RC24104 (W0701-015) January 3, 2007
Computer Science

IBM Research Report

A New Abstraction for Summary-Based Pointer Analysis

Marcio Buss[1], Daniel Brand[2], Vugranam Sreedhar[3], Stephen A. Edwards[1]

[1]Columbia University
New York, NY

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

[3]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

LIMITED DISTRIBUTION NOTICE: This report has been submitted for publication outside of IBM and will probably be copyrighted if accepted for publication. It has been issued as a Research Report for early dissemination of its contents. In view of the transfer of copyright to the outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or legally obtained copies of the article (e.g., payment of royalties). Copies may be requested from IBM T. J. Watson Research Center , P. O. Box 218, Yorktown Heights, NY 10598 USA (email: reports@us.ibm.com). Some reports are available on the internet at http://domino.watson.ibm.com/library/CyberDig.nsf/home .

A New Abstraction for Summary-Based Pointer Analysis

Marcio Buss
Columbia University
New York, NY
marcio@cs.columbia.edu

Daniel Brand
IBM T.J. Watson
Yorktown Heights, NY
danbrand@us.ibm.com

Vugranam Sreedhar
IBM T.J. Watson
Hawthorne, NY
vugranam@us.ibm.com

Stephen A. Edwards
Columbia University
New York, NY
sedwards@cs.columbia.edu

ABSTRACT

We propose a new abstraction for pointer analysis based on the principle of matching pointer dereferences or "memory fetches" with pointer assignments. Our abstraction, the Assign-Fetch Graph (AFG), has several advantages over traditional points-to graphs. It leads to concise procedure summaries for pointer analysis and similar computations; each procedure's summary information can be used effectively in arbitrary calling contexts. Different analysis variations can be formulated on the AFG— we present two possible variations. One is based on a standard flow-insensitive view of the program; the other takes some statement ordering into account and produces results that are both more precise *and* more quickly computed. Our abstraction also facilitates incremental summary-based pointer analysis, which is not generally possible with conventional points-to graphs.

We show how the AFG simplifies certain steps in updating pointer graphs and computing procedure summaries. For efficiency and compactness, we build a single summary for each procedure under the optimistic assumption that distinct pointers from the environment are not aliased. We restore soundness when the summary is used in a context with aliases. We present some results of the practical impact of our technique.

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: General; D.2.0 [**Software Engineering**]: General

General Terms

Algorithms, Languages, Performance

Keywords

Points-to analysis, summary-based analysis, incremental analysis

1. INTRODUCTION

Pointer analysis is a necessary step in most language processing tools, including bug finding tools, program optimizers, program understanding and refactoring tools. Pointer analysis consists of computing points-to information—given two program variables, p and q, we say that p points-to q if p can contain the address of q. For heap allocated objects, we say that p can point-to an object O if p can contain the address of O. Although the address of a heap-allocated object is generally not known at compile-time, static analyses assign analysis-time addresses to such objects.

Most previous pointer analysis techniques focus on interpreting program statements that affect pointers by constructing a *points-to graph* [7], where nodes correspond to pointer variables and objects, and edges correspond to the points-to relation. In this paper, we describe a new representation for the behavior of a function. This flexible representation lends itself to a variety of pointer analysis algorithms, including a classical flow-insensitive analysis and a new "flow-aware" analysis that is both faster and more precise.

In our representation, called assign-fetch graph (AFG), nodes represent locations and values, and edges represent read and write operations. Pointer analysis amounts to matching pointer dereferences ("fetch edges") with pointer assignments ("assign edges").

After quickly introducing our techniques through an example, we describe our assign-fetch graph (Section 2) and discuss the general approach of performing pointer analysis on it (Section 3). From there, we describe two particular instances of pointer analysis on the AFG: classical flow-insensitive (Section 4) and our new flow-aware technique that is both faster and more accurate (Section 5). Finally, we present experimental results showing our technique working well on large programs (Section 6).

1.1 An Example

Figure 1 is a simple example illustrating our pointer analysis techniques. They are centered around the assign-fetch graph (AFG), Figure 1(b), an abstract procedure representation that captures assignments and memory dereferences. Our approach has three main advantages that lead to faster, more precise pointer analysis: the AFG is simpler to construct than a points-to graph; unlike a points-to graph, whose structure depends on a procedure's context (i.e., aliases present when the procedure is called), the AFG is context-agnostic and therefore can be reused in any context; and finally our AFG makes it easy to vary the precision (and hence cost) of the analysis by using different analysis algorithms on the same underlying representation. Figure 1 illustrates this last point: we perform two analyses from the same AFG.

An AFG is a directed graph whose nodes represent values (including addresses) and whose edges represent read and write operations. We call these *fetch* and *assign* edges, respectively. Consider the first statement in Figure 1(a): `*z = &x;`. This fetches the contents of `z` and stores the address of `x`, a global variable, as one of the locations to which `z` may point. In the AFG of Figure 1(b), we represent variable `z` with a *location node* labeled `z`, the fetch of

.

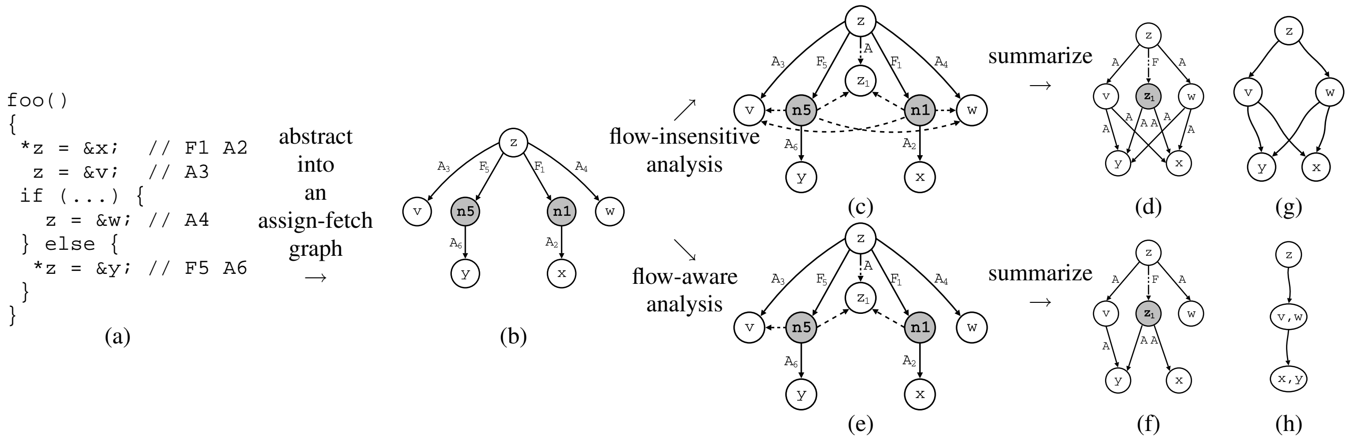


Figure 1: An illustration of our pointer-analysis technique. A procedure (a) is first abstracted as an assign-fetch graph (b), whose nodes represent addresses and values and whose edges represent memory operations. An assign-fetch graph can be analyzed in at least two ways: a classical flow-insensitive analysis (c), where potential aliases are calculated ignoring statement ordering to produce a summary (d); and a new flow-aware analysis, which considers statement execution order (e) to produce a more accurate summary (f). Contrast these summaries with the pointer analysis solutions of (g) Andersen and (h) Steensgaard.

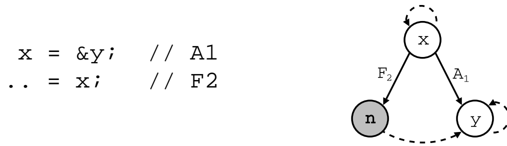


Figure 2: The simplest case: `x` is assigned in A_1 and fetched in F_2; an *alias edge* indicate that n can be an alias for `y`. Self-loops indicate trivial aliasing of location nodes.

z with the fetch edge labeled F_1, the fetched value of z with a *fetch node* labeled n_1 (we shade each fetch node gray to indicate we do not know its values when we construct the graph), the address of x with the location node labeled x, and the assignment to *z with the assign edge labeled A_2. We construct the rest of Figure 1(b) using similar reasoning.

