Flexible Pointer Analysis Using Assign-Fetch Graphs

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 that represents reads and writes to memory instead of traditional points-to relations. Compared to points-to graphs, our Assign-Fetch Graph (AFG) leads to concise procedure summaries that can be used in any calling context. Also, its flexibility supports new analysis techniques with different trade-offs between speed and precision.

For efficiency, we build a summary for each procedure that assumes distinct pointers from the environment are not aliased and restore soundness when the summary is used in a context with aliases.

We present two pointer analysis techniques based on our AFG. The first takes the flow-insensitive view adopted by many authors; the second considers statement ordering. In addition to being more precise, we find that this "flow-aware" analysis runs faster. We conclude with experimental results showing it is practical.

Categories and Subject Descriptors

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—Program Analysis

General Terms

Algorithms, Languages, Performance

Keywords

Static analysis, pointer analysis, summary-based analysis

1. INTRODUCTION

Pointer analysis [1, 5, 6, 7, 8, 10, 11] is necessary for most language processing tools, including optimizing compilers and tools for bug finding, program understanding, and refactoring. Such analysis consists of computing *points-to* information—given two program locations, p and q, we say p points-to q if p can contain the address of q. Pointer analysis statically estimates such possible set of locations a pointer can point to during program's execution.

Pointer analysis is of greatest concern for a language such as C that exposes pointers to the programmer, but it is also helpful for tools for a language such as Java that hides pointers behind object references. We consider C, but our techniques could be adapted.

Instead of the *points-to graph* [6, 7, 10] commonly used in pointer analysis, we propose a new representation for a function's behavior: the assign-fetch graph (AFG). A strength of our AFG is that it support many different algorithms, including the well-known flow-insensitive technique and the faster, more precise flow-aware approach that we present for the first time in this paper.

Unlike a points-to graph, whose nodes represent pointer variables and whose edges represent points-to relations, the nodes in our AFG represent locations and values and edges represent reads and writes to memory. Pointer analysis amounts to matching pointer dereferences ("fetch edges") to pointer assignments ("assign edges"). The AFG allows varying levels of precision by allowing different matchings; a more selective matching may require more resources. Our analysis is *summary-based* [4, 11]: it computes a representation for each procedure that summarizes its effects on pointers.

After introducing our technique with an example, we describe the AFG (Section 2) and how it is used for pointer analysis (Section 3). We implemented two kinds: classical flow-insensitive (Section 3.2), which we use as a baseline, and our faster and more accurate flow-aware technique (Section 3.3). Section 4 discusses the interprocedural nature of the analysis. We conclude with results from running our technique on large programs (Section 5).

1.1 An Example

Our AFG is designed to let us compute what values a program might read from memory while it is executing. Except for I/O and constants, the program itself must have written any value that is read, so pointer analysis can be thought of as an attempt to understand which writes could be seen by each read. One approximation is that each write to a location can be seen by every read of that location, but this is usually an overapproximation: a read and write may occur in different branches of a conditional, or a write might occur in sequence after a read. With the AFG, it is easy to approximate these relationships differently, producing different pointer analysis algorithms. In this paper we explore two: our baseline, classical flow-insensitive analysis, and a new flow-*aware* analysis.

Our AFG abstracts a procedure's reads and writes instead of points-to relations, which has many advantages: the AFG is simpler to construct, our summaries work in all possible calling contexts (in particular, when arguments are aliased), and the AFG facilitates trading off analysis precision for efficiency.

Figure 1 shows the AFG in action. Figure 1(b) depicts the AFG for the C code in Figure 1(a). The AFG's nodes represent values and addresses and its edges represent read and write operations. The first statement in Figure 1(a), `*z=&x`, stores the address of the global variable `x` at the address in `z`. We represent `z` with the location node `z`, the dereference of `z` with the fetch edge F_1, the address read from `z` with the *fetch node* n_1, the address of `x` with the location node `x`, and the write to `*z` with the assign edge A_2.

In our figures, we shade each fetch node as a reminder that we do not know its value when we construct the graph. The basic question for pointer analysis then becomes "*given a fetch, which assigns should it match*?" The AFG abstraction allows us to answer this question differently depending on speed/precision trade-offs.

