Speeding Up Dataflow Analysis Using Flow-Insensitive Pointer Analysis

Stephen Adams[1], Thomas Ball[1], Manuvir Das[1], Sorin Lerner[2], Sriram K. Rajamani[1], Mark Seigle[2], and Westley Weimer[3]

[1] Microsoft Research
[2] University of Washington
[3] UC Berkeley

Abstract. In recent years, static analysis has increasingly been applied to the problem of program verification. Systems for program verification typically use precise and expensive interprocedural dataflow algorithms that are difficult to scale to large programs. An attractive way to scale these analyses is to use a preprocessing step to reduce the number of dataflow facts propagated by the analysis and/or the number of statements to be processed, before the dataflow analysis is run. This paper describes an approach that achieves this effect. We first run a scalable, control-flow-insensitive pointer analysis to produce a conservative representation of value flow in the program. We query the value flow representation at the program points where a dataflow solution is required, in order to obtain a conservative over-approximation of the dataflow facts and the statements that must be processed by the analysis. We then run the dataflow analysis on this "slice" of the program.
We present experimental evidence in support of our approach by considering two client dataflow analyses for program verification: typestate analysis, and software model checking. We show that in both cases, our approach leads to dramatic speedups.

1 Introduction

In recent years, static analysis has increasingly been applied to the problem of program verification. Two kinds of algorithms for static program analysis have been proposed. Flow-insensitive algorithms such as [DLFR01,HT01,OJ97] scale to large programs, but do not handle strong updates precisely. Flow-sensitive algorithms [CRL99,WL95], on the other hand, can handle strong update precisely but do not scale to large programs. The absence of strong update adversely affects software engineering and program verification tools, which produce many false positives with conservative analysis. Therefore, these tools often employ costly interprocedural dataflow algorithms, even though this choice precludes application to large programs. This paper is based on two key insights. The first insight is that the points-to algorithm of Das [Das00] can be viewed as producing a lightweight, conservative representation of the value flow in a program, called the value-flow graph (VFG). The second insight is that by querying the

VFG before a client dataflow analysis is run, both the number of dataflow facts propagated by the analysis and the part of the program that must be analyzed can be reduced.

Two small examples that demonstrate our approach are given in Figure 1.

Figure 1(a) shows a procedure whose file I/O related behavior must be summarized for all callers. By performing a "typestate slice" in the VFG, we can remove dataflow facts corresponding to the states of the file handle `g2` (since the state of `g2` is not changed by `bar`) and statements that do not affect the state of `g1`, before analyzing `bar`.

Figure 1(b) shows a program fragment on which we wish to determine whether the `abort` function is called. One way to do this is to determine a set of predicates over program variables, construct a boolean program abstraction [BR01a] over these predicates, and then model check the abstraction using dataflow analysis. The predicates required for proving that the `abort` statement is unreachable are `x == 5`, `y == 5`, `x == 10`, and `y == 10`. These predicates can be discovered in two iterations of iterative refinement (see Section 4). By performing a "predicate slice" in the VFG, we can infer the set of constants that flow into variables `x` and `y` and generate these predicates in advance, eliminating the iteration overhead.

These examples show the sort of information we hope to provide to client dataflow analyses. The typestate slice and predicate slice can both be obtained from the VFG. Each slice reduces the workload of the client dataflow analysis.

Two alternative approaches for achieving the same effect are demand-driven versions of the client dataflow analyses or standard program slicing. Both approaches have drawbacks. Frameworks for demand-driven dataflow [DGS95, HRS95] do not handle a general enough class of dataflow problems to allow application to problems that involve value flow. Program slicing [HRB90], which is based on dataflow analysis, it typically too expensive to apply on large programs.

```
(a)  FILE *g1, *g2;
     void bar() {
       FILE *l, *m;
       l = g1;
       m = g2;
       fclose(l);
     }
```

```
(b)  void baz(bool b) {
       int x , y;
       if (b) {
         x = 5;
         y = 5;
       } else {
         x = 10;
         y = 10;
       }
       if (x != y)
         abort();
     }
```

Fig. 1. Examples of speeding up dataflow analysis. Figure (a) above shows a fragment of C code to be sliced w.r.t. file I/O behavior. Figure (b) above shows a fragment of C code to be sliced w.r.t. reachability of calls to `abort`.

The main contributions of this paper are:

– We show how the flow-insensitive pointer analysis of Das can be used to produce a program-point-independent graph representation of value flow (the VFG) in a program.
– We show how the VFG can be used to perform client-specific program slices that significantly speed up client dataflow analyses.
– We demonstrate the effectiveness of our technique with two specific applications of slicing using the VFG:
  - Typestate slicing: We use typestate slicing to reduce the number of objects whose state is tracked by ESP [DLS02], a typestate checker for large programs. We apply ESP to check file I/O properties of the `gcc` compiler. Typestate slicing reduces the average number of objects tracked per procedure from 1100 to less than 1, making the dataflow analysis in ESP practical.
  - Predicate slicing: We use predicate slicing to generate a candidate set of predicates for the initial boolean program abstraction used by SLAM [BR01a], a software model checker. We apply SLAM to check properties of several Windows device drivers. Predicate slicing reduces the running time of SLAM by a factor of 2-10, and allows SLAM to terminate in some cases where it did not terminate before.

