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 `S1:x=z` only if the fact $z \rightarrow y$ holds at `S1`. 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.

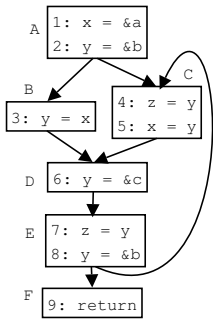

Figure 1: A motivating example

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 computing solutions using tools like *bddbddb*[9].

2. PREVIOUS WORK

Our current work carries ideas similar to [5]: while Burke

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

SAC' 08, March 16-20, 2008, Fortaleza, Ceará, Brazil.

$$ReachSet(x) = \{x\} \bigcup \sum_{y \in pred(x)} ReachSet(y)$$

$$Dom_x(y) = \begin{cases} \{y\} \bigcup \prod_{z \in pred(y)} Dom_x(z) & if\ x \in ReachSet(y), x \neq y \\ \{x\} & if\ x = y \\ \emptyset & otherwise \end{cases}$$

Figure 2: Computing the ReachSet and the All Node Dominator relations. The solution is the maximum fixpoint solution of the above equations.

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 intraprocedural 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 $y \in Dom(x)$), 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 $y \in Dom_z(x)$) 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.

$$\begin{gathered} Control\text{-}Flow\ Graph,\ G = (N,\ E,\ start) \\ N = Set\ of\ all\ basic\ blocks. \\ DataFlowFact = \{d_i | d_i\ is\ a\ possible\ data\ flow\ fact\} \\ GenSet(d_i) = \{n | n \in N, d_i\ might\ be\ generated\ at\ n\} \\ KillSet(d_i) = \{n | n \in N, d_i\ is\ unambiguosly\ killed\ at\ n\} \\ Dependent(d_i) = \{d_j | d_j \in DataFlowFact, \\ generation\ of\ d_i\ is\ dependent\ on\ d_j\} \\ Statement(n) = \{stm | stm\ is\ a\ statement\ in\ the\ basic\ block\ 'n'\} \\ EnvFact = Set\ of\ all\ dataflow\ facts\ from\ the\ environment. \end{gathered}$$

$$\frac{d_i \in DataFlowFact \quad gen, curr, kill \in N \quad curr \neq gen \quad kill \neq gen \quad kill \neq curr \quad gen \in ReachSet(curr) \quad kill \in KillSet(d_i) \quad kill \in Dom_{gen}(curr)}{SureKilledFact(d_i, gen, curr)}$$

$$\frac{d_i \in DataFlowFact \quad start \in N \quad d_i \in EnvFact}{ValidFact(d_i, start, start)}$$

$$\frac{d_i \in DataFlowFact \quad gen, curr \in N \quad gen = curr \quad gen \in GenSet(d_i)}{ValidFact(d_i, gen, curr)}$$

$$\frac{d_i \in DataFlowFact \quad gen, curr \in N \quad gen \in GenSet(d_i) \quad gen \in ReachSet(curr) \quad \neg SureKilledFact(d_i, gen, curr)}{ValidFact(d_i, gen, curr)}$$

$$\frac{d_i, d_j \in DataFlowFact \quad gen, curr \in N \quad stm \in Statement(curr) \quad \prod_{d_j \in Dependent(d_i)} ValidFact(d_j, gen, curr)}{ValidFact(d_i, gen, curr)}$$

$$\frac{d_i \in DataFlowFact \quad gen \in N \quad ValidFact(d_i, gen, *)}{gen \in GenSet(d_i)}$$

$$\frac{d_i \in DataFlowFact \quad ValidFact(d_i, *, *)}{d_i \in DataFlowSolution}$$

Figure 3: Improved Flow-Insensitive Analysis Semantics. The solution is obtained by fixpoint computation over these rules.

$$\frac{d_i \in DataFlowFact \quad gen, curr \in N \quad gen \in (GenSet(d_i) \cap ReachSet(curr))}{d_i \in SolutionSet(curr)}$$

Figure 4: Recovering a better solution at specific program point.

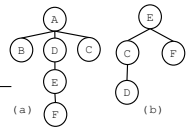

Figure 5: Dom. trees for Figure 1 w.r.t A & E.

For each dataflow fact d_i, we find the set $KillSet(d_i)$ — all basic blocks where d_i is unambiguously killed. The relation $SureKilledFact(d_i, gen, curr)$ holds if the fact d_i, generated at a basic block `gen` is certainly killed at another basic block `curr` — if $(Dom_{gen}(curr) \cap KillSet(d_i)) \neq \emptyset$ (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

not kill information within a basic block);

- 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 `stm` 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 `<gen,curr>` 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.

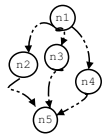

Figure 6: Limitation of the improved flow-insensitive analysis.