Fetch nodes correspond to unknown values that need to be *resolved* during the analysis. The resolution is done through *alias edges* that connect fetch nodes to location nodes. The simplest example is shown in Figure 2—an alias edge (dashed line) is created from n to y to indicate that fetching x (F_2) after it has been assigned the address of y (A_1) should be an alias for y. Self-loops indicate trivial aliasing of location nodes; we omit them in all other figures.

The bulk of the analysis is to determine aliases between fetch nodes and location nodes. This produces the *resolved* AFG; depending on the precision of the analysis, different resolved AFGs can be generated from the same *initial* AFG, such as Figure 1(c) and (e).

1.2 Flow-Insensitive Analysis

We first show how to perform classical flow-insensitive analysis (Figure 1(c) and (d)) on the AFG. Such an analysis does not consider the order in which assignments and fetches will execute, which makes for a fast analysis at the cost of some precision. First, we add to the AFG of Figure 1(b) the *initial value node* z_1 and the assign edge labeled A in Figure 1(c) to model the environment's (assumed) initialization of the global variable z. Next, we add the alias edges (n_1, v), (n_1, z_1), and (n_1, w) by repeatedly applying the rule in Figure 2. For example, alias edge (n_1, w) in Figure 1(c) is added due to edges F_1 and A_4, respectively fetching from and assigning to location z. The AFG is designed to make such relationships easy to see. In total, there are three assignments to z (edges A, A_3, and A_4), so there are three potential aliases for fetches F_1 and F_5: nodes v, z_1, and w.

The final step is to produce the procedure *summary* AFG. We delete any information that would be invisible to a potential caller, e.g., fetch nodes n_1 and n_5 in Figure 1(c), will be deleted. Before deleting these nodes, the information they carry is "transfered" to location nodes in the graph, which are visible to callers. Figure 1(d) shows the result. The basic idea is the following: if an assignment is made to some fetch node n, and n can be an alias for a location node n', then the assignment is equivalent to an assignment to n'. For example, in Figure 1(c), n_1 is assigned the address of x and can be an alias for z_1, v, and w. Assign edges are therefore added from z_1, v, and w to x. Similar reasoning connects edges from z_1, v, and w to y. Finally, the fetch nodes n_1 and n_5 are removed and node z_1 is demoted to a fetch node to represent the fact that z has been dereferenced inside foo; this produces the final flow-insensitive summary in Figure 1(d).

The right side of Figure 1 also shows the points-to graphs for the code example generated by two well-known techniques. Figure 1(h) shows the result of Steensgaard [17], and Figure 1(g) shows the result of Andersen [1]. The latter is equivalent to our flow-insensitive result (Figure 1(d), although it would not usually include something like an initial value node z_1). Steensgaard's merges nodes pointed-to by the same pointer into equivalence classes, as seen for {v,w} and {x,y} in Figure 1(h); it is therefore less accurate although generates smaller graphs.

In Section 4, we describe flow-insensitive analysis using the AFG in detail.

1.3 Flow-Aware Analysis

Flow-insensitive analysis ignores the fact that assignments and fetches in a program happen in sequence: later assignments cannot be "seen" by earlier fetches. This leads to pessimistic results: flow-insensitive analysis often arrives at many more aliases than actually possible. For example, in Figure 1(a), the first fetch of z (F_1) can only "see" the initial value of z, since it is the first statement in the

procedure. Thus, the only possible alias edge from n_1 is to z_1, as shown in Figure 1(e). In Figure 1(c), flow-insensitive analysis also introduced edges from n_1 to w and v, which can never happen.

In addition, the dereference *z at statement *z = &y (represented by F_5) can only access two locations—the initial value of z, and v; we know assignment z = &w (represented by A_4) cannot be seen by fetch F_5 because they are in separate branches of the conditional. Hence the alias edges from n_5 do not include w in Figure 1(e).

The result, after performing the same graph clean-up as flow-insensitive analysis, is the more precise summary in Figure 1(f). This has three fewer assign edges than the flow-insensitive result in Figure 1(d).

Our flow-aware analysis takes advantage of the fact that each operation has a distinct edge in the AFG and we decompose loops into tail-recursive procedure calls. Therefore all procedures are acyclic and their statements can be considered in order in the intraprocedural phase of the analysis. Recursive procedures (in general, strongly-connected components in the call graph, i.e., mutually recursive procedures) are handled by iterating the intraprocedural phase until convergence.

The flow-aware analysis is faster than flow-insensitive analysis as well as being more precise. We present these results in Section 6 after describing the analysis in detail in Section 5.

2. THE ASSIGN-FETCH GRAPH

Our pointer analysis is interprocedural. It propagates procedure summaries through the call graph of the program. More precisely, the input to our analysis analysis is a program we assume has been partitioned into procedures. Each procedure is treated as consisting of three types of operations: assignments, fetches, and procedure calls. We ignore pointer arithmetic by making the common simplification of treating all elements of an array as one location. In this section, we will merge fields of structures and classes, so expressions such as p->next is treated as *p. Procedure return is modeled by creating a special location *ret*; heap locations are modeled through a naming scheme similar to that of Choi et al. [5].

To analyze procedures, we represent them initially using an asssign-fetch graph. An AFG is an abstraction of a procedure in the form of a directed graph. Its nodes represent addresses and values and its edges represent memory operations, initial values, and aliases between nodes. A *location node* is simplest: it represents the address of global or stacked variables, or heap-resident data. In Figure 1(b), v, w, x, y, and z are each location nodes that represent the address of distinict global variables.

Fetch nodes, shaded gray, represent the results of reading a value from memory. Since in general the result of such a fetch depends on the behavior of the program at runtime, we think of the "value" of a fetch node as being unknown. Resolving the possible values returned by each fetch is the bulk of the analysis. In Figure 1(b), n_1 and n_5 are fetch nodes. By definition, each fetch node has exactly one incoming fetch edge (defined below).

Interface nodes are location nodes that correspond to information that crosses the caller-callee interface of a procedure. They are the global variables, parameters values, and heap locations allocated inside the callee. Nodes for local (automatic) variables are therefore never interface nodes. In Figure 1, nodes v, w, x, y, and z are all interface nodes, assuming they all represent global variables.

Initial value nodes are placeholders for the values of global variables and parameters supplied by the environment and represent chains of dereferences starting from interface nodes. We add them in the process of analyzing the initial assign-fetch graph. At that point, we think of them as location nodes (e.g., z_1 in Figure 1(c)).

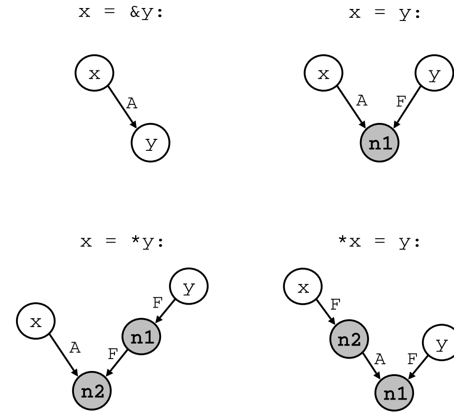


Figure 3: The conventional canonical statements and their respective Assign-Fetch Graph representations.

After we summarize the results of an analysis, we "demote" them to fetch nodes (e.g., z_1 in Figure 1(d)). Note that we only add those initial value nodes that are actually needed by the procedure, e.g., even though the variable x appears in Figure 1(a), we do not construct an initial value node for it because the procedure never reads its value.