The rest of this paper is structured as follows. In Section 2, we describe our value flow representation, its computation, and its interface to dataflow clients. We then present two concrete applications of our value flow slicing approach. In Section 3, we demonstrate the use of typestate slicing in ESP. In Section 4, we show how predicate slicing can be used to aid predicate discovery in SLAM. We discuss related work in Section 5, and conclude in Section 6.

2 Value Flow via Pointer Analysis

Pointer analysis algorithms typically produce graph representations ("points-to graphs") of pointer relationships in programs. These graphs encode information about which memory locations hold references to other memory locations. More importantly, algorithms based on subtyping encode constraints that arise from value flow through assignments in the program: every assignment that causes flow of pointer values is represented either implicitly or explicitly in the graph. The key insight of this paper is that if pointer analyses based on subtyping are modified to process all assignments, rather than just pointer assignments, and if constant values are represented explicitly in the graph, the resulting points-to graph encodes a conservative approximation of all value flow in the program (we call this the value flow graph, or VFG). In addition, if the fragments of the graph that result from processing constraints are labeled with the identities of the statements that generated the constraints, the VFG encodes slices of the program: each slice represents the set of program statements that contribute to the value of a given expression. In the terminology of program slicing [Tip95], these slices encode flow dependences, but not control dependences. When the client dataflow analysis ignores control dependences (this is typically true of many dataflow analyses, including the examples considered in this paper) the flow dependence slices encoded in the VFG preserve the results produced by the client dataflow analysis.

We focus on flow-insensitive analyses, which produce a single representation of all of the value flow in the program that is applicable at every point in the program. Flow-sensitive analyses could be used to produce VFGs as well: they would produce separate graphs at each program point, though every graph needs to encode only values relevant at the program point.

2.1 One-Level Value-Flow Analysis

In this paper, we use the one-level-flow subtyping analysis of Das [Das00] to produce value flow information. The points-to graphs produces by this algorithm make value flow through assignments explicit, via so-called "flow" edges.

The graph includes two kinds of nodes: source nodes, and expression nodes. Source nodes represent variables in the program, and are labeled by the variable name. Expression nodes represent pointer dereferences of source nodes. An expression node can be named by a path starting from a source node with a dereference (*) prefixed for every points-to edge on the path from the source node. Due to aliasing, expression nodes do not have unique names, and are thus not labeled.

The graph includes two kinds of edges: points-to edges (written $\rightarrow$) and flow edges (written $\rightarrow_F$). A points-to edge connects (the node of) an expression `e` with (the node of) an expression `*e`. A flow edge $*$`e1` $\rightarrow_F$ $*$`e2` encodes an assignment from `e1` to `e2` in the program. Because the algorithm sometimes merges

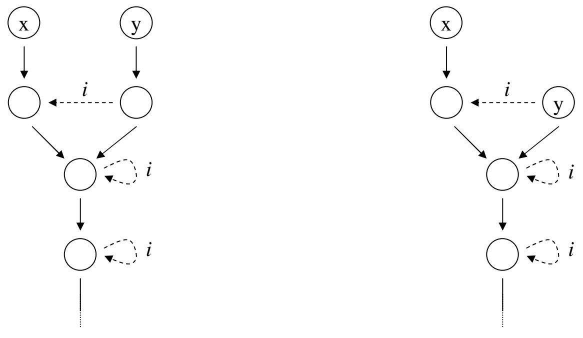


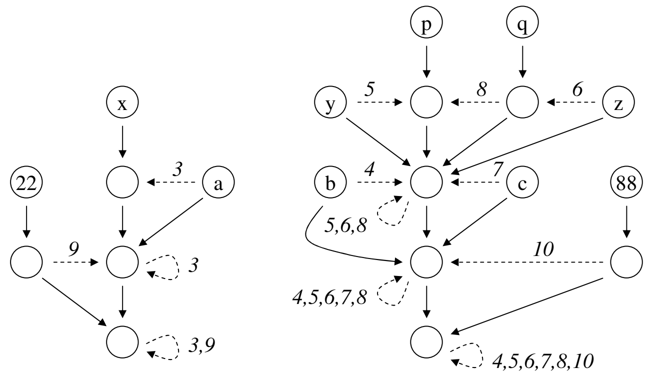
Fig. 2. Figures (a) and (b) above show the VFGs computed by the one level flow algorithm for i: `x = y` and i: `x = &y`, respectively. Points-to edges are solid arrows. Flow edges are labeled dashed arrows.

expressions into the same node, value flow is also encoded in node equality: every node can be viewed as having a flow edge to itself.

The graph fragments produced by the analysis for two kinds of assignments are shown in Figure 2. The assignment `x = y` induces an edge $*\mathtt{y} \rightarrow_F *\mathtt{x}$, indicating that the value of `y` may flow to `x`, and induces node equality for `**x` and `**y`, `***x` and `***y`, and so on, indicating that the assignment also implicitly creates value flow between `*x` and `*y`, `**x` and `**y`, and so on. The assignment `x = &y` induces an edge $\mathtt{y} \rightarrow_F *\mathtt{x}$, indicating that the address of `y` may flow to `x`. A more detailed discussion of the algorithm can be found in [Das00].