The above analysis tends to be more precise than a flow-insensitive analysis — assume that a dataflow fact d_j could get generated by a statement at a basic block `curr`, and d_j is dependent on another fact d_i. In that case, d_j would get generated only if d_i might hold at `curr`, i.e. $ValidFact(d_i, gen, curr)$ holds for any $gen \in GenSet(d_i)$ — in contrast to a flow-insensitive 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 $n_1 \in GenSet\{d_i\}$ and $n_2, n_3, n_4 \in KillSet(d_i)$. So, although the fact d_i does not reach n_5 as it gets killed along all paths from n_1 to n_5, our analysis fails to discover this fact as none of $\{n_2, n_3, n_4\}$ is a member of $Dom_{n_1}(n_5)$.

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 being 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:

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

Statement	Dependent Fact	Gen. Fact
$v_1 = \&v_2$	—	$v_1 \rightarrow v_2$
$v_1 = v_2$	$v_2 \rightarrow v_x$	$v_1 \rightarrow v_x$
$v_1 = {}^*v_2$	$v_2 \rightarrow v_x \wedge v_x \rightarrow v_y$	$v_1 \rightarrow v_y$
${}^*v_1 = v_2$	$v_1 \rightarrow v_x \wedge v_2 \rightarrow v_y$	$v_x \rightarrow v_y$

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 $f^B_{v_1 v_2}$. Note that the presence of the fact $f^B_{v_1 v_2}$ simply implies that $B \in GenSet(f_{v_1 v_2})$.

The facts f^A_{xa}, f^A_{yb} , f^E_{yb} and f^D_{yc} would be the first facts to be generated (by the statements `1`, `2`, `8` and `6` respectively) as they are not dependent on any other facts. In conventional flow-insensitive analysis, the statement "`5:x=y`", in the presence of the fact f_{yc} would have generated the fact f_{xc}. For our analysis, $Dom_D(C) = \{D, E, C\}$ and $KillSet(f_{yc}) = \{A, B, D, E\}$. Hence, the relation $SureKilledFact(f^D_{yc}, D, C)$ holds, which prevents f^C_{xc} from being added to $ValidFact$.

$x \rightarrow a$	$x \rightarrow b$	$[x \rightarrow c]$
$y \rightarrow a$	$y \rightarrow b$	$y \rightarrow c$
$[z \rightarrow a]$	$z \rightarrow b$	$z \rightarrow c$

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.

Figure 8 shows the points-to facts generated for Figure 1 by the conventional flow-insensitive analysis and by our analysis. Our algorithm was 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 $\{f^A_{xa}, f^A_{yb}\}$ and those generated at B is $\{f^B_{ya}\}$. So, the solution $\{f^A_{xa}, f^A_{yb}, f^B_{ya}\}$ is very close to the flow-sensitive solution at B, that is $\{f_{xa}, f_{ya}\}$.

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

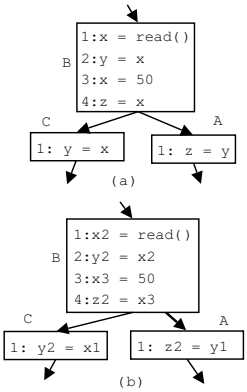

Figure 9: Motivation for intra-block variable renaming

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

Figure 10: Variable Renaming Algorithm

concern. We use a scheme similar to the SSA form, but limited within basic-blocks, to address this issue. This representation is much cheaper to build than the SSA form. Also, it does not lead to any increase in code size (as no ϕ-functions are required) and avoids the tremendous increase in the number of variables as caused by the full SSA form. Both the above factors are crucial for scalability when using tools like *bddbddb* [9].

Our variable renaming algorithm simply traverses over all the basic-blocks, numbering the uses for the variable, incrementing the version number at each definition. The algorithm is shown in Figure 10. Note that the sets `IsVersion` records the versions of a particular variable, the set `FirstVersion` records the first version of a variable and `LastVersion` records the last version of the variable within a basic block. We would refer to the program variables as *base variables* and the variables generated due to variable renaming as *versioned variables*. Together they form the set `Variables`.

The version numbers assigned to a variable within a block are "local" i.e. they are not "visible" outside the respective basic-blocks. The first and the last version of a variable have a few additional properties:

- only the first version of a variable in a basic-block is capable of reading dataflow facts from outside the block;
- only the last version is capable of "exporting" dataflow facts to the rest of the basic-blocks.