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

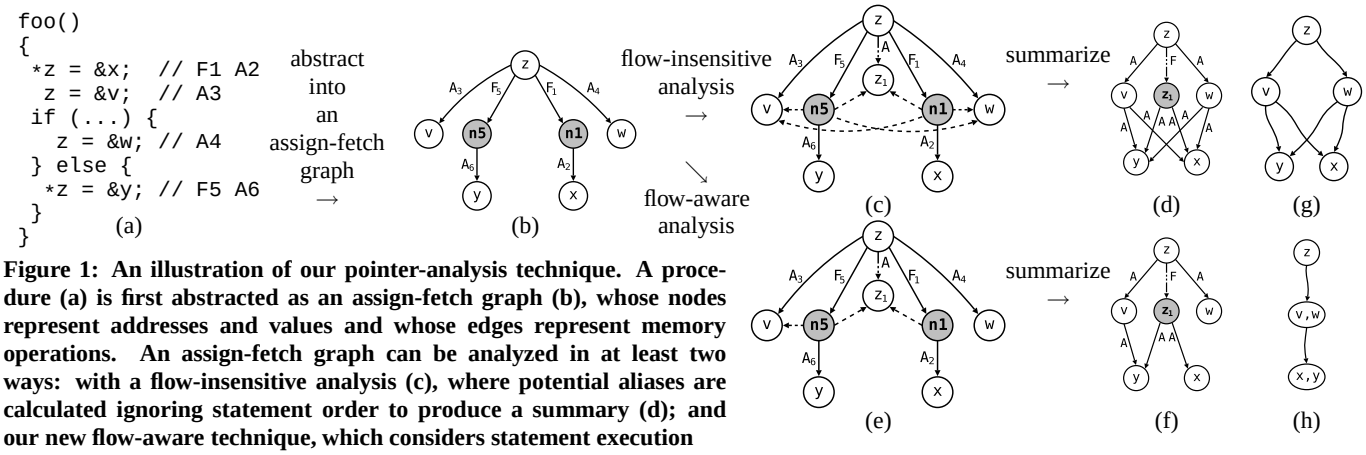

Figure 1: An illustration of our pointer-analysis technique. A procedure (a) is first abstracted as an assign-fetch graph (b), whose nodes represent addresses and values and whose edges represent memory operations. An assign-fetch graph can be analyzed in at least two ways: with a flow-insensitive analysis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

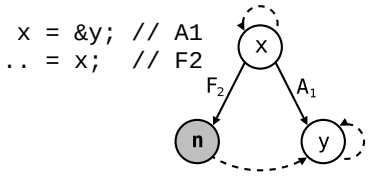

Figure 2: The simple case: `x` is assigned in A_1 and fetched in F_2; an *alias edge* indicate that n can be an alias for `y`. Self-loops represent trivial aliasing of locations.

Answering this question is the goal of the *resolution phase*, which adds *alias edges* from fetch nodes to location nodes to indicate what values could be fetched. Each fetch of the same variable in a procedure generates a distinct fetch node, allowing the AFG to represent variables that take on different values at different times.

Figure 2 shows the simplest alias case: we add a (dashed) alias edge from n to `y` to indicate fetching `x` (F_2) after assigning it the address of `y` (A_1) returns the address of `y`. The self-loops indicate, e.g., the address of `x` is itself. We omit them in all other figures.

Adding aliases from fetch to location nodes produces a *resolved* AFG. Using different levels of precision can generate different resolved AFGs, such as Figures 1(c) and 1(e). The former is a flow-insensitive view of the program, where a fetch from a location matches any assignment to the same location; the latter is a more precise result obtained by considering statement ordering and mutually exclusive operations, which leads to a smaller number of fetch/assign matchings. E.g., because they appear in separate branches of a conditional, `*z=&y` and `z=&w` are mutually exclusive, so fetch F_5 cannot see assign A_4 and there is no alias edge from n_5 to `w` in Figure 1(e). Also, the first fetch of `z` in the code (F_1) can only "see" the (unknown) initial value of `z` coming from the environment (represented by z_1). Thus, n_1 to z_1 is the only alias edge.

Figure 1 also shows points-to graphs from two well-known techniques. Figure 1(g) is the result from Andersen [1], which is equivalent to Figure 1(d), but we also include the initial value node z_1. Figure 1(h) is the result of Steensgaard [10], who merges nodes to which the same pointer points to, such as $\{\mathtt{v},\mathtt{w}\}$ and $\{\mathtt{x},\mathtt{y}\}$. It generates smaller but less accurate graphs.