We extend the analysis in two ways to create the VFG: First, we add nodes for values that are not stored in any memory location (*e.g.* constants). Second, we label flow edges with the assignments that induced the edges. We also transitively label all of the self-loop flow edges that are induced at lower (in terms of points-to edges) levels of the graph.

```
 0: void main() {
 1:  int a,b,c,*x,
 2:   *y,*z,**p,**q;
 3:  x = &a;
 4:  y = &b;
 5:  p = &y;
 6   q = &z;
 7:  z = &c;
 8:  p = q;
 9:  *x = 22;
10:  **p = 88;
11: }
```



(a) (b)

Fig. 3. (a) A fragment of C code and its VFG, augmented with source code labels and nodes for constants, and (b) its value-flow graph.

Example 1 A small C program is shown in Figure 3(a). The VFG computed for this program by the one-level value-flow analysis is shown in Figure 3(b). This graph encodes all of the value flow in the program. For instance, the flow of the value 22 to variable `a` at line 9 is encoded in the flow edge from `*22` to `*a` labeled 9. Notice that the statement at line 3, where `x` is assigned the address of `a`, also contributes to this value flow. The role of this statement is captured by the self-loop flow edge on `*a` labeled 3. □

For our purposes, the VFG produced by the modified one-level-flow algorithm has several interesting properties:

– The graph can be computed in almost-linear time.

- Points-to sets can be obtained from the graph in linear time: the set of variables pointed-to by p (written $\mathsf{Vars}(*\mathsf{p})$) may be obtained by performing backwards reachability along flow edges from `*p` and including all of the variables encountered along the way.
- Value flow can be queried in linear time: the set of expressions whose values may flow to e may be obtained by performing backwards reachability along flow edges from `*e` and including all of the expressions with points-to edges to nodes encountered along the way.

Example 2 The VFG in Figure 3 can be used to answer a number of points-to and value flow queries. A backwards flow query from `**x` picks up `*22`, which means that the value 22 can flow to any variable in the points-to set `*x`, including a. Because the query does not pick up any of b, c, y, z, p, or q, all of these variables can be ignored by a dataflow client that is tracking the flow of the value 22 through the program. □

2.2 Slicing Interface

Although the entire value flow graph could be presented to a client analysis, in practice most clients make a series of structured queries to the graph. This interface to the value flow graph is summarized below:

- $\mathsf{N}(\mathsf{v})$ is the node representing v, $\forall \mathsf{v} \in \mathit{ProgVar}$.
- $\mathsf{Deref}(\mathsf{p}) = \mathsf{q}$ where $\mathsf{p} \to \mathsf{q}$.

  $\mathsf{Deref}(\mathsf{p})$ performs a single pointer dereference from p in the VFG.
- $\mathsf{N}(\mathsf{e}) = \begin{cases} \mathsf{N}(\mathsf{v}) & \text{if } \mathsf{e} = \mathsf{v} \\ \mathsf{Deref}(\mathsf{N}(\mathsf{e}')) & \text{if } \mathsf{e} = *\mathsf{e}' \end{cases}$

  $\mathsf{N}(\mathsf{e})$ maps expression e to its node in the VFG.
- $\mathsf{Vars}(\mathsf{e}) = \{\mathsf{v} \in \mathit{ProgVar} \mid \mathsf{v} \to_F^* \mathsf{e}\}$,

  $\mathsf{Vars}(\mathsf{e})$ is the set of variables that may be aliases of expression e.
- $\mathsf{FlowsInto}(\mathsf{e}) = \{\mathsf{q} \mid \mathsf{Deref}(\mathsf{q}) \to_F^* \mathsf{Deref}(\mathsf{e})\}$.

  $\mathsf{FlowsInto}(\mathsf{e})$ returns all expressions whose values may flow into e.
- $\mathsf{FlowsTo}(\mathsf{e}) = \{\mathsf{q} \mid \mathsf{Deref}(\mathsf{e}) \to_F^* \mathsf{Deref}(\mathsf{q})\}$.

  Conversely, $\mathsf{FlowsTo}(\mathsf{e})$ returns all expressions into which the value of e may flow.

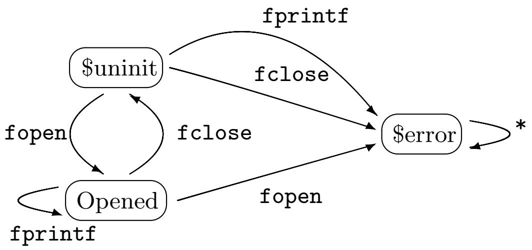
The case studies presented in the next two sections use this interface to the value flow graph. In this description we have not considered objects with structure. The value flow representation can be extended in a straightforward manner to encode structure field accesses.

3 Typestate Slicing in ESP

ESP [DLS02] is a verification tool that identifies violations of programmer specified properties in large C programs. ESP tracks "typestate" [SY86]: As a program executes, it creates values. Every value is associated with a type that is invariant during program execution. Some values are additionally associated with a state that may be updated by certain operations on values. Transitions between states are encoded in a state machine. The machine has a special error state; transitions to the error state indicate violations of the property. An example of a user specified property (valid file output) is given in Figure 4(a).

