Partial Flow Sensitivity

Subhajit Roy* and Y.N. Srikant

Department of Computer Science and Automation
Indian Institute of Science
{subhajit,srikant}@csa.iisc.ernet.in

Abstract. Compiler optimizations need precise and scalable analyses to discover program properties. We propose a partially flow-sensitive framework that 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.

Keywords: compilers, dataflow analysis, compiler optimizations, points-to analysis.

1 Introduction

Compiler optimizations largely depend on the program properties that the compiler could discover. Precision and scalability are two conflicting goals that such analyses have to meet. Control flow abstraction is an useful technique to attain high scalability, though at the cost of precision. A flow-sensitive algorithm takes the program control-flow into account to come up with a highly precise solution at each program point. On the other hand, a flow-insensitive algorithm neglects all control-flow leading to a high degree of scalability, while coming up with a summary solution for the whole flow-graph.

It is the need for scalability that forces many of the analyses to be implemented as flow-insensitive algorithms. However, it is possible that many opportunities for optimization could 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.

Fig. 1 presents an 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.

* Supported in part by doctoral fellowship provided by Philips Research, India.

S. Aluru et al. (Eds.): HiPC 2007, LNCS 4873, pp. 245–256, 2007.

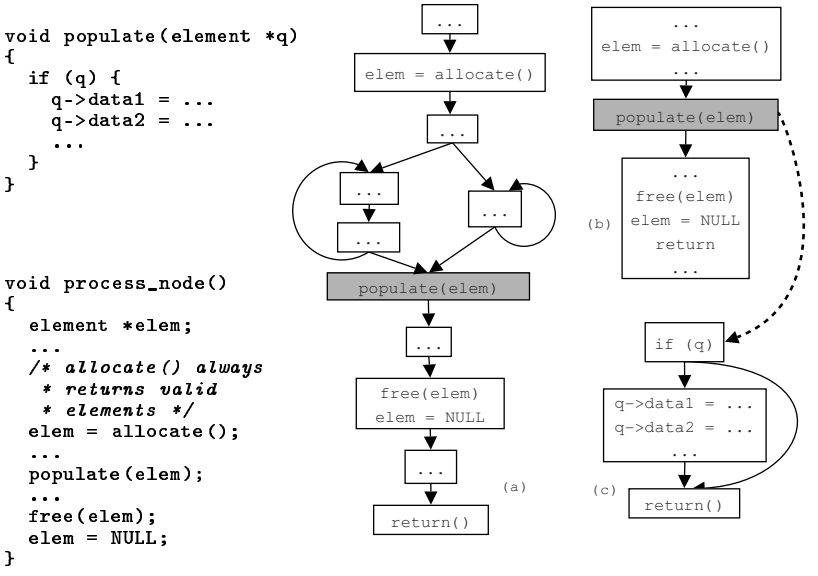


Fig. 1. A motivating example: Fig. (a) shows the original CFG for `process_node()`. Fig. (b) shows the reduced CFG for `process_node()` obtained via the PFS algorithm. Fig. (c) shows the CFG for the function `populate()`. The gray node is the critical node selected. The dotted line shows the parameter binding of 'elem' to 'q'.

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[2]. A flow-insensitive analysis would simply miss the fact. A flow-sensitive algorithm would surely indicate that fact; however, it would be hugely expensive. This leads us to an important observation : even if `process_node()` was analyzed well enough to provide a better estimate of program properties at only the program points where `populate()` is called, our purpose would be served.

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 a few points in a program, without losing much on scalability, we might be able to discover opportunities to optimize the program that a flow-insensitive algorithm would miss. 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 to have a better estimate of the program properties. We term such basic-blocks as *critical nodes* in the program's flow-

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

[2] We assume that the analyzer is made aware of the fact that `allocate()` will always return valid elements using suitable annotations.

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 a 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. 1(a) is shown in fig. 1(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.

