Partial Flow-Sensitivity

Subhajit Roy Y. N. Srikant

IISc-CSA-TR-2006-12
http://archive.csa.iisc.ernet.in/TR/2006/12/

Computer Science and Automation
Indian Institute of Science, India

November 2006

Partial Flow-Sensitivity

Subhajit Roy Y. N. Srikant
Department of Computer Science and Automation
Indian Institute of Science
{subhajit,srikant}@csa.iisc.ernet.in

Abstract

We propose a partial flow-sensitivity algorithm which tries to draw on the scalability of flow-insensitive algorithms while providing more precision at some specific program points. Provided with a set of critical nodes - basic blocks at which more precise information is desired - our partially flow-sensitive algorithm computes a reduced control-flow graph by collapsing some sets of non-critical nodes. The algorithm is more scalable than a fully flow-sensitive one as, assuming that the number of critical nodes is small, the reduced flow-graph is much smaller than the original flow-graph. At the same time, a much more precise information is obtained at certain program points than would had been obtained from a flow-insensitive algorithm.

1 Introduction

Precision and scalability are the two goals that any program analysis algorithm strives for. Flow abstraction is an useful technique to attain a very high scalability, though at the cost of precision. A flow-sensitive algorithm takes the program control-flow into account to come up with a solution with high precision at each program point. On the other hand, a flow-insensitive algorithm neglects all control-flow leading to a high degree of scalability, thus coming up with a summary solution for the whole flow-graph.

It is the need for scalability that forces a lot of the analyses to be implemented as flow-insensitive algorithms. However, there are situations where a lot of opportunities for optimizations can be exploited if precise solutions are known even at a very few program points. For instance, if a profile run for a program selects a few "hot" methods, aggressive optimizations of these few methods would give high runtime gains. However, one needs to analyze well, not only the specific method, but also the caller method to discover all such opportunities. For instance, note the code shown in fig. 1.

If the only call-site for `bar()` is the one shown in the skeleton program, one could easily see that it is only '*z*' with the constant value of `5` that 'q' can ever point to. Fig. 1(b) shows the possible optimizations on `bar()`. (The pointer

```
void foo()
{
  int *x;
  int y = 3;
  int z = 5;
  ...
  x = &z;
  bar(&x);
  ...
  x = &y;
  ...
}
```

```
void bar(int *q
{
  ... = *q;
}
```

(a)

```
void bar()
{
  ... = 5;
}
```

(b)

Figure 1: A motivating example. Fig (a) shows the original source code. Fig. (b) shows the possible optimizations on procedure `bar()`.

```
typedef struct element
{
  int data1, data2, ...;
  ...
} element_t;

void populate(element_t *q)
{
  if (q) {
    q->data1 = ...
    q->data2 = ...
    ...
  }
}
```

```
void process_node()
{
  element_t *elem;
  ...
  /* allocate() always return
   * a valid element      */
  elem = allocate();
  ...
  populate(elem);
  ...
  free(elem);
  elem = NULL;
}
```

Figure 2: Yet another motivating example

dereference on `q` is replaced by a constant and also the parameter `q` is omitted from the parameter list). However, to optimize `bar()`, we not only need to analyze `bar()`, but also need to analyze `foo()` reasonably well.

Note that a flow-insensitive pointer analysis would miss the fact that the parameter 'q' in `bar()` can only point to 'z'. A flow-sensitive pointer analysis would surely indicate that fact; however, it would be hugely expensive. This leads us to an important observation : even if `foo()` was analyzed well enough to provide a better estimate of program properties at only the program points where `bar()` is called, our purpose would be served.

Fig. 2 presents another example. The function `process_node()` allocates a new element and passes it to the `populate()` method. The `populate()` method checks if the argument passed is not `NULL` and then proceeds to populate the same. The important thing to note is that it is never possible that the argument `q` passed to `populate()` is `NULL` (as the `allocate()` method is supposed to always return a valid element[1]). Hence the check "`if (q)`" could be eliminated from `populate()` if the above fact can be discovered. As in the previous example, a flow-insensitive analysis would simply miss the fact and a flow-sensitive algorithm would be very expensive.

We propose a middle path between full flow-sensitivity and flow-insensitivity - that of *partial flow-sensitivity.* If we preserve some partial flow-sensitivity at some few points in a program, we might be able to discover opportunities to optimize which a flow-insensitive algorithm would miss, without losing much on scalability. Our partially flow-sensitive algorithm expects from the user a set of program points (currently we accept the specification in the granularity of basic-blocks) where we are interested in having a better estimate of the program-properties. We term such basic-blocks as *critical nodes* in the program's flow-graph. Our algorithm then goes about merging some sets of non-critical nodes to compute a *reduced flow-graph (r-CFG)*, on which we perform flow-sensitive analysis. Assuming such critical nodes to be a very small fraction of all the nodes in the flow-graph, the algorithm achieves the desired scalability by running over a very small flow-graph. In fact, the scalability is then a function of the fraction of the nodes selected as critical - the algorithm reduces to a completely flow-insensitive algorithm when none of the nodes are critical, to a completely flow-sensitive one, when all the nodes are selected critical.

The reduced flow graph for fig. 2 is shown in fig. 3(b) with the node containing the call to `populate()` marked critical. One can see that a flow-sensitive points-to analysis on this much smaller flow-graph of the method `process_node()` can deduce the fact that the parameter `q` in `populate()` can only point to valid memory locations and hence the null-check in `populate()` is redundant.

Here we provide a road-map for the paper. Section 2 discusses previous work. Section 3,4 and 5 explain the core algorithm. It starts off by explaining how the non-critical nodes are merged to form a reduced flow-graph and also describe the types of the transfer functions to be used for the critical and the merged non-critical nodes in the reduced CFG. Section 6 demonstrates a partially flow-sensitive points-to analysis algorithm. This section gives a concrete example on how a practical analysis can be implemented using the framework developed in sections 3,4 and 5. Section 7 talks of our implementation. Section 8 lays out our future plans and other directions of research in this area.

---

[1] The code for `allocate()` is not shown here. However, such a method can be simply be implemented by spinning in a loop till the `malloc()` call succeeds.

