded.

We define two functions π_s^m and *add* that are used for describing edge flow function as follows: π_s^m returns a *part* from s with label m.

$$\pi_s^m = \begin{cases} \langle d, m\rangle & \langle d, m\rangle \in s \\ \langle \top, m\rangle & otherwise \end{cases} \tag{14}$$

Function *add* merges two parts.

$$add(\langle d_1, m_1\rangle, \langle d_2, m_2\rangle) = \langle d_1 \sqcap d_2, m_1 \cup m_2\rangle \tag{15}$$

$$\overline{g}_e(s) = (s \setminus \{\pi_s^{m'}\}) \cup \{add(\pi_s^{m'}, \langle d, m'\rangle)\} \tag{16}$$
$$\textit{where } m' = m \cap live(e)$$

In Figure 4, at edge e_7, *live*(e_7) $\cap \{\mu_1, \mu_2\} = \phi$, so the part $\langle l \mapsto [0, 0], \{\mu_1, \mu_2\}\rangle$ is merged with *unlabeled* part $\langle l \mapsto [2, 2], \{\}\rangle$ resulting in $\langle l \mapsto [0, 2], \{\}\rangle$ at e_7.

The IPPMFP solution is computed in data flow variables $\overline{\overline{In}},\overline{\overline{Out}}$ using equations 1,2.

5 ANALYSIS GUARANTEES

In this section, we prove that IPPMFP solution is sound

5.1 Proof of Soundness

At each program node n, the data flow values in original analysis In_n/Out_n are an over approximation of data flow values computed in new analysis ($\overline{\overline{In_n}}/\overline{\overline{Out_n}}$) i.e. for all nodes n in the CFG of program

$$In_n \sqsubseteq \overline{\overline{In_n}},\ Out_n \sqsubseteq \overline{\overline{Out_n}}$$

We argue that three fundamental functions 1) node flow, 2) edge flow and 3) meet are sound.

Node Flow Function($\overline{f}_n$)

To Prove:

$$d \sqsubseteq fold_{\sqcap}(s) \implies f_n(d) \sqsubseteq fold_{\sqcap}(\overline{f}_n(s)) \tag{17}$$
$$where\ s = \{\langle d_1, m_1\rangle, \langle d_2, m_2\rangle, ...\}$$

- Case1: n is procedure entry ($n = Start_p$):

$$\begin{aligned}
&LHS \\
&= BI \sqsubseteq fold_{\sqcap}(\{\langle BI, \{\}\rangle\}) && from\ 5, 7 \\
&= BI \sqsubseteq BI && from\ 3 \\
&= True \\
&RHS \\
&= f_n(BI) \sqsubseteq fold_{\sqcap}(\overline{f}_n(\{\langle BI, \{\}\rangle\})) && from\ 5, 7 \\
&= f_n(BI) \sqsubseteq fold_{\sqcap}(\{\langle f_n(BI), \{\}\rangle\}) && from\ 9 \\
&= f_n(BI) \sqsubseteq f_n(BI) && from\ 3 \\
&= True
\end{aligned}$$

- Case2: n is not procedure entry node

$$\begin{aligned}
&LHS \\
&= d \sqsubseteq fold_{\sqcap}(s) \\
&= d \sqsubseteq \bigsqcap_{\langle d_i,\, m_i\rangle \in s} d_i \\
&\implies f_n(d) \sqsubseteq f_n(\bigsqcap_{\langle d_i,\, m_i\rangle \in s} d_i) \\
&\qquad ...f \text{ is monotonous function} \\
&\implies f_n(d) \sqsubseteq \bigsqcap_{\langle d_i,\, m_i\rangle \in s} f_n(d_i)
\end{aligned}$$

$$\begin{aligned}
&RHS \\
&= f_n(d) \sqsubseteq fold_{\sqcap}(\overline{f}_n(s)) && from\ 5, 7 \\
&= f_n(d) \sqsubseteq fold_{\sqcap}(s'),\ where \\
&\quad s' = \{\langle f_n(d_i), m_i\rangle \mid \langle d_i, m_i\rangle \in s\} && from\ 9 \\
&= f_n(d) \sqsubseteq \bigsqcap_{\langle f_n(d_i),\, m_i\rangle \in s'} f_n(d_i) && from\ 3 \\
&LHS \implies RHS
\end{aligned}$$

Edge Flow Function($\overline{g}_e$): For simplicity, we assume original edge flow function g_e is identity.