2 Previous Work

[1] proposes a flow-sensitive interprocedural alias analysis that they claimed was, at that time, the most precise and efficient interprocedural method known. Andersen[2] described a flow-insensitive subset based algorithm based on constraint solving that computes a single solution for the whole program. [3] propose an algorithm to improve the precision of flow-insensitive interprocedural alias analysis using precomputed kill information.

[4,5,6] propose techniques towards solving a demand dataflow analysis algorithm, that answers a query about a single given dataflow fact holding at a single given program point. However, this technique differs from our work as our partial flow sensitive framework computes a solution for all program points, though of varying precision.

[7] proposed to use the SSA form to improve the precision of flow-insensitive pointer analysis. The algorithm uses repeated iterations to improve the precision of the analysis, and the final result could even be as good as that computed using a flow-sensitive analysis. However, the worst case time requirement for translating a code in SSA form is cubic. Also, the SSA translation could result in a program that is quadratic in the size of the original program. Moreover, as the algorithm has to be primed with points-to relations, it requires a points-to analysis in its initial phase. Recently, [8] reported a new approach to solving subset-based points-to analysis for Java using Binary Decision Diagrams (BDDs).

[9] proposes a client-driven pointer analysis, where the analysis adapts to the need of the client analyses. However, our work differs from this work in the way flow-sensitivity is provided; while the mentioned work looks at using the SSA from to provide flow-sensitivity to some variables at all programs points, our work looks at providing better precision to the dataflow solutions of all variables at some program points. Also, we do not need to use the SSA form, construction of which itself needs a prior pointer analysis phase [7].

3 The Reduced Control-Flow Graph (r-CFG)

The Partially Flow-Sensitive Algorithm (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. The selected functions could be call-sites of "hot" methods (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) or basic blocks having a high execution count identified through a profile run.

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.

The Partial Flow Sensitive algorithm is safe as all the control-flow edges in the CFG are also preserved in the r-CFG; any path in the CFG being traceable in the latter[3]. Thus, the PFS algorithm does not "miss" any flow-path along which the program properties could propagate.

3.1 Yardsticks for a Reduced Control-Flow Graph (r-CFG)

Both the scalability and the precision of the Partial Flow Sensitive algorithm depends on the r-CFG. We define the notions of Precision and Size optimality to understand how good is a given r-CFG for a given flow-graph.

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. [4]

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.

3.2 Algorithm for Computing the r-CFG

Our algorithm is shown Fig. 3. Equation 1 and 2 compute the set $\rho(n)$ for all the nodes $n \in G_{orig}$; $\rho(n)$ represents the set of all critical nodes that are reachable from the node n. This computation can be done efficiently by setting it up as a simple bit-vector dataflow problem where each bit in the bit-vector stands for a particular critical-node and an extra bit for distinguishing 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

[3] 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.

[4] This also implies that if there exists a path, n to c, in the reduced graph, such a path surely exists in the original graph.

specifies when two nodes can actually be merged — when none of them is critical and the predicate *not_mergeable*() does not forbid their merge.

The reduced CFG is created by representing all the nodes merged together by a single aggregate node. 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 *Aggregate Node ID (anid).* The *anid* 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 *anid*s.

3.3 Analyzing the Algorithm

Let us define some notations: N and C denote the set of all nodes and the set of critical nodes respectively in the original CFG (G_{orig}). The r-CFG is denoted by $G_{reduced}$. The relation $path(n_i, n_j) \in G$ represents that there exists a path from n_i to n_j in the graph G. The relation ρ is computed over G_{orig}.

Claim: *The above algorithm produces a precision optimal reduced CFG.*

Proof. Assume $\exists n_i \in N$ s.t. for some $c \in C$, $path(n_i, c) \in G_{reduced}$ and $path(n_i, c) \notin G_{orig}$. Such a path is only possible due to a merge of n_i with some $n_j \in N$ where $path(n_j, c) \in G_{orig}$ (see Fig. 2). This implies that $\rho(n_i) = \rho(n_j)$ as otherwise the algorithm would not have merged the nodes. This causes a contradiction as the critical node $c \in \rho(n_j)$ but $c \notin \rho(n_i)$.