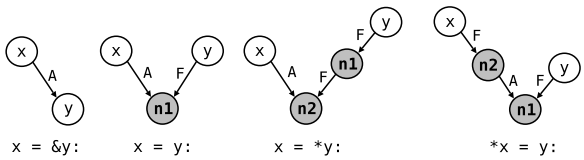

Figure 3: The usual statements considered in pointer analysis represented in AFG notation

2. ASSIGN-FETCH GRAPHS

Our assign-fetch graphs represent data and operations in a procedure in a way that makes it easy to compute the relationships among them. It is a directed graph whose nodes represent values and locations and whose edges represent fetches and assignments to this data. To compute points-to information for a function, we mechanically build an AFG, add alias edges to "resolve" it, and remove information hidden from a caller, producing a summary.

Before building AFGs, we decompose a program into loop-free procedures. We transform loops to tail-recursive procedure calls, so a procedure's statements can be ordered in a useful way. We handle mutually recursive procedures by iterating their analysis to convergence. Details are given in Section 4.5.

Our AFGs abstract many things. We ignore pointer arithmetic by considering all elements of an array to be one location, merge fields of structures so an expression such as `p->next` is treated like `*p`, and model heap locations using a naming scheme like Choi et al. [5]. We model return values with a special location *ret*.

Figure 3 shows AFG fragments for the four statements typically considered in pointer analysis. For `x=&y`, we represent the lvalue `x` and the rvalue `&y` as location nodes and connect them with an assign edge indicating `x` points to the memory location for `y`. Note that this does not read or change the contents of `y`. By contrast, since `x=y` reads `y`, we add the fetch node n_1 to represent its value, add a fetch edge indicating a read of `y`, and add an assign edge to indicate they are written to `x`. We treat other statemenets similarly.

In alias analysis, executing statement `p=&r` creates the alias relation $\langle \mathtt{*p}, \mathtt{r} \rangle$, meaning `*p` is *an alias for* `r`. Computing points-to sets using the AFG amounts to determining the locations for which a fetch node could be an alias. We represent such relations by adding directed *alias edges* to the AFG. In Figure 2, the dashed edge (n, `y`) represents the alias relation $\langle \mathtt{*x}, \mathtt{y} \rangle$. Each (non-trivial) alias edge starts at a fetch node and terminates at a location node.

The central goal of pointer analysis is to determine a small set of aliases that includes every possible one (i.e., remains sound). While we could proclaim that every fetch node aliases every location, such a gross overapproximation would not be very helpful. In the next section, we discuss various ways to compute this set more precisely.

3. DETERMINING ALIASES

Determining aliases between fetch nodes and location nodes is the main step in pointer analysis on AFGs. In this section, we discuss procedures with varying levels of precision, and show how the AFG lends itself to such variants, one of its key strengths.

3.1 Alias Analysis

Let x, y,... denote *location* nodes, $n_1, n_2, \ldots$ denote *fetch* nodes; and $\alpha, \beta, \ldots$ denote arbitrary nodes. We write $al(\alpha)$ to indicate the set of nodes that α can be an alias for. In Figure 2, $al(n) = \{y\}$, $al(y) = \{y\}$ and $al(x) = \{x\}$.

An alias edge from a node α to a node x indicates $x \in al(\alpha)$; an alias edge's target is always a location. We assume variables are distinct, so a location node only aliases itself: $al(x) = \{x\}$.

A fetch node *n* can be an alias for many locations; computing them is the main purpose of any analysis. Because a fetch can only return a value that the program wrote to memory, any alias of a fetch node must be the target of an assign edge (we model the initialization of global variables with assign edges).

We write $affects(\sigma_A, \sigma_F)$ to indicate the assign edge σ_A could write a value that fetch edge σ_F could read. This relation can be many-to-many: one assignment could be seen by many fetches, and a fetch might see many assignments. Unfortunately, *affects* is not effectively computable, so any pointer analysis must approximate it. A sound analysis demands an overapproximation: it should be true when *affects* is true, but not necessarily vice-versa.

When an assignment affects a fetch, the fetch can return anything written by the assignment, so aliases for the fetch must include all aliases of the assignment's "right-hand side." Put formally,

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

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*. The solution to pointer analysis is the minimal set of alias edges that satisfies this rule.

3.2 Flow-Insensitive Analysis on the AFG

A simple approximation of the *affects* relation gives Andersen-style flow-insensitive analysis, which will use as a baseline for evaluating our new flow-aware analysis (described in the next section). Define the predicate *aliases* as

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

This says nodes α and γ are aliases for the same thing if they are identical or if they are aliases for some common location node.

This relationship is a flow-insensitive (over-) approximation of the exact $affects(\sigma_A, \sigma_F)$, so [ALIAS] can be approximated by

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

Because this rule is recursive (the premise refers to the *aliases* relation, which depends on *al*), finding the minimal resolved AFG requires computing a fixed point. Our implementation uses the usual worklist algorithm that iterates to convergence.