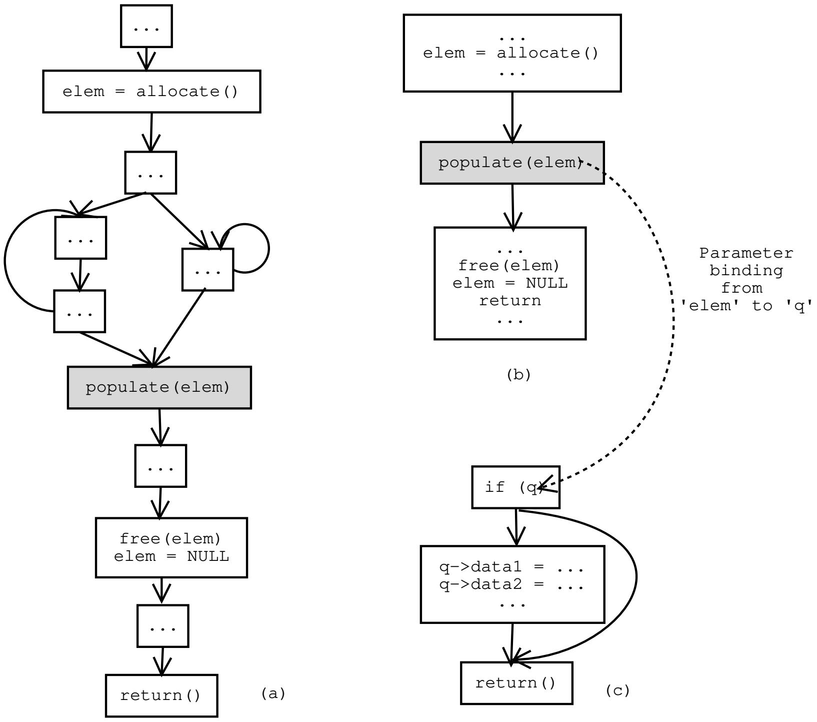


Figure 3: The reduced CFG for the program in Fig. 2. They gray node is the critical node selected. Fig. (a) shows the original CFG for `process_node()`. Fig. (b) shows the reduced of `process_node()` obtained via the PFS algorithm. Fig. (c) shows the CFG for the function `populate()`.

2 Previous Work

[4, 6, 8] provide a good survey of various control-flow analysis techniques.

We used the Datalog logic programming language and the bddbddb[12] tool for part of our implementation. Ullman [19] suggested that dataflow-analyses can be formulated as database queries. [21] used the Corel logic programming language to implement demand-driven interprocedural analysis. However, good scalability for implementing program analyses using logic language came with bddbddb[12], a solver for Datalog with stratified negation. [14] describes techniques on developing program analyses using BDDs and Datalog. [11, 15] also provide valuable information on using BDDs for developing scalable analyses.

We show the applicability of our algorithm by implementing a partially flow-sensitive pointer analysis. [18] proposes a flow-sensitive interprocedural alias analysis which they claimed was, at that time, the most precise and efficient interprocedural method known. Anderson[1] described a flow-insensitive subset based algorithm based on constraint solving which computes a single solution for the whole program. Steensgard [3] came up with an almost linear time algorithm for computing points-to relations. He used the fast union-find data-structure to represent an alias relation. However, it is less precise than Andersons's algorithm due to the merging of alias sets in the union-find data-structure. More recently, [9] reported a new approach to solving subset-based points-to analysis for Java using Binary Decision Diagrams (BDDs). [20] provides a nice comparative survey of the various pointer analysis algorithms.

[16] provides two valuable schemes for interprocedural analyses : summary-based and call-string based. The call-string based approach computes the interprocedural solution by propagating the information over an interprocedural flow-graph. However, each such propagation of information is tagged with a call-string - an encoded history of procedure calls encountered during the propagation. This enables it to determine the valid interprocedural paths at procedure returns, thus weeding out all paths that have a conflicting call history.

3 The Partial Flow-Sensitivity Algorithm (PFS Algo.)

The PFS algorithm allows the user to specify a set of *critical nodes* from the program's flow-graph. We define *Critical Nodes* as basic blocks at which the user is interested in having a more precise information about some program properties. For instance, we implement a points-to analysis over the PFS algorithm where we select basic blocks having calls to some specific functions as critical nodes. The selected functions could be "hot" methods identified through a profile run. A better estimate of the points-to sets of the passed arguments at their call-sites could enable us to drive better optimizations within these "hot" methods.

3.1 Computing the reduced CFG (r-CFG)

Depending on the critical nodes selected, the PFS algorithm computes a reduced control-flow graph by collapsing some sets of non-critical nodes. Finally a flow-sensitive analysis algorithm is run on the reduced flow-graph. We propose two techniques for creating the r-CFG, an heuristic approach which could generally result in a smaller r-CFG but is possibly more imprecise and an optimal algorithm which satisfies two optimality critiria which we define. We discuss these two techniques in the next two sections.

3.2 The Analysis Phase

3.2.1 Intraprocedural Analysis

We define either of two types of transfer functions for each basic-block in the r-CFG - a *weak transfer function* and *strong transfer function.* The weak transfer functions perform flow-insensitive updates over all the nodes belonging to a aggregate node in the r-CFG while a strong update does flow-sensitive updates.

The non-critical aggregate nodes in the r-CFG - with the control-flow within the constituent nodes smudged - use the weak transfer function. The critical nodes retain their identity even in the r-CFG and hence use the strong transfer functions. Finally, we perform a flow-sensitive analysis the over the whole r-CFG to generate the required program properties.

3.2.2 Interprocedural Analysis

Interprocedural analysis is performed using the k-limit call-string approach [16]. For critical nodes, the actual parameters carry the program properties existing at the call-site into the callee. For non-critical nodes, if the call-site is in $x \in nodes(G_{orig})$ and $x \in x', x' \in nodes(G_{red})$, then the analysis information from $OUT_{x'}$ is made to pass into the callee.

For procedure return at a basic block n in the callee, it is always the $OUT_{\{x|x=collaped\_id(n)\}}$ value that is passed back to the caller.

3.3 Safety

<u>Claim:</u> The PFS algorithm is safe.

Consider a single edge from a node from $a_x \to a_y$. If any of conditions enumerated in section 3.1 holds, it will disallow the merge and hence the edge would be preserved. If, however, all the required conditions are met and the merge actually takes place, then the aggregate node would be marked non-critical (as both original nodes must be non-critical for the conditions to hold) in the r-CFG. Hence, the aggregate node will still preserve the edge $a_x \to a_y$ as a flow-insensitive analysis would be performed within the nodes of this non-critical aggregate node.[2]

As all the control-flow edges in the CFG are also preserved in the r-CFG, any path in the CFG is traceable in the latter. Thus, the PFS algorithm does not "miss" any flow-path along which the program properties could propagate and hence is safe.

4 An heuristic apporach to compute the r-CFG

4.1 Computing the reduced flow-graph (r-CFG)

To compute the reduced CFG (r-CFG), the algorithm starts-off by removing all the backedges from the original control-flow graph (CFG) to form a directed acyclic graph (DAG). Each node in the DAG is initialized to a unique value for an identifier (`collapse_id`).

