Probabilistic Points-to Analysis

Yuan-Shin Hwang[†] Peng-Sheng Chen[‡] Jenq Kuen Lee[‡] Roy Dz-Ching Ju[§]

[†]Department of Computer Science, National Taiwan Ocean University, Keelung 202 Taiwan
[‡]Department of Computer Science, National Tsing Hua University, Hsinchu 300 Taiwan
[§]Microprocessor Research Lab., Intel Corporation, Santa Clara, CA 95052 U.S.A

Abstract. Information gathered by the existing pointer analysis techniques can be classified as *must* aliases or *definitely*-points-to relationships, which hold for all executions, and *may* aliases or *possibly*-points-to relationships, which might hold for some executions. Such information does not provide quantitative descriptions to tell how likely the conditions will hold for the executions, which are needed for modern compiler optimizations, and thus has hindered compilers from more aggressive optimizations. This paper addresses this issue by proposing a probabilistic points-to analysis technique to compute the probability of each points-to relationship. Initial experiments are done by incorporating the probabilistic data flow analysis algorithm into SUIF and MachSUIF, and preliminary experimental results show the probability distributions of points-to relationships in several benchmark programs. This work presents a major enhancement for pointer analysis to keep up with modern compiler optimizations.

1 Introduction

There have been considerable efforts on pointer analysis by researchers [1, 4, 6, 7, 8, 12, 17, 18, 20, 22, 24, 25]. They have proposed various algorithms to compute either aliases or points-to relationships at program points. They categorize aliases or points-to relationships into two classes: *must* aliases or *definitely*-points-to relationships, which hold for all executions, and *may* aliases or *possibly*-points-to relationships, which might hold for some executions. However, the information gathered by these algorithms based on this classification does not provide the quantitative descriptions needed for modern compiler optimizations, e.g. data speculation, data prefetching, etc., and thus has hindered compilers from more aggressive optimizations. Neither *may* aliases nor *possibly*-points-to relationships can tell how likely the conditions will hold for the executions, and consequently compilers have to make a conservative guess and assume the conditions hold for all executions. This paper addresses this issue by proposing a *probabilistic points-to analysis* approach to give a quantitative description for each points-to relationship to represent the probability that it holds.

Useful optimizations and transformations can be performed if it is known that certain alias or points-to relationships hold with high or low probabilities. One application is to guide data speculation on advanced architectures. For example, IA-64 [5], which relies on static scheduling, may provide hardware support for speculative motion of loads across possibly aliasing stores. This allows the loads to be executed early but with potentially incorrect values. The hardware in conjunction with software provides a recovery mechanism to recover from any mis-speculation. This feature allows a compiler

[1] The work was supported in part by NSC of Taiwan under grant no. NSC-89-2213-E-019-019, NSC-90-2213-E-019-016, NSC-89-2218-E-007-023, NSC-89-2219-E-007-012, and MOE research excellent project under grant no. 89-E-FA04-1-4.

to generate optimal code by breaking memory dependences, which are often on performance critical paths. However, a mis-speculation on such architecture typically incurs a large recovery penalty. Therefore, to properly guide data speculation, it is important for a compiler to derive the aliasing probability for a pair of data speculation candidates (i.e. a load and a store) and compare an amortized recovery cost with the benefit of a 'good' speculation. A probabilistic memory disambiguation approach was proposed for numeric applications [11]. However, the problem remains open for pointer-induced memory references.

```
foo(int a, int b, int c) {
   int *p; ...
   p = ..
   if( a < b ) { p = &c; }
   st c = ..;
   ld  = *p
}
```

Above is an example of using aliasing probability to guide data speculation. Before the if-clause, p does not point to c, but it does so in the clause. After the if-clause, a store to c is followed by a load from $\star$p. Assume that the load is on a critical path, and hence a compiler wants to schedule the load before the store. However, since $\star$p may alias with c, a compiler would not be able to do so without a support like data speculation (or alternatively some code duplication). The compiler must be able to estimate the aliasing probability between the load and the store and hence how often p points to c. If the amortized recovery cost outweighs the benefit of the shortened critical path after moving the load across the store, this data speculation is unprofitable and should not be performed.

Another application will be optimizations for pointer-based objects on distributed shared memory parallel machines. With affinity analysis [3] and data distribution analysis [13], the advanced analyzer will be able to know or estimate which processor an object is resided in. For task allocations, the optimizer will attempt to assign the processor that owns most of the objects for that task for executions. For programs employing pointer usages, a pointer will be pointing to a set of objects with may-aliases. Therefore, the ability for the analyzer to be able to tell the probability of the aliasing objects for a pointer reference will help the analyzer calculate the amortized amount of objects a processor owns for a task execution.

Probabilistic points-to analysis can be applied to compiler optimizations with memory hierarchies as well. Suppose a pointer, p, points to a set of may-aliasing objects. With the limited amount of the fast memory and working set, only the objects among the aliased objects with high probabilities should be brought into the faster memory of the memory hierarchies. In that case, we should use the information gathered during the probabilistic points-to analysis to have the one with higher probability. In general, the probabilistic points-to information help the compilers to estimate an amortized cost for object placements among memory hierarchies.