To Prove:

$$d \sqsubseteq fold_{\sqcap}(s) \implies d \sqsubseteq fold_{\sqcap}(\overline{g}_e(s)) \tag{18}$$
$$where\ s = \{\langle d_1, m_1\rangle, \langle d_2, m_2\rangle, ...\} \tag{19}$$

We divide the proof in sub cases that cover all scenarios

- step 1: These operations are performed for each *part* $\langle d_i, m_i\rangle \in s$, we prove that equation 18 holds in each iteration of step 1.

(1) if e is not *end* edge of any MIPS in m_i then

(a) if e is not *live* edge for some MIPSs from m_i, $m' = m_i \cap live(e), m' \neq m_i$ then $\langle d_i, m_i\rangle$ is merged with *part* labeled m' using equation 16

$$\overline{g}_e(s) = (s \setminus \{\pi_s^{m'}, \pi_s^{m_i}\}) \cup \{add(\pi_s^{m'}, \pi_s^{m_i})\}$$

Applying $fold_{\sqcap}$we get,

$$\begin{aligned}
&fold_{\sqcap}(\overline{g}_e(s)) \\
&= fold_{\sqcap}((s \setminus \{\pi_s^{m'}, \pi_s^{m_i}\}) \cup \{add(\pi_s^{m'}, \pi_s^{m_i})\}) \\
&= \bigsqcap_{\langle d_i,\, m_i\rangle \in s} d_i \\
&= fold_{\sqcap}(s)
\end{aligned} \tag{20}$$

(b) e is *inner* edge for all MIPSs in m_i

$$\overline{g}_e(s) = s \qquad ...from\ 13 \tag{21}$$

from 20, 21 we get,

$$d \sqsubseteq fold_{\sqcap}(s) \implies d \sqsubseteq fold_{\sqcap}(\overline{g}_e(s))$$

(2) if e is *end* edge of some MIPS in m_i: discard *part* $\langle d_i, m_i\rangle$. Operations in above cases 1a,1b (step 1) prove that each discarded *part* has traversed along an infeasible control flow path—without leaving the path in between— and hence could be discarded. (If the *part* had left the MIPS in between it would have been merged with other *parts* as shown in case 1a in step1 before reaching the *end* edge.)

- Step 2: This step is performed once per edge. We prove equation 18 holds after step 2.

(1) if e is not *start* edge on any MIPS

$$\overline{g}_e(s) = s \qquad from\ 11 \tag{22}$$

(2) if e is *start* edge of non-empty set of MIPSs m

$$\overline{g}_e(s) = attach_s(m) \cup \{\langle \top, \{\}\rangle\}$$
$$...from\ 11$$

Applying $fold_{\sqcap}$, we get

$$\begin{aligned}
& fold_{\sqcap}(\overline{g}_e(s)) \\
&= fold_{\sqcap}(attach_s(m) \cup \{\langle \top, \{\}\rangle\}) \\
&= fold_{\sqcap}(attach_s(m)) \sqcap \top \quad \ldots from\ 3 \\
&= fold_{\sqcap}(attach_s(m)) \\
&= fold_{\sqcap}(\{\langle d_1, m_1 \cup m\rangle, \langle d_2, m_2 \cup m\rangle, ..\}) \\
&\qquad\qquad \ldots from\ 19, 12 \\
&= \bigsqcap_{\langle d_i,\, m_i \cup m\rangle \in attach_s(m)} d_i \quad \ldots from\ 3 \\
&= fold_{\sqcap}(s) \qquad \ldots from\ 3, 19
\end{aligned} \tag{23}$$

from 22, 23 we get,

$$d \sqsubseteq fold_{\sqcap}(s) \implies d \sqsubseteq fold_{\sqcap}(\overline{g}_e(s))$$