Then, edges in the DAG are picked in an arbitrary order, and its constituent nodes are merged into a non-critical aggregate node in the r-CFG, provided a certain conditions (discussed below) are met. This operation is carried out

---

[2] Using flow-insensitive analysis over a set of statements can be seen as a flow-sensitive analysis over a complete flow-graph formed with these set of statements.

by setting the `collapsed_id` of both these nodes to the maximum of their `collapsed_id` values. At the end this phase, `collapse_id` values are used to mark the nodes belonging to the same aggregate node in the r-CFG. Fig. 5 shows how the nodes `5` and `6` are conceptually merged into the same aggregate node by setting both their `collapse_id` as `6`.

Finally, the backedges in the original flow-graph are re-inserted into the DAG (making it a graph). A backedge is included in the r-CFG if it maps to an edge between two nodes with different `collapse_id` values (any self-loops on critical nodes are obviously retained).

Any of the following conditions disallows the collapsing of an edge $a_y \rightarrow a_z$for creating an aggregate node:

Condition I: The predecessor node a_y is critical
Condition II: Any of the successors of the predecessor is critical
Condition III: For the edge $a_y \rightarrow a_z$, there exists another path from a_y to a_z which contains a critical node

Condition I is required as we never merge any node with a critical node so as to preserve more precise information at critical nodes.

A violation of condition II is shown in fig. 6. We are interested in merging the edge `5->6`, where `7` is a successor of `5` and is a critical node. Note that if we merge nodes `5` and `6`, we end up traversing an edge `6->7` which does not actually exist in the actual flow-graph, thus losing precision for the critical node `7`.

A violation of condition III is shown in fig. 7. For the nodes `5` and `7`, other than the edge `5->7`, there exists another path which contains the critical node `6`. We collapse the edge `5->7`, merging the corresponding nodes. Note that this merge generates a pseudo-path from 7 to 6 . (This aggregate node could again merge with other successors of 7, making the matter worse). Still worse, it causes a cycle in the DAG (which thus ceases to be a DAG).

However, condition III is more difficult to check than conditions I and II. It reduces to solving the problem stated below for each edge in the DAG.

Problem: Given a directed acyclic graph G, and an edge $a_i \rightarrow a_j$, find if there exists a path from a_i to a_j such that it passes though any node in the set of critical nodes $C = \{c_1, c_2, \ldots, c_n\}$.

The algorithm for the same is described in fig. 4.

Fig. 8 describes the algorithm for computing the reduced graph. The relation $collapse(x, y)$ holds if the edge $x \rightarrow y$ is collapsed to form the aggregate node containing `x` and `y` in the r-CFG. $collapsed\_id(x, y)$ states that the collapsed_id of node x is set to y. Equation 3.5 enumerates the conditions under which a node x_f in the original flowgraph becomes a member of the aggregate node x'_f in the reduced flowgraph. The final equation defines the edges in the r-CFG.

Note that the r-CFG is a mere conceptual flow-graph; it need not be created in practice. The r-CFG computation phase of the PFS algorithm just assigns the `collapse_id` values to the nodes. The rest of the algorithm works by identifying

We use the following scheme of weight assignment in the adjecency matrix of the graph.

$$w[a_i, a_j] = \begin{cases} -1 & \text{if the edge } a_i \to a_j \text{ exists and either } a_i \in C \text{ or } a_j \in C \\ 0 & \text{if the edge } a_i \to a_j \text{ exists and } a_i \notin C \text{ and } a_j \notin C \\ \infty & \text{if the edge } a_i \to a_j \text{ does not exist} \end{cases}$$

We run the Floyd-Warshall all-pairs shortest path algorithm on the above weighted adjacency matrix, which computes the all-pairs shortest-distance matrix *dist*. If possible, the shortest path between any two nodes now must pass through the nodes $n \in C$ as the edges containing such nodes have a negetive cost. Hence, for an edge $a_i \to a_j$, we can conclude that an alternate path containing a node $n \in C$ exists iff $dist[a_i, a_j] < 0$.

Figure 4: Algorithm for detecting the existence of paths though critical-nodes

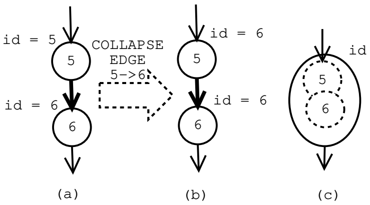


Figure 5: Collapsing nodes

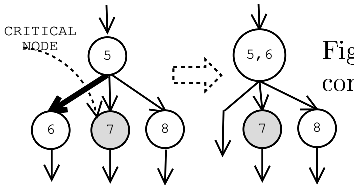


Figure 6: Violation of condition 2

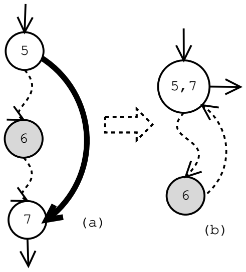


Figure 7: Violation of condition 3

aggregate nodes as the set of all nodes sharing the same `collapse_id` value.

Fig. 3 shows the original and the reduced CFG for the example code in fig. 2. The call to the `populate()` function is selected as the critical node.

4.2 Taking care of special cases

There are a few special cases which requires transforming the original control-flow graph before starting to collapse nodes to get the r-CFG.

$$\frac{\begin{array}{l} f \in \mathcal{F} \quad x_f, y_f \in \mathcal{N} \quad x_f \in \mathit{pred}(y_f) \quad \neg \mathit{backedge}(x_f, y_f) \quad \neg \mathit{critical}(x_f)) \\ \mathit{no\_alt\_path\_with\_critical\_node}(x_f, y_f) \end{array}}{\mathit{collapse\_allowed}(x_f, y_f)} \tag{4.1}$$

$$\frac{f \in \mathcal{F} \quad x_f, z_f \in \mathcal{N} \quad x_f \in \mathit{pred}(z_f) \quad \mathit{critical}(z_f)}{\mathit{has\_critical\_succ}(x_f)} \tag{4.2}$$

$$\frac{f \in \mathcal{F} \quad x_f, y_f \in \mathcal{N} \quad \mathit{collapse\_allowed}(x_f, y_f) \quad \neg \mathit{has\_critical\_succ}(x_f)}{\mathit{collapse}(x_f, y_f)} \tag{4.3 a}$$

$$\frac{f \in \mathcal{F} \quad x_f, y_f \in \mathcal{N} \quad \mathit{collapse}(x_f, z_f) \quad \mathit{collapse}(z_f, y_f)}{\mathit{collapse}(x_f, y_f)} \tag{4.3 b}$$

