Improving Flow-Insensitive Solutions for Non-Separable Dataflow Problems

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

ABSTRACT
Flow-insensitive solutions to dataflow problems have been known to be highly scalable; however also hugely imprecise. For non-separable dataflow problems this solution is further degraded due to spurious facts generated as a result of dependence among the dataflow facts. We propose an improvement to the standard flow-insensitive analysis by creating a generalized version of the dominator relation that reduces the number of spurious facts generated. In addition, the solution obtained contains extra information to facilitate the extraction of a better solution at any program point, very close to the flow-sensitive solution. To improve the solution further, we propose the use of an intra-block variable renaming scheme. We illustrate these concepts using two classic non-separable dataflow problems — points-to analysis and constant propagation.

Keywords
Compilers, Dataflow Analysis, Compiler Optimizations

1. INTRODUCTION
Separability is an important property of dataflow frameworks. As discussed in [8], a dataflow analysis is said to be separable if it has a separable function space i.e. the component functions work only on the component lattices and so the aggregate information can simply be viewed as a function product on the individual dataflow items. Separability allows one to perform analysis on the components independently and finally combine the result without any loss of precision. Analyses like liveness analysis and reaching definitions are separable; faint variable analysis, constant propagation and alias analysis are classic examples of non-separable analysis.

Hence, non-separable dataflow facts have an element of dependence on other dataflow facts. For instance, for points-
to analysis, a dataflow fact \( x \rightarrow y \) (i.e. \( x \) may point to \( y \)) would be generated for a statement \( S_1: x = z \) only if the fact \( z \rightarrow y \) holds at \( S_1 \). Moreover, for flow-insensitive analyses — as any dataflow fact generated at any point in a procedure is essentially included in the solution and is assumed to hold at all the points in the procedure — the element of dependence tends to generate more spurious facts, worsening the precision of the solution further.

Let us explain the motivation for the current work with an example (Figure 1) for points-to analysis. The points-to set for the variable ‘\( z \)’ is generated by the statements 4 and 7. So, the inclusion of the facts \( z \rightarrow a, z \rightarrow b, z \rightarrow c \) in the solution is dependent on the facts \( y \rightarrow a, y \rightarrow b, y \rightarrow c \) respectively. As all the above facts for ‘\( y \)’ do get generated (at 3, 2 and 6 respectively), a flow-insensitive solution will include all the above mentioned facts for ‘\( z \)’.

However, note that the fact \( z \rightarrow a \) is spurious as the fact \( y \rightarrow a \) (generated at 3) surely gets killed at 6 before reaching either 7 or 4. This power of using kill information is available to a flow-sensitive algorithm which flow-insensitive algorithms lack.

We propose the following ideas in this paper:

• We propose a generalized version of the dominator relationship and use it to obtain an improved summarized solution over the conventional flow-insensitive solution.

• As the above solution is “tagged” — i.e. each dataflow fact is annotated with its generation site — it also allows the generation of a weak flow-sensitive solution (WeakFS). The WeakFS solution was found to be very close to the actual flow-sensitive solution on most of the benchmarks.

• To improve the solutions further, we propose an intra-block variable renaming scheme (essentially an intra-block SSA) which is cheaper to construct than the SSA form and also restricts the enormous increase in the number of variables encountered in the SSA form. We argue that this form is pretty useful when comparing solutions using tools like bbdiff[9].

2. PREVIOUS WORK
Our current work carries ideas similar to [5]; while Burke
et al. used precomputed kill information based on dominator [3, 10] relationship to limit the alias relations that are propagated interprocedurally, we use a generalized version of the dominator relationship (which we call the all-node dominator relationship) to improve the precision of non-separable intra-procedural analysis.

[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 translation to 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 a points-to relation, it requires a points-to analysis in its initial phase. The above mentioned algorithm is much more expensive than our scheme; however, it scores over our scheme in the precision of the solution obtained. We also propose a variable renaming scheme, similar to the SSA form but the variable renaming being limited only within basic blocks. Our representation is much cheaper to construct than the full SSA form. Also, it does not cause any increase in the code size and a slight increase in the number of variables.