This paper proposes a probabilistic points-to analysis approach to address these open issues by giving quantitative descriptions which represent the probabilities that points-to relationships might hold. A probabilistic data flow analysis framework is presented for the probabilistic points-to analysis. In this framework, transfer functions are first computed to identify the probabilities each points-to relationship will be generated and preserved respectively, and then the probabilities of each points-to relationship that might hold at program points will be computed from the transfer functions. This work, to the authors' best knowledge, is the first algorithm for probabilistic points-to analysis. Initial experiments are done by incorporating the intraprocedural probabilistic points-to analysis algorithm into SUIF [9] and MachSUIF [21]. Preliminary experimental results reporting the probability distributions of probabilistic points-to relationships will be given as well.

2 Probabilistic Points-to Analysis

2.1 Problem Specifications

The goal of probabilistic points-to analysis is to compute at every program point the probability of each points-to relationship that might hold. For each points-to relationship, say that p points to v, denoted as a tuple $(p,\, v)$, it computes the probability that pointer p points to v at every program point during the program execution. In other words, a *probabilistic points-to relationship* $(p,\, v,\, P)$ is computed for each points-to relationship $(p,\, v)$ at every program point, where P is the probability that $(p,\, v)$ holds. When P is equal to 1, the points-to relationship $(p,\, v)$ always holds every time the program point is visited. On the other hand, if P is equal to 0, then p will never points to v at this program point. Consequently, if P is between 0 and 1, p will point to v at some instances when the program control reaches the program point, while p will not point to v at other instances.

The Domain The probability P of each probabilistic points-to relationship $(p,\, v,\, P)$ at a program point s can be defined as follows:

$$P = \frac{E(s,\, (p,\, v))}{E(s)}$$

where $E(s)$ is the number of times s is expected to be visited during program execution and $E(s,\, (p,\, v))$ denotes the number of times the points-to relationship $(p,\, v)$ holds at s [15]. Consequently, all the possible values of P for each probabilistic points-to relationship will be the real numbers ranging from 0 to 1. In addition, before the probability P is computed, it is set as $\perp$, and hence the domain of P will be

$$Domain(P) = \{p \mid p \in [0,\, 1] \vee p = \perp\}$$

Program Representations Programs will be represented by control flow graphs (CFGs) whose edges are labeled with a static assigned execution frequency [15, 23] or an actual frequency from profiling. An empty node will be added at the entry of every loop as the *header* node, while an empty node will be augmented as the header node and an empty node as the *join* node.

Meet Operator $\sqcap$ Although the domain of the probabilistic points-to analysis is not a semilattice, the notion of *meet* operations is used to represent the actions of merging values at join nodes. Suppose the probabilities that the points-to relationship $(p,\, v)$ holds at the program point right after B_1 and B_2 in the control flow graph shown in Figure 1 are P_1 and P_2, respectively. In other words, $(p,\, v,\, P_1) \in \mathit{OUT}_{B_1}$ and $(p,\, v,\, P_2) \in \mathit{OUT}_{B_2}$, where OUT_{B_1} and OUT_{B_2} are the sets of probabilistic points-to relationships at the program points right after B_1 and B_2. Then the possibility P that the points-to relationship

$(p,\, v)$ holds at the join node will be

$$P = \frac{P_1 \cdot E(B_1) + P_2 \cdot E(B_2)}{E(B_1) + E(B_2)}$$

where $E(B_1)$ and $E(B_2)$ are the numbers of times B_1 and B_2 are expected to be visited during program execution. Consequently, the probabilistic points-to relationship $(p,\, v,\, P)$ at the join node can be computed by the following the meet operation:

$$(p,\, v,\, P) = E(B_1) \cdot (p,\, v,\, P_1) \sqcap E(B_2) \cdot (p,\, v,\, P_2) = (p,\, v,\, \frac{P_1 \cdot E(B_1) + P_2 \cdot E(B_2)}{E(B_1) + E(B_2)})$$

where the scalar multiplication operator '$\cdot$' over probabilistic points-to relationships is defined as

$$E(B_1) \cdot (p,\, v,\, P_1) \stackrel{def}{=} (p,\, v,\, E(B_1) \cdot P_1)$$

Furthermore, the meet operation on OUT_{B_1} and OUT_{B_2} can be defined:

$$\begin{aligned} OUT_{B_1} \sqcap\ OUT_{B_2} = \{&E(B_1) \cdot (p,\, v,\, P_1) \sqcap E(B_2) \cdot (p,\, v,\, P_2) \mid \\ &(p,\, v,\, P_1) \in OUT_{B_1} \wedge (p,\, v,\, P_2) \in OUT_{B_2}\} \end{aligned}$$

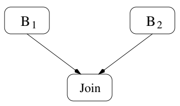


Fig. 1. Meet Operation

$$\begin{aligned} \perp \sqcap \perp &= \perp \\ P_1 \sqcap \perp &= \perp \\ \perp \sqcap P_2 &= \perp \\ P_1 \sqcap P_2 &= P \equiv \frac{P_1 \cdot E(B_1) + P_2 \cdot E(B_2)}{E(B_1) + E(B_2)} \end{aligned}$$

Fig. 2. Rules for $\sqcap$

The meet operator $\sqcap$ merges the possibilities of each probabilistic points-to relationship of different inedges at a join node. If the possibilities of any incoming probabilistic points-to relationships are unknown, i.e. $\perp$, the possibility P at the join node will be computed following the rules for $\sqcap$ operator shown in Figure 2.