These rules ignore statement order. Consider Figure 1(c), which shows the resolved AFG for Figure 1(b) under the [FI-ALIAS] rule. Every assign to a location is seen by all fetches from that location, so in Figure 1(b), both n_1 and n_5 will resolve to both v and w. In the implementation, we also create the (unknown) initial value node z_1 since global z is dereferenced within the function (we lazily initialize environment variables). Nodes n_1 and n_5 also resolve to z_1.

3.3 Flow-Aware Analysis on the AFG

Here, we describe our second key contribution: flow-aware analysis, which considers statement ordering when computing aliases.

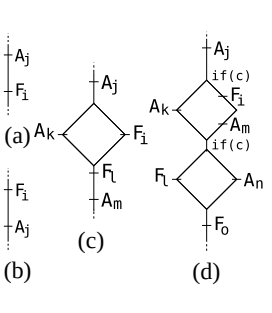

Figure 4: Motivation for flow-aware analysis. (a) An assignment before a fetch can affect the fetch; (b) one after a fetch cannot. Conditionals (c, d) further constrain which assignments are visible to a fetch. Our flow-aware analysis approximates the execution order (d) with the total order (e).

From experiments, which we describe in Section 5, we find flow-aware analysis runs faster than flow-insensitive analysis and produces more precise results, making it all-around superior. Reducing the number of matchings between fetches and assigns leads to faster convergence with fewer alias edges.

The basic idea in flow-aware analysis is to number all the assignments and fetches in a procedure (i.e., assign them a total order) and only consider assignments whose labels are less than that of a matching fetch. This is a quick-to-compute conservative approximation (the true relationship is a complex partial order due to conditionals and data) that is more precise than the [FI-ALIAS] rule, which completely ignores statement order. The AFG abstraction is key for the flow-aware analysis.

The control-graph fragments in Figure 4 illustrate the motivation for this approximation. Figure 4(a) is the simplest 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, such as in Figure 4(b), cannot be affected by the assignment. A flow-insensitive analysis treats these two cases identically; our flow-aware analysis does not build an alias edge in the second case.

Conditionals add complexity. In Figure 4(c), fetch F_i should resolve to assign A_j since the latter occurs strictly before the former, but *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 4(d) is slightly different. Although F_l occurs after A_m, A_m cannot affect F_l because they are mutually exclusive: the expression *c* controls both conditionals. However, A_k does affect F_l because A_k comes before F_l along a feasible path. *Affects* is always an approximation since path-feasibility is undecidable.

Figure 4(e) shows one possible total order for the control-flow graph of Figure 4(d): we numbered statements in the true branch of each conditional before its false branch.

Figure 1 illustrates the precision advantage of a flow-aware analysis using conditionals. Figure 1(f) is the summary of Figure 1(e), which is more precise than Figure 1(d).

Let $affects_0(\sigma_A, \sigma_F)$ be true when the assignment σ_A occurs before the fetch σ_F in the total order. This approximation can produce spurious results. For example, in the linearization of Figure 4(d) in Figure 4(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 4(d), are false. Thus, $affects_0$ allows a fetch edge to resolve to extra assignments, but it is a sound solution with substantially improved precision over flow-insensitive analysis.

To implement flow-aware analysis, we simply augment each edge σ in the AFG with an index $rank(\sigma)$ from a topological sort of the statements in the procedure. We write these labels as subscripts on *F*'s and *A*'s. The [ALIAS] rule for flow-aware analysis is

$$\frac{\sigma_A : \gamma \xrightarrow{A} \beta \qquad \sigma_F : \alpha \xrightarrow{F} n \qquad aliases(\alpha, \gamma) \qquad affects_0(\sigma_A, \sigma_F)}{al(\beta) \subseteq al(n)}$$

[FA-ALIAS]

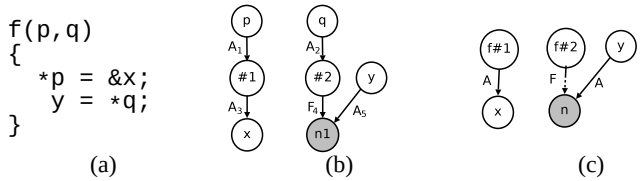

Figure 5: (a) A procedure, (b) its AFG, and (c) summary.