There are two type of *operation edges*. A *fetch edge*, which we label with an "F," represents a memory read operation. Its source node represents the address being read and its target is a fetch node that represents whatever value is stored at that location. An *assign edge*, labeled "A," represents a memory write operation. Its source node represents the target address and its destination represents the value being written.

Figure 2 shows AFG fragments for the four canonical statements used by traditional pointer analyses. Consider the assignment x=&y. We represent both the lvalue x and the rvalue &y as location nodes and connect them with an assign edge indicating that x now points to the memory location for y. Note that this assignment does not read or change the contents of y. By contrast, since the statement x = y does read the contents of y, we introduce the fetch node n_1 to represent these contents, indicate with a fetch edge that y's contents are read, and indicate they are written to x with an assign edge. The other two statements in the bottom of Figure 2 follow a similar idea.

Alias edges, which we draw as dashed lines, represent aliasing information among nodes and can be read "can be an alias for." Each location node has an implicit self-loop alias edge that represents the fact that the address of each variable is unique. Proper alias edges always leave fetch nodes and terminate at location nodes that the fetch node can alias. For example, the alias edge from n_5 to v in Figure 1(e) means the F_5 fetch of z can return the address of v.

Finally, *initial value edges* represent environment initialization. Like initial value nodes, we think of them one way (as assign edges) when we are analyzing the AFG and another way (as fetch edges) after we summarize the graph. We draw them with dashed/dotted lines, e.g., from z to z_1 in Figure 1(c), (d), (e), and (f).

Using the present terminology, the final *summary* AFG for a procedure only has interface nodes and initial value nodes. Each interface node may have a chain of initial value nodes represented by a chain of fetch edges; assign edges may connect any two pair of nodes in the summary (Figure 5).

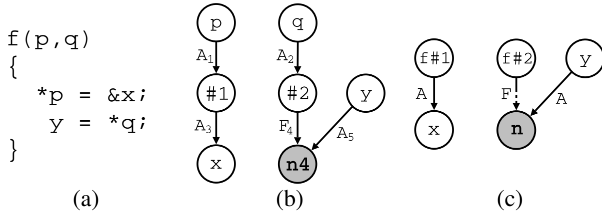


Figure 4: (a) A procedure with arguments, (b) its AFG, and (c) its summary.

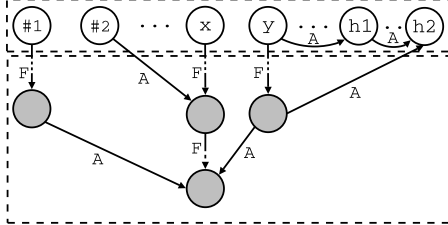


Figure 5: A generic procedure's summary.

2.1 Procedure Parameters

We treat procedure parameters a little differently than global variables. While both are assumed to come from the environment, we always introduce initial value nodes for procedure parameters since they are always assigned by a caller when a procedure is called. During execution, those initial values are assigned to formal parameters, which are local variables. Since local variables are popped off the stack as the procedure returns, the location nodes representing formal parameters are removed in the summarization process.

Figure 4 illustrates the summarization process for a simple procedure. Location nodes are created for the formal parameters `p` and `q` and assigned initial value nodes `#1` and `#2`, which represent the values passed by the caller through these two arguments. This plus adding nodes and edges for the two statements in the procedure gives the AFG for function `f` in Figure 4(b).

Since formal parameters are assumed initialized through "#*i*" nodes, the AFG representation for `*p = &x` in Figure 4(b) does not include a fetch edge; `*p` yields directly its initial value node, #1, originally assigned to `p` through assign edge A_1.

To summarize the procedure, whose analysis did not find any aliases, we remove the nodes for the formal parameters `p` and `q` and rename the initial value nodes to indicate the name of the procedure. Also, fetch edge F_4 in Figure 4(b) generates an initial value for node #2, which in Figure 4(c) is labeled n. Figure 4(c) shows the final summary for procedure `f`. This summary is built assuming that parameters `p` and `q` point-to different locations at function entry (i.e., #1 and #2, which are treated as location nodes).

Figure 5 shows the general structure of a procedure summary, which consists of two layers. The top layer contains interface nodes, which include nodes #1, #2, etc., for the values of procedure parameters. Also in this layer are nodes for global variables (`x`, `y`, etc.) and heap locations h_1, h_2, etc.

The bottom layer represents the initial values for the interface nodes, which must all be fetch nodes. These are the only fetch nodes in the summary since all others are removed as part of the summary process; those that remain were demoted from (initial value) location nodes, as described earlier. Since each interface node is assumed to have at most one initial value, each node in the summary has at most one outgoing fetch edge.

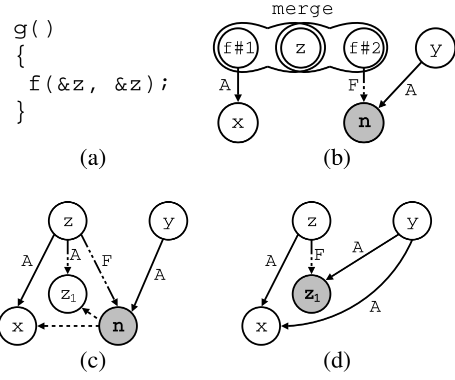


Figure 6: (a) A procedure `g`, which calls the procedure `f` from Figure 4, (b) its initial AFG, (c) after resolving, and (d) its final summary.

2.2 Procedure Calls and Summaries

To construct an AFG for a procedure *P*, we add nodes and edges for each statement that reads and writes memory and instantiate (make a copy of) the summary for each procedure *Q* called by *P*. Instantiating the summary amounts to merging common global variables as well as merging actual parameters to formal parameters. The usual advantage of procedure summaries is that we do not have to analyze the body of a procedure repeatedly—the summary captures all we need to know about its behavior.

Figure 6 demonstrates this with a simple procedure `g` that calls the procedure `f` from Figure 4. Starting from the summary of `f` (Figure 4(c)), we bind the actual parameters to formal parameters by merging the formal parameter initial value nodes (`f#1` and `f#2`) with their corresponding actual arguments. The procedure `g` passes the address of the global variable `z` to both parameters. In Figure 6(b), nodes `f#1` and `f#2` are marked to be merged into node `z`. The merging is complete in Figure 6(c).

Global variables are handled similarly: their nodes in the callee are merged with matching nodes in the caller. This is vacuous for the example of Figure 6 since `g` does nothing with the global variables `x` and `y`.

Note that node merging is the only operation we need for argument/parameter binding, even in cases where the actual parameter is an arbitrary expression. A function call such as `f(z, z)`, for instance, would represent its actual arguments as two fetch edges (`z`, n_1) and (`z`, n_2), respectively; n_1 would be merged with `f#1`, and n_2 merged with `f#2`. Later, when determining aliases between fetch nodes and location nodes, the aliasing introduced at the function call `f(z, z)` is directly recovered by the analysis.

After instantiating the summaries for the callees, computing a summary for the calling procedure proceeds normally. In Figure 6(c), we have applied flow-insensitive analysis: added an initial value node z_1 and initial assign edge for global variable `z`, and alias edges from n to `x` and z_1 since both are assigned to `z`. Finally, summarizing `g` produces the graph in Figure 6(d). The fetch node n has been removed and the effects of its aliases now manifest themselves as the assign edges from `y` to z_1 and `x`.

This example illustrates a novel and useful aspect of our procedure summaries: they are agnostic about parameter aliasing and can be reused in contexts with arbitrary alias relationships. In this example, our initial summary for the `f` procedure did not explicitly consider the case that its two parameters were aliased. However the summary for `g` correctly ascertains that `y` can point to `x`, which is only possible if the two arguments are aliases at the call to `f`.