The core engine of ESP is an interprocedural dataflow analysis that computes the possible states of every value at every point in the program. Because ESP is intended for application on large programs, the dataflow analysis is performed bottom-up on the call graph, one strongly connected component at a time. As a result, ESP must summarize the typestate behavior of a function `foo` for all possible callers before any of the callers of `foo` are analyzed. The typestate summary of a function `foo` includes (a) the state changes caused by execution of `foo` on values live before the execution of `foo`, and (b) any new stateful values created by execution of `foo`, and their states on exit from `foo`. In order to compute the summary of a function, ESP first computes the set of memory locations that may hold stateful values at entry to the function, and then performs dataflow analysis for each such value. In a language with type coercion such as C, if ESP is computing the states of values of a particular type, for instance file handles with type `FILE *`, it must also consider values of all other possible types.

The set of locations that may hold a stateful value at entry to a function `foo` includes all globals, formal parameters of `foo`, and transitive dereference targets of these. We refer to this set as *inNodes* (input interface nodes):



```
FILE *e, *f, *g, *h;

void foo () {
   int *x, y;
   y = 3;  x = &y;
   g = fopen(...);
   bar(x);
}

void bar (int* p) {
   fprintf (h, "%d", *p);
   fclose (h);
}
```

(a) (b)

Fig. 4. (a) A finite state property that specifies correct usage of a file output library, and (b) an example program that uses the library. Function `foo` calls `bar` with an integer pointer. `bar` prints its dereferenced parameter to `h` and then closes `h`.

$$\begin{aligned}
&\mathit{inNodes}(f) \;=\; \bigcup_{p \in params(f)} \mathit{reached}(p) \;\cup\; \bigcup_{g \in globals} \mathit{reached}(g)\\
&\mathit{reached}(v) = \{v\} \cup \mathit{reached}(\mathsf{Deref}(v))\\
&\mathit{reached}_p(v) = \mathit{reached}(v) - \{v\}
\end{aligned}$$

Similarly, the set of locations that may hold a freshly created value at exit from `foo` includes all globals, the return value of `bar`, and dereference targets of these and the formal parameters of `bar`. We refer to this set as *outNodes* (output interface nodes):

$$\begin{aligned}
\mathit{outNodes}(f) = \mathit{reached}(\mathit{return}_f) \;&\cup\; \bigcup_{p \in params(f)} \mathit{reached}_p(p)\\
&\cup\; \bigcup_{g \in globals} \mathit{reached}(g)
\end{aligned}$$

Example 3 A small program that manipulates file handles is given in Figure 4(b). The sets of possible interface nodes for the functions in this program are:

inNodes(`foo`): {`e, *e, f, *f, g, *g, h, *h`}
outNodes(`foo`): {`e, *e, f, *f, g, *g, h, *h`}
inNodes(`bar`): {`e, *e, f, *f, g, *g, h, *h, p, *p`}
outNodes(`bar`): {`e, *e, f, *f, g, *g, h, *h, *p`}

In large programs with many globals, the interface node sets can be large enough to make typestate checking infeasible. Clearly, these sets include many locations that contain values whose state is not changed by the called function. A preprocessing step that can eliminate many of the spurious locations from the interface node sets will increase the efficiency of the subsequent typestate analysis.

3.1 Eliminating Interface Nodes via Value Flow

ESP uses a slicing procedure over the VFG to eliminate nodes from *inNodes* before the typestate analysis is run. The slicing procedure is based on the following observation: A location `l` must be included in *inNodes(*`f`*)* only if there is some expression `e` such that `e` is an argument to a state changing operation performed during execution of `f`, and the value held by `l` at entry to `f` can flow to `e`. Therefore, we can query the VFG to obtain an over-approximation of the set of locations that must be included in *inNodes(*`f`*)*. We refer to this procedure as “typestate slicing”.

A similar procedure can be used to eliminate nodes from *outNodes*. A location `l` must be included in *outNodes(*`f`*)* only if there is some expression `e` such that `e` is the result of a value creation operation, and the value of `e` can be held by `l` at exit from `f`.

ESP uses a language of syntactic patterns to identify state changing operations in the code. The control flow graphs produced by the ESP front-end contain two kinds of distinguished, mutually exclusive, pattern nodes:

– PATTERN(name, p): Represents a call to a function whose name appears along a transition in the protocol, i.e. `fclose(f)`. `p` is the expression on which the operation is applied.

– CPATTERN(name, p): Represents a call to a function which creates fresh stateful values, i.e. `f = fopen(...)`. In this case `p` represents the recipient of the new value.