$$\frac{f \in \mathcal{F} \quad x_f, y_f \in \mathcal{N} \quad \prod_{\mathit{collapse}(x_f, y_f) \vee \mathit{collapse}(y_f, x_f)} (y_f > x_f)}{\mathit{collapsed\_id}(x_f, y_f)} \tag{4.4}$$

$$\frac{f \in \mathcal{F} \quad x_f, z_f \in \mathcal{N} \quad x_f \in \mathit{nodes}(G_{orig}) \quad x'_f \in \mathit{nodes}(G_{red}) \quad \mathit{collapsed\_id}(x_f, x'_f)}{x_f \in x'_f} \tag{4.5}$$

$$\frac{\begin{array}{llll} f \in \mathcal{F} & x_f, y_f \in \mathit{nodes}(G_{orig}) & x'_f, y'_f \in \mathit{nodes}(G_{red}) & \\ x_f \in \mathit{pred}(y_f) & \mathit{collapsed\_id}(x_f, x'_f) & \mathit{collapsed\_id}(y_f, y'_f) & x'_f \neq y'_f \end{array}}{x'_f \in \mathit{pred}(y'_f)} \tag{4.6}$$

Figure 8: Computing the reduced CFG(r-CFG). The algorithm is described as a fix-point iteration over a set of functions F and a set of CFG nodes N.

4.2.1 Inserting Merge-Dummy nodes

Consider fig. 9(a). The critical node, `8`, does not allow `5,6,7` to get collapsed although the program properties flowing through these nodes are anyway merged at the beginning of the critical node `8`. Hence, we introduce a dummy non-critical node 'D' to allow `5,6,7` to collapse (shown in fig. 9(b)).

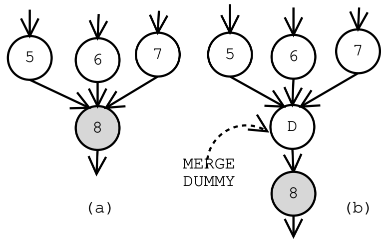


Figure 9: Inserting Merge-Dummy nodes

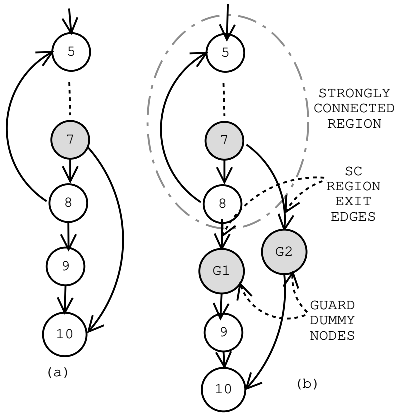


Figure 10: Inserting Guard-Dummy node

4.2.2 Inserting Guard-Dummy nodes

Consider fig. 10(a). The loop consisting of the nodes `5,..,7,8` contains a critical node `7`. However, the CFG reduction algorithm works only on the DAG, neglecting the backedge `8` to `5`. Hence, the algorithm will lead to the merging of nodes `8, 9, 10` etc., causing program properties generated due to statements in `9, 10` etc. to enter `7`, passing through the backedge.

To prevent the same, we compute the strongly-connected components (SCC) for the program's flow-graph. If any of the SCCs contain a critical node, we insert a guard dummy node at each exit edge from the SCC and mark it critical. (If the destination of such an edge is already critical, inserting the dummy node is not required for that edge.). Fig. 10(b) shows how a guard node is inserted.

4.3 Results on the r-CFG

<u>Claim</u>: The reduced CFG obtained is canonical.

The fact that the reduced CFG obtained by the PFS algorithm is canonical can be easily understood by the fact that assigning a node a new `collapse_id` still keeps the node non-critical. Moreover, as the new `collapse_id` is assigned as per an associative function (the `max()` function), all the nodes get the same `collapse_id` assigned irrespective of the order of picking the edges in the CFG for collapsing.

$$\frac{\begin{array}{l} y_f \in \mathit{pred}(x_f) \qquad\qquad z_f \in \mathit{pred}(x_f) \qquad\qquad y_f \neq z_f \\ \neg \mathit{backedge}(y_f, x_f) \qquad\qquad \neg \mathit{backedge}(z_f, x_f) \\ \mathit{critical}(x_f) \quad \neg \mathit{critical}(y_f) \quad \neg \mathit{critical}(z_f) \quad \mathit{dummy}_f = \mathit{mg\_dummy}_{x,f} \end{array}}{\mathit{insert\_merge\_dummy}(x_f, \mathit{dummy}_f)} \tag{4.7}$$

$$\frac{\begin{array}{l} \mathit{edge\_from} \in \mathit{pred}(\mathit{edge\_to}) \\ \mathit{id}_f = \mathit{scc\_id}(\mathit{edge\_from}_f) \quad \mathit{id} \neq \mathit{scc\_id}(\mathit{edge\_to}_f) \end{array}}{\mathit{edge}(\mathit{edge\_from}_f, \mathit{edge\_to}_f) \in \mathit{scc\_exit\_edge}(\mathit{id}_f)} \tag{4.8 a}$$

$$\frac{\begin{array}{l} \mathit{critical}(n_f) \quad \mathit{id}_f \in \mathit{scc\_id}(n_f) \quad \mathit{edge}(x_f, y_f) \in \mathit{scc\_exit\_edge}(\mathit{id}_f) \\ \neg \mathit{critical}(y_f) \quad x_f \neq n_f \qquad\qquad \mathit{dummy}_f = \mathit{gd\_dummy}_{y,f} \end{array}}{\mathit{insert\_guard\_dummy}(x_f, y_f, \mathit{dummy}_f)} \tag{4.8 b}$$

$$\frac{\begin{array}{l} x_f, y_f \in G \quad x_f, y_f, \mathit{dummy}_f \in H \quad x_f \in \mathit{pred}_G(y_f) \\ \mathit{insert\_merge\_dummy}(y_f, \mathit{dummy}_f) \end{array}}{x_f \in \mathit{pred}_H(\mathit{dummy}_f)} \tag{4.9 a}$$

$$\frac{\begin{array}{l} x_f, y_f \in G \quad x_f, y_f, \mathit{dummy}_f \in H \quad x_f \in \mathit{pred}_G(y_f) \\ \qquad\qquad \mathit{insert\_merge\_dummy}(x_f, \mathit{dummy}_f) \end{array}}{\mathit{dummy}_f \in \mathit{pred}_H(y)} \tag{4.9 b}$$