3. POINTER ANALYSIS USING THE AFG

We perform pointer analysis on a procedure through a series of graph transformations on the AFG for the procedure. To construct this *initial* AFG, we create edges and nodes for each pointer-related statement in the procedure and replace procedure calls with a summary for the called procedure. The *resolved* and *summary* AFGs (the intermediate steps) are obtained by performing two transformations on the initial AFG. First, we augment the initial AFG with alias edges from each fetch node n to the set of location nodes to which n can be an alias; initial values are lazily constructed by observing which interface nodes have been fetched. The resulting graph is the resolved AFG for the procedure. Then to form the summary AFG, we first remove all fetch nodes and subsequently demote to fetch nodes all initial value nodes that represent environment initialization.

Different techniques for adding alias information to an AFG lead to different analysis variations with different precision levels; constructing the AFG itself and building the summary after computing alises are straightforward, mechanical procedures.

Below, we discuss the general problem of adding aliases to an AFG. In Section 4, we describe a specific approach to adding aliases that gives classical flow-insensitive analysis. In Section 5, we describe a more precise analysis we have dubbed "flow-aware analysis" that takes some control flow into account to produce a more precise analysis that is also more efficient.

In what follows, x,y,z,... denote *location* nodes (i.e., interface nodes or initial value nodes before demotion); $n_1, n_2, n_3, \ldots$ are *fetch* nodes; and $\alpha, \beta, \gamma, \ldots$ are arbitrary nodes.

3.1 Determining Aliases

In our framework, the main step in pointer analysis is to calculate the set of locations each node in the graph could alias with; this set is denoted $al(\alpha)$. In general $al(n)$ for a fetch node n is not reflexive, i.e., a fetch node n can simultaneously be an alias for multiple locations x, y, z, etc., much as *p is an alias for r, s and $\mathtt{p}_1$, the initial value of p, in the following code fragment:

```
void foo() { p=&r; ... p=&s; ... *p=&k; }
```

However, x can never be an alias for y or vice-versa since they refer to different variables, and therefore are guaranteed to be placed at different memory locations.

An alias edge from a node α to a node x indicates $\mathtt{x} \in al(\alpha)$; the target x is always a location node. The common assumption of no aliasing among location nodes implies $al(\mathtt{x}) = \{\mathtt{x}\}$.

A fetch node n can only alias with the target of an assign edge. This aliasing occurs if in some execution the fetch returns the value of the assignment, i.e., if the assignment is the last to change the location before it is fetched. To indicate such an aliasing can occur, we write $affects(\sigma_A, \sigma_F)$, where σ_A is an assign edge and σ_F a fetch edge (assignment σ_A "affects" fetch σ_F).

In general, the relation $affects(\sigma_A, \sigma_F)$ is not effectively computable and therefore any pointer analysis uses an approximation. The more accurate the approximation, the more accurate the analysis. For a sound analysis, it should be an over-approximation, i.e., the approximation should always be true when the relation is, but not necessarily vice versa.

If the relation $affects(\sigma_A, \sigma_F)$ (or its approximation) is available, determining aliases can be described by the following simple rule.

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad affects(\sigma_A, \sigma_F)}{al(\beta) \subseteq al(n)} \quad [\text{ALIAS}]$$

where $\gamma \xrightarrow{A} \beta$ indicates an assign edge from γ to β and $\alpha \xrightarrow{F} n$ indicates a fetch edge from α to n. This states that whenever there is an assign edge affecting a fetch edge then the fetch node n aliases with everything the assigned value β does. The solution to pointer analysis is then the minimal resolved AFG that satisfies this rule. Different approximations of *affects* result in different minimal resolved AFGs for the same initial AFG, i.e., different pointer analysis solutions.

In the next two sections we present two ways to approximate the *affects* relation with different precisions and costs. The less precise analysis we describe in Section 4 is equivalent to Andersen's flow-insensitive analysis [1]. Our more precise solution, which we describe in Section 5, is actually more efficient. In it, the control-flow graph is "linearized," i.e., conditions are removed by imposing an artificial total order in the procedure's statements. We call such an approximation *total order* flow-aware analysis.

4. FLOW-INSENSITIVE ANALYSIS

Section 3 described pointer analysis of arbitrary precision that depends on the uncomputable *affects* relation. In this section, we describe one possible approximation to *affects* that gives Andersen-style flow insensitive analysis. In Section 5, we describe a more precise approximation.

Define the predicate *aliases* as follows.

$$aliases(\alpha, \gamma) \Leftrightarrow \alpha = \gamma \vee al(\alpha) \cap al(\gamma) \neq \emptyset.$$

This says that nodes α and γ are aliases for the same thing if they are the same node (the trivial case), or if they are aliases for at least one common location node, i.e., $al(\alpha) \cap al(\gamma) \neq \emptyset$ or equivalently, there is some node x present in both $al(\alpha)$ and $al(\gamma)$.

The $aliases(\alpha, \gamma)$ relationship is a flow-insensitive approximation to the exact $affects(\sigma_A, \sigma_F)$ for edges $\sigma_A : \gamma \xrightarrow{A} \beta$ and $\sigma_F : \alpha \xrightarrow{F} n$. Hence the [ALIAS] rule can be approximated as follows.

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha, \gamma)}{al(\beta) \subseteq al(n)} \quad [\text{FI-ALIAS}]$$

This rule is recursive in that the premise refers to the relation *aliases*, which is defined in terms of *al*, which is to be computed. Therefore finding the minimal resolved AFG requires a fixed point computation. Below, we describe our implementation of this: a worklist algorithm that iterates to convergence.

Figure 7 illustrates the [FI-ALIAS] rule graphically. Existing alias relationships are shown with thin dashed lines and the rule generates the edges in bold. Figure 7(d) is the most general case; Figure 7(a) is the special case when α is an arbitrary node and $\beta = \mathtt{y}$; Figure 7(b) is the special case when $\alpha = n_0$, $\gamma = \mathtt{x}$ and $\beta = \mathtt{y}$; and Figure 7(c) is the special case when α is an arbitrary node and β is a fetch node n_1; assume $al(n_1) = \mathtt{y}$ for this figure.

4.1 Implementation

Here, we illustrate our worklist-based implementation of the flow-insensitive inference rule through the example in Figure 8. Our algorithm maintains a set of pairs $\langle \sigma_A, \sigma_F \rangle$ that are known to satisfy the premise of the [FI-ALIAS] rule. Technically, we maintain a worklist of pairs of the form $\langle \sigma_F, S_{\sigma_F} \rangle$ where S_{σ_F} is a *set* of assignments. Intuitively, instead of considering pairs $\langle \sigma_A, \sigma_F \rangle$ separately, we group together all assignments that should resolve to σ_F in the current iteration of the algorithm. We will refer to pairs $\langle \sigma_A, \sigma_F \rangle$ and $\langle \sigma_F, S_{\sigma_F} \rangle$ interchangeably; the meaning should be clear from context.