The set of expressions that are arguments to pattern nodes can then be defined as follows:

$$pNodes(f) = \{n | PATTERN(\_, n) \in f\}$$
$$cpNodes(f) = \{n | CPATTERN(\_, n) \in f\}$$

The sliced sets $inNodes_s$ and $outNodes_s$ are given by:

$$inTargets(f) = pNodes(f) \cup \bigcup_{g \in callees(f)} inNodes_s(g)$$
$$inNodes_s(f) = inNodes(f) \cap \bigcup_{n \in inTargets(f)} \mathsf{FlowsInto}(n)$$

$$outSrcs(f) = cpNodes(f) \cup \bigcup_{g \in callees(f)} outNodes_s(g)$$
$$outNodes_s(f) = outNodes(f) \cap \bigcup_{n \in outSrcs(f)} \mathsf{FlowsTo}(n)$$

The equations above describe a slicing procedure that is applied bottom-up on the call graph, one strongly connected component (SCC) at a time. Although the equations appear to require a fixpoint computation for SCCs containing more than one function, the solution can be obtained by combining a single flow query each for $inNodes_s$ and $outNodes_s$ with intersection operations for each function in the SCC.

Example 4 After typestate slicing is applied, the sets of interface nodes for the functions in the program from Figure 4(b) are:

$inNodes$(`foo`): {`h`}	$outNode$(`foo`): {`g`}
$inNodes$(`bar`): {`h`}	$outNodes$(`bar`): {}

3.2 Experiments

We have implemented typestate slicing as a preprocessing step in ESP. The dataflow engine in ESP performs an exhaustive dataflow analysis of the entire code in a strongly connected component for every node in the interface node sets of functions in the SCC. Therefore, the performance of ESP is directly related to the number of interface nodes. ESP also uses typestate slicing to dramatically reduce the number of relevant CFG nodes before dataflow analysis. We do not discuss those results here.

File output in gcc. ESP has been applied to the problem of verifying the file output behavior of a version of the `gcc` compiler, taken from the SpecInt95 benchmark suite, using the property specification given in Figure 4. `gcc` has 140,000 LOC in 2149 functions over 66 files; there are 1,086 global and static variables; the call graph contains a single SCC with over 450 functions.

Typestate slicing applied bottom-up on `gcc` requires 200 seconds on a Toshiba Tecra 8200 laptop with a 1GHz Pentium III processor and 512MB RAM, running Windows XP. Slicing reduces the number of interface nodes from roughly 1100 to < 1 on average, with a median of 15 for functions with non-empty interface node sets. This dramatic reduction in the sizes of the interface node sets allows ESP to successfully verify `gcc` in less than 15mins and 750MB of memory. Verification of `gcc` w.r.t. the file output property would be infeasible if the set of global interface nodes were not pruned.

Registry key leakage in cmd. We have also used ESP to find resource leaks in the command shell interpreter (`cmd`) of a version of the Windows operating system. `cmd` has 40,000 LOC; there are 483 global and static variables. Typestate slicing applied bottom-up on `cmd` requires 33 seconds on a Compaq Evo W6000 desktop PC with a 2.2GHz Pentium IV processor and 2GB RAM, running Windows XP. Slicing reduces the number of interface nodes from roughly 500 to $<< 1$ on average, with a median of 1 for functions with non-empty interface node sets.

4 Predicate Slicing in SLAM

4.1 SLAM Overview

The SLAM toolkit validates temporal safety properties of C programs through a process of boolean abstraction [BMMR01,BR01a], interprocedural dataflow analysis, and counterexample-driven refinement. The first step in this process involves abstracting the C program to a boolean program. Boolean programs have all of the control structure of C programs but contain only boolean variables. These boolean variables represent predicates over expressions in the original program. For example, a boolean variable might represent "`(*ptr)==2`". The soundness of the boolean abstraction means that if a variable "`(*ptr)==2`" is true at a point L in the boolean program then `(*ptr)==2` will be true at the same point L in the original C program. Given a set of predicates $\mathcal{P}$ and a C program, SLAM generates the corresponding boolean program. A key property of this transformation is that if the error state is *not* reachable in the boolean program then it is *not* reachable in the original program. Assuming $\mathcal{P}$ has been well-chosen, checking the safety of the original program reduces to checking the safety of the boolean program.

Since the boolean program involves only control flow (`if`, `goto`, function calls) and a finite set of boolean variables, interprocedural dataflow analysis can be used to explore its state-space exhaustively [BR01b].[1] If the error state is not reachable, the program adheres to the safety policy. If there is a path to the error state then the path can be checked for feasibility in the original C program. If the path is feasible then the SLAM produces an error trace demonstrating how the original program violates the safety policy. Otherwise, the path is an

[1] When viewed as a dataflow problem, $\mathcal{P}$ is the set of dataflow facts and the boolean abstraction process gives the transition function for every statement in the program.

infeasible counterexample (or false positive) and SLAM generates new predicates to increase the precision of the boolean program abstraction on the subsequent iterations of SLAM.

4.2 Better Predicate Generation

The SLAM process is an example of "counterexample-driven refinement". When SLAM was first implemented, the initial set of predicates $\mathcal{P}$ was the empty set, and counterexample-driven refinement was used to expand the set of predicates as necessary. This process required many iterations in some cases (over 20 iterations for some Windows device drivers).

If, instead, we could start SLAM off with a "good" set of predicates that included most (if not all of the predicates) that would be discovered by the counterexample-driven refinement process, then SLAM would terminate more quickly. On the other hand, if we start with too many predicates (for example, all boolean relations between all pairs of expressions in the program) the search space may be too large and SLAM's dataflow analysis might exhaust system resources. Such a general approximation of predicates would link variables that are never meaningfully related by the flow of values in the original program.