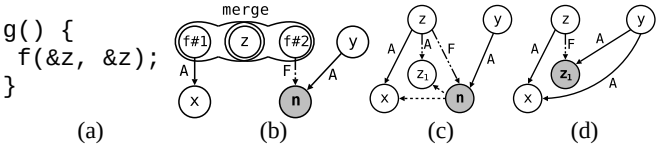

Figure 6: (a) A procedure g, which calls f from Figure 5 (b) Its initial AFG (c) After resolving (d) Its summary

4. INTERPROCEDURAL ANALYSIS

Below, we describe how we perform interprocedural analysis with our AFG. We describe how we summarize procedures, handle function parameters, and use summaries at procedure call sites. Although our summaries assume their parameters do not alias, they can be used in settings where parameters are aliases and remain sound, a key advantage of our approach.

4.1 Computing Summaries

To prepare a procedure's AFG to be used at a call site, after computing aliases using flow-insensitive or -aware analysis, we summarize the AFG for each procedure by deleting anything that a caller could not see, such as temporary memory fetches n_1 and n_5 in Figure 1(c). Before we delete such nodes, we transfer their effects to nodes that will remain. Figure 1(d) shows this. In general, if an assignment is made to a fetch node n, and n can be an alias for a location node n', the assignment is equivalent to one to n'. E.g., in Figure 1(c), n_1 is assigned the address of `x` and can be an alias for z_1, `v`, and `w`, so we add assign edges from z_1, `v`, and `w` to `x`. Similarly, we add edges from z_1, `v`, and `w` to `y`. Finally, we remove n_1 and n_5 and "demote" z_1 to a fetch node to indicate the dereference of `z`. This produces the flow-insensitive summary in Figure 1(d).

4.2 Modeling Parameters

We treat procedure parameters almost like global variables. While we assume each comes from the environment, a caller always initializes formal parameters so we add explicit initial value nodes for them; we only add an initial value for each global variable that is fetched by the procedure. Since formal parameters are local variables, i.e., stacked and discarded when a procedure returns, we remove their location nodes during the summarization process.

Figure 5 illustrates how we handle parameters. We add location nodes for formal parameters `p` and `q` and initial value nodes `#1` and `#2` that represent their initial caller-passed values. Figure 5(b) also includes nodes and edges for the two statements.

Since formal parameters are initialized by "#*i*" nodes, the AFG representation for `*p=&x` in Figure 5(b) does not include a fetch edge for `p`; `*p` directly yields its initial value, #1.

In summarizing this (alias-free) procedure, we remove the nodes for the formal parameters `p` and `q` and rename the initial value nodes to include the procedure's name. Also, fetch edge F_4 in Figure 5(b) generates an initial value for node #2, which in Figure 5(c) is labeled *n*. Figure 5(c) is the final summary.

4.3 Modeling Procedure Calls

When building the initial AFG for a procedure, a call to a function is replaced by the callee's summary. Instantiation a summary involves merging any global variables shared by both and connecting formal parameters to actual parameters. Figure 6 illustrates calling procedure `f` from Figure 5. The address of global variable `z` is passed to both `p` and `q`, so when we copy the summary of `f` from Figure 5(c), we mark the nodes for the initial values of `p` and `q`, `f#1` and `f#2`, to be merged with `z`.

We perform the same process for each global variable: its node in the callee is merged with its node in the caller. This is vacuous in Figure 6 since `g` does not touch globals `x` or `y`.

Node merging works even when an actual parameter is an expression. The call `f(z,z)` would represent its actual arguments as two fetch edges from `z` to nodes, say, n_1 and n_2, which would be merged with the value nodes for the formals: `f#1` and `f#2`. Computing aliases on the AFG would find the two parameters aliased.

Once each callee's summary has been instantiated, we compute the caller's summary. In Figure 6(c), we added an initial value node for global `z` and used flow-insensitive analysis to add alias edges from from *n* to `x` and z_1. Figure 6(d) is the summary. We removed fetch node *n*; its aliases now manifest themselves as the assign edges from `y`. A caller of `g` knows that `z` is dereferenced somewhere down the line by looking at `g`'s summary.

This example illustrates how a summary is agnostic about parameter aliasing and can be used in any context. The summary in Figure 5(c) treated parameters `p` and `q` as distinct, but we merged them at the call site for `f` in `g` and found that running `g` makes `y` point to `x`. Existing solutions either use information from the environment while building a summary, or build multiple summaries for each function, one for each possible environment.

4.4 Interprocedural Flow-Aware Ordering

Performing flow-aware analysis across procedure calls requires us to label statements on both sides of a call site. To get this right, we increase the indices of a callee by the maximum index that occurs in the caller before the call site, then increase the indices in the caller that appear after the call. Figure 7 illustrates this.