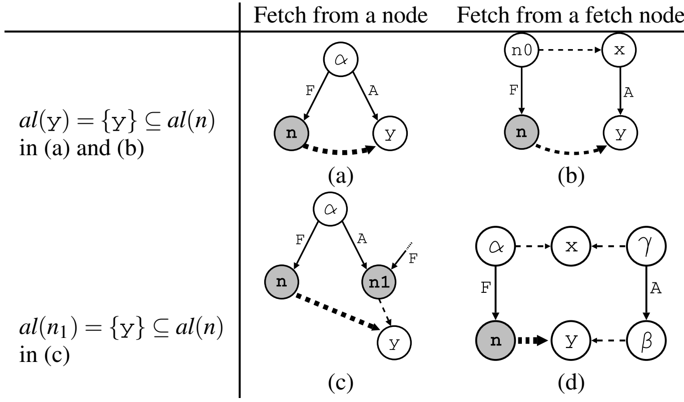
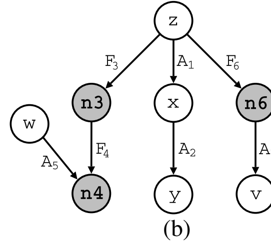


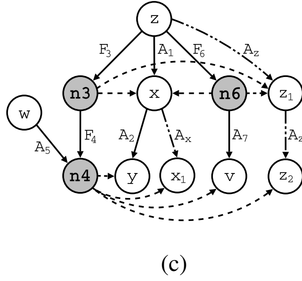
Figure 7: Four cases of applying the [FI-ALIAS] rule. A node is fetched in (a) and (c) ($\alpha = \gamma$); the result of a fetch is itself fetched in (b); (d) is the general case.

```
bar()
{
  z = &x; // A1
  x = &y; // A2
  w = *z; // F3 F4 A5
 *z = &v; // F6 A7
}
```

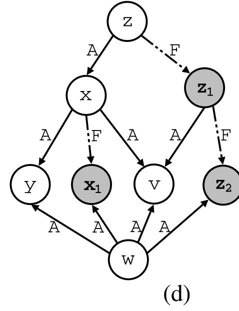
(a)



(b)



(c)



(d)

Figure 8: (a) A code fragment and its AFG (b), after resolution (c), and after summarization (d).

We initialize the worklist with all pairs of edges $\langle \sigma_A, \sigma_F \rangle$ that share their source nodes. The algorithm is incremental in the sense that addition of new aliases triggers additions to the worklist. We will refer to determining aliases as the *resolution phase*.

Consider the code for `bar` in Figure 8(a). Its initial AFG is shown in Figure 8(b). The worklist starts containing exactly two elements: $\langle F_3, \{A_1, A_\mathtt{z}\}\rangle$ and $\langle F_6, \{A_1, A_\mathtt{z}\}\rangle$. The assign edge labeled $A_\mathtt{z}$ is the initial value edge for interface node `z`, lazily created by the algorithm because `z` is fetched by either of F_3 or F_6.

For the initial AFG of `bar`, the algorithm produces the resolved AFG in Figure 8(c) as follows. The first element, $\langle F_3, \{A_1, A_\mathtt{z}\}\rangle$, is taken from the worklist, and alias edges (n_3,x) and (n_3,$\mathtt{z}_1$) are added. Doing so requires three actions: first, we add the pair $\langle F_4, \{A_2\}\rangle$ to the worklist because n_3 is now an alias for `x`, and therefore F_4 should resolve to A_2—an application of the rule in Figure 7(b). Second, because $al(n_3) = \{\mathtt{x}, \mathtt{z}_1\}$ at this point, fetching n_3 through edge F_4 means indirectly fetching both `x` and $\mathtt{z}_1$. Our algorithm therefore creates two initial values: $\mathtt{x}_1$ and $\mathtt{z}_2$, and adds the assign edges ($\mathtt{x}$, $\mathtt{x}_1$) and ($\mathtt{z}_1$, $\mathtt{z}_2$). In Figure 8(c) these edges are labeled $A_\mathtt{x}$ and $A_{\mathtt{z}_1}$. Third, the worklist is augumented with $\langle F_4, \{A_\mathtt{x}, A_{\mathtt{z}_1}\}\rangle$ for the same reason it was augumented with $\langle F_4, \{A_2\}\rangle$: applications of the rule in Figure 7(b). Since we group together common elements based on σ_F, this means the new worklist element is in fact $\langle F_4, \{A_2, A_\mathtt{x}, A_{\mathtt{z}_1}\}\rangle$

Assume the worklist is augumented at its head; $\langle F_4, \{A_2, A_\mathtt{x}, A_{\mathtt{z}_1}\}\rangle$ is hence the next to be processed. This adds alias edges (n_4, `y`), (n_4, $\mathtt{x}_1$) and (n_4, $\mathtt{z}_2$).

Next, $\langle F_6, \{A_1, A_\mathtt{z}\}\rangle$ is processed and alias edges (n_6, `x`) and (n_6, $\mathtt{z}_1$) are added. Doing so requires us to return F_4 to the worklist, this time paired up with A_7—i.e, $\langle F_4, \{A_7\}\rangle$, an instance of Figure 7(d) with $\alpha = n_3$, $\gamma = n_6$, $n = n_4$, and $\beta = \mathtt{y}$, which is replaced by `v`. This happens because $aliases(n_3, n_6)$ is true, i.e., $al(n_3) \cap al(n_6) \neq \emptyset$. Specifically, $\mathtt{x} \in al(n_3), al(n_6)$ and $\mathtt{z}_1 \in al(n_3), al(n_6)$.

Processing $\langle F_4, \{A_7\}\rangle$ adds the alias edge (n_4, `v`), completing the resolution phase. Because $al(n_3) \cap al(n_6) \neq \emptyset$, both n_3 and n_6 are two aliases for at least one common location node `x`. Therefore, any assignments to or dereferences of n_3 and n_6 indirectly afect each other.

We returned F_4 to the worklist because we need to arrive at a fixed point for the [FI-ALIAS] rule. In the graph of Figure 8(b), note that a solution with $al(n_3) = al(n_4) = al(n_6) = \emptyset$ does not satisfy the rule. A valid solution must include an alias edge, e.g., from n_3 to `x` for two reasons: there is a fetch edge ($\mathtt{z}$, n_3) and an assign edge ($\mathtt{z}, \mathtt{x}$), and $aliases(\mathtt{z}, \mathtt{z})$ is true. A similar argument demands the other alias edges in Figure 8(c).

We produce the summary AFG in Figure 8(d) by adding assignment edges around each fetch node based on its aliases, deleting all fetch nodes, and demoting assignment edges $(\mathtt{x}, \mathtt{x}_1)$, $(\mathtt{z}, \mathtt{z}_1)$ and $(\mathtt{z}_1, \mathtt{z}_2)$ to fetch edges, turning nodes $\mathtt{x}_1$, $\mathtt{z}_1$ and $\mathtt{z}_2$ into fetch nodes.

Our algorithm adds elements to the worklist when new aliases are discovered by the analysis. This means newly-added alias edges will "trigger" the inference rule. Figure 9 shows the most general case: a new alias edge $(n_i, \mathtt{x})$ is highlighted—due to this new alias, the worklist is augumented with $\langle F_1, \{A_5, A_6\}\rangle$, $\langle F_2, \{A_4\}\rangle$ and $\langle F_3, \{A_4\}\rangle$. These three elements are the new facts that need to be considered because of the new alias between $(n_i, \mathtt{x})$. For this incremental procedure, node mergings occuring at summary instantiation can be seen as preceded by an alias edge from the callee node to the corresponding caller node.

Let $L(\mathtt{x}) = \{\varphi | \mathtt{x} \in al(\varphi)\}$ be the inverse of the *al* function. Since the range of *al* is sets of location nodes, L is only defined for location nodes. In Figure 9, $L(\mathtt{x}) = \{n_i, \mathtt{x}, \gamma\}$ (γ not necessarily distinct from `x`). When we add a new alias $(n_i, \mathtt{x})$, we add to the worklist fetch edges of the form (φ, n_j) where $\varphi \in L(\mathtt{x})$. In Figure 9, such set of edges corresponds to F_1, F_2 and F_3. The sets of assignments that pair up with each fetch edge (φ, n_j) are defined as follows. If $\varphi = n_i$, all $\eta \xrightarrow{A} \mu$ s.t. $\eta \in L(\mathtt{x}) \backslash n_i$. In Figure 9, this means $\langle F_1, \{A_5, A_6\}\rangle$. If $\varphi \neq n_i$, all $n_i \xrightarrow{A} \delta$. This means the other two worklist elements. Note that if no $n_i \xrightarrow{A} \delta$ exists, then both F_2 and F_3 in Figure 9 are not affected by the new alias $(n_i, \mathtt{x})$.