We present a predicate slicing algorithm that works for a restricted subset of the C language. Given a VFG G for a program written in that subset, the algorithm produces a set of predicates $\mathcal{P}$ that is provably sufficient to avoid false positives but avoids linking unrelated terms. For C programs that fall outside this restricted subset, the algorithm produces a good set of initial predicates, and the remaining false positives can be eliminated using iterative refinement. This algorithm has been used in practice to produce predicate sets that allow for the rapid analysis of programs for which the naive analysis was either very slow or infeasible. By using the VFG, we were able to hit a "sweet-spot" and generate an initial set of predicates detailed enough to eliminate false positives but small enough to make the analysis scale.

4.3 Input Language

For the purposes of presentation, we consider a restricted subset of C which contains local scoping, procedural abstraction and the following statements:

$$\begin{array}{rll} s ::= & v_i \leftarrow n & (n \in \mathcal{Z}) \\ \mid & v_i \leftarrow v_j & \\ \mid & \mathsf{if}\ (\star)\ s_1\ \mathsf{else}\ s_2 & \\ \mid & v_i \leftarrow \mathsf{fun}(v_j, ...) & \\ \mid & \mathsf{return}(v_j) & \\ \mid & \mathsf{abortif}(v_i \approx v_j) & \end{array}$$

v_i and v_j represent variables. All non-parameter variables are assumed to be initialized before use. Integer constants n form the set of *ground terms* in the program. The $\mathsf{if}\ (\star)$ construct represents a non-deterministic `if`. The $\mathsf{abortif}(v_i \approx v_j)$ statement represents the safety policy: if such a statement can be reached when $(v_i \approx v_j)$ is true, the program violates the safety property. The relational operator $\approx$ is left abstract but is assumed to be deterministic. While severely restricted, this language is motivated by device driver handling of status codes. The codes are enumerated constants defined in header files and no transforming operations (e.g., plus, bitwise-and) may legally be performed on them. However, the safety policy may insist that status codes be propagated or that certain functions be called (or not called) depending on status code values. In such a device driver example the set of ground terms would be the set of status code values considered by the program.

4.4 Predicate Slicing Algorithm

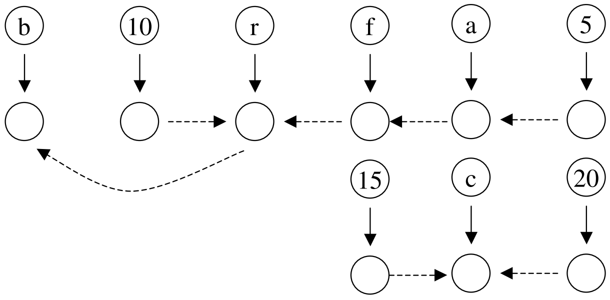
Given a program C in this language, let G be its VFG. Intuitively, the algorithm uses G to track all variables and ground terms that can flow into v_i and v_j for every $\mathsf{abortif}(v_i \approx v_j)$ statement and generates predicates to keep track of that flow of values.

Ideally the algorithm would emit the predicate "a==b" if $\mathsf{Deref}(\mathsf{N}(\mathsf{a})) \rightarrow_F \mathsf{Deref}(\mathsf{N}(\mathsf{b})) \in G$. Unfortunately, such predicates are often meaningless. For example, in Figure 5, although $\mathsf{Deref}(\mathsf{N}(\mathtt{r})) \rightarrow_F^* \mathsf{Deref}(\mathsf{N}(\mathsf{b}))$ will be in G, the predicate "r==b" cannot be interpreted at any point in the program since `r` is a local variable in `select` and `b` is a local variable in `main`. That predicate is thus not suitable as a dataflow fact for the SLAM toolkit. A similar scoping problem occurs between actual arguments and formal parameters.

We will use the set of ground terms (constant values) to bridge the gap between variables in different scopes. For example, given that the predicates

```
int select(int f)
{
  int r;
  if (⋆) r ← f;
  else r ← 10;
  return(r);
}
void main() {
  int a,b,c;
  a ← 5;
  b ← select(a);
  if (*) c ← 15;
  else c ← 20;
  abortif(b > c);
}
```

(a)



(b)

Fig. 5. An example of predicate slicing. An example program is shown in (a) above. The value flow representation for this program produced by one level flow points-to analysis is shown in (b) above.

"`10==r`" and "`10==b`" are true, we can logically conclude "`r==b`" even if we cannot express it as a well-scoped predicate. The predicate slicing algorithm will emit special predicates linking all formal parameters and function return values to an appropriate set of ground terms. These predicates and the transitivity of equality will allow for reasoning that crosses scope boundaries. In the example above, with the predicates "`10==r`", "`5==r`", "`10==b`" and "`5==b`" we have enough dataflow facts to reason about the value of `b` after the call to `select`.