Claim: *The above algorithm produces a size optimal reduced CFG.*

Proof. Assume $\exists n_i, n_j \in N, n_i \neq n_j$, not merged by the above algorithm, s.t. merging them still maintains precision optimality for $G_{reduced}$. Obviously, $\exists c \in C$ s.t. $c \in \rho(n_i)$ and $c \notin \rho(n_j)$ — otherwise the nodes n_i and n_j would had been merged by the algorithm. This causes a contradiction as then precision optimality w.r.t the node $c \in C$ is compromised.

3.4 Size of the Reduced CFG

Lemma: *The above algorithm partitions the non-critical nodes 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[5] is 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.*

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

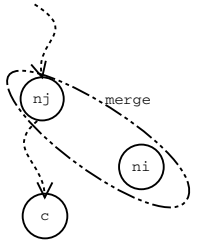


Fig. 2. Proving that the r-CFG is precision optimal

$$\frac{crit, node \in G_{orig} \quad critical(crit) \quad node = crit}{crit \in \rho(node)} \quad (1)$$

$$\frac{crit, node, succ \in G_{orig} \quad node \in Pred(succ) \quad crit \in \rho(succ)}{crit \in \rho(node)} \quad (2)$$

$$\frac{n_1, n_2, c \in G_{orig} \quad critical(c) \quad c \in \rho(n_1) \quad c \notin \rho(n_2)}{not\_mergeable(n_1, n_2)} \quad (3)$$

$$\frac{n_1, n_2 \in G_{orig} \quad not\_mergeable(n_2, n_1)}{not\_mergeable(n_1, n_2)} \quad (4)$$

$$\frac{\begin{array}{l} n, n_1, n_2 \in G_{orig} \quad n_1 \neq n_2 \\ \neg critical(n_1) \qquad \neg critical(n_2) \quad \neg not\_mergeable(n_1, n_2) \end{array}}{merge(n_1, n_2)} \quad (5)$$

Fig. 3. The optimal algorithm for computing the r-CFG. The solution can be found by performing a fixpoint computation over the above rules.

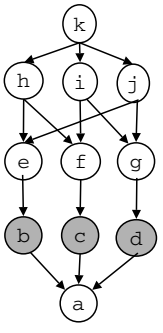
Proof. According to the above lemma, as each of the non-critical node must belong to one of the equivalence classes and there are at most 2^n such classes — the reduced CFG can at most have 2^n aggregate nodes formed by merging of the non-critical nodes. None of the n critical nodes is merged with any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

Also, the above bound is tight. Fig. 4 shows a case where the above bound is actually reached for `n=3`. The critical nodes are marked gray. The table shows the value of $\rho(x)$ for each non-critical node (x). Note that none of the nodes can be merged. A graph of similar structure can be constructed for any value of n.

Surely, the total number of nodes possible in the r-CFG is also bounded by the total number of nodes in the graph. If all the nodes are selected critical, then the r-CFG is same as the original CFG and the PFS algorithm reduces to a flow-sensitive algorithm.

4 The Analysis Phase in the PFS-Algorithm

We explain the analysis phase of the PFS algorithm using a classic compiler analysis — points-to analysis. We choose points-to analysis for the purpose as most compilers implement a flow-insensitive version of this analysis to attain scalability. Our experiments show that the PFS algorithm not just manages to get much better solutions at the critical nodes (arbitrarily chosen), but also improves the solution at most of the other nodes as a side-effect.