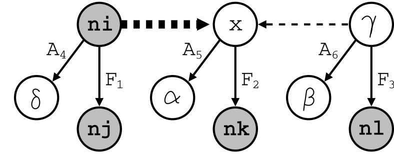


Figure 9: With the addition of new alias (n_i, x), the worklist is augmented with $\langle F_1, \{A_5, A_6\}\rangle$, $\langle F_2, \{A_4\}\rangle$ and $\langle F_3, \{A_4\}\rangle$.

In the example of Figure 8, the new alias discovered between n_6 and x while processing $\langle F_6, \{A_1, A_z\}\rangle$ augments the worklist with $\langle F_4, \{A_7\}\rangle$ (an instance of Figure 9 with $n_i = n_6$, x=x, and $\gamma = n_3$).

Note that F_4 corresponds to the second dereference of variable z in the statement `w=*z`. In traditional incremental techniques, the addition of a new alias would trigger the (re)analysis of at least one entire statement [14, 18]. In our finer-grain algorithm, it only triggers the local (re)analysis of *sub-expressions* within statements.

5. FLOW-AWARE ANALYSIS

Our AFG has the advantage of enabling a variety of analyses. In this section, we describe a more accurate analysis that uses a more precise, but not any harder to compute, approximation of the *affects* relation defined in Section 3.

Flow-insensitive analysis, by definition, ignores the order of statements in the program, a pessimistic simplification that leads to spurious aliases. A "flow-aware" analysis considers the order of statements in the program and therefore greatly improves the precision of the analysis.

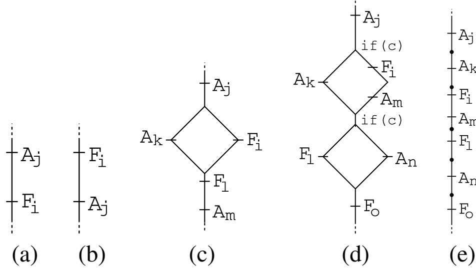


Figure 10: Motivation for flow-aware analysis. (a) An assignment before a fetch can affect the fetch, but (b) an assignment after a fetch cannot. The presence of conditionals (c, d) further constrain which assignments are visible to each fetch. Our total-order flow-aware analysis approximates the execution order of (d) with the total order of (e).

Figure 10 illustrates the motivation for flow-aware analysis. Each drawing represents a fragment of the control-flow graph of a procedure. Figure 10(a) illustrates the most basic case: the assignment A_j runs before the fetch F_i, so A_j can affect F_i, i.e., *affects*(A_j, F_i) holds. However, a fetch that runs before an assignment, as in Figure 10(b), cannot be affected by the assignment. A flow-insensitive analysis treats these two cases identically; our total-order flow-aware analysis does not build an alias in the second case.

Conditionals further complicate the situation. In Figure 10(c), fetch F_i should resolve to assign A_j, since the latter occurs strictly before the former. On the other hand, *affects*(A_k, F_i) is false because the two operations are mutually exclusive. Finally, *affects*(A_m, F_i) and *affects*(A_m, F_l) are both false because A_m runs after F_i and F_l.

The situation in Figure 10(d) is similar to Figure 10(c) with a few noteworthy exceptions. Although F_l occurs (not strictly) after A_m, A_m cannot possibly affect F_l because they are mutually exclusive: the same expression c controls the two conditionals. On the other hand, A_k does affect F_l because A_k comes before F_l along a feasible path. Of course, path-feasibility is in general undecidable, so *affects* will always be an over-approximation.

In this section, we use a flow-aware analysis that simply imposes a total order on the statements in each procedure to produce a simple approximation of *affects* that can result in significant accuracy and perfomance benefits. Totally-ordering the statements in a procedure is itself an approximation (e.g., it ignores mutual exclusivity between conditional branches, data-dependent infeasible paths, etc.), but is inexpensive and produces good results. Figure 10(e) shows the linearization we choose for the control-flow graph of Figure 10(d): we place the code for the true branch of each conditional before the code for its false branch.

Let $affects_0(\sigma_A, \sigma_F)$ be true whenever the assignment σ_A occurs before the fetch σ_F in the total order imposed on its procedure. This is an approximation that leads to some spurious results. For example, in the linearization of Figure 10(d) shown in Figure 10(e), $affects_0(A_k, F_i)$ and $affects_0(A_m, F_l)$ are true, yet $affects(A_k, F_i)$ and $affects(A_m, F_l)$, the exact relations in Figure 10(d), are false. Thus, using $affects_0$ means a fetch edge can resolve to more assignments than the tightest analysis would, but it is a sound solution that turns out to greatly improve precision over a flow-insensitive analysis. We present experimental results in Section 6.

To implement flow-aware analysis, we label each edge σ in the AFG with an integer $rank(\sigma)$ obtained from a topological sort of the statements in the procedure. In this paper, we have written these labels as subscripts on F's and A's in the graphs. As mentioned earlier, our procedures are loop-free: a preprocessing step transforms loops into tail-recursive procedure calls.

In flow-aware analysis, the [ALIAS] inference rule becomes

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha, \gamma) \qquad affects_0(\sigma_A, \sigma_F)}{al(\beta) \subseteq al(n)} \quad \text{[FLOW-AWARE]}$$

We implement this rule using a variant of the worklist algorithm presented in Section 4.1 in which a pair (σ_A, σ_F) is inserted into the worklist only if $affects_0(\sigma_A, \sigma_F)$ is true. Analyzing the example of Figure 8 follows almost the same steps as those in Section 4.1. However, the addition of alias edge $(n_6, \texttt{x})$ does not trigger the insertion of $\langle F_4, \{A_7\}\rangle$ into the worklist. Indeed, $affects_0(A_7, F_4)$ is false, since A_7 occurs after F_4. Note this implies both faster convergence and a more accurate solution, since spurious alias $(n_4, \texttt{v})$ is avoided.

5.1 Ordering Aliases

To further increase the accuracy of our analysis, we also maintain ordering information on the alias edges in the AFG. This information indicates at what point in the procedure's execution the alias is created. We use such information to make later aliases invisible to earlier fetches, much as we do for later assignments.

Consider the resolved graph in Figure 8(c). The initial value edge for z—assignment edge A_z—is demoted into a fetch edge in Figure 8(d). This fetch edge "condenses" both F_3 and F_6 from Figure 8(c), given that either (or both) of them would set the initial value for z. To record this fact in the final summary, initial value edges are annotated with the interval that they represent—in Figure 8(d), the fetch edge (z, $\texttt{z}_1$) would be labeled as F_6^3. Operation edges can be seen as representing a unitary interval, e.g., A_1^1. We will refer to the superscript and subscript as *min* and *max* indices.

Figure 12 further illustrates alias ordering. Figure 12(a) represents the initial AFG for function h in Figure 11(c), obtained after the summaries for f and g have been computed. It is assumed that the summary for g was constructed by previously instantiating the summary for f. Figure 12(a) therefore shows the call to g that occurs in the body of h.

The resolved AFG for h (Figure 12(b)) shows the alias edge derived by the resolution phase. Like the other edges in the graph, it has been annotated with two indices. Such indices are obtained from the assign edge participating in the resolution, here A_8^8.

```
f(r,s,t)
{                    g(p,q)            h()
  **r = &y;          {                 {
   *s = &z;            f(p,q,p);         g(&x,&x);
  **t = &w;          }                 }
}
      (a)                 (b)              (c)
```

Figure 11: The code for procedures f, g and h.

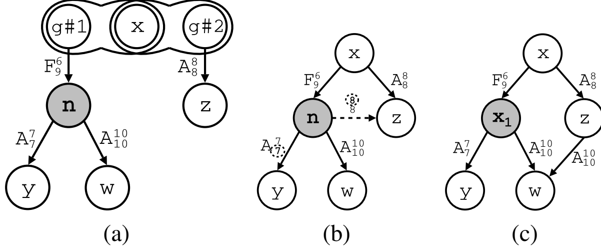


Figure 12: (a) Initial, (b) resolved and (c) summary AFG for h.