In Figure 7(b), `x`'s value is read by `q=x` in `bar()` then modified by `f()` at the call site `f(&x)`. When statement `*q=&w` executes, the original value of `x`, `&v`, is set to point to `w`. By ignoring order information, an interprocedural flow-insensitive analysis would pessimistically include `a` and `b` as values that could be read by `q=x`. Our flow-aware analysis avoids these. Figures 7(c) and 7(d) show the resolved and summary AFGs for function `f`. We sort the edges in a function summary and number them starting from 1, being careful to preserve the order among statements. Figure 7(e) shows how the summary for `f` is instantiated at the call site `f(&x)`. Statements before the call are labeled A_1, F_2, and A_3.

To place a callee's statements in the total order, we add the highest index before the call to every statement in the callee's summary when we instantiate it. In Figure 7(e), this index is 3, so we label them $A_{1\backslash 4}$ and $A_{2\backslash 5}$ to indicate A_1 and A_2 will become A_4 and A_5.

Figure 7(f) shows the result after flow-aware alias analysis. Note the fetch of `x` in `q=x` (F_2) resolves to A_1, the only assignment occurring before that fetch. Figure 7(g) is the points-to sets we compute with our flow-aware analysis, which is more precise than the flow-insensitive result in Figure 7(h).

4.5 Loops and Recursive Procedures

We convert loops into tail-recursive procedures and iteratively analyze (such) recursive procedures until we reach a fixed-point. The first time a recursive procedure is analyzed, we do not have

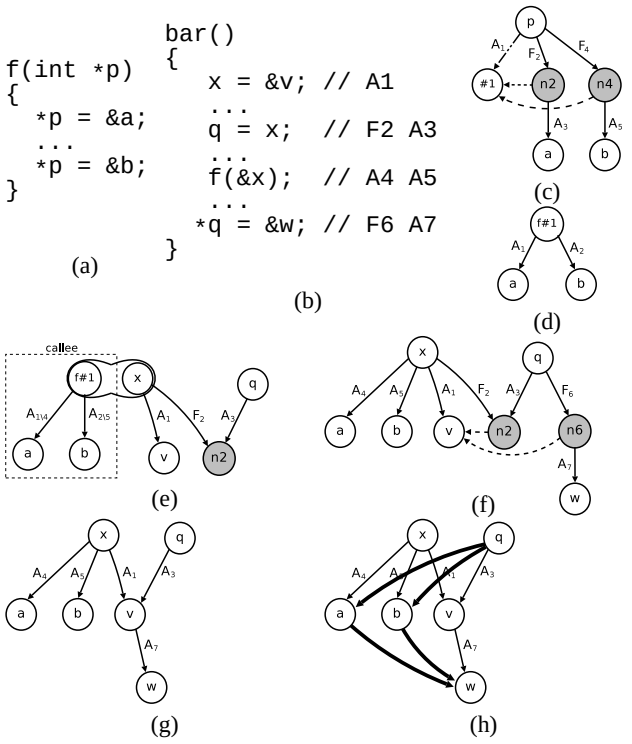

Figure 7: Propagating flow-aware ordering across procedure calls. (a) Function `f()` is called by (b) `bar()`. The AFG for `f()` (c) is summarized (d), inserted in the AFG for `bar()` (e), the indices in the summary are updated and flow-aware analysis is performed (f). The flow-aware points-to set (g) is more precise than the flow-insensitive (h).

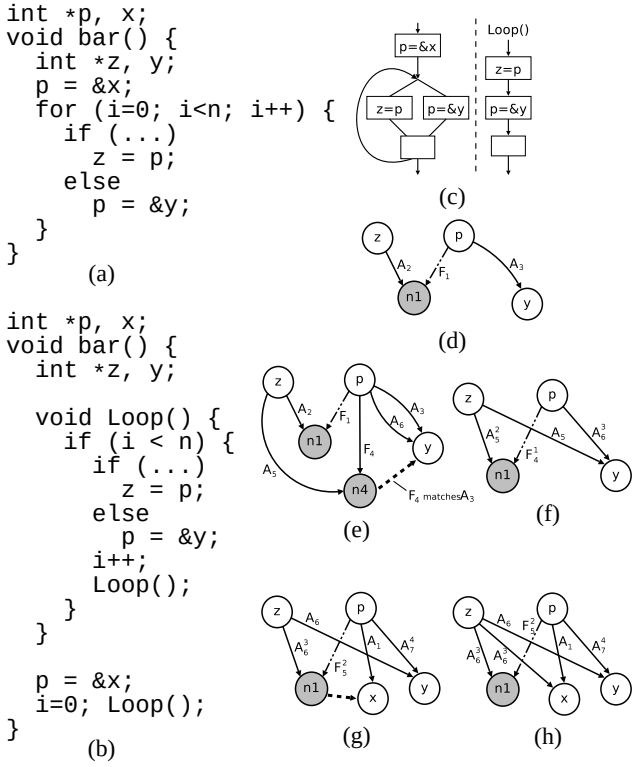

Figure 8: Handling loops and recursive functions.

a summary for it, so we only consider the other statements in the procedure. This gives a better summary for the procedure, which we then instantiate at recursive call sites and summarize again.