2.2 Approach

The probabilistic points-to analysis can be formulated as a data flow framework [4, 10, 14]. The data flow framework for the probabilistic points-to analysis includes *transfer functions*, which formulate the effect of statements on probabilistic points-to relationships. Suppose the sets of probabilistic points-to relationships at the program points right before and after S are IN_S and OUT_S, respectively. Then the effect of S on probabilistic points-to relationships can be represented by the transfer function F_S:

$$OUT_S = F_S\ (IN_S)$$

Transfer Functions For every statement S, a transfer function will be computed for each points-to relationship. Therefore, a transfer function $\langle p,\, v,\, P_{gen}(S),\, P_{prv}(S)\rangle$ will be computed at S for the points-to relationship $(p,\, v)$, where $P_{gen}(S)$ and $P_{prv}(S)$ are defined as follows:

- $P_{gen}(S) \equiv$ probability that $(p,\, v)$ will be generated at S.
- $P_{prv}(S) \equiv$ probability that $(p,\, v)$ will be preserved at S.

where $P_{gen}(S) + P_{prv}(S) \leq 1$. Consequently, the transfer function F_S of statement S consists of the transfer functions for all probabilistic points-to relationships, i.e.

$$F_S = \{\langle p,\, v,\, P_{gen}(S),\, P_{prv}(S)\rangle \,|\, (p,\, v,\, P_{in}(S)) \in IN_S\}$$

Suppose the probability that the points-to relationship $(p,\, v)$ holds at the program point before S is $P_{in}(S)$, i.e. $(p,\, v,\, P_{in}(S)) \in IN_S$. Then the probabilistic points-to relationship $(p,\, v,\, P_{out}(S))$ holds at the program point after S will be

$$(p,\, v,\, P_{out}(S)) = F_S\,((p,\, v,\, P_{in}(S))) = (p,\, v,\, P_{gen}(S) + P_{in}(S) \cdot P_{prv}(S))$$

where $\langle p,\, v,\, P_{gen}(S),\, P_{prv}(S)\rangle \in F_S$.

Default Transfer Functions The transfer functions that are computed in this paper model how the probabilistic points-to relationships are modified by statements. When a probabilistic points-to relationship, say $(p,\, v,\, P)$, will not be modified by a statement S, the transfer function F_S of the statement S will not include the transfer function $\langle p,\, v,\, 0,\, 1\rangle$ for $(p,\, v,\, P)$. Instead, $\langle p,\, v,\, 0,\, 1\rangle$ will be considered as a *default transfer function* of S, and hence will not be explicitly listed. For the rest of the paper, when the transfer functions for any probabilistic points-to relationships are not specified, the default transfer functions will be applied.

Composition of Transfer Functions The composition of transfer functions F_{S_1} and F_{S_2} of two contiguous statements $S_1;\, S_2$ can be denoted as $F_{S_1} \circ F_{S_2}$ and is defined as

$$F_{S_1} \circ F_{S_2}\,(x) \stackrel{def}{=} F_{S_2}(\,F_{S_1}\,(x))$$

Suppose $\langle p,\, v,\, P_{gen}(S_1),\, P_{prv}(S_1)\rangle$ and $\langle p,\, v,\, P_{gen}(S_2),\, P_{prv}(S_2)\rangle$ are the transfer functions of two contiguous statement $S_1; S_2$ for the points-to relationship $(p,\, v)$, respectively. That is, $\langle p,\, v,\, P_{gen}(S_1),\, P_{prv}(S_1)\rangle \in F_{S_1}$ and $\langle p,\, v,\, P_{gen}(S_2),\, P_{prv}(S_2)\rangle \in F_{S_2}$. Then the transfer function of $S_1;\, S_2$ for the points-to relationship $(p,\, v)$ can be computed by the following formula

$$P_{gen}(S_1;\, S_2) = P_{gen}(S_1) \cdot P_{prv}(S_2) + P_{gen}(S_2)$$
$$P_{prv}(S_1;\, S_2) = P_{prv}(S_1) \cdot P_{prv}(S_2)$$

and consequently $\langle p,\, v,\, P_{gen}(S_1;\, S_2),\, P_{prv}(S_1;\, S_2)\rangle \in F_{S_1;\, S_2}$.

Meet Operator $\sqcap$ of Transfer Functions Given transfer functions F_{B_1} and F_{B_2}, the merge of F_{B_1} and F_{B_2} is the transfer function $E(B_1) \cdot F_{B_1} \sqcap E(B_2) \cdot F_{B_2}$ which is defined by

$$(E(B_1) \cdot F_{B_1} \sqcap E(B_2) \cdot F_{B_2})(x) \stackrel{def}{=} E(B_1) \cdot F_{B_1}(x) \sqcap E(B_2) \cdot F_{B_2}(x)$$

Therefore, the corresponding transfer functions for each probabilistic points-to relationship in F_{B_1} and F_{B_2} will be merged. Suppose the transfer functions of F_{B_1} and F_{B_2} for the probabilistic points-to relationship $(p,\, v,\, P)$ are $\langle p,\, v,\, P_{gen}(B_1),\, P_{prv}(B_1)\rangle$ and $\langle p,\, v,\, P_{gen}(B_2),\, P_{prv}(B_2)\rangle$, respectively. Then the merge of the transfer functions F_{B_1} and F_{B_2} for the probabilistic points-to relationship $(p,\, v,\, P)$ will be defined as follows:

$$P_{gen}(E(B_1) \cdot F_{B_1} \sqcap E(B_2) \cdot F_{B_2}) = \frac{P_{gen}(B_1) \cdot E(B_1) + P_{gen}(B_2) \cdot E(B_2)}{E(B_1) + E(B_2)}$$
$$P_{prv}(E(B_1) \cdot F_{B_1} \sqcap E(B_2) \cdot F_{B_2}) = \frac{P_{prv}(B_1) \cdot E(B_1) + P_{prv}(B_2) \cdot E(B_2)}{E(B_1) + E(B_2)}$$

and hence $\langle p,\, v,\, P_{gen}(E(B_1)\cdot F_{B_1} \sqcap E(B_2)\cdot F_{B_2}),\, P_{prv}(E(B_1)\cdot F_{B_1} \sqcap E(B_2)\cdot F_{B_2})\rangle \in E(B_1)\cdot F_{B_1} \sqcap E(B_2)\cdot F_{B_2}$.