With this intuition in mind, we present the complete algorithm. For each $\mathsf{abortif}(v_i \approx v_j)$:

1. Let $F_i = \mathsf{FlowsInto}(v_i)$.
2. For every $x, y \in F_i$ with $\mathsf{Deref}(x) \rightarrow_F \mathsf{Deref}(y)$, emit the set of predicates { "`a==b`" $\mid$ $\mathtt{a} \in \mathsf{Vars}(x) \wedge \mathtt{b} \in \mathsf{Vars}(y) \wedge \mathsf{ShareScope}(\mathtt{a}, \mathtt{b})$}.[2]
3. Let $T_i = \{\ \mathtt{n} \mid \mathsf{N}(\mathtt{n}) \in F_i\}$ be the set of ground terms `n` that can flow into v_i.
4. For every $x \in F_i$, consider every $\mathtt{a} \in \mathsf{Vars}(x)$. If $\mathsf{return}(\mathtt{a}) \in C$ or `a` is a formal parameter then emit the set of predicates { "`n==a`" $\mid \mathtt{n} \in T_i$}.
5. Repeat steps 1–4 with v_j and F_j.

Since the relation $\approx$ is kept abstract, the algorithm generates predicates that keep track of the flow of ground terms throughout the program. Intuitively, the algorithm walks along chains of flow edges and generates equality predicates for every such edge (step 2). Edges that cross scope boundaries are linked using ground terms (step 4). If this set of predicates is used for dataflow analysis, when the $\mathsf{abortif}(\mathsf{v_i} \approx \mathsf{v_j})$ statement is reached some chain of predicates of the form "v_i==x", "x==y", "y==n" should be true. By transitivity, v_i==n. A similar value can be obtained for v_j. Given ground term values for v_i and v_j, the safety of the $\mathsf{abortif}$ statement can be decided statically.

For this restricted language the above algorithm can be proved correct: the set of generated predicates is always sufficient to statically verify all $\mathsf{abortif}(\mathsf{v_i} \approx \mathsf{v_j})$ statements. The proof is by induction on the length of chains of flow edges leading into v_i in G and makes use of the fact that in this language v_i must always take on a value from T_i (no other values are possible). The algorithm generates a sufficient number of predicates to keep track of the flow of ground terms through the program.

Example 5 Consider the program in Figure 5(a) and the associated value flow representation in 5(b). Let `b` be the variable under consideration. In step 1, $F_{\mathtt{b}}$ will be $\{10, \mathtt{r}, \mathtt{f}, \mathtt{a}, 5\}$. In step 3 we will emit the predicates "`5==a`", "`f==r`", "`10==r`" (predicates like "`a==f`" have no valid scope). In step 3, $T_{\mathtt{b}}$ is $\{10, 5\}$. So in step 4, `f` and `r` qualify so we emit the predicates "`5==f`", "`10==f`", "`5==r`" and "`10==r`". The algorithm then repeats with `c` as the variable under consideration and generates "`15==c`" and "`20==c`". □

The algorithm can be optimized to produce a smaller set of sufficient predicates if the relation $\approx$ is not abstract (e.g., if it is `<` or `==`) or if one of the

[2] $\mathsf{ShareScope}(\mathtt{a}, \mathtt{b})$ is true if either `a` or `b` is a global variable or if `a` and `b` are declared in the same procedure.

arguments is constant. For example, to handle abortif$(x == 7)$, step 3 can be replaced by "Let $T_i = \{7\}$," since at every point we only care if the value that will flow into x is 7 or not. In addition, because value flow representations often merge source variables of different types into the same node, the algorithm can be refined to emit equality predicates over variables of matching types only.

4.5 Experiments

Figure 6 shows the performance of the predicate slicing algorithm on Windows NT device drivers that have been instrumented with input–output request and locking safety properties. In reality, these programs do not fit within the restricted subset of C we presented. For example, the subset does not capture correlations between conditionals: in such cases the algorithm will generate an insufficient set of predicates. As mentioned before, SLAM uses a form of counter-example driven refinement to add in those missing predicates. We compare SLAM with the predicate slicing algorithm against SLAM with all predicates generated by counter-example driven refinement.

"Original Runtime" shows the time for SLAM to either find a bug or prove the driver correct using only counter-example driven refinement. In the case of `iscsiprt`, SLAM does not terminate. "Improved Runtime" shows the execution time when SLAM begins with the predicate slice computed using the previous algorithm. "Generated Predicates" gives the number of distinct predicates in the slice. "Missing Predicates" gives the number of necessary predicates not in the slice that must be found by counter-example driven refinement (e.g., those predicates that correlate conditionals).

Using this technique SLAM was able to scale to some previously unreachable real-world device drivers and performs 2–10 times better on others. The predicate slice provides between all and two-thirds of the necessary predicates. However, since SLAM must generally iterate once to find 3–7 missing predicate and each iteration is exponential in the number of predicates already discovered, the net performance increase is more than linear.

Driver Name	Lines of code	Original Runtime	**Improved Runtime**	Generated Predicates	Max Preds In Scope	Missing Predicates
`apmbatt`	2207	299 s	**22 s**	85	10	0
`pnpmem`	3849	1132 s	**125 s**	143	9	4
`floppy`	7562	1063 s	**600 s**	154	16	33
`iscsiprt`	4543	**	**729 s**	146	10	42

Fig. 6. Performance of predicate slicing. The table above compares the performance of SLAM with predicate slicing against the performance of SLAM with counter-example driven refinement.