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

Fig. 4. An example illustrating the tight bound of the number of nodes in the reduced CFG

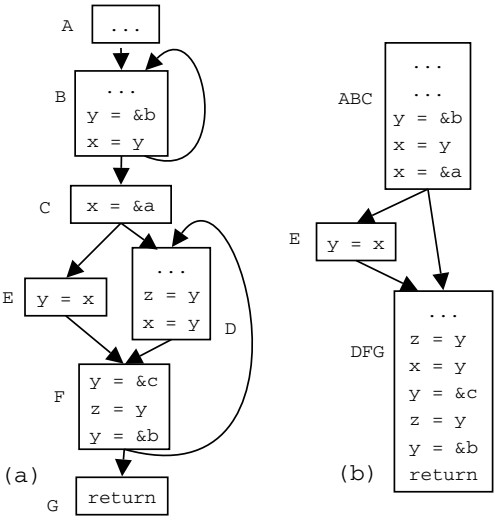


Fig. 5. An Example: The (a) original CFG and the (b) reduced CFG, taking the node 'E' as the critical node

4.1 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 [2,10]. Some of the rules in the update semantics for the various statements is shown in fig. 6. The complete set of rules for all the important statement templates is given in [11]. $\pi_{cs,pp}(x, y)$ denotes the points-to relation that `x` points-to `y` at the program point `pp` under the call-string `cs`.

$$\frac{stm_{pp}(\text{"}x = y\text{"}) \quad \pi_{cs,pp}(y, z)}{\pi_{cs,pp}(x, z)} \quad (1)$$

$$\frac{stm_{pp}(\text{"}x = \&y\text{"})}{\pi_{cs,pp}(x, y)} \quad (4)$$

$$\frac{stm_{pp}(\text{"}x = *y\text{"}) \quad \pi_{cs,pp}(y, y1) \quad \pi_{cs,pp}(y1, z)}{\pi_{cs,pp}(x, z)} \quad (2)$$

$$\frac{stm_{pp}(\text{"}*x = y\text{"}) \quad \pi_{cs,pp}(x, z) \quad \pi_{cs,pp}(y, y1) \quad \neg garbage\_value(z) \quad \neg null\_value(z)}{\pi_{cs,pp}(z, y1)} \quad (5)$$

$$\frac{stm_{pp}(\text{"}x = *y\text{"}) \quad \pi_{cs,pp}(y, g) \quad garbage\_value(g)}{\pi_{cs,pp}(x, g)} \quad (3)$$

Fig. 6. 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 S is statement encountered at program point *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[6].

4.2 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 transfer function 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. Fig. 7 shows the semantics of these transfer functions for points-to analysis. Finally, we perform a flow-sensitive analysis over the whole r-CFG to generate the required program properties.

$$\pi^{strong}_{cs,pp}(x,y) = \pi_{cs,pp}(x,y) \;\vee\; \begin{cases} \pi^{strong}_{cs,pp-1}(x,y) & \texttt{if } \neg killed_{cs,pp}(x) \\ \phi & \texttt{if } killed_{cs,pp}(x) \end{cases}$$

$$\vee \begin{cases} \pi^{IN}_{cs,node}(x,y) & \texttt{if } pp \in node \texttt{ and pp is first stm in node} \\ \phi & \texttt{otherwise} \end{cases}$$

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

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

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

Fig. 7. The semantics for the strong and weak transfer functions for points-to analysis. $\pi(x,y)$ denotes that the variable x points-to y. *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`. The relation *id* returns a unique identifier for all nodes. A relation $\pi(x,y)$ is killed if there exists an unambiguous definition to x. The solution can be computed by a fixpoint computation over the rules.

4.3 Interprocedural Analysis

Interprocedural analysis is performed using the k-limit call-string approach [12]. 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 $x \in$

[6] We summarize heap locations by summarized heap variables per allocation-site.

$$\pi^{strong}_{cs,pp}(x,y) = critical_{func\_caller}(caller\_node) \wedge (\pi^{RET}_{cs,pp}(x,z) \vee \pi^{MOD}_{cs,pp}(x,z) \vee \pi^{strong}_{cs,pp}(x,y))$$
$$\pi^{weak}_{cs,node}(x,y) = \neg critical_{func\_caller}(caller\_node) \wedge (\pi^{RET}_{cs,pp \in node}(x,z) \vee \pi^{MOD}_{cs,pp \in node}(x,z) \vee \pi^{weak}_{cs,node}(x,y))$$

Fig. 8. Interprocedural PFS Points-to Analysis Semantics. The solution can be computed by a fixpoint computation over the rules.