Comparison with Bitvector Data Flow Framework The data flow analysis framework proposed in this paper can be called as the *probabilistic data flow analysis framework*, which is adapted from the bitvector data flow analysis framework [14]. As a bitwise transfer function f is computed for every bit of bitvectors with the bitvectors GEN_f and THRU_f in the bitvector data flow analysis framework, where f is defined by bitwise logical operations:

$$f(x) = \mathit{GEN}_f \vee (x \wedge \mathit{THRU}_f)$$

a probabilistic transfer function is computed for every points-to relationship in probabilistic data flow analysis framework. Therefore, the relationships between the transfer functions of these two data flow analysis frameworks are listed in the following table:

	P_{gen}	P_{prv}
GEN_f	1	0
THRU_f	0	1

The main difference is for every condition the probabilistic data flow analysis framework computes a real number ranging from 0 to 1 as the possibility that the condition might hold, whereas the bitvector data flow analysis framework computes a boolean number either true or false to indicate whether the condition might hold or not.

Similar to the composition of probabilistic transfer functions defined in Section 2.2, the composition $f \circ g$ of bitvector transfer functions f and g is defined by

$$f \circ g\,(x) \stackrel{def}{=} g(\,f\,(x))$$

while the computation of $f \circ g$ can be computed by

$$\begin{aligned} \mathit{GEN}_{f\circ g} &= (\mathit{GEN}_f \wedge \mathit{THRU}_g) \vee \mathit{GEN}_g \\ \mathit{THRU}_{f\circ g} &= \mathit{THRU}_f \wedge \mathit{THRU}_g \end{aligned}$$

2.3 Algorithm

The algorithm is adapted from the elimination methods [14, 19]. It performs probabilistic points-to analysis in two phases:

1. Regions are reduced to *abstract CFG nodes* repeatedly to obtain a sequence of *abstract CFGs (ACFGs)* [14]. Their transfer function will be computed during the transformation process and then annotated to the corresponding ACFG nodes.
2. Traverse the sequence of ACFGs to compute the probabilistic points-to relationships at every region using the transfer functions computed in the first phase.

Basic Pointer Assignment Statements Basic pointer assignment statements can be classified into four types: *address-of assignment*, *copy assignment*, *load assignment*, and *store assignment* [18]. For every basic pointer assignment of the first three types, i.e. $p = \cdots$, it first kills all the points-to relationships of the pointer p, before generating any new points-to relationships. Therefore, it will be semantically equivalent if it is preceded immediately by the statement $p = nil$. Similarly, it will be semantically equivalent if any store assignment $\star p = q$ is preceded immediately by the statement $\star p = nil$. Therefore, programs will be normalized such that each basic pointer assignment of the first three types $p = \cdots$ will be preceded by a $p = nil$ while each store assignment $\star p = q$ will be preceded by a $\star p = nil$. Consequently, in addition to the transfer functions of the four basic types of pointer assignment statements, transfer functions of statement types $p = nil$ and $\star p = q$ will be computed as well.

• $S\colon\ p = nil$ Statement $S\colon\ p = nil$ kills all the points-to relationships of p. Therefore, the transfer function F_S of S is

$$F_S = \{\langle p,\, \star,\, 0,\, 0\rangle\}$$

where $\star$ is a wildcard character that means that p points to every variable.

• **Address-of Assignment** $S:\ p = \&q$ Statement $S:\ p = \&q$ generates a points-to relationship $(p,\, q)$. Therefore, the transfer function F_S of S for $(p,\, q)$ will be

$$F_S = \{\langle p,\, q,\, 1,\, 0\rangle\}$$

while the transfer functions for other points-to relationships are default transfer functions $\langle \neg p,\, \star,\, 0,\, 1\rangle$, where $\neg p$ represents the pointers other than p, and consequently are not listed explicitly.

• **Copy Assignment** $S:\ p = q$ The copy assignment $S:\ p = q$ will generate new points-to relationships of p by copying all the points-to relationships of q. Consequently, the transfer function F_S of S will be

$$F_S = \{\langle p,\, v,\, P,\, 0\rangle \mid (q,\, v,\, P) \in IN_S\}$$

• **Load Assignment** $S\colon\ p = \star q$

$$F_S = \{\langle p,\, v,\, \sum_x P_1^x \cdot P_2^x,\, 0\rangle \mid \forall_x\, (q,\, x,\, P_1^x) \in IN_S \wedge (x,\, v,\, P_2^x) \in IN_S\}$$