3. IMPROVING THE PRECISION OF FLOW-INSENSITIVE ANALYSIS

3.1 The All-Node Dominator Relation

A node 'y' is called the dominator [3, 10] of node 'x' (denoted by $y \in \text{Dom}(x)$), if any path from the start node to the node 'x' must pass through node 'y'. We define a generalized version of the domination relation: node 'y' is called the dominator of node 'x' w.r.t node 'z' (denoted by $y \in \text{Dom}_z(x)$) if any path starting from the node 'z' and reaching node 'x' must pass through 'y'. We call the above relation the All-Node Dominator Relation. An algorithm to compute the relation is shown in Figure 2. ReachSet(x) computes the set of all nodes that can reach the node x.

3.2 The Analysis

3.2.1 Computing an improved flow-insensitive solution

The improved flow-insensitive analysis is shown in Figure 3. We define the set DataFlowFact as the universe of all dataflow facts that are possible in the solution (which is essentially the facts forming the top element in the semilattice). GenSet(d_i) for a dataflow fact $d_i$ contains the set of all basic blocks where $d_i$ could be generated. This set is built incrementally as the analysis progresses and new dataflow facts are generated.

For each dataflow fact $d_i$, we find the set KillSet(d_i) — all basic blocks where $d_i$ is unambiguously killed. The relation $\text{SureKilledFact}(d_i, gen, curr)$ holds if the fact $d_i$, generated at a basic block $gen$ is certainly killed at another basic block $curr$ — if $(\text{Dom}_gen(curr) \cap \text{KillSet}(d_i)) \neq \emptyset$ (provided that gen, curr and the node at which $d_i$ is killed are all in different basic blocks, i.e. we do not kill facts within a basic block).

ValidFact(d_i, gen, curr) represents the validity of the fact $d_i$ generated at the basic block gen, holding at a basic block curr.

- All the dataflow facts flowing from the environment (parameters and global variables) hold true at the start node of the CFG.
- All the dataflow facts that are generated at a basic block gen surely hold within that basic block (we do...
A fact \( d_i \) generated at the basic block \( gen \) stands valid at another basic block \( curr \) if \( curr \) is reachable from \( gen \) and the fact \( d_i \) is not surely-killed while flowing from \( gen \) to \( curr \) (i.e. \( SureKilledFact(d_i, gen, curr) \) is false);

- A dataflow fact \( d_i \) — that is dependent on a set of facts \( D_j \) — can be generated as a result of a program statement \( stmt \) at \( curr \) only if all the facts \( d_j \in D_j \) is valid at \( curr \). This is the relation that improves the solution by killing facts where it is not valid.

The relation \( ValidFact \) may be computed lazily, i.e. on an on-need basis. Once computed for a certain \( \langle gen, curr \rangle \) pair, the result may be memoized for any future use. Finally, \( DataflowSolution \) aggregates all the dataflow facts that are valid at any point in the program.

The above analysis tends to be more precise than a flow-sensitive analysis — assume that a dataflow fact \( d_i \) could get generated by a statement at a basic block \( curr \), and \( d_i \) is dependent on another fact \( d_j \). In that case, \( d_j \) would get generated only if \( d_j \) might hold at \( curr \), i.e. \( ValidFact(d_j, gen, curr) \) holds for any \( gen \in GenSet(d_j) \) — in contrast to a flow-sensitive analysis that would invariably include \( d_j \) in the solution set.

However, the results would still be less precise than a flow-sensitive solution. Figure 6 illustrates such a scenario. Let \( n_1 \in GenSet(d_j) \) and \( n_2, n_3, n_4 \in KillSet(d_j) \). So, although the fact \( d_j \) does not reach \( n_2 \) as it gets killed along all paths from \( n_1 \) to \( n_4 \), our analysis fails to discover this fact as none of \( \{n_2, n_3, n_4\} \) is a member of \( Dom_{n_1}(n_2) \).

In our implementation, rather than maintaining the \( GenSet \) as described above, we simply annotate all the dataflow facts generated with their appropriate generation sites. We will refer to this annotated solution as the “tagged” flow-insensitive solution. Note that a tuple would get duplicated with two different annotations if it may get generated at two different basic blocks.