Meet Operation($\overline{\sqcap}$): Following holds trivially from the definition of $\overline{\sqcap}$ because it does not discard any data flow value present in s, s' during computation of $s \,\overline{\sqcap}\, s'$.

$$fold_{\sqcap}(s \,\overline{\sqcap}\, s') \sqsubseteq fold_{\sqcap}(s)$$
$$fold_{\sqcap}(s \,\overline{\sqcap}\, s') \sqsubseteq fold_{\sqcap}(s')$$

5.2 MIPS EQUIVALENCE

In section 2, we declared that two MIPSs are equivalent if they have the same *end* edge, we give a formal proof for the same here. Firstly, we state the following lemma that will be useful for our equivalence proof.

Lemma 1: For any MIPS: $\sigma : x_1 \rightarrow x_2 \rightarrow \ldots \xrightarrow{e} n$, where e is *end* edge of MIPS. If the necessary condition for execution of edge e be *cond*(e) then σ is infeasible if *cond*(e) evaluates to False at node x_1 (*cond*(e) $\models False$ at x_1) and variables in *cond*(e) are not modified along σ. This case holds for all MIPSs considered by our approach so the implication is bidirectional.

Let $[\![p]\!]_n$ represent evaluation of condition expression p at program node n, and *modified*(p, σ) be set of variables in p that are modified along MIPS σ then

$$\begin{aligned}
\sigma \models \text{MIPS} \iff & [\![cond(e)]\!]_{x_1} \models False, \\
& modified(cond(e), \sigma) = \phi
\end{aligned} \tag{24}$$

Theorem 1: If σ_1, σ_2 are two MIPSs with the same *end* edge e, and they intersect at some node x_k, then σ_3, σ_4—

constructed as shown below using split and join of σ_1, σ_2 at intersecting node—are also MIPSs.

$$\begin{aligned}
&\sigma_1 : \underline{x_1 \rightarrow x_2 \rightarrow ...x_k \rightarrow x_{k+1} \rightarrow ... \xrightarrow{e} n} \\
&\sigma_2 : y_1 \rightarrow y_2 \rightarrow ...x_k \rightarrow y_j \rightarrow ... \xrightarrow{e} n
\end{aligned} \tag{25}$$

$$\begin{aligned}
&\sigma_3 : \underline{x_1 \rightarrow x_2 \rightarrow ...x_k} \rightarrow y_j \rightarrow ... \xrightarrow{e} n \\
&\sigma_4 : y_1 \rightarrow y_2 \rightarrow ...\underline{x_k \rightarrow x_{k+1} \rightarrow ... \xrightarrow{e} n}
\end{aligned}$$

To Prove :

$$\sigma_1, \sigma_2 \models \text{MIPS} \implies \sigma_3, \sigma_4 \models \text{MIPS}$$

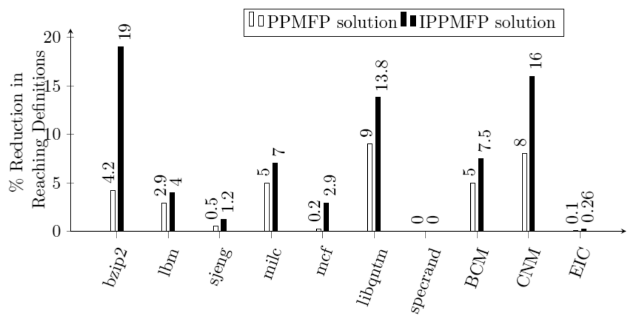


Figure 5: Precision gain in PPMFP vs IPPMFP solution over MFP solution in Reaching Definition Analysis.

$$\begin{aligned}
& LHS: \\
& \sigma_1, \sigma_2 \models \text{MIPS} && \text{given} \\
& \implies [\![cond(e)]\!]_{x_1} \equiv [\![cond(e)]\!]_{y_1} \equiv False, \\
& \qquad modified(cond(e), \sigma_1) = \phi, \\
& \qquad modified(cond(e), \sigma_2) = \phi && \text{from } 24 \\
& \implies [\![cond(e)]\!]_{x_1} \equiv [\![cond(e)]\!]_{y_1} \equiv False, \\
& \qquad modified(cond(e), \sigma_3) = \phi, \\
& \qquad modified(cond(e), \sigma_4) = \phi && \text{from } 26 \\
& \implies \sigma_3, \sigma_4 \models \text{MIPS} && \square
\end{aligned}$$