Indices for alias edges are used when re-directing assign edges in the resolved AFG to form the summary AFG. To remove a fetch node n, we re-direct its incoming and outgoing assign edges according to $al(n)$. In the graph of Figure 12(b), this would mean creating two new assign edges: $\texttt{z} \xrightarrow{A_7^7} \texttt{y}$ and $\texttt{z} \xrightarrow{A_{10}^{10}} \texttt{w}$. However, edge $\texttt{z} \xrightarrow{A_7^7} \texttt{y}$ is a spurious pointer relation that emerges because multiple dereferences have been encapsulated into a single dereference (i.e., multiple fetches have been condensed into F_9^6 much as F_6^3 represents $A_{\texttt{z}}$ in Figure 8(d)). Indeed, a quick look at Figure 11 verifies that z does not point to y because A_8^8, representing the assignment in `*s=&z`, occurs between A_7^7 and A_{10}^{10}, representing the assignments in `**r=&y` and `**t=&w`, respectively.

We avoid the spurious edge $\texttt{z} \xrightarrow{A_7^7} \texttt{y}$ by noting the *min* index for the resolution edge (n_1, z) is greater than the *max* index for A_7^7 (both encircled in Figure 12(b)). Intuitively, this means the aliasing between n and z in Figure 12(b) is created only after A_7^7 executes, and therefore $\texttt{z} \xrightarrow{A_7^7} \texttt{y}$ is invalid. Maintaining this additional ordering information increases the accuracy of our total order flow-aware analysis.

6. EXPERIMENTAL RESULTS

We implemented the pointer analysis framework presented in this paper in a static analysis (bug-finding) tool called BEAM [2], which is in widespread use within IBM Corporation. Our experiments show a number of things: first, our AFG-based pointer analysis technique can be applied on real-world programs; second, our flow-aware analysis (Section 5) has both better performance and accuracy than a flow-insensitive analysis; third, our AFG-based, summary-based analysis, generates pointer abstractions that have considerably smaller points-to sets when compared to existing techniques.

Table 1 lists the benchmark programs we analyzed. Paraffins is an implementation of the Salishan Paraffins problem, Compress and Gzip are well-known file compression programs, Ispell is a spelling checker, Pcre is a library that implements regular expression pattern matching, Make is the well-known build tool, Bison is the well-known gnu parser generator, Tar is a unix utility to concatenate multiple files into a single file, and Balsa is an electronic mail client. "Source functions" refers to the number of procedures in the original code; "internal functions" refers to the total number of procedures after replacing loops with tail-recursive functions. "SCCs" refers to the number of strongly-connected components in the program's call graph. Some of the benchmarks contain SCCs with as many as 52 functions (Make), 19 functions (Tar), and 14 functions (Balsa). This means, for instance, that Make has a "cluster" with 52 mutually recursive functions where the analysis must first converge before proceeding.

Table 1: Benchmark programs

Benchmark	Lines	Source functions	Internal functions	SCCs
paraffins	1.5K	13	36	36
compress	2.2K	30	66	66
gzip	8.3K	126	331	330
ispell	10.1K	117	337	337
pcre	15.4K	63	300	299
make	22.1K	309	853	799
bison	25.4K	700	1297	1296
tar	32.7K	651	1145	1124
balsa	110.0K	2659	4682	4648

Table 2: Flow-insensitive experiments

Benchmark	Analysis Time	Avg. Size	Max. Size
paraffins	0.22s	6.10	19
compress	0.13s	3.36	9
gzip	0.72s	3.86	18
ispell	13s	6.72	101
pcre	21s	5.76	19
make	114s	15.5	279
bison	44s	7.60	356
tar	27s	11.6	178
balsa	31s	4.32	51

Table 2 lists runtimes for the flow-insensitive analysis described in Section 4. The analysis time is for BEAM running on a 2.2 GHz Pentium 4 machine with 4 GB of memory running Linux. The Avg. Size and Max. Size columns lists the average and maximum number of nodes in the summary graphs for the program's procedures. Analysis times include the time to calculate the points-to sets as initial, resolved, and summary AFGs, and to propagate summaries until convergence. They do not include time taken to read source files, parsing, and building the program's intermediate representation in BEAM.

Analysis times are short enough to make our technique practical, but vary widely depending on the size of the input program and other characteristics. For example, although Make is only twice as long as Ispell, it takes more time to analyze Make because its call graph has large SCCs (ie., mutually recursive functions) on which the analysis must iterate.

The third and fourth columns of Table 2 suggest that procedure summary graphs are small on average and appear to grow polynomially with program size, i.e., much slower than the exponential worst case. Wilson and Lam [20] observed similar behavior with their partial transfer functions.

Table 3: Total Order Flow-aware compared to flow-insensitive

Benchmark	Speedup		Accuracy	
	Overall	Slowest Proc.	Average	Peak
paraffins	–	–	1%	20%
compress	3%	5%	1%	50%
gzip	8%	13%	7%	40%
ispell	60%	127%	34%	122%
pcre	243%	360%	10%	45%
make	109%	65%	23%	446%
bison	159%	503%	19%	337%
tar	59%	145%	5%	75%
balsa	23%	78%	12%	80%

Table 4: Points-to Set Sizes

Benchmark	AFG	Andersen	Steensgaard
paraffins	1.11	–	–
compress	0.524	1.22	2.1
gzip	2.06	2.96	25.17
ispell	1.92	2.25	16.45
pcre	2.21	–	–
make	26.11	74.70	414.04
bison	1.88	1.72	20.51
tar	2.58	17.41	53.7
balsa	0.92	–	–

For heap locations, we use a naming scheme that is similar to the technique of Choi et al. [5]. Basically, a heap location is named by continuously prepending the name of the caller function whenever a summary for a callee is embedded into the caller. A limit is given to the number of prefixes allowed, and distinct heap nodes are merged together based on this limit. The full handling of heap locations is out of the scope of this paper but can be found elsewhere [3].

Table 3 shows the results of experiments that compare flow-insensitive analysis to our flow-aware analysis (see Section 5). For these experiments, we used the simple-minded total linearization of the statements in each procedure and considered *min* and *max* times for each operation and alias edge.

Table 3 shows that our total-order flow-aware analysis is both more efficienct and more precise than the flow-insensitive analysis implementation. The second column lists the overall decrease in runtime for our flow-aware analysis compared to our implementation of flow-insensitive analysis. The third column indicates how much faster flow-aware analysis is at analyzing the procedure that took the most time.

The "accuracy" columns in Table 3 compare the size of the summary graphs obtained by flow-aware analysis compared with those from flow-insensitive analysis. We computed these numbers as follows: let R be the ratio of assign edges to the total number of nodes in the final summary graph for a procedure. If R^I is this ratio after flow-insensitive analysis and R^A is this ratio after flow-aware analysis, then the increase in accuracy is $Q = (R^I - R^A)/R_A$. The peak accuracy (fifth column) is the highest such Q over all procedures; the average accuracy is the average increase over all procedures: $(Q_1 + \cdots + Q_n)/n$.

Table 4 shows that the AFG abstraction for pointer analysis generates points-to sets that are considerably smaller than those obtained through Andersen or Steensgaard analysis[6, 13]. We compute points-to set sizes by observing that assign edges in the AFG for a function correspond to the set of locations pointed-to by the source node of the assign edge. Clearly, there are several summary AFGs in a given benchmark – one for each function in the program. To calculate the points-to set sizes shown in column 1 of Table 4, we compute an average among all AFGs. The table shows the resulting numbers for the flow-insensitive analysis on the AFG.