• $S\colon\ \star p = nil$

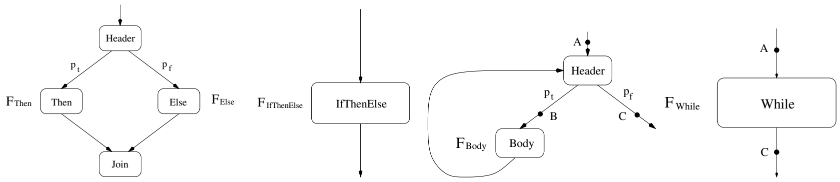
$$F_S = \{\langle x,\, \star,\, 0,\, 1 - P\rangle \mid (p,\, x,\, P) \in IN_S\}$$

• **Store Assignment** $S\colon\ \star p = q$

$$F_S = \{\langle x,\, v,\, P_1 \cdot P_2,\, 0\rangle \mid (p,\, x,\, P_1) \in IN_S \wedge (q,\, v,\, P_2) \in IN_S\}$$

Sequence of ACFG Nodes $S_1;\ S_2;\ \cdots;\ S_n$

$$F_{S_1;\, S_2;\, \cdots;\, S_n} = F_{S_1} \circ F_{S_2} \circ \cdots \circ F_{S_n}$$



(a) *if-then-else* Region (b) ACFG Node (c) *while* Region (d) ACFG Node

Fig. 3. Computing Transfer Functions of *if-then-else* and *while* Regions

if-then-else **Construct** The merge of the $Then$ and $Else$ branches summarizes all paths through the *if-then-else* construct shown in Figure 3(a) and (b).

$$F_{IfThenElse} = p_t \cdot F_{Then} \sqcap p_f \cdot F_{Else}$$

where F_{Then} and F_{Else} are the transfer functions of Then and Else branches respectively while p_t and p_f are the branching probabilities of Then and Else branches respectively and $p_t + p_f = 1$.

Suppose $\langle p, v, P_{gen}(\mathit{Then}), P_{prv}(\mathit{Then})\rangle$ and $\langle p, v, P_{gen}(\mathit{Else}), P_{prv}(\mathit{Else})\rangle$ are the transfer functions for the points-to relationship (p, v) at *Then* and *Else* branches, respectively. Then the transfer function for (p, v) of the *if-then-else* construct will be $\langle p, v, P_{gen}(\mathit{IfThenElse}), P_{prv}(\mathit{IfThenElse})\rangle$, where $P_{gen}(\mathit{IfThenElse})$ and $P_{prv}(\mathit{IfThenElse})$ can be computed by the follow equations:

$$P_{gen}(\mathit{IfThenElse}) = p_t \cdot P_{gen}(\mathit{Then}) + p_f \cdot P_{gen}(\mathit{Else})$$
$$P_{prv}(\mathit{IfThenElse}) = p_t \cdot P_{prv}(\mathit{Then}) + p_f \cdot P_{prv}(\mathit{Else})$$

Once the transfer function of an *if-then-else* construct is computed at the first phase, the sets of probabilistic points-to relationships at program points within the *if-then-else* region can be computed:

$$\begin{aligned} \mathit{OUT}_{IfThenElse} &= F_{IfThenElse}(\mathit{IN}_{IfThenElse}) \\ \mathit{IN}_{Then} &= \mathit{IN}_{IfThenElse} \\ \mathit{IN}_{Else} &= \mathit{IN}_{IfThenElse} \end{aligned}$$

while **Loops** Figure 3(c) and (d) depicts the process of summarizing the region of a *while* loop at the first phase. Since a loop can iterate an arbitrary number of times, its transfer function can be defined by the following equation:

$$F_{While} = \prod_{i=0}^{\infty} (p_f \cdot p_t^i) \cdot (F_{Body})^{\,i}$$

where F_{Body} is the transfer function of the loop body, while the probability of entering the loop from the header is p_t and the probability of leaving the loop is p_f.

Although the above equation merges an infinite number of transfer functions, it can be easily reduced into very simple expressions. Suppose the transfer function of the loop body *Body* for each probabilistic points-to relationship, say (p, v, P), is $\langle p, v, P_g(B), P_p(B)\rangle$, then transfer function $\langle p, v, P_{gen}(\mathit{While}), P_{prv}(\mathit{While})\rangle \in F_{While}$ of the loop for the points-to relationship (p, v) can be computed:

$$\begin{aligned} P_{prv}(\mathit{While}) &= p_f + p_t \cdot P_p(B) \cdot p_f + (p_t \cdot P_p(B))^2 \cdot p_f + \cdots + (p_t \cdot P_p(B))^n \cdot p_f + \cdots \\ &= p_f(1 + p_t \cdot P_p(B) + (p_t \cdot P_p(B))^2 + \cdots + (p_t \cdot P_p(B))^n + \cdots) \\ &= p_f \,/\, (1 - p_t \cdot P_p(B)) \\ P_{gen}(\mathit{While}) &= p_t \cdot P_g(B) \cdot (p_f + p_t \cdot P_p(B) \cdot p_f + (p_t \cdot P_p(B))^2 \cdot p_f + \cdots) + \\ &\quad\; p_t^2 \cdot P_g(B) \cdot (p_f + p_t \cdot P_p(B) \cdot p_f + (p_t \cdot P_p(B))^2 \cdot p_f + \cdots) + \;\cdots + \\ &\quad\; p_t^n \cdot P_g(B) \cdot (p_f + p_t \cdot P_p(B) \cdot p_f + (p_t \cdot P_p(B))^2 \cdot p_f + \cdots) + \;\cdots \\ &= P_g(B) \cdot P_{prv}(\mathit{While}) \cdot (p_t + p_t^2 + \cdots + p_t^n + \cdots) \\ &= p_t \cdot P_g(B) \cdot P_{prv}(\mathit{While}) \,/\, (1 - p_t) = p_t \cdot P_g(B) \cdot p_f \,/\, (1 - p_t \cdot P_p(B)) \cdot (1 - p_t) \\ &= p_t \cdot P_g(B) \,/\, (1 - p_t \cdot P_p(B)) \end{aligned}$$