σ_3, σ_4 are constructed using edges and nodes from σ_1, σ_2 so

$$\begin{aligned}
& modified(cond(e), \sigma_1) = \phi \ \wedge\ modified(cond(e), \sigma_2) = \phi \\
& \implies \\
& modified(cond(e), \sigma_3) = \phi \ \wedge\ modified(cond(e), \sigma_4) = \phi
\end{aligned} \tag{26}$$

5.3 Complexity Analysis

In our analysis, at most one *part* is created at the *start* edge of MIPSs; no extra partitions are created at other edges, nodes, or during meet operation. In worst case, each edge in the CFG of program is *start* edge of some MIPS, so the upper bound on the number of *parts* at any program point is $E + 1$ (E is number of edges in the program CFG).

The worst case time and space complexity of our analysis is equivalent to that of computing MFP solutions for $E + 1$ data flow analyses in parallel which is:

$O(E * (complexity\ of\ MFP\ solution))$.

6 EXPERIMENTS AND RESULTS

In this section, we empirically compare the IPPMFP solution with the PPMFP and the MFP solutions with respect to precision and efficiency.

6.1 Experimental Setup

We performed our experiments on a 64 bit machine with 8GB RAM running Windows 7 with Intel core i7-5600U processor having clock speed of 2.6 GHz. We used ten C programs of up to 74kLoC size which contained seven SPEC CPU-2006 benchmarks,

Table 2: Increased Precision in Reaching Definition Analysis: 1) MFP column shows set of all reaching definitions computed using ($In_n \cup Out_n$) over all program nodes n. The reduction shown in PPMFP and IPPMFP solution is with respect to MFP solution. 2) Benchmarks 1 to 7 are spec benchmarks and benchmarks 8 to 10 marked by * are industry code bases.

Benchmark Properties					Reaching Definition Analysis		
						#reduction(%)	
Name	KLOC	#MIPSs	#Overlap-ing MIPSs	%Funcs Impacted	#defs in MFP	PPMFP	IPPMFP
1.bzip2	16	712	660	43	1469090	61441(4.2)	285818(19)
2.lbm	2.3	60	50	63	3734	107(2.9)	151(4)
3.sjeng	27	1504	1359	40	922089	4917(0.5)	11020 (1.2)
4.milc	30	685	445	42	218565	12731(5)	15290(7)
5.mcf	5.4	26	23	29	28278	65(0.2)	832(2.9)
6.libqntm	8.7	98	95	14	42551	4134(9)	5911(13.8)
7.specrnd	0.1	0	0	0	74	0(0)	0(0)
8.BCM*	74	65	41	26	8641	515(5)	655(7.5)
9.CNM*	52	181	97	18	36596	2890(8)	5861 (16)
10.EIC*	13	479	420	17	4116468	6016(0.1)	10774(0.26)

and three industry code bases; same as that of the PPMFP solution [24]. These programs have up to 2,900 function calls. The number of distinct MIPSs range from 0 for the *specrand* benchmark to 1.5k for the *sjeng* benchmark. In these benchmarks, up to 63% (average 29%) functions had at least one MIPS in their CFG. On average 70% MIPSs were overlapping with each other.

We used the iterative fix-point data flow computation algorithm [2, 20] implemented in Java in commercial static analysis tool *TCS Embedded Code Analyzer* [1] for our data flow analysis. We also implemented an intermediate framework that takes underlying data flow specifications (Equations 5, 6 in Section 4), and automatically lifts them to the IPPMFP solution specifications (equations 7, 8 in section 4).