$nodes(G_{orig})$ 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 $OUT_{\{x'|n \in x', x' \in nodes(G_{red})\}}$ value that is passed back to the caller.

Fig. 8 shows the interprocedural semantics of the weak and strong transfer functions for our PFS 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. The $\pi^{RET}_{cs,pp}$ relation defines the semantics of a return statement — if the callee returns with a statement "`return(y)`" to the caller who had initiated the call with "`x=func_callee(...)`", the equation updates the points-to set of the variable `x` with that of the return parameter `y`. The π^{MOD} relation updates all points-to relations generated due to indirect references in the called procedure. The detailed description of the interprocedural semantics is given in [11].

We give an example to illustrate the effect of the PFS analysis. Fig. 5(a) shows the original flow-graph; fig. 5(b) shows the reduced flow-graph (taking the node '`E`' as the critical node). The results for flow-sensitive, flow-insensitive and partially flow-sensitive analyses are shown in table 1. Note that for the PFS case, flow-insensitive analysis is done on the aggregate nodes "`ABC`" and on "`DFG`" while we perform flow-sensitive analysis on the critical node '`E`' to compute the local properties. A flow-sensitive analysis is then performed on the r-CFG.

The reduced CFG is much simplified as the number of nodes drop to three and the two loops in the original CFG get dissolved. In fact, the r-CFG in this case gets reduced to a DAG; note that the number of iterations required for a flow-sensitive analysis to reach a fixpoint depends on the loop-depth. Hence, performing a flow-sensitive analysis over the r-CFG is a lot cheaper than doing the same on the original CFG.

Let us look at the results of the analysis in table 1. For the critical node '`E`', the PFS solution is much better than that obtained by the flow-sensitive analysis but is not as good as that obtained by the flow-insensitive analysis. The reason for the loss in precision is the inability of the PFS algorithm to use "kill" information within the aggregate nodes reaching the critical node. In the given example, the dataflow fact `x may-point-to b`, generated at the node '`B`', is killed at the node '`C`'. However, the PFS algorithm is unable to use this information as a flow-insensitive analysis is carried on the aggregate node "`ABC`". However, the result is better than the flow-insensitive case, as the dataflow facts from the nodes '`D`', '`F`', and '`G`' were not allowed to pollute the information at the critical

Table 1. The dataflow facts discovered for the CFG and r-CFG in fig. 5. The program points corresponding to a node 'X' is the point just after 'X'. The notation $V_1 \rightarrow V_2$ implies that any variable $v_1 \in V_1$ may point to any of the variables $v_2 \in V_2$.

BB	Flow-Sensitive	Flow-Insensitive	Partial Flow-Sensitive
A	ϕ	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x\} \rightarrow \{a, b\}$, $\{y\} \rightarrow \{b\}$
B	$\{x, y\} \rightarrow \{b\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x\} \rightarrow \{a, b\}$, $\{y\} \rightarrow \{b\}$
C	$x \rightarrow \{a\}$, $y \rightarrow \{b\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x\} \rightarrow \{a, b\}$, $\{y\} \rightarrow \{b\}$
D	$\{x, y, z\} \rightarrow \{b\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$
E	$\{x, y\} \rightarrow \{a\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x, y\} \rightarrow \{a, b\}$
F	$\{x\} \rightarrow \{a, b\}$, $\{y\} \rightarrow \{b\}$, $\{z\} \rightarrow \{c\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$
G	$\{x\} \rightarrow \{a, b\}$, $\{y\} \rightarrow \{b\}$, $\{z\} \rightarrow \{c\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$	$\{x, y, z\} \rightarrow \{a, b, c\}$

node 'E'. As a bonus, the solutions at the nodes 'A', 'B' and 'C' are also improved over the flow-insensitive one.