The first equation computes $P_{prv}(\mathit{While})$ as the summation of the probabilities that $(p,\, v)$ is preserved at the loop exit after zero, one, two, $\cdots$ iterations. The second equation specifies that $(p,\, v)$ will be generated by the loop when it is generated at the first iteration and preserved hereafter or it is generated at the second iteration and preserved hereafter, and so on. It also can be proved that the ranges of $P_{prv}(\mathit{While})$ and $P_{gen}(\mathit{While})$ fall between 0 and 1.

Once the transfer function of a *while* loop is computed at the first phase, the sets of probabilistic points-to relationships at program points within the *while* region can be computed

$$\begin{aligned} \mathit{OUT}_{While} &= F_{While}(\mathit{IN}_{While}) \\ \mathit{IN}_{Body} &= F_{Header}(\mathit{IN}_{While}) \end{aligned}$$

where the function F_{Header} (which summarizes the effects from the program point A to B in Figure 3(c)) is defined as follows:

$$F_{Header} = \prod_{i=0}^{\infty} (p_t \cdot p_t^i) \cdot (F_{Body})^i$$

Suppose the probabilistic points-to relationship $(p,\, v,\, P_{in})$ is in IN_{While}, then the probabilistic points-to relationship $(p,\, v,\, P_{out}) \in \mathit{OUT}_{While}$ can be calculated:

$$\begin{aligned} (p,\, v,\, P_{out}) &= F_{While}((p,\, v,\, P_{in})) \\ &= (p,\, v,\, \frac{p_f \cdot P_{in} + p_t \cdot P_g(B)}{1 - p_t \cdot P_p(B)}) \end{aligned}$$

Similarly, $(p,\, v,\, P_i) \in \mathit{IN}_{Body}$ can be calculated:

$$(p,\, v,\, P_i) = F_{Header}((p,\, v,\, P_{in})) = (p,\, v,\, \frac{p_f \cdot P_{in} + p_t \cdot P_g(B)}{1 - p_t \cdot P_p(B)})$$

***do-while* or *repeat-until* Loops** The transfer function of a *do-while* or *repeat-until* loop can be defined by the following equation:

$$F_{DoWhile} = \prod_{i=1}^{\infty} (p_f \cdot p_t^{i-1}) \cdot (F_{Body})^i$$

***for* Loops** The transfer function of a $for\,(E_{init\_stmt};\, E_{cond.};\, E_{iteration\_stmt})\, Body$ loop can be defined by the following equation:

$$F_{For} = F_{E_{init\_stmt}} \circ \prod_{i=0}^{\infty} (p_f \cdot p_t^i) \cdot (F_{Body} \circ F_{E_{iteration\_stmt}})^i$$

Computing Transitive Transfer Functions The transfer functions that are generated by copy, load, and store statements can be called as *transitive transfer functions* since they depends on the sets of probabilistic points-to relationships right before the statements. These transfer functions complicate the process of the probabilistic points-to analysis especially for loops since the sets of probabilistic points-to relationships, which will be computed at the second phase, must be known before transfer functions are generated at the first phase. This problem can be solved by assigning a symbolic probability for each probabilistic points-to relationship at the entry of a loop body. Consider the program shown in Figure 4. The symbolic probabilities P_1, P_2, P_3, and P_4 are assigned as the probabilities of the probabilistic points-to relationships for $(p,\, v,\, P_1)$, $(p,\, u,\, P_2)$, $(q,\, v,\, P_3)$, and $(q,\, u,\, P_4)$ respectively at the loop entry, i.e. OUT_{S_3}. The statement S_5 : $p = q$; generates a set of transitive transfer functions $F_{S_5} = \{\langle p,\, v,\, P_3,\, 0\rangle\ \langle p,\, u,\, P_4,\, 0\rangle\}$, while S_7 generates a set of transitive transfer functions $F_{S_7} = \{\langle q,\, v,\, P_1,\, 0\rangle\ \langle q,\, u,\, P_2,\, 0\rangle\}$. Consequently, $F_{S_4} = \{\langle p,\, v,\, 0.5P_3,\, 0.5\rangle\langle p,\, u,\, 0.5P_4,\, 0.5\rangle\langle q,\, v,\, 0.5P_1,\, 0.5\rangle\langle q,\, u,\, 0.5P_2,\, 0.5\rangle\}$ will be the transfer function of the loop body. Furthermore, F_{S_3} will be the transfer function of the loop with elements $\langle p,\, v,\, 9P_3/11,\, 2/11\rangle$, $\langle p,\, u,\, 9P_4/11,\, 2/11\rangle$, $\langle q,\, v,\, 9P_1/11,\, 2/11\rangle$, and $\langle q,\, u,\, 9P_2/11,\, 2/11\rangle$.