6.2 Precision Measurements

6.2.1 Reaching Definitions Analysis. We randomly chose the classic reaching definitions analysis [2, 20] (non SSA based) as underlying data flow analysis, and measured the number of reaching definitions reported at each program point and added them up over all program points. A precise solution should eliminate definitions that reach a program point from infeasible CFPs. We illustrate the relative precision gain in the PPMFP and the IPPMFP solution in Figure 5. The IPPMFP solution reported up to 19% (average 7.2%) less reaching definitions compared to the MFP solution, whereas, the PPMFP solution reported a reduction of up to 9% (average 3.5%) over the MFP solution.

The detailed reaching definition numbers are presented in Table 2. In all benchmarks that had infeasible CFPs the IPPMFP solution

Table 3: Improvement in Client Analyses: reduction in 1) def-use pairs, and 2) uninitialized variable Alarms

	Def-Use Pairs			Uninitialized Variable Alarms		
	MFP	reduction(%)		MFP	reduction(%)	
		PPMFP	IPPMFP		PPMFP	IPPMFP
bzip2	72650	2267(3.1)	3632(5)	162	12(7.4)	49(30)
lbm	807	3(0.3)	24(3)	5	0(0)	3(60)
sjeng	12979	52(0.4)	130(1)	289	0(0)	0(0)
milc	14186	83(0.6)	279(2)	450	7(1.5)	53(11.7)
mcf	2064	0(0)	0(0)	30	0(0)	0(0)
libqntm	3032	0(0)	0(0)	68	0(0)	0(0)
specrnd	18	0(0)	0(0)	0	0(0)	0(0)
BCM	966	2(0.2)	2(0.2)	9	0(0)	0(0)
CNM	1525	32(2)	76(5)	7	0(0)	0(0)
EIC	229142	133(0)	169(0)	968	12(1.2)	16(1.6)
Total	337369	2572(0.7)	4312(1.3)	1988	31(1.5)	121(6)

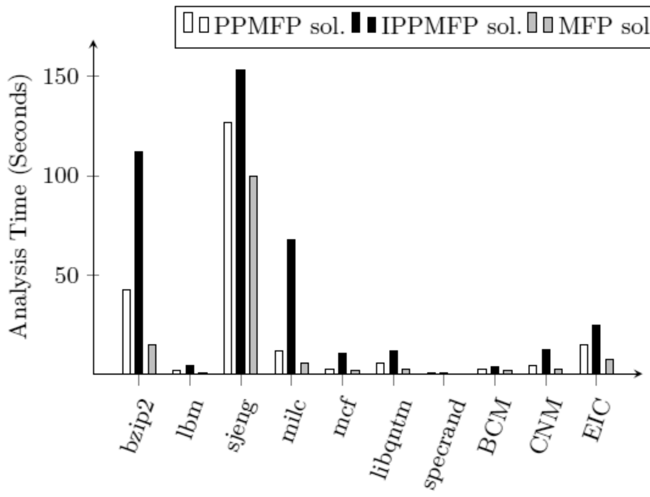
Table 4: Performance MFP vs PPMFP vs IPPMFP Solution: Prep. represents the MIPS detection phase.

	Time (sec)				Memory (MB)			
	Prep.	MFP	PPMFP	IPP-MFP	Prep.	MFP	PPMFP	IPP-MFP
bzip2	24	15	43	112	85	46	146	154
lbm	2	1	2	5	17	15	18	19
sjeng	164	100	127	153	187	46	357	373
milc	25	6	12	68	75	36	94	139
mcf	5	2	3	11	18	15	21	30
libqntm	10	3	6	12	28	18	32	29
specrnd	1	0.6	1	1	10	10	10	11
BCM	8	2	3	4	48	40	50	56
CNM	14	3	5	13	45	22	51	102
EIC	35	8	15	25	154	174	206	231

achieved better precision than the PPMFP solution. In general, the precision increased with increase in the number of MIPSs, however, on *sjeng* benchmark which had a large number of MIPSs, neither PPMFP nor IPPMFP solution show significant precision gain, because in most cases, the definitions blocked within MIPSs were still reached along alternate feasible CFPs at various nodes.