5 Experimental Results and Conclusion

We have implemented a framework for partial flow-sensitive points-to analysis using the Lance compiler framework [13] and the bddbddb [14] tool. The details of the implementation can be found in [11]. The results are shown in Fig. 9, 11 and 10. We selected the critical nodes arbitrarily for computing the solutions using the PFS algorithm. We used Andersen's algorithm [2] for the flow-insensitive analysis within the aggregate nodes. The results were obtained by performing an intraprocedural analysis on the respective functions by setting all the pointer arguments and the global pointer variables used in the procedure to be pointing to *undefined* (implying that they could potentially point to any location). The analysis was performed on the intermediate code generated by the Lance Compiler Framework [13].

Fig. 9 shows the effect of partial flow-sensitivity on the function "`SwapNode()`" from the "`ks`" benchmark (from [15]) with nodes `5` and `8` arbitrarily selected as the critical nodes. The results for a fully flow-sensitive and a flow-insensitive analysis are also shown. The partial flow-sensitive analysis yields a very precise solution for the critical nodes. In fact, they are as good as the flow-sensitive solution in this case. Also, as a side-effect, the solution at many of the other nodes are much better than that obtained using a flow-insensitive algorithm. However, though close, the FS and PFS solutions may not coincide in all cases as the PFS algorithm is unable to use kill information within the aggregate nodes.

Fig. 10 compares the number of nodes in the original and the reduced graphs. For instance, note that for the function `PrintChannel()`, the PFS solution is almost the same as the flow-sensitive one, even though the analysis was done on a reduced CFG with 5 nodes while the original CFG had 88 nodes. However, how close is the PFS solution to the flow-sensitive solution is hugely dictated by the choice of the critical nodes.

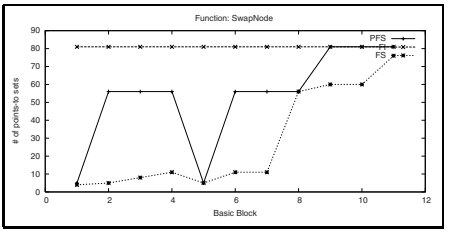


Fig. 9. The effect of Partial Flow Sensitivity (on function "`SwapNode`" from the "`ks`" benchmark suite) : The plot shows the number of `may-point-to` relations that hold at each basic-block in the program for the flow-sensitive (FS), flow-insensitive (FI) and the partially flow-sensitive (PFS) algorithms.

Function	Benchmark	N_{crit}	$\#N_o$	$\#N_r$
SwapNode	ks	{5,8}	11	5
DensityChannel	yacr2	{7,28}	32	5
PrintChannel	yacr2	{6,22,89}	88	5
HasVCV	yacr2	{6,11}	13	5
gen_bitlen	gzip	{4,15}	32	5
build_tree	gzip	{4,8}	16	5
init_block	gzip	{4,7}	10	5

Fig. 10. Details on the benchmarks used. The column 'N_{crit}' denotes the nodes selected as critical in the original flow-graph. $\#\ N_o$ and $\#\ N_r$ denotes the number of nodes in the original and the reduced flow-graphs.