$$\frac{\begin{array}{l} x_f, y_f \in G \qquad x_f, y_f, dx_f, dy_f, \mathit{dum}_f \in H \qquad x_f \in \mathit{pred}_G(y_f) \\ \neg \mathit{insert\_merge\_dummy}(x_f, dx_f) \quad \neg \mathit{insert\_merge\_dummy}(y_f, dy_f) \\ \neg \mathit{insert\_guard\_dummy}(x_f, y_f, \mathit{dum}_f) \\ dx_f = \mathit{mg\_dummy}_{x,f} \quad dy_f = \mathit{mg\_dummy}_{y,f} \quad \mathit{dum}_f = \mathit{gd\_dummy}_{y,f} \end{array}}{x_f \in \mathit{pred}_H(y_f)} \tag{4.9 c}$$

$$\frac{\begin{array}{l} x_f, y_f \in G \qquad\qquad x_f, y_f, \mathit{dummy}_f \in H \qquad\qquad x_f \in \mathit{pred}_G(y_f) \\ \quad \mathit{insert\_guard\_dummy}(x_f, y_f, \mathit{dummy}_f) \quad \mathit{dummy}_f = \mathit{gd\_dummy}_{y,f} \end{array}}{x_f \in \mathit{pred}_H(\mathit{dummy}_f)} \tag{4.9 d}$$

$$\frac{\begin{array}{l} x_f, y_f \in G \qquad\qquad x_f, y_f, \mathit{dummy}_f \in H \qquad\qquad x_f \in \mathit{pred}_G(y_f)) \\ \quad \mathit{insert\_guard\_dummy}(x_f, y_f, \mathit{dummy}_f) \quad \mathit{dummy}_f = \mathit{gd\_dummy}_{y,f} \end{array}}{\mathit{dummy}_f \in \mathit{pred}_H(y))} \tag{4.9 e}$$

5 An Optimal Algorithm to compute the r-CFG

5.1 Yardsticks for a Partial Flow Sensitivity algorithm

We define the notions of Precision and Size optimality to determine how good a Partial Flow Sensitive algorithm is.

5.1.1 Precision Optimality

The reduced CFG is said to be precision optimal if the following holds for each critical node c: if there does not exist a path from any node $n \to c$ in the original flow graph, such a path would not exist even in the reduced flow-graph. In other words, if there exists a path, $n \to c$, in the reduced graph, such a path surely exists in the original graph.

5.1.2 Size Optimality

The reduced CFG is said to be size optimal, if there do not exist nodes n_i and n_j s.t. merging them still maintains precision optimality. In other words, we cannot find a smaller reduced CFG which still satisfies precision optimality.

5.2 The Algorithm

We observed the following property must hold for precision optimality.

To maintain precision optimality w.r.t a critical node c, two non-critical nodes n_i and n_j can be merged iff either of the two conditions hold:

1. Both n_i and n_j have paths to the critical node c
2. None of n_i or n_j have a path to the critical node c

Without loss of generality, we assume that n_i has a path to the critical node c while n_j has no such path. Merging n_i and n_j (thus disobeying the property) creates a pseudo-path from n_j to c. The solution thus fails to have precision optimality w.r.t the critical node c.

Our algorithm (Fig. 12) works by merging nodes while maintaining the property 5.2 for all critical nodes.

Equation 1 and 2 compute the set $\rho(n)$ for all the nodes $n \in G_{orig}$. Here $\rho(n)$ represents a set of all critical-nodes which are reachable from the node n. This computation can be set up as a simple bit-vector dataflow problem where each bit in the bit-vector stands for a particular critical-node. An extra bit could be used to distinguish critical nodes from non-critical nodes.

Equation 3 represents the condition when two nodes are not mergeable - when one of them reaches a certain critical node c and the other does not. Equation 4 represents the symmetric nature of this relation.

Equation 5 finally specifies when two nodes can actually be merged - when none of them is critical and the predicate $not\_mergeable()$ does not forbid their merge.

$$\frac{crit, node \in G_{orig} \quad critical(crit) \quad node = crit}{crit \in \rho(node)} \tag{5.1}$$

$$\frac{crit, node, succ \in G_{orig} \quad node \in Pred(succ) \quad crit \in \rho(succ)}{crit \in \rho(node)} \tag{5.2}$$

$$\frac{n_1, n_2, c \in G_{orig} \quad critical(c) \quad c \in \rho(n_1) \quad c \notin \rho(n_2)}{not\_mergeable(n_1, n_2)} \tag{5.3}$$

$$\frac{n_1, n_2 \in G_{orig} \quad not\_mergeable(n_2, n_1)}{not\_mergeable(n_1, n_2)} \tag{5.4}$$

$$\frac{n_1, n_2 \in G_{orig} \quad n_1 \neq n_2 \quad \neg critical(n_1) \quad \neg critical(n_2) \quad \neg not\_mergeable(n_1, n_2)}{merge(n_1, n_2)} \tag{5.5}$$

Figure 12: The algorithm

The reduced CFG is created by representing all the nodes merged together by a single aggregate node and suppressing all edges between these merged nodes. Actually, this reduced CFG is just a conceptual graph - it need not be created. All the nodes belonging to the same aggregate node are simply assigned a common unique *merge ID*. This *merge ID* identifies each of the aggregate node in the original CFG. All the later algorithms actually work on the original CFG, identifying the aggregate nodes by the *merge IDs*.

5.3 Analyzing the algorithm

5.3.1 The above algorithm produces a precision optimal reduced CFG.

We prove the same by contradiction. Assume there exists a node n_i such that, for some critical node c, there exists a path $n_i \rightarrow c$ in $G_{reduced}$ but not in G_{orig}. Such a path could only be created due to a merge of the node n_i with some node n_j, where n_j had a path to c in G_{orig}(See Fig. 13). This implies that $\rho(n_i) = \rho(n_j)$ i.e both n_i and n_j reach the same set of critical nodes (as otherwise the algorithm would not have merged the nodes). This causes a contradiction as the critical node $c \in \rho(n_j)$ but $c \notin \rho(n_i)$.

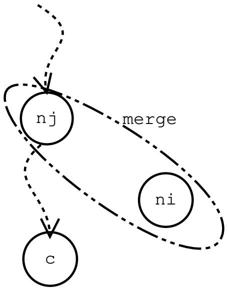


Figure 13: Proving that the r-CFG is precision optimal

5.3.2 The above algorithm produces a size optimal reduced CFG.

The prove the same by contradiction. Assume that there exists two distinct nodes n_i, n_j, not merged by the above algorithm, such that merging them still maintains precision optimality of the reduced CFG. Obviously, there exists atleast one critical node c such that $c \in \rho(n_i)$ and $c \notin \rho(n_j)$ [3] - otherwise the above algorithm would have merged n_i and n_j. In that case, merging these nodes would compromise the precision optimality w.r.t the critical node c as it would create a pseudo-path from n_j to c. This contradicts our assumption.