### 3.2.2 WeakFS: Recovering a more precise solution at a given program point

The dataflow facts having been tagged with the generation site — the program-point (basic-block) where it was generated — allows us to recover a better solution at any program point at a later time if the tagged points-to relation is cached as summary information for the procedure. This sort of information could be useful for a JIT compiler when it finds that some of the basic-blocks are hot enough to benefit from a higher level of optimization.

The WeakFS algorithm is shown in Figure 4. The algorithm simply takes a union of all dataflow facts that are generated at sites which reach the current node (the program point where we are interested in a more precise solution). To avoid the cost of caching the all-node dominator relation, we ignore the potential improvement on using killing information made possible by the same. The reachability relation can be computed in linear time for any basic-block and hence need not be cached.

Note that the tagged dataflow facts form a summarized solution — much smaller than a flow-sensitive solution; however, together with the reachability relation, it allows the computation of a close approximation to the actual flow-sensitive solution at any program point.

### 3.3 Example: Points-to Analysis

We show the applicability of our framework for a classic non-separable analysis — points-to analysis. Figure 7 shows the dependent facts and the facts that get generated (if the dependent facts hold) for points-to analysis for the basic statement types.

Consider Figure 1. To compute the points-to sets using our improved flow-insensitive analysis, we first need to compute the relation \( KillSet \). Here, any fact of the form \( v_1 \rightarrow * \) (where * indicates any other variable) gets killed whenever \( v_1 \) is unambiguously defined. Hence, we maintain just the following kill sets:

- \( KillSet(x \rightarrow *) = \{A, C\} \)
- \( KillSet(y \rightarrow *) = \{A, B, D, E\} \)
- \( KillSet(z \rightarrow *) = \{C, E\} \)

Figure 7: Dependent Facts and Generated Facts for Points-to Analysis for the basic statement types.

We use Andersen’s algorithm [4] as the basic algorithm to compute the points-to sets. Let us illustrate how the generation of the dataflow fact \( x \rightarrow c \) is prevented. We denote the dataflow fact \( v_1 \rightarrow v_2 \) generated at a basic block \( B \) as \( f_{v_1 \rightarrow v_2}^B \). Note that the presence of the fact \( f_{v_1 \rightarrow v_2}^B \) simply implies that \( B \in GenSet(f_{v_1 \rightarrow v_2}) \).

The facts \( f_{A, v}^A, f_{A, y}^A, f_{B, x}^B, f_{E, b}^E, f_{E, c}^E \) would be the first facts to be generated (by the statements 1, 2, 3 and 4 respectively) as they are not dependent on any other facts. In conventional flow-insensitive analysis, the statement \( 5: x = y \) prevents \( f_{E, c}^E \) from being added to \( ValidFact \).

\[
\begin{align*}
  x &\rightarrow a \\
  y &\rightarrow b \\
  z &\rightarrow c \\
  x &\rightarrow c \quad [x \rightarrow c] \\
  y &\rightarrow b \quad y \rightarrow c \quad [y \rightarrow c] \\
  z &\rightarrow b \quad z \rightarrow c
\end{align*}
\]

Figure 8: The points-to facts discovered by a conventional flow-insensitive analysis for the program in Figure 1. The bracketed items are the facts that our analysis is able to eliminate.

We are able to prune out the two facts \( x \rightarrow c \) and \( z \rightarrow a \) now, let us illustrate the extraction of a more precise solution at the basic block \( B \). The set of basic-blocks reaching \( B \) is \( \{A, B\} \). The dataflow facts generated at \( A \) are \( \{f_{x, a}^A, f_{x, y}^A\} \) and those generated at \( B \) is \( \{f_{y, b}^B\} \). So, the solution \( \{f_{x, a}^A, f_{x, y}^A, f_{y, b}^B\} \) is very close to the flow-sensitive solution at \( B \), that is \( \{f_{x, a}^A, f_{y, b}^B\} \).