We use another classic dataflow problem — constant propagation — to explain the scheme. We, however, use a weaker form of constant propagation where a variable `x` is marked non-constant if it is reachable from two different definitions (even if both the definitions would always evaluate to the same value). Figure 9(a) provides a motivation for our scheme. We assume that the only definitions of the variables `x`, `y` and `z` are that shown in the figure. Note that in the basic-block `B`, the use of `x` at statement number `2` is a non-constant. However, at statement number `4`, the use of `x` is constant — but our analysis as explained in the previous section would fail to detect the same. Moreover, it would also mark the use of `x` at basic-block `C` as non-constant.

Figure 9(b) shows the partial flow-graph after intra-block variable renaming. Now, `x2` is identified as non-constant and `x3` as constant in basic-block `B`. Also, as only the last version of `x` in basic-block `B` (which is `x3`) is allowed to "export" its state outside its owner block, the use of `x` in statement number `1` of basic-block `C` is also marked as a constant. Also note that the state of `x1` remains constant irrespective of the other statements in the basic-block `C` (as other definitions to `x` would get different versions).

The semantics for the analysis with intra-block renamed variables for intraprocedural constant propagation is shown in Figure 11. We assume that alias information is available and each assignment to a variable dereference is substituted by conditional assignments to each possible pointee (see [7]).

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

- `Expr(bb,pc,y,x)` implies that at `<bb,pc>`, `x` is assigned to an expression involving `y`;
- `Def(bb,pc,x)`/`IsUsed(bb,pc,x)` imply that the variable `x` is defined/used at the program point `<bb,pc>`;
- `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;
- `IsNotConst(bb,x)` implies that the variable `x` is non-constant in the basic block `bb`;
- `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 `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 "`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 set[1] 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 propagation[2]. 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 inlined 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].

[1]Heap locations are summarized by special per-allocation site `HEAP` variables. For `HEAP` variables h_1 and h_2, we generate a dataflow fact $h_1 \rightarrow h_2$ if any heap location summarized by h_1 may point-to a location summarized by h_2.

[2]We use a weaker definition of constant propagation wherein the use of a variable is constant iff it is reached by a single definition.

Bench.	Program	Func.	fs	fi	im	vr
fibcall	fibcall.c	fib	4	0	4	4
expint	expint.c	expint	18	10	17	18
ns	ns.c	foo	13	1	4	7
cover	cover.c	swi10	3	0	3	3
crc	crc.c	icrc	18	9	13	16
ndes	ndes.c	cyfun	77	65	71	77
fdct	fdct.c	fdct	87	57	72	87

Figure 13: Results for intraprocedural Constant Propagation. The comparison is between the number of uses that were discovered by our improved analysis using the all-node dominator relation without (im) and with variable renaming (vr) against that discovered using a flow-sensitive (fs) and a flow-insensitive analysis (fi).

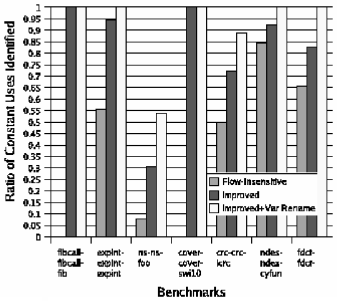

Figure 14: The ratio of constant uses identified for each analysis, normalized by that obtained by flow-sensitive analysis

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

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

[2] WCET project/Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[4] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[5] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-Insensitive Interprocedural Alias Analysis in the Pressence of Pointers. In David Gelertner, Alexandru Nicolau, and David Padua, editors, *LNCS, 892*. Springer-Verlag, 1995.

$$\frac{x, x_5 \in Variables \quad gen, curr, knode \in BasicBlocks \quad curr \neq gen \quad gen \in ReachSet(curr) \quad knode \in KillSet(x_5) \quad x_5 \in IsVersion(x) \quad knode \neq gen \quad knode \neq curr \quad knode \in dom_{gen}(curr)}{KilledDefinition(x, gen, curr)}$$

$$\frac{x, y \in Variables \quad curr \in BasicBlocks \quad pc \in StmNum \quad Expr(curr, pc, y, x) \quad IsNotConst(curr, y)}{IsNotConst(curr, x)}$$

$$\frac{x \in Variables \quad curr \in BasicBlocks \quad pc \in StmNum \quad PtrUse(curr, pc, x)}{IsNotConst(curr, x)}$$

$$\frac{x, x_1, x_4, x_9 \in Variables \quad gen, curr \in BasicBlocks \quad gen \neq curr \quad Def(curr, *, x_4) \quad x_4 \in IsVersion(x) \quad x_1 \in FirstVersion(x) \quad x_9 \in LastVersion(gen, x) \quad IsUsed(curr, x_1) \quad Def(gen, *, x_9) \quad IsInsideLoop(curr) \quad gen \in ReachSet(curr) \quad \neg KilledDefinition(x, gen, curr)}{IsNotConst(curr, x_1)}$$