Program	IN_{S_i}	OUT_{S_i}	F_{S_i}
S_1: p = &v;		$(p,\, v,\, 1)$	$\langle p,\, v,\, 1,\, 0\rangle$
S_2: q = &u;	$(p,\, v,\, 1)$	$(p,\, v,\, 1)\ (q,\, u,\, 1)$	$\langle q,\, u,\, 1,\, 0\rangle$
S_3: while (...) {	$(p,\, v,\, 1)\ (q,\, u,\, 1)$	$(p,\, v,\, P_1)\ (p,\, u,\, P_2)\ (q,\, v,\, P_3)\ (q,\, u,\, P_4)$	$\langle p,\, v,\, 9P_3/11,\, 2/11\rangle\ \langle p,\, u,\, 9P_4/11,\, 2/11\rangle$ $\langle q,\, v,\, 9P_1/11,\, 2/11\rangle\ \langle q,\, u,\, 9P_2/11,\, 2/11\rangle$
S_4: if (...)	$(p,\, v,\, P_1)\ (p,\, u,\, P_2)\ (q,\, v,\, P_3)\ (q,\, u,\, P_4)$	$(p,\, v,\, P_1)\ (p,\, u,\, P_2)\ (q,\, v,\, P_3)\ (q,\, u,\, P_4)$	$\langle p,\, v,\, 0.5P_3,\, 0.5\rangle\ \langle p,\, u,\, 0.5P_4,\, 0.5\rangle$ $\langle q,\, v,\, 0.5P_1,\, 0.5\rangle\ \langle q,\, u,\, 0.5P_2,\, 0.5\rangle$
S_5: p = q;	$(p,\, v,\, P_1)\ (p,\, u,\, P_2)\ (q,\, v,\, P_3)\ (q,\, u,\, P_4)$	$(p,\, v,\, P_3)\ (p,\, u,\, P_4)\ (q,\, v,\, P_3)\ (q,\, u,\, P_4)$	$\langle p,\, v,\, P_3,\, 0\rangle\ \langle p,\, u,\, P_4,\, 0\rangle$
S_6: else			
S_7: q = p;	$(p,\, v,\, P_1)\ (p,\, u,\, P_2)\ (q,\, v,\, P_3)\ (q,\, u,\, P_4)$	$(p,\, v,\, P_1)\ (p,\, u,\, P_2)\ (q,\, v,\, P_1)\ (q,\, u,\, P_2)$	$\langle q,\, v,\, P_1,\, 0\rangle\ \langle q,\, u,\, P_2,\, 0\rangle$
S_8: }			

Fig. 4. Solving Transitive Transfer Functions

The symbolic probabilities P_1, P_2, P_3, and P_4 can be solved by the linear system $OUT_{S_3} = F_{Header}(IN_{S_3})$, where $F_{Header} = \prod_{i=0}^{\infty}(p_t \cdot p_t^i) \cdot (F_{S_4})^{\,i}$. The set of probabilistic points-to relationships at the loop entry will be $OUT_{S_3} = \{(p,\, v,\, 0.55)\ (p,\, u,\, 0.45)\ (q,\, v,\, 0.45)\ (q,\, u,\, 0.55)\}$ if the branching probabilities are $p_t = 0.9$ and $p_f = 0.1$.

3 Experimental Results

3.1 Platform and Benchmarks

A prototype compiler has been implemented upon the SUIF system [9] and CFG library of MachSUIF [21] to perform the intraprocedural probabilistic points-to analysis (PPA). Programs are first transformed from the high-SUIF format to the low-SUIF format by SUIF and then represented by CFGs using the CFG library of MachSUIF. The compiler will then traverse the CFGs to compute the probability of each probabilistic points-to relationship at every program point. This section will present the preliminary experimental results of this implementation.

Several applications have been chosen as the benchmarks and a procedure of each benchmark program has been instrumented, as listed in Table 1. These benchmark programs will then be executed to gather the detailed points-to information of these procedures at runtime. The runtime results will be compared with the following three variations of points-to analysis:

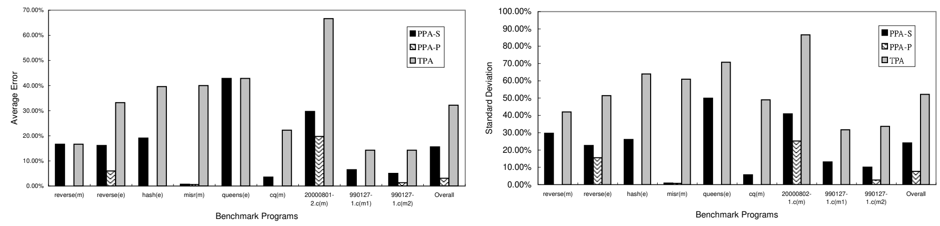
- Probabilistic points-to analysis based on static probabilities (PPA-S)
  A probability will be assigned to each outgoing edge of CFG, say $p_t = p_f = 0.5$ for *if* statements and $p_t = 0.9$ and $p_f = 0.1$ for loops, and the probabilistic points-to analysis algorithm described in Section 2 will be executed based on these edge probabilities.

Program	Procedure	Description
reverse	*InsertElement*	A small program that builds a binary tree and then recursively swaps the left and right children of each node. (McGill [8])
hash	*AddToTable*	A program builds a hash table (McGill)
misr	*create_link_list*	A program creates and uses linked list. (McGill)
queens	*find*	A program that finds solutions to the eight-queens chess problem.
cq	*s81*	A test program from lcc-4.0 testsuite.
20000108-2.c	*test*	Test programs from gcc-3.0 snapshot testsuite (from directory: c-torture/execute/)
990127-1.c	*main*	

Table 1. Benchmark Programs and Selected Procedures

- Probabilistic points-to analysis based on profiling information (PPA-P)
  The TCOVSUIF profiling tool [2] is used to gather loop counts and branch frequencies, and probabilistic points-to analysis will be performed based on the profiling information to compute the probabilities of points-to relationships in these selected procedures.
- Traditional points-to analysis (TPA)
  The probability of each points-to relationship is assumed to be 1.