### 3.4 Improving Solutions within a basic-block using renamed variables

The above scheme of using the all-node dominators improves the solution by killing information across basic-blocks; however, the spurious facts generated within basic-blocks due to flow-insensitive analysis is also a major cause for
for all $x \in \text{ProgramVariables}$ do
  Add $x_1$ to FirstVersion($x$)
  Add $x_1$ to IsVersion($x$)
for all $bb \in \text{BasicBlocks}$ do
  $v = 1$
  $stm = \text{FirstStatement}(bb)$
  while $stm \neq \text{null}$ do
    Replace uses of $x$ in $stm$ by $x_0$
    if $x$ is defined in $stm$ then
      $v = v + 1$
    end if
    Replace $x$ in the definition by $x_0$
    Add $x_0$ to IsVersion($x$)
  end while
  $stm = \text{NextStatement}(bb)$
  while $stm$ do
    $x_n$ to LastVersion($bb, x$)
  end for
end for

for all $x \in \text{ProgramVariables}$ do
  Add $x_1$ to FirstVersion($x$)
  Add $x_1$ to IsVersion($x$)
for all $bb \in \text{BasicBlocks}$ do
  $v = 1$
  $stm = \text{FirstStatement}(bb)$
  while $stm \neq \text{null}$ do
    Replace uses of $x$ in $stm$ by $x_0$
    if $x$ is defined in $stm$ then
      $v = v + 1$
    end if
    Replace $x$ in the definition by $x_0$
    Add $x_0$ to IsVersion($x$)
  end while
  $stm = \text{NextStatement}(bb)$
  while $stm$ do
    $x_n$ to LastVersion($bb, x$)
  end for
end for

The tuple $(<bb, pc>, x)$ denotes a program point, where $bb$ is a basic-block and $pc$ is the statement number of a particular statement within that basic block. $\text{KilledDefinition}(x)$ denotes the set of all basic-blocks where definitions of the versioned variable $x$ would be killed. The rule $\text{KilledDefinition}(x, \text{gen}, \text{curr})$ evaluates to true if there exists a definition of the base variable $x$ in the basic-block $\text{bb}$, which is surely killed while reaching the basic-block curr. Some of the other notations are explained below:

- $\text{Expr}(bb, pc, y, x)$ implies that at $(bb, pc)$, $x$ is assigned to an expression involving $y$.
- $\text{Def}(bb, pc, x) / \text{IsUsed}(bb, pc, x)$ imply that the variable $x$ is defined/used at the program point $(bb, pc)$.
- $\text{PtrUse}(bb, pc, x)$ refers to the fact that there exists an assignment to $x$ at $(bb, pc)$ to an expression involving the use of a pointer dereference.
- $\text{IsNotConst}(bb, x)$ implies that the variable $x$ is non-constant in the basic-block $bb$.
- $\text{ConstUse}(bb, pc, x)$ refers to the fact that the use of $x$ at the program point $(bb, pc)$ can be replaced by a constant.

The set of rules for $\text{IsNotConst}(bb, x)$ decides when a variable $x$ is non-constant within a basic-block: when $x$ occurs in a definition of an expression involving a use of a previously marked non-constant variable or a pointer dereference. Also, if there exists both a use of the first version of a base variable and a subsequent definition to the same base variable within a basic-block contained in a loop, and there also exists another reaching definition to the base variable outside this basic-block, the former definition is marked non-constant. This sort of situation could occur if there exists a statement of the form "if $i = i + 1" within a loop. Across basic-blocks, the first version of variable is non-constant if there are two non-killed definitions reaching it. The first version of the variable would also be non-constant if it has any definition reaching it from a basic-block whose last version for the same variable is non-constant. Any use of heap regions or undefined values (like unassigned pointers) is non-constant. Finally, we mark the use of a variable at a particular program point $(bb, pc)$ as constant if it survives being marked non-constant. Note that across basic-blocks, we read off the state of only the last version of the variables from a basic-block (generating the dataflow fact) into only the first version of the variables affected by the respective fact (in the reachable blocks).