5.4 Size of the reduced CFG

Following are some results on the size of the r-CFG.

Lemma: The above algorithm partitions the non-critical node into 2^n equivalence classes, where n is the number of critical nodes.

Proof:The above algorithm allows two non-critical nodes to be collapsed iff they reach exactly the same set of critical nodes. It is obvious that the relation[4] is reflexive, symmetric and transitive - thus, an equivalence relation.

As each equivalence class corresponds to a possible subset of the set of critical nodes, there can only be 2^n such equivalence classes.

Claim: The number of nodes in the reduced CFG is bounded by $2^n + n$ where n is the number of critical nodes. Also, the same is a tight bound.

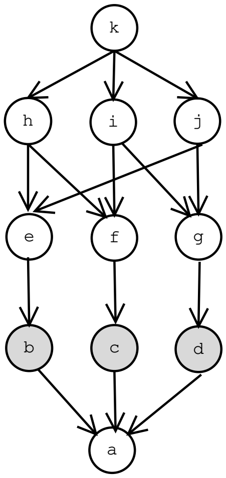
Proof: According to above lemma, as each of the non-critical node must belong to one of the equivalence classes and there are atmost 2^n such classes - the reduced CFG can atmost have 2^n aggregate nodes formed by merging of the non-critical nodes. None of the n critical nodes is merged with any other node. Hence, the reduced CFG can almost have $2^n + n$ nodes.

Also, the above bound is tight. Fig. 14 shows a case where the above bound is actually reached for n=3. The value of $\rho(n)$ for all non-critical nodes is shown in the table. Note that none of the nodes can be merged. A graph of similar structure can be constructed for any value of n.

---

[3] the other side is symmetric

[4] nodes a and b are related iff they reach exactly the same set of critical nodes



Node (n)	$\rho(n)$
a	{}
e	{b}
f	{c}
g	{d}
h	{b,c}
i	{c,d}
j	{b,d}
k	{b,c,d}

Figure 14: An example illustrating the tight bound of the number of nodes in the reduced CFG

6 Partially Flow-sensitive Points-to Analysis

We selected points-to analysis for demonstrating the PFS algorithm. One primary reason for selecting points-to analysis is that, currently, this analysis is invariably being implemented as a flow-insensitive, context-sensitive analysis. However, as discussed in the introduction, good optimization opportunities might be discovered if more precise points-to sets can be found, even at a very few program points.

6.1 Updating points-to sets for each statement type

The operational semantics for updating of points-to sets is very similar to that proposed earlier in literature [1, 14, 2, 9, 3]. We introduce two new predicates : `garbage_value()` and `null_value()`, which indicate that the respective variable holds an unpredictable value and the value `NULL` respectively. For heap allocations, we summarize the heap locations allocated at a particular allocation-site by a heap variable $heap\_var_{alloc-site}$.

The update semantics for the various statements is shown in fig. 15. $\pi_{cs,pp}(x, y)$ denotes the points-to relation that `x` points-to `y` at the program point `pp` under the call-string `cs`.

Note that our semantics of the `malloc()` call is similar to the `allocate()` call (discussed in the introduction) as we assume that `malloc()` always succeeds and returns with the address of a valid memory location.

$$\frac{stm_{pp}(\text{''}x = y\text{''}) \quad \pi_{cs,pp}(y,z)}{\pi_{cs,pp}(x,z)} \tag{6.1}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}x = *y\text{''}) \\ \pi_{cs,pp}(y,y1) \quad \pi_{cs,pp}(y1,z) \end{array}}{\pi_{cs,pp}(x,z)} \tag{6.2}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}*x = y\text{''}) \\ \pi_{cs,pp}(x,z) \quad \pi_{cs,pp}(y,y1) \\ \neg garbage\_value(z) \\ \neg null\_value(z) \end{array}}{\pi_{cs,pp}(z,y1)} \tag{6.3}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}*x = *y\text{''}) \\ \pi_{cs,pp}(x,x1) \\ \pi_{cs,pp}(y,y1) \quad \pi_{cs,pp}(y1,y2) \\ \neg garbage\_value(x1) \\ \neg null\_value(x1) \end{array}}{\pi_{cs,pp}(x1,y2)} \tag{6.4}$$

$$\frac{stm_{pp}(\text{''}x = \&y\text{''})}{\pi_{cs,pp}(x,y)} \tag{6.5}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}x = malloc(\cdots)\text{''}) \\ h = heap\_var_{pp} \end{array}}{\pi_{cs,pp}(x,h)} \tag{6.6}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}x = malloc(\cdots)\text{''}) \\ garbage\_value(g) \end{array}}{\pi_{cs,pp}(x,g)} \tag{6.7}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}free(x)\text{''}) \\ garbage\_value(g) \end{array}}{\pi_{cs,pp}(x,g)} \tag{6.8}$$

$$\frac{\begin{array}{l} stm_{pp}(\text{''}x = NULL\text{''}) \\ null\_value(NULL) \end{array}}{\pi_{cs,pp}(x,NULL)} \tag{6.9}$$

Figure 15: Points-to set update semantics. $\pi_{cs,pp}(x,y)$ denotes the points-to relation that `x` points-to `y` at the program point `pp` under the call-string `cs`. The $stm_{pp}(S)$ indicates that the $\boldsymbol{S}$ is statement encountered at program point $\boldsymbol{pp}$.

For a critical node (which undergoes flow-sensitive analysis), a variable gets its previous points-to set killed if it gets assigned a new value unambiguously and it is not a heap variable.

$$\frac{\begin{array}{r} stm_{pp}(\text{''}x = y\text{''}, \text{''}x = \&y\text{''}, \text{''}x = *y\text{''}, \text{''}x = NULL\text{''}, \text{''}x = malloc(\cdots)\text{''}, \text{''}free(x)\text{''}) \\ \neg heap\_node(x) \end{array}}{killed_{cs,pp}(x)}$$

$$
\begin{aligned}
\pi^{strong}_{cs,pp}(x,y) = & \pi_{cs,pp}(x,y) \\
& \vee \begin{cases} \pi^{strong}_{cs,pp-1}(x,y) & \text{if } \neg killed_{cs,pp}(x) \\ \phi & \text{if } killed_{cs,pp}(x) \end{cases} \\
& \vee \begin{cases} \pi^{IN}_{cs,node}(x,y) & \text{if } pp \in node \text{ and pp is first statement in node} \\ \phi & \text{otherwise} \end{cases}
\end{aligned}
$$

$$
\pi^{weak}_{cs,node}(x,y) = \sum_{\forall nodes_f(n), pp \in n, critical\_id_f(n) = critical\_id_f(node)} \{\pi_{cs,pp}(x,y) \vee \pi^{IN}_{cs,n}(x,y)\}
$$

$$
\pi^{OUT}_{cs,node}(x,y) = \begin{cases} \pi^{strong}_{cs,pp}(x,y) & \text{if } critical_f(node) \wedge \text{ pp is last stm in node} \\ \pi^{weak}_{cs,node}(x,y) & \text{if } \neg critical_f(node) \end{cases}
$$

$$
\pi^{IN}_{cs,node}(x,y) = \sum_{p \in pred_f(node)} \pi^{OUT}_{cs,p}(x,y)
$$

Figure 16: The transfer functions. ***cs*** refers to the call-string, ***pp*** and ***node*** refer to the program point or the basic-block where the relation is being computed and ***f*** indicates that the relation is being defined for the procedure `f`.