$$\frac{x, x_1, x_8, x_9 \in Variables \quad gen_1, gen_2, curr \in BasicBlocks \quad x_1 \in FirstVersion(x) \quad x_8 \in LastVersion(gen_1, x) \quad Def(gen_1, *, x_8) \quad Def(gen_2, *, x_9) \quad x_9 \in LastVersion(gen_2, x) \quad gen_1 \neq gen_2 \quad gen_1 \neq curr \quad gen_2 \neq curr \quad gen_1 \in ReachSet(curr) \quad \neg KilledDefinition(x, gen_1, curr) \quad gen_2 \in ReachSet(curr) \quad \neg KilledDefinition(x, gen_2, curr)}{IsNotConst(curr, x_1)}$$

$$\frac{x, x_1, x_9 \in Variables \quad gen, curr \in BasicBlocks \quad x_1 \in FirstVersion(x) \quad Def(gen, *, x_9) \quad gen \neq curr \quad x_9 \in LastVersion(gen, x) \quad gen \in ReachSet(curr) \quad \neg KilledDefinition(x, gen, curr) \quad IsNotConst(gen, x_9)}{IsNotConst(curr, x_1)}$$

$$\frac{x \in Variables \quad curr \in BasicBlocks \quad curr \in BasicBlock \quad x \in \{HEAP,\ UNDEF\}}{IsNotConst(curr, x)}$$

$$\frac{x \in Variables \quad bb \in BasicBlocks \quad pc \in StmNum \quad IsUsed(bb, pc, x) \quad \neg IsNotConst(bb, x)}{ConstUse(bb, pc, x)}$$

Figure 11: Intraprocedural Constant Propagation using intra-block variable renaming

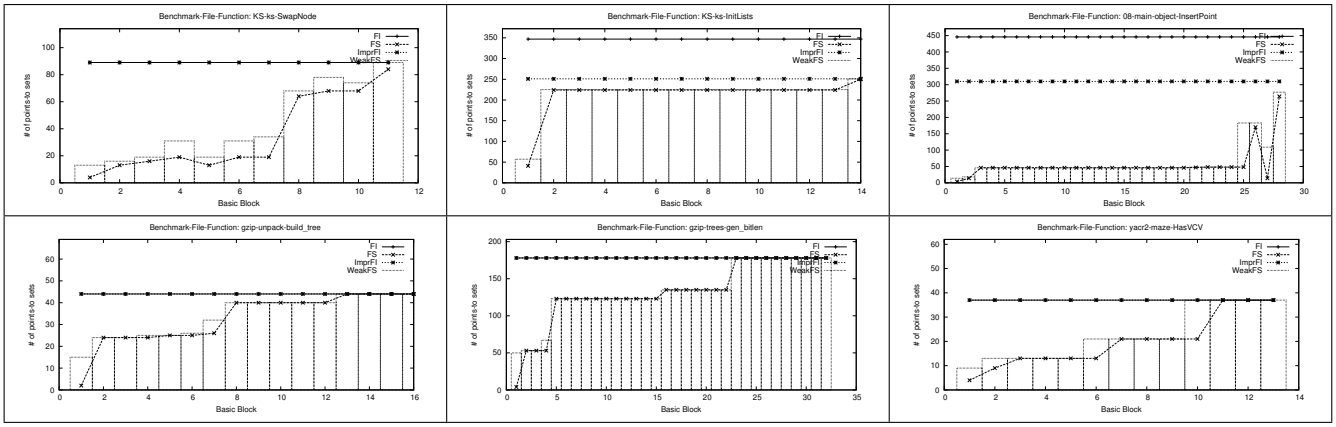

Figure 12: Experimental Results for intraprocedural points-to analysis. The plots show the sizes of the points-to relation discovered by the various analyses at (the end of) each basic-block. (FI: Flow-insensitive, FS: Flow-sensitive, ImprFI: Improved Flow-insensitive, WeakFS: Weak Flow-sensitive extracted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

[6] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Conf. on Prog. Lang. Design and Impl.*, pages 242–256, 1994.

[7] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *PLDI '98: Proc. of Conf. on Prog. Lang. Design and Impl.*, pages 97–105, 1998.

[8] Uday P. Khedker. Data flow analysis. In Y. N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook*, pages 1–59. CRC Press, 2002.

[9] Monica S. Lam, John Whaley, V. Benjamin Livshits, Michael C. Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. Context-sensitive program analysis as database queries. *PODS '05: Proc. of Sym. on Principles of database systems*, pages 1–12, 2005.

[10] Thomas Lengauer and Robert Endre Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Trans. Program. Lang. Syst.*, volume 1, pages 121–141, New York, NY, USA, 1979. ACM Press.

[11] T. Austin, et al. Pointer-intensive benchmark suite. http://pages.cs.wisc.edu/~austin/ptr-dist.html, 1995.