4. EXPERIMENTAL RESULTS AND CONCLUSIONS

We implemented a few analyses to measure the benefits offered by our scheme. We used the Lance Compiler Framework [1] as the front-end and used its intermediate representation to generate the required relations from the subject program. As some of the relations tend to be large,
we compute the solution using bddbddb [9] that uses Binary Decision Diagrams to store and operate on large relations efficiently.

4.1 Points-to Analysis

We use Andersen’s algorithm [4] as the basic algorithm to compute the annotated points-to set1 as described in the previous section and use it to compute the improved flow-insensitive solution. We then extract a “weaker” flow-sensitive solution, WeakFS, at each node using just the reachability relation. For this analysis, we did not use the intra-block variables renaming scheme. Use of the renaming scheme for points-to analysis needs a preliminary alias analysis and handling of assignments to dereferences (similar to [7]) before renaming can be attempted.

The graphs in Figure 12 show a plot of the size of the points-to relation obtained at the end of each basic block on a set of benchmarks taken from the Pointer-intensive [11], SPEC and McCat [6] benchmark suites. A lower value implies that the analysis is more precise. We have selected some programs on which an expensive flow-sensitive analysis provides a much better solution than a flow-insensitive one. Our analyses yield solutions which lie between the two.

The horizontal lines show the flow-insensitive solutions. The size of the points-to relation is the same for all basic-blocks as they compute a single summary solution for the whole procedure. The improved analysis is able to significantly improve the flow-insensitive solutions for KS-ks-InitLists and 08-main-object-InsertPoint (which can be seen by the lower horizontal line denoting the improved flow-insensitive solution). However, the other programs did not “kill” enough dataflow facts and hence our solution is the same as that from the flow-insensitive one.

The dotted line shows the flow-sensitive solutions obtained at the end of each basic block. The bars display the precision of the WeakFS solution extracted from the tagged summary obtained from the improved flow-sensitive analysis. The WeakFS results can be seen to be very close to the actual flow-sensitive results. This shows that a very precise solution at any basic-block can be obtained “on-demand” if the tagged summary solution is cached.

4.2 Constant Propagation

Here we present some preliminary results of using the above framework for an intraprocedural version of constant propagation2. All variables were assumed to be defined before use. All the global variables and the formal parameters were assumed to be non-constants. Also, to speed up the analysis, the temporaries introduced by Lance were not renamed by our intra-block variable renaming scheme. As our current implementation is intraprocedural, we inline any functions called from the subject programs before running the analysis. The analysis is done by first performing a flow-insensitive alias analysis (similar to [4]) to disambiguate assignments to pointer dereferences. The uses of pointer dereferences are assumed to be non-constant. The flow-sensitive analysis results were obtained by performing analysis on the SSA form as described in [7].

We recorded the number of uses of variables which our analysis could discover as a constant over that discovered using a flow-insensitive analysis over a set of programs taken from [2], selected at random. The results are shown in Figures 13 and 14.

The results are better than that from a flow-insensitive analysis and very close to the flow-sensitive solutions. As the situation is overly pessimistic (all formal parameters and globals declared non-constants) for an intraprocedural case, the number of constants which could be discovered even by a fully flow-sensitive analysis is small in many cases. The large percentage of constants identified by the flow-insensitive analysis is also misleading, as most of them were temporaries introduced in the Lance IR. We intend to study the effect of our scheme on inter-procedural constant propagation in the future and we feel that the benefits would be more pronounced in that case.

5. REFERENCES

\[ x, x_5 \in \text{Variables} \quad \text{gen, curr, knode} \in \text{BasicBlocks} \quad \text{curr} \neq \text{gen} \quad \text{gen} \in \text{ReachSet}(\text{curr}) \]

\[ \text{knode} \in \text{KillSet}(x_5) \quad x_5 \in \text{IsVersion}(x) \quad \text{knode} \neq \text{gen} \quad \text{knode} \neq \text{curr} \quad \text{knode} \in \text{dom}_{\text{gen}}(\text{curr}) \]

\[ \text{KilledDefinition}(x, \text{gen}, \text{curr}) \]