7. RELATED WORK

7.1 Pointer Analysis

Existing (flow-insensitive) pointer analysis algorithms can be roughly divided into Steensgaard's [17], Andersen's [1] and Das's [6]. Each computes one solution for the entire program with different degrees of precision and performance. The analysis solution is usually in the form of a large *points-to graph*, a directed graph where nodes represent memory locations and edges represent "poins-to" relationships. An edge from node `p` to node `q` means that `p` points-to `q`. Precision refers to the ratio between valid points-to relationships and spurious information (e.g., when the graph says that `p` points-to `q` but that can never happen in the actual code). Steensgaard's analysis unions two objects that are pointed to by the same pointer into one object (i.e., the outdegree of any node in the points-to graph is at most one). This leads to the unioning of the points-to sets of these formerly distinct objects, and therefore loss of precision. To implement that, Steensgaard's analysis employes a fast union-find data structure to represent all alias relations. Andersen's analysis also uses one solution for the entire program, but it does not merge objects that have a common pointer pointing to them. This leads to a better precision, although worse running time [1]. Das adds a small amount of directionality to a unification-based analysis [6], creating an analysis with precision close to Andersen's while remaining scalable.

Our AFG-based approach is a general one that can be specialized into a variety of analysis techniques. We presented two: an Andersen-style flow-insensitive analysis as well as a more precise, more efficient version called flow-aware analysis. The *affects* relation of Section 3.1 can be also used to derive analysis that are sensitive to conditions in the code, including a form of path-sensitive, flow-aware analysis.

A number of works present implementations of Andersen's analysis that are not whole-program. In [12], *program fragments*, i.e., arbitrary collections of functions, are analyzed using worst-case assumptions about callees and callers. Demand-driven versions of Andersen's analysis have been proposed for C [9] and Java [15, 16]. For each query, these analysis evaluate only those statements that are relevant for the particular query.

7.2 Summary-based analysis

The main goal of summary-based pointer analysis is to avoid re-analyzing the body of a function each time the function is invoked. One approach to this idea is to analyze the body of a function the *minimum number of times possible*, which is implemented by recording the set of input/output behaviors of a procedure. I.e., once a summary $\langle I, O\rangle$ has been computed for a procedure `pr`, it is not necessary to reanalyze the body of `pr` if context I arises at another call to `pr`. Instead, the summary $\langle I, O\rangle$ is consulted and the corresponding output context O is used. On the other hand, if a yet unseen input I' arises, then the body of `pr` needs to be re-evaluated in order to compute and store the new pair $\langle I', O'\rangle$. This is basically the idea behind the algorithm of Wilson and Lam [20], where

multiple summaries for a function are constantly being memoized during a top-down traversal of the program. For any given function `f`, there is one summary for each distinct calling context I under which `f` has been invoked (a calling context is defined in terms of the alias relations among function parameters, which are determined by the callers). This signifies that the procedure is not summarized for all potential aliases among its inputs, but only for those that actually occur in the program—which also implies that a function may need to be re-summarized upon the addition of new callers (*e.g.*, in a modular development).

Instead of recording multiple input/output behaviors for each possible input (i.e., alias contexts) that occur in the program [20], we create a single summary for each procedure that can be specialized at any call site. However, our summaries are soundly used at a call site containing aliases. In contrast, a single points-to graph cannot summarize a C function for all possible calling contexts.

Another approach to summary-based analysis is to evaluate the body of a function once, but to store *multiple* analysis results (i.e., multiple summaries); each result is indexed under the possible alias relations that may exist at function entry [4, 8]. In contrast to Wilson and Lam [20], summaries are built using no information "from above" (i.e., from callers). For example, for a function `f` with two pointer parameters `p1` and `p2`, two "initial conditions" may be assumed: either `p1` and `p2` point to the same location(s), or they don't. From these starting points, two initial summaries are generated, which may later branch into more derived summaries. Basically, such techniques use *alias contexts* to lazily enumerate *potential* aliases among parameters in order to distinguish transfer functions for different calling contexts. In [10], the authors formulate a "conditional may alias analysis" for C, handling multiple levels of indirection, for both scalar and aggregate data types. Given a set of alias pairs that are true at function entry, they compute the set of the alias pairs that hold at function exit (i.e., *conditioned* on the alias pairs at the entry).

Our technique builds a single summary for each function that uses no information from above and analyzes the body of each function once[1]. Like Livshits et al. [11] and Whaley and Rinard [19], we assume no aliasing among locations from the environment while analyzing a procedure, a seemingly unsound assumption. Later, aliasing is (incrementally) taken into account when incorporating the summary into a calling environment containing aliases, restoring the soundness of the analysis. The approach has the advantage of computing just one summary that can be applied in all situations, simplifying the summary representation as well as speeding the analysis, and it applies for languages such as C.

8. CONCLUSIONS

We presented a new approach to pointer analysis based on the assign-fetch graph. By representing the operations in a procedure rather than points-to relations, the AFG enables context-agnostic procedure summaries and several pointer analysis variations.

We present two pointer analysis techniques on the AFG: a standard flow-insensitive one and a new approach we call flow-aware. The flow-aware analysis considers an approximation of the control-flow in a procedure to reduce the number of spurious alias relationships that would otherwise arise in a flow-insensitive analysis of a procedure.

Experimental results on real-world C programs showed that our flow-aware analysis is both faster (3–243%) and more precise than flow-insensitive analysis. We measured precision by looking at the number of edges in the procedure summaries. Since both analyses are sound, fewer edges corresponds to a more precise summary.

In the future, we plan to develop yet more precise analyses by refining the approximation of the *affects* relation of Section 3.1. For instance, a path-sensitive approximation can be made by including the conditions under which fetches and assignments occur. Moreover, we are currently adding field-sensitivity to the analysis by inserting one more type of edge in the AFG, a *field-dereference* edge, which will handle arbitrary casting possible in C.

[1] Modulo achieving a fixed-point for recursive procedures.

9. REFERENCES

[1] L. O. Andersen. Program analysis and specialization for the C programming language. PhD thesis, DIKU, University of Copenhagen, May 1994. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

[2] Daniel Brand. A software falsifier. In *International Symposium on Software Reliability Engineering*, pages 174–185, October 2000.

[3] Marcio Buss. Phd thesis (to appear). Department of Computer Science, Columbia University.

[4] Ramkrishna Chatterjee, Barbara Ryder, and William A. Landi. Relevant context inference. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 133–146, 1999.

[5] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual ACM Symposium on Principles of Programming Languages*, pages 232–245, 1993.

[6] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 35–46, 2000.

[7] Maryam. Emami, Rakesh Ghiya, and Laurie Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[8] Mary Jean Harrold and Gregg Rothermel. Separate computation of alias information for reuse. *IEEE Transactions of Software Engineering*, 22(7):442–460, 1996.

[9] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 254–263, 2001.

[10] William Landi and Barbara Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 235–248, 1992.

[11] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, 2003.

[12] Atanas Rountev, Barbara Ryder, and William Landi. Data-flow analysis of program fragments. In *7th European Software Engineering Conference/ACM SIGSOFT Symposium on the Foundations of Sofware Engineering, ESEC/SIGSOFT FSE*, pages 235–252, 1999.

[13] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of Principles of Programming Languagues (POPL)*, pages 1–14, 1997.

[14] Jyh shiarn Yur, Barbara Ryder, and Willim Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International Conference on Software Engineering*, pages 442–451, 1999.

[15] Manu Sridharan, , and Rastislav Bodik. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 387–400, 2006.

[16] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. Demand-driven points-to analysis for java. In *Proceedings of the SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2005.

[17] Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[18] Frederic Vivien and Martin Rinard. Incrementalized pointer and escape analysis. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 35–46, 2001.

[19] John Whaley and Martin Rinard. Compositional pointer and escape analysis for java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA)*, 1999.

[20] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 1–12, 1995.