It may appear this procedure may not terminate, but this is not the case. It turns out the number of edges and arcs that can be added is bounded. The number of heap nodes is bounded because of the heap naming scheme we adopt. The number of fetch edges is bounded because the final summary allows at most one fetch edge out of any node, and there is a limit on the length of any chain of fetch edges. Finally, we prohibit duplicate assign edges. Together, these constraints bound the summary and guarantee convergence. If duplicate assignments between a pair of nodes is allowed, such as in Figure 8(e), the comparison between two summaries must only consider whether $x \xrightarrow{A} y$ exists and not the number of such edges. Details can be found in the first author's thesis [3].

Figure 8 illustrates summarizing a *for* loop. We transform the function in Figure 8(a) into the tail-recursive procedure in Figure 8(b). We nested the definition of `Loop` inside `bar` to emphasize that it has access to `bar`'s local variables.

Figure 8(c) is a simplified control-flow graph for this code. On the left is the structure of the loop; on the right is a linearized version of the `Loop` procedure that assumes flow-aware analysis orders the *then* branch of the *if* before the *else*.

Figure 8(d) is the first summary of `Loop`—the assignment `p=&y` is hidden from `z`. We now have a summary of `Loop`, so we insert it at its call site. This gives Figure 8(e). Edges with subscripts 1, 2, and 3 correspond to the loop body statements within the function. Instantiating the earlier summary adds edges F_4, A_5, and A_6 (the indices are shifted as in Section 4.4). This time, fetch edge F_4 matches assignment A_3, and `z` will point to `y` as a result; edges F_4 and A_3 belong to different iterations of the original loop.

Figure 8(f) is the summary of Figure 8(e) and also the fixed-point—the final summary for the function. Some edges have two numerical labels because they are the result of merging multiple edges. For example, A_6^3 represents the merge of A_3 and A_6.

Figure 8(g) shows the graph for `bar` after we inserted the summary for `Loop`. The fetch of `p` resolves to `p=&x` since the assigment occurs before the fetch (i.e., we check that the subscript index of the fetch is greater than the superscript on the assign, in case one exists, or the subscript otherwise). Finally, Figure 8(h) shows the summary for `bar`, which notes that the global variable `p` is fetched.

5. EXPERIMENTAL RESULTS

We implemented our pointer analysis framework in a static analysis (bug-finding) tool called BEAM [2] developed at IBM. Our experiments show our AFG-based pointer analysis technique can be applied on real-world programs; our flow-aware analysis (Section 3.3) has both better performance and accuracy than a flow-insensitive analysis; and our AFG-based analysis generates considerably smaller points-to sets than existing techniques.

Table 1 lists our benchmarks. Paraffins is an implementation of the Salishan Paraffins problem, compress and gzip are file compression programs, ispell is a spelling checker, pcre is a library for regular expression pattern matching, make is a build tool, bison is a parser generator, tar is a file archive utility, and balsa is an electronic mail client. "Source functions" count procedures in the original code; "internal functions" count procedures after replacing loops with tail-recursive functions. "SCCs" count strongly-connected components in the program's call graph. Some benchmarks contain large SCCs: make has a cluster of 52 mutually-

Table 1: Experimental Results

Benchmark	Statistics				Flow-insensitive Analysis			Flow-aware Analysis				Avg. Points-to Set Sizes		
	Lines	Functions		SCCs	Time	Nodes/Summary		Speedup		"Precision"		AFG	Steensgaard	
		Source	Internal			Avg.	Max.	Overall	Slowest	Avg.	Peak	Andersen		
paraffins	1.5k	13	36	36	0.22s	6.1	19	–	–	1%	20%	1.11	—	—
compress	2.2	30	66	66	0.13	3.4	9	3%	5%	1	50	0.524	1.22	2.1
gzip	8.3	126	331	330	0.72	3.9	18	8	13	7	40	2.06	2.96	25.17
ispell	10.1	117	337	337	13	6.7	101	60	127	34	122	1.92	2.25	16.45
pcre	15.4	63	300	299	21	5.8	19	243	360	10	45	2.21	—	—
make	22.1	309	853	799	114	16	279	109	65	23	446	26.11	74.70	414.04
bison	25.4	700	1297	1296	44	7.6	356	159	503	19	337	1.88	1.72	20.51
tar	32.7	651	1145	1124	27	12	178	59	145	5	75	2.58	17.41	53.7
balsa	110.0	2659	4682	4648	31	4.3	51	23	78	12	80	0.92	—	—

recursive functions on which the analysis must converge, tar has a cluster of 19 functions, and balsa has one with 14.