6.2.2 Def-Use Pairs. We measured the effect of reduction in number of reaching definitions on Def-Use pair computation, which uses the results of reaching definitions analysis to pair the definition of a variable with its reachable use point. The details are presented in Table 3. In summary, we found up to 5% (average 1.62%) less def-use pairs were reported when using IPPMFP solution in place of the MFP solution. The PPMFP solution has shown up to 3% (average 0.7%) reduction in the number of def-use pairs over the MFP solution. The def-use pairs are useful for program slicing, program dependency graph creation, program debugging, and test-case generation in program testing.

6.2.3 Uninitialized Variables. We implemented the *must defined variables* analysis that computes set of variables that are defined along all CFPs reaching a program point. The uninitialized variable analysis uses *must defined* data flow results to report used variables



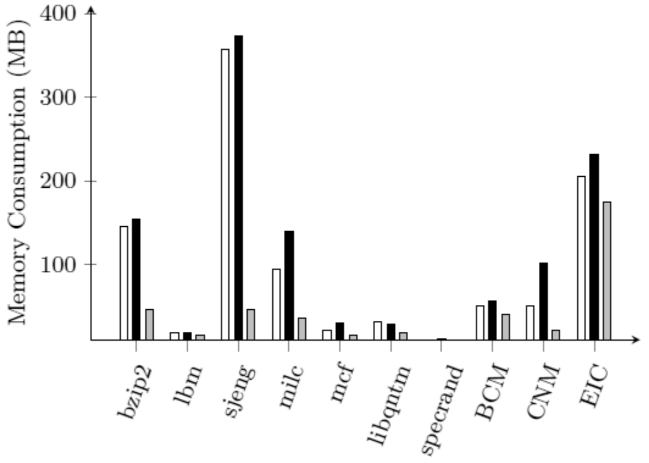


Figure 6: Performance comparison between MFP, PPMFP and IPPMFP solution. Note: The analysis times are not comparable with gcc, clang because our implementation is in Java.

at a program point *p* that are not initialized along atleast one CFP that reaches *p*, such CFPs are called witness CFPs. An uninitialized variable analysis that used MFP solution of *must defined* analysis reported 1988 alarms. When using PPMFP solution 1957 alarms were reported (a reduction of 1.5%) because in 1.5% of the cases all the witness CFPs were found infeasible. When we used IPPMFP solution 1867 alarms were reported which was a reduction of 6%. The detailed numbers of uninitialized variables are presented in Table 3.

6.3 Performance Measurements

Computation of the IPPMFP solution was less efficient compared to the MFP and the PPMFP solutions, because during IPPMFP solution computation equivalence classes with a finer granularity are created which results in a larger number of data flow values compared to that of PPMFP and MFP. The relative memory and time consumption of our analysis is illustrated in Figure 6, and details are presented in Table 4 which indicate that our analysis took an average 100% extra time and an average 20% more memory over the PPMFP computation.

6.4 Summary of Empirical Observations

Previous empirical evidence [5] on linux kernel code shows 9-40% of conditional statements contribute to at least one infeasible CFP. In our benchmarks up to 63% (average 29%) functions had at least one MIPS in their CFG. Precise elimination of data flow values reaching from infeasible CFPs has allowed us to reduce the number of reaching definitions by up to 19% with an average of 7.2% reduction over MFP solution. The improvement over recently proposed PPMFP solution was on average of 300%. Similar improvement was observed in the def-use pair computation, and the uninitialized variable analysis.

7 RELATED WORK

Trace partitioning [17, 19, 22] provides a general framework for reasoning about equivalence classes of program executions at a time; however, choosing the right criteria for executions that go in an equivalence class is usually the difference between an effective and not so effective use of trace partitioning. Towards this end, we propose a novel and effective approach for creating equivalence classes of feasible and infeasible CFPs.

The existence of the infeasible paths in practical programs is well known [12, 13, 18, 21] and various approaches have been proposed in the literature to detect them [3–7, 9, 21, 23, 25, 28, 29].

Bodik et.al [4, 5] proposed an approach to detect *minimal infeasible path segments* (MIPS), we used the same algorithms for detecting MIPSs. They used a demand driven def-use analysis to discard definitions across MIPSs. Our approach proposes an equivalence class creation that is applicable to exhaustive data flow analysis and is generically applicable to a large class of data flow analyses.