\[ x, y \in \text{Variables} \quad \text{curr} \in \text{BasicBlocks} \quad \text{pc} \in \text{StmNum} \quad \text{Expr}(\text{curr}, \text{pc}, y, x) \quad \text{IsNotConst}(\text{curr}, y) \]

\[ \text{IsNotConst}(\text{curr}, x) \]

\[ x, x_1, x_2, x_3 \in \text{Variables} \quad \text{gen, curr} \in \text{BasicBlocks} \quad x_1 \in \text{FirstVersion}(x) \quad x_3 \in \text{LastVersion}(\text{gen}, x) \quad \text{IsUsed}(\text{curr}, x_1) \quad \text{Def}(\text{gen}, x, x_3) \quad \text{IsInsideLoop}(\text{curr}) \quad \text{gen} \in \text{ReachSet}(\text{curr}) \]

\[ \neg \text{KilledDefinition}(x, \text{gen}, \text{curr}) \]

\[ x, x_1, x_2, x_3 \in \text{Variables} \quad \text{gen}_1, \text{gen}_2, \text{curr} \in \text{BasicBlocks} \quad x_1 \in \text{FirstVersion}(x) \quad x_3 \in \text{LastVersion}(\text{gen}_1, x) \quad \text{Def}(\text{gen}_1, x, x_3) \quad \text{Def}(\text{gen}_2, x, x_3) \quad x_3 \in \text{LastVersion}(\text{gen}_2, x) \quad \text{gen}_1 \neq \text{gen}_2 \quad \text{gen}_1 \neq \text{curr} \quad \text{gen}_2 \neq \text{curr} \quad \text{gen}_1 \in \text{ReachSet}(\text{curr}) \]

\[ \neg \text{KilledDefinition}(x, \text{gen}_1, \text{curr}) \quad \neg \text{KilledDefinition}(x, \text{gen}_2, \text{curr}) \]

\[ x, x_1, x_2, x_3 \in \text{Variables} \quad \text{gen, curr} \in \text{BasicBlocks} \quad x_1 \in \text{FirstVersion}(x) \quad x_3 \in \text{LastVersion}(\text{gen}, x) \quad \text{gen} \in \text{ReachSet}(\text{curr}) \quad \neg \text{KilledDefinition}(x, \text{gen}, \text{curr}) \quad \text{IsNotConst}(\text{gen}, x_3) \]

\[ \text{IsNotConst}(\text{curr}, x_1) \]

\[ x, x_1, x_2, x_3 \in \text{Variables} \quad \text{gen, curr} \in \text{BasicBlocks} \quad x_1 \in \text{FirstVersion}(x) \quad x_2 \in \text{LastVersion}(\text{gen}, x) \quad \text{IsUsed}(\text{curr}, x_1) \quad \text{Def}(\text{gen}, x, x_2) \quad \text{IsInsideLoop}(\text{curr}) \quad \text{gen} \in \text{ReachSet}(\text{curr}) \]

\[ \neg \text{KilledDefinition}(x, \text{gen}, \text{curr}) \quad \text{IsNotConst}(\text{curr}, x_2) \]

\[ x \in \text{Variables} \quad \text{curr} \in \text{BasicBlocks} \quad \text{curr} \in \text{BasicBlock} \quad x \in \{\text{HEAP, UNDEF}\} \]

\[ \text{IsNotConst}(\text{curr}, x) \]

\[ x \in \text{Variables} \quad \text{bb} \in \text{BasicBlocks} \quad \text{pc} \in \text{StmNum} \quad \text{IsUsed}(\text{bb}, \text{pc}, x) \quad \neg \text{IsNotConst}(\text{bb}, x) \]

\[ \text{ConstUse}(\text{bb}, \text{pc}, x) \]

\[ \text{Figure 11: Intraprocedural Constant Propagation using intra-block variable renaming} \]

\[ \text{Figure 12: Experimental Results for intraprocedural points-to analysis. The plots show the sizes of the points-to relation discovered by the various analyses at (the end of) each basic-block. (FI: Flow-insensitive, FS: Flow-sensitive, ImprFI: Improved Flow-insensitive, WeakFS: Weak Flow-sensitive extracted from the tagged points-to summary)} \]