Table 1 lists run times for flow-insensitive analysis (Section 3.2). The analysis time is for BEAM running on a 2.2 GHz Pentium 4 machine with 4 GB of memory running Linux. The "nodes/summary" columns list the average and maximum number of nodes in each procedure's summary graph. Analysis times includes calculating the points-to sets as initial, resolved, and summary AFGs, and to propagate summaries until convergence, but does not include the time for BEAM to read source files, parse, and build the IR.

Analysis times are short enough to make our technique practical, but vary widely. 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 (i.e., groups of mutually recursive functions).

The nodes/summary columns suggest procedure summaries are small on average and grow polynomially with program size—much slower than the exponential worst case. Wilson and Lam [11] observed similar behavior on partial transfer functions.

We name heap locations following Choi et al. [5]: a location's name is the list of callers at its allocation site. We limit name length and allow colliding names to merge. Details are elsewhere [3].

The "flow-aware analysis" columns compare our flow-aware analysis to flow-insensitive. It shows flow-aware analysis is both more efficient and more precise than our implementation of flow-insensitive analysis. The "Overall" column lists the overall decrease in run time for our flow-aware analysis compared to our implementation of flow-insensitive analysis. Our flow-aware analysis can be over $2\times$ faster. The "Slowest" column lists how much faster flow-aware analysis is at analyzing the procedure that took the most time.

The "precision" columns compare the size of the summary graphs from flow-aware analysis with those from flow-insensitive analysis. If R is the ratio of assign edges to the total number of nodes in the final summary graph for a procedure, R^I is this ratio after flow-insensitive analysis, and R^A is this ratio after flow-aware analysis, then the increase in precision is $Q = (R^I - R^A)/R_A$. The peak precision is the highest such Q over all procedures; the average precision is the average increase over all procedures: $(Q_1 + \cdots + Q_n)/n$.

The "Avg. points-to set sizes" columns show our AFG abstraction generates smaller points-to sets than those of Andersen or Steensgaard [6, 9]. 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. There are several summary AFGs in a given benchmark—one for each function in the program. To calculate the points-to set sizes in Table 1, we compute an average among all AFGs. The table shows the numbers for the flow-insensitive analysis on the AFG.

6. CONCLUSIONS

We presented a new approach to pointer analysis based on the assign-fetch graph. By representing the operations in a procedure instead of points-to relations, the AFG enables context-agnostic procedure summaries and several pointer analysis variations. Since an assign-fetch graph abstracts code along with the possible results of code, it is very natural for inter-procedural analysis.

We described two AFG pointer analysis techniques, a standard flow-insensitive analysis that ignores statement ordering and a novel flow-aware technique that approximates the control flow in a procedure to reduce the number of spurious alias relationships.

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 are sound, fewer edges mean a more precise summary.

In the future, we will develop more precise analyses by refining the approximation of the *affects* relation of Section 3. For instance, a path-sensitive approximation can be made by including the conditions under which fetches and assignments occur. 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.

7. REFERENCES

[1] L. O. Andersen. Program analysis and specialization for the C programming language. PhD thesis, DIKU, U. Copenhagen, 1994.

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

[3] M. Buss. Summary-based pointer analysis framework for modular bug finding. PhD thesis, Columbia University (to appear), 2008.

[4] R. Chatterjee, B. Ryder, and W. Landi. Relevant context inference. In *Principles of Prog. Languages*, 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 *Principles of Prog. Languages*, pages 232–245, 1993.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Prog. Language Design and Impl.*, pages 35–46, 2000.

[7] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Prog. Language Design and Impl.*, pages 242–256, 1994.

[8] W. Landi and B. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Prog. Language Design and Implementation*, pages 235–248, 1992.

[9] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Principles of Prog. Languages*, pages 1–14, 1997.

[10] B. Steensgaard. Points-to analysis in almost linear time. In *Principles of Prog. Languages*, pages 32–41, 1996.

[11] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Prog. Language Design and Impl.*, pages 1–12, 1995.