Our work builds on the work done by Komal et.al [24], they created equivalence classes of CFPs, in which the MIPSs with the same *end* edge were considered equivalent, and the CFPs sharing equivalent MIPSs were considered equivalent. Our results indicate that our approach of creating equivalence classes is more effective and does not loose precision in case the MIPSs are intersecting, whereas their work looses precision in such cases.

Many other partially path sensitive analyses exist in literature [7, 8, 10, 11, 14–16, 26, 27]. 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. However, these solutions target some specific data flow analysis, and are not as generically applicable as ours.

Some generically applicable methods [4, 26] for removing infeasible paths use CFG restructuring. CFG restructuring has exponential worst case complexity and is not performed in our approach.

Our key differentiating factor is that we maintain the same level of general applicability offered by the PPMFP solution and achieve more precision without falling back to undecidable solutions like MoP or MoFP.

8 CONCLUSIONS

Achieving a balance between scalability and precision in data flow analysis is a difficult art. In this work, we proposed a data flow solution that achieves an unique tradeoff of scalability and precision that is as generically applicable as MFP and PPMFP solutions, offers better precision than both MFP and PPMFP, and its computation is decidable unlike MoP and MoFP.

We used infeasible paths —a property of programs and not of any specific data flow analysis —to achieve this tradeoff. Any future work that allows detection of more infeasible program paths will improve our solution. Customizing the IPPMFP solution to consider data flow specific details like which infeasible paths do not affect precision, could help ignore such paths during equivalence class creation leading to increased scalability without loosing precision, our future efforts will be in the same direction.

ACKNOWLEDGMENTS

We wish to thank Kumar Madhukar, Divyesh Unadkat, Tukaram Muske, and anonymous reviewers for their valuable suggestions towards improvement of this paper.

REFERENCES

[1] 2017. TCS Embedded Code Analyzer. Retrieved 2017-09-30 from tcs.com/product-engineering/tcs-embedded-code-analyzer
[2] Alfred V Aho. 2003. *Compilers: principles, techniques and tools (for Anna University), 2/e.* Pearson Education India.
[3] 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.
[4] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. 1997. Interprocedural conditional branch elimination. In *ACM SIGPLAN Notices*, Vol. 32. ACM, 146–158.
[5] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. 1997. Refining data flow information using infeasible paths. In *Software EngineeringâĂŤESEC/FSE'97*. Springer, 361–377.
[6] 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.
[7] 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.
[8] 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.
[9] 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.
[10] Dinakar Dhurjati, Manuvir Das, and Yue Yang. 2006. Path-sensitive dataflow analysis with iterative refinement. In *International Static Analysis Symposium.* Springer, 425–442.
[11] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *ACM SIGPLAN Notices*, Vol. 43. ACM, 270–280.
[12] S Ding, H.B.K. Tan, and K.P. Liu. 2012. A survey of infeasible path detection. (01 2012), 43–52.
[13] Sun Ding and Hee Beng Kuan Tan. 2013. Detection of Infeasible Paths: Approaches and Challenges. In *Evaluation of Novel Approaches to Software Engineering*, Leszek A. Maciaszek and Joaquim Filipe (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 64–78.
[14] 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.
[15] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. 2005. Joining dataflow with predicates. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 227–236.
[16] 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.
[17] Maria Handjieva and Stanislav Tzolovski. 1998. Refining static analyses by trace-based partitioning using control flow. In *SAS*, Vol. 98. Springer, 200–214.
[18] 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.
[19] L Howard Holley and Barry K Rosen. 1981. Qualified data flow problems. *IEEE Transactions on Software Engineering* 1 (1981), 60–78.
[20] Uday Khedker, Amitabha Sanyal, and Bageshri Sathe. 2009. *Data flow analysis: theory and practice.* CRC Press.
[21] 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.
[22] Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming.* Springer, 5–20.
[23] 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.
[24] Komal Pathade and Uday P Khedker. 2018. Computing partially path-sensitive MFP solutions in data flow analyses. In *Proceedings of the 27th International Conference on Compiler Construction.* ACM, 37–47.
[25] 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.
[26] 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.
[27] 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.
[28] 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.
[29] 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.