6.2 Intraprocedural Analysis

As discussed in section 3, we have two transfer functions: strong and weak. The strong transfer function only adds points-to sets from the previous program point if the same is not killed. The weak transfer function performs flow-insensitive analysis (similar to Anderson's [1]) by performing a fixed point iteration over all the statements in all the nodes belonging to the respective aggregate node.

Note that the strong points-to relations are defined for each program point while the weak points-to relations are defined for aggregate nodes. (The equations define it for each node for the original CFG; however, each such node belonging to the same aggregate node in the r-CFG will share the same points-to sets).

Finally, the program properties of a node in the r-CFG is summarized into π^{IN} and π^{OUT} sets. See fig. 16 for the respective equations.

6.3 Interprocedural Analysis

Fig. 17 shows the semantics for our partially flow-sensitive points-to analysis. For simplicity, we assume that a function call is the first statement in a basic block, global variables are absent and parameters are passed by value.

For a given caller and callee call-strings, at a particular program point `caller_pp`, the $param\_binding_{cs\_caller,cs\_callee,caller\_pp}()$ relations binds the formal parameters to the actual parameters. The parameter list is indexed by the identifier ***param_id***. The corresponding actual and formal parameters bind via the same value of $param\_id$ in the relations $actual\_param_{f,param\_id}$ and $formal\_param_{f,param\_id}$. The $create\_call\_string(cs_{caller}, func_{caller}, caller_{pp}, func_{called}, cs_{callee})$ predicate evaluates to `true` if the $func_{caller}$, operating with call-string cs_{caller} calls the function $func_{called}$ at the program point $caller_{pp}$, and the called functions starts operating with a call-string cs_{callee} (The call-string is created according to the k-limit call-string approach [16], i.e. append the caller procedure to the call-string; if the string length exceeds some maximum allowed length 'k', then remove a call from the beginning of the call-string).

The $\pi^{IN}_{cs,n}$ relation defines the function call semantics, i.e. when the callee with call string $\boldsymbol{cs}$ and entry node $\boldsymbol{n}$ is called. The equation first binds each formal parameter `x` with the respective actual parameter `y`. It then picks up the respective point-to sets of the actual parameter from the caller and assigns it to its corresponding formal parameter in the callee. A more precise estimate of the parameters is obtained for the critical nodes as the $\pi^{IN}_{cs,n}$ relation of the callee picks up the points-to relations from the $\pi^{IN}_{cs,n}$ of the caller; while for the non-critical aggregate nodes the $\pi^{OUT}_{cs,n}$ of the calling function is picked. We assume that the basic-blocks is split at each function call; hence, we can have atmost one function call per basic-block.

The $\pi^{RET}_{cs,pp}$ relation defines the semantics of a return statement. The relation $del\_last\_call()$ generates call-strings on returning to the caller as per the k-limit call-string approach (remove the callee from the call-string and create call-strings by prepending functions calling the caller procedure). If the callee returns with a statement "`return(y)`" to the caller who had initiated the call with "`x=func_callee(...)`", the equation simply sets the the points-to set of the the variable `x` to that of the return parameter `y` .

The π^{MOD} relation updates all points-to relations generated due to indirect references in the called procedure. It simply updates points-to relations in the caller from that in the callee if the variables being updated are not local variables or formal parameters in the callee. (We omit a case in the equation for simplicity: if any of the points-to relations being imported from the callee to the calling procedure is a local variable or formal parameter in the callee, set the same to a special variable indicating a garbage value as these stack variables would not be available to the caller).

The final equation finally updates the $\pi^{strong}_{cs,pp}$ or the $\pi^{weak}_{cs,node}$ relations appropriately.

Using the analysis on our running example in fig. 3(b) and (c), one can infer

$$\begin{aligned}
&\mathit{param\_binding}_{cs\_caller,cs\_callee,caller\_pp}(x,y) = \\
&\qquad \mathit{formal\_param}_{func\_called,param\_id}(x) \\
\wedge &\qquad \mathit{actual\_param}_{func\_caller,call\_node,param\_id}(y) \\
\wedge &\qquad \mathit{create\_call\_string}(cs\_caller, func\_caller, caller\_pp, func\_called, cs\_callee)
\end{aligned}$$

$$\begin{aligned}
&\pi^{IN}_{cs\_callee,callee\_start\_node}(x,z) = \\
&\qquad \mathit{formal\_param}_{func\_called,param\_id}(x) \\
&\qquad \wedge \quad \mathit{actual\_param}_{func\_caller,call\_node,param\_id}(y) \\
&\qquad \wedge \quad \mathit{create\_call\_string}(cs\_caller, func\_caller, caller\_pp, func\_called, cs\_callee) \\
&\qquad \wedge \begin{cases} \pi^{IN}_{cs\_caller,call\_node}(y,z) \text{ if } \mathit{critical}_{func\_caller}(caller\_node) \\ \pi^{OUT}_{cs\_caller,call\_node}(y,z) \text{ if } \neg \mathit{critical}_{func\_caller}(caller\_node) \end{cases}
\end{aligned}$$

$$\begin{aligned}
\pi^{RET}_{cs\_caller,call\_site\_pp}(x,z) = & \mathit{del\_last\_call}(call\_str\_callee, cs\_caller) \\
& \wedge \mathit{stm}_{func\_callee,return\_stm\_pp}(''\texttt{return(y)}'') \\
& \wedge \mathit{stm}_{func\_caller,call\_site\_pp}(''\texttt{x = func\_callee(}\cdots\texttt{)}'') \\
& \wedge \pi^{OUT}_{cs\_callee,exit\_node\_callee}(y,z)
\end{aligned}$$

$$\begin{aligned}
\pi^{MOD}_{cs\_caller,call\_site\_pp}(x,z) = & \mathit{formal\_param}_{func\_callee,*}(x) \\
& \wedge \mathit{local\_var}_{func\_callee}(x) \\
& \wedge \mathit{del\_last\_call}(cs\_callee, cs\_caller)) \\
& \wedge \pi^{OUT}_{cs\_callee,exit\_node\_callee}(x,z)
\end{aligned}$$

$$\begin{aligned}
&\text{if } \mathit{critical}_{func\_caller}(caller\_node)) \\
&\quad \pi^{strong}_{cs,pp}(x,y) = \pi^{RET}_{cs,pp}(x,z) \vee \pi^{MOD}_{cs,pp}(x,z) \vee \pi^{strong}_{cs,pp}(x,y) \\
&\text{else} \\
&\quad \pi^{weak}_{cs,node}(x,y) \vee = \pi^{RET}_{cs,pp \in node}(x,z) \vee \pi^{MOD}_{cs,pp \in node}(x,z) \vee \pi^{weak}_{cs,node}(x,y)
\end{aligned}$$