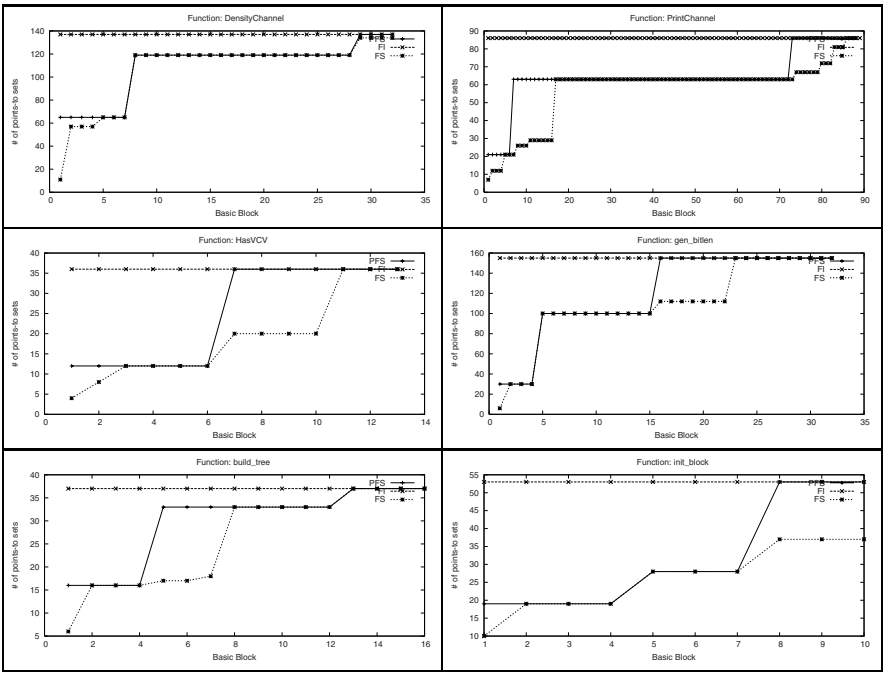


Fig. 11. Precision of the flow-sensitive (FS), flow-insensitive (FI) and the partially flow-sensitive (PFS) algorithms for various benchmarks. The plot shows the number of `may-point-to` relations that hold at each basic-block in the program.

References

1. Choi, J.D., Burke, M., Carini, P.: Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects. In: Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, pp. 232–245 (1993)
2. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)
3. Burke, M., Carini, P., Choi, J.D., Hind, M.: Flow-Insensitive Interprocedural Alias Analysis in the Pressence of Pointers. In: Pingali, K.K., Gelernter, D., Padua, D.A., Banerjee, U., Nicolau, A. (eds.) Languages and Compilers for Parallel Computing. LNCS, vol. 892, Springer, Heidelberg (1995)
4. Babich, W.A., Jazayeri, M.: The Method of Attributes for Data Flow Analysis: Part II. Demand Analysis. Acta Inf. 10, 265–272 (1978)
5. Horwitz, S., Reps, T., Sagiv, M.: Demand interprocedural dataflow analysis. In: SIGSOFT 1995. Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering, pp. 104–115. ACM Press, New York (1995)
6. Duesterwald, E., Gupta, R., Soffa, M.L.: Demand-driven computation of interprocedural data flow. In: POPL 1995. Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 37–48. ACM Press, New York (1995)
7. Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: PLDI 1998. Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation, pp. 97–105. ACM Press, New York (1998)
8. Berndl, M., Lhoták, O., Qian, F., Hendren, L., Umanee, N.: Points-to analysis using BDDs. SIGPLAN Not. 38(5), 103–114 (2003)
9. Guyer, S.Z., Lin, C.: Client-Driven Pointer Analysis. In: International Static Analysis Symposium, pp. 214–236 (2003)
10. Muchnick, S.S.: Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers Inc., San Francisco (1997)
11. Roy, S., Srikant, Y.N.: Partial Flow Sensitivity. Tech. Report CSA-TR-2006-12, Dept. of Computer Science & Automation, Indian Institute of Science (2006)
12. Sharir, M., Pnueli, A.: Two Approaches to Interprocedural Data Flow Analysis. In: Muchnick, S., Jones, N. (eds.) Program Flow Analysis: Theory and Applications, pp. 189–234. Prentice-Hall, Englewood Cliffs (1981)
13. LANCE Retargetable C Compiler: `http://www.lancecompiler.com/`
14. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of the ACM Symposium on Principles of Database Systems, pp. 1–12 (2005)
15. Austin, T., et al.: The Pointer-intensive Benchmark Suite, `http://www.cs.wisc.edu/~austin/ptr-dist.html`