5 Related Work

The main idea behind this paper is the use of cheap flow-insensitive value flow information to speed up dataflow. The previous work most similar to our work is that of Ruf [Ruf97] and Zhang et al. [ZRL96]. Ruf showed how a unification-based non-standard type inference procedure such as Steensgaard's pointer analysis could be used to partition the data in a program in such a way that dataflow analysis could be scheduled one partition at a time. This reduces the memory footprint of the dataflow analysis [Ruf97]. Ruf's work can be viewed as producing slices using a points-to graph where all flow edges are replaced by node merging. In general, this will lead to larger slices. Ruf also had no mechanism for identifying constraints and using this information to generate slices of the program.

Zhang et al. used a unification-based pointer analysis to divide the pointer variables in a program into equivalence classes, such that the points-to sets for each equivalence class could be computed separately using a more expensive pointer analysis [ZRL96]. Our work generalizes their result by introducing directional flow, and extends their idea to value flow analysis and clients of value flow analysis in general.

Recent work by Foster et al. [FTA01] uses a unification-based analysis to compute a set of dataflow facts that are then fed to a flow-sensitive type qualifier system. We believe that their work, which appears similar to the typestate slicing used in ESP, could be classified as an instance of our approach.

Rountev et al. developed a framework for combining flow-insensitive global information with flow-sensitive local information [RRL99]. Our work differs from theirs in that our use of flow-insensitive information does not affect the precision of the client analysis, as is the case in their framework. We are merely interesting in improving the efficiency of the subsequent dataflow analysis.

An alternative approach to the one we have presented is to develop demand-driven versions of client dataflow analyses. Previous work on demand-driven dataflow frameworks includes [HRS95] and [DGS95], among others. These frameworks restrict the class of dataflow problems handled, usually to distributive problems, whereas we are interested in value flow problems that are not distributive. It is possible that one could design distributive approximations of value flow analysis that could then be performed from program points of interest, in order to yield more precise slices than those obtained using our method.

Another alternative approach is to use standard program slicing techniques, surveyed in [Tip95]. The drawback of program slicing is that it is based on flow-sensitive reaching definition computation, which is likely to be too expensive to scale to large programs.

The precision of our value flow can be improved through the use of an SSA form [CFR$^+$91].

6 Conclusions

This paper is based on a simple hypothesis: systems that employ heavyweight interprocedural dataflow analyses can benefit greatly by using a inexpensive flow-insensitive value-flow analysis to prune the set of dataflow facts and program statements over which they must operate. We have presented experiments using two different client dataflow analyses to validate our hypothesis. In both cases, we obtain significant gains in performance.

References

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of c programs. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 35, pages 203–213, 2001.

[BR01a] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proceedings of SPIN '01, 8th Annual SPIN Workshop on Model Checking of Software*, May 2001.

[BR01b] Thomas Ball and Sriram K. Rajamani. Bebop: A path-sensitive interprocedural dataflow engine. In *Proceedings of PASTE '01, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, June 2001.

[CFR$^+$91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CRL99] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *Proceedings of POPL '99, 26st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[Das00] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation (PLDI-00)*, 2000.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Symposium on Principles of Programming Languages*, pages 37–48, 1995.

[DLFR01] Manuvir Das, Ben Liblit, Manuel Fahndrich, and Jakob Rehof. Estimating the impact of scalable pointer analysis on optimization. In *Proceedings of the 8th International Symposium on Static Analysis*, 2001.

[DLS02] M. Das, S. Lerner, and M. Seigle. ESP: Path sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN '02 Conference on Programming Language Design and Implementation (PLDI-02)*, June 2002.

[FTA01] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-Sensitive Type Qualifiers. Technical Report UCB//CSD-01-1162, University of California, Berkeley, November 2001.

[HRB90] Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[HRS95] Susan Horwitz, Thomas Reps, and Mooly Sagiv. Demand-driven interprocedural dataflow analysis. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering Notes*, volume 20, 1995.

[HT01] Nevin Heintze and O. Tardeau. Ultra fast aliasing analysis using CLA: a million lines in a second. In *Proceedings of the ACM SIGPLAN '01 Conference on Programming Language Design and Implementation (PLDI-01)*, 2001.

[OJ97] R. O'Callahan and D. Jackson. Lackwit: A program understanding tool based on type inference. In *Proceedings of the 1997 International Conference on Software Engineering*, 1997.

[RRL99] Atanas Rountev, Barbara G. Ryder, and William Landi. Data-flow analysis of program fragments. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 235–252, 1999.

[Ruf97] E. Ruf. Partitioning dataflow analyses using types. In *Conference Record of the Twenty-Fourth ACM Symposium on Principles of Programming Languages*, 1997.

[SY86] R. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, 12(1):157–171, 1986.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[WL95] R. Wilson and Monica Lam. Efficient context-sensitive pointer analysis for C progams. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI-95)*, 1995.

[ZRL96] S. Zhang, B. Ryder, and W. Landi. Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses. In *Fourth Symposium on the Foundations of Software Engineering (FSE4)*, 1996.