Figure 17: Interprocedural PFS Points-to Analysis Semantics

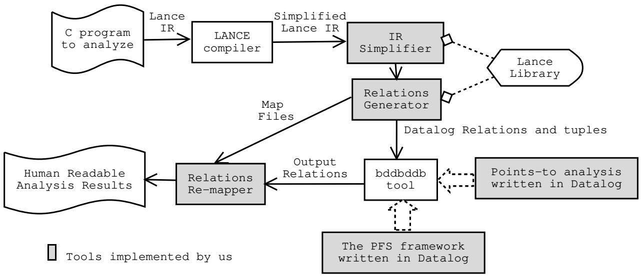


Figure 18: Our tool-chain for implementing Partial Flow-Sensitive Points-to analysis . We implemented our framework using the Lance Compiler Framework [13] and the bddbddb [12] tools.

that the parameter '`q`' in `populate()` which binds to '`elem`' in `process_node()`, will always point to a valid element. Hence, an optimizer can use this information to eliminate the "`if`" check in `populate()` as discussed in the introduction.

7 Implementation Details

Fig. 18 shows our implementation. The IR simplifier and the Relations-Generator are written using the Lance library. The IR Simplifier consumes the Lance generated IR and further simplifies the same. Some of its functions are enlisted below:

1. Heap Abstraction: It summarizes each heap location by its allocation-site, creating a unique heap variable for each such location.
2. Field-insensitive analysis : It maps all the field-locations (of pointer type) in a structure to the zeroth field.

The Relations-Generator works on the simplified IR to generate relations for the abstraction of the program being analyzed. It generates relations for control-flow, function-calls and call-strings, program variables and function call parameters and selected critical nodes. These relations are consumed by the Datalog programs run using the bddbddb tool which generates output relations as per the PFS algorithm.

The Relations Re-mapper, with the help of the map files (also generated by the Relations-Generator), interprets these results to provide human-readable outputs.

8 Future Work

We presented a partially flow-sensitive algorithm which is useful in analyzing programs where, at a small number of points in the program, it is beneficial to obtain a more precise information of the program properties (than what is obtained by flow-insensitive analysis). The algorithm is scalable if the number of such program points is small - in fact, the algorithm can be tuned to operate at various "settings" of scalability and precision by choosing appropriate number of critical nodes.

The paper opens up a lot many directions for future research. One such direction is on developing more heuristics on selecting critical nodes. We suggested an option of selecting the call-sites of "hot" methods as critical so that a better estimate on the parameters of these hot methods could be obtained. We feel that there are still a lot many other such opportunities to explore.

In this paper, we explained points-to analysis using the PFS framework. We plan to implement other analysis algorithms using the framework. We finally aim to build a profile-driven compilation scheme, where, for expensive analyses, each method is assigned a compilation-time budget as per the execution time spent in the same. For PFS, this time-budget would be directly translatable to the number of critical nodes the method is allowed to select.

Acknowledgments

This work is funded in part by doctoral fellowship provided by Philips Research, India.

References

[1] Lars Ole Andersen. Program Analysis and Specialization for the C Programming Language, May 1994. Ph.D thesis, DIKU, University of Copenhagen.

[2] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In Proceedings of PLDI '94, pages 242-256, 1994.

[3] Bjarne Steensgaard. Points-to analysis in almost linear time. In Proceedings of POPL '96, pages 32-41, Jan 1996.

[4] A. V. Aho, R. Sethi, and J. D. Ullman. Compilers, Principles, Techniques, and Tools. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[5] T. H. Cormen and C. E. Leiserson and R. L. Rivest: Introduction to Algorithms. McGraw-Hill, 2000.

[6] Cooper, Keith D. and Torczon, Linda. Engineering a Compiler. Morgan Kaufmann. 2005.

[7] Muchnick, Steven S. Advanced Compiler Design and Implementation. Morgan Kaufmann. 1997.

[8] Hecht, Matthew S. Flow Analysis of Computer Programs. Elsevier North-Holland Inc. 1977.

[9] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation, pages 103-114, June 2003.

[10] T. W. Reps. Demand Interprocedural Program Analysis Using Logic Databases, pages 163-196. Kluwer, 1994.

[11] Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analyses using binary decision diagrams. PLDI '2004.

[12] J. Whaley, C. Unkel, and M. S. Lam. A BDD-based deductive database for program analysis. *http://suif.stanford.edu/bddbddb*, 2004.

[13] LANCE Retargetable C Compiler. *http://www.lancecompiler.com/*

[14] Whaley, Avots, Carbin, Lam. Using Datalog with Binary Decision Diagrams for Program Analysis. Programming Languages and Systems: Third Asian Symposium (APLAS 2005), Tsukuba, Japan.

[15] Lam, Whaley, Livshits, Martin, Avots, Carbin, Unkel. Context-Sensitive Program Analysis as Database Queries. Principles of Database Systems (PODS 2005). Invited paper.

[16] M. Sharir and A. Pnueli, "Two Approaches to Interprocedural Data Flow Analysis," Program Flow Analysis: Theory and Applications, S. Muchnick and N. Jones, eds., Prentice Hall, pp. 189-234, 1981.

[17] Kim S. Henriksen, John P. Gallagher: Analysis and specialisation of a PIC processor. SMC (2) 2004: 1131-1135

[18] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side-effects. In Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages, pages 232–245. ACM Press, January 1993.

[19] J. D. Ullman. Principles of Database and Knowledge-Base Systems. Computer Science Press, Rockville, Md., volume II edition, 1989.

[20] Michael Hind, Anthony Pioli. Which Pointer Analysis Should I Use? International Symposium on Software Testing and Analysis, 2000.

[21] T. Reps. Demand interprocedural program analysis using logic databases. Applications of Logic Databases, pages 163-196, 1994.