The preciseness of these points-to analysis methods respective to the runtime results will be compared by the statistics *average error* $\xi = \sum_{i=1}^{n} |P_{estimated}(i) - P_{runtime}(i)|/n$ and *standard deviation* $\sigma = \sqrt{\sum_{i=1}^{n} (P_{estimated}(i) - P_{runtime}(i))^2/(n-1)}$.



(a) Average Errors (b) Standard Deviations

Fig. 5. Experimental Results

3.2 Results

Figure 5(a) shows the average errors of estimated probabilities of points-to relationships by these methods respective to the profiled frequencies at runtime. At each chosen program point, the estimated probabilities of all points-to relationships will be compared with the profiled probabilities. For example, *reverse(m)* in Figure 5(a) is computed at the middle point (randomly selected) of the instrumented procedure in the program *reverse*, while *reverse(e)* compares the errors at the end of the procedure in *reverse*. Similarly, Figure 5(b) depicts the standard deviations of these points-to analysis techniques respective to the profiled frequencies at runtime. Table 2 summarizes the average errors and standard deviations depicted in Figure 5(a) and Figure 5(b) in a tabular format.

The above figures and table show that probabilistic points-to analysis approach can estimate how likely each points-to relationship would hold with relatively small errors.

Even with statically assigned edge probabilities, the average error of estimated probabilities by PPA-S compared to the runtime frequencies is about 15%. With the aid of edge profiling information, PPA-P reduces the average error down to 3%. Furthermore, the 7.69% standard deviation of PPA-P demonstates that almost all of estimated probabilities are quite accurate, with errors less than 7.69%.

Programs	Average Errors			Standard Deviations		
	PPA-S	PPA-P	TPA	PPA-S	PPA-P	TPA
reverse(m)	16.67%	0%	16.67%	29.70%	0%	42.01%
reverse(e)	16.16%	5.99%	33.19%	22.60%	15.55%	51.46%
hash(e)	19.10%	0%	39.58%	26.11%	0%	63.96%
misr(m)	0.72%	0.61%	40%	0.86%	0.71%	60.91%
queens(e)	42.86%	0.0002%	42.86%	50%	0.0004%	70.71%
cq(m)	3.56%	0.0174%	22.22%	5.66%	0.0277%	49.01%
20000801-2.c(m)	29.7%	19.75%	66.67%	40.91%	25.19%	86.60%
990127-1.c(m1)	6.49%	0%	14.29%	13.12%	0%	31.71%
990127-1.c(m2)	5%	1.30%	14.29%	10.10%	2.62%	33.67%
Overall	15.58%	3.07%	32.19%	24.11%	7.69%	52.20%

Probability Range	PPA-S	PPA-P	PPA-S	PPA-P
0%∼10%	83.33%	100%	80%	96.30%
10%∼20%	0%	0%		
20%∼30%	0%	50%	0%	33.33%
30%∼40%	0%	0%		
40%∼50%	0%	50%	0%	100%
50%∼60%	0%	50%		
60%∼70%	0%	100%	10%	100%
70%∼80%	0%	100%		
80%∼90%	0%	100%	98.21%	100%
90%∼100%	100%	100%		

Table 2. Average Errors and Standard Deviations **Table 3.** Accuracy of Estimated Probabilities

This result is significant since most compiler optimizations can benefit from the ability to determine if points-to relationships hold with high or low probabilities. For instance, data speculation can be performed on reads and writes with low possibilities of conflicts to avoid costly mis-speculation penalties. Let $Points\text{-}to_{PPA}(l\%{\sim}h\%)$ be the set of points-to relationships that are estimated by PPA to hold with the probabilities within the range $l\%{\sim}h\%$, and $Points\text{-}to_{Runtime}(l\%{\sim}h\%)$ be the set of points-to relationships with runtime-profiled probabilities within the range $l\%{\sim}h\%$ and are also in the set $Points\text{-}to_{PPA}(l\%{\sim}h\%)$. Then the *accuracy within the probability range* $l\%{\sim}h\%$ of PPA is defined as the ratio of the size of the sets $Points\text{-}to_{Runtime}(l\%{\sim}h\%)$ over the size of $Points\text{-}to_{PPA}(l\%{\sim}h\%)$, i.e. $|Points\text{-}to_{Runtime}(l\%{\sim}h\%)|$ / $|Points\text{-}to_{PPA}(l\%{\sim}h\%)|$. Table 3 presents the accuracy of PPA-S and PPA-P within different probability ranges based on the above definition. The first section of Table 3 shows the accuracy of PPA-S and PPA-P in the probability range 0%∼10% are 83.33 and 100% respectively, while the accuracy of both PPA-S and PPA-P in the range 90%∼100% are 100%. If the interval of the probability ranges is extended to 20%, the accuracy of PPA-S and PPA-P in the probability range 0%∼20% is 80% and 96.30% respectively, and while the accuracy of both PPA-S and PPA-P in the range 80%∼100% is 98.21% and 100%, respectively, as shown in the second section of Table 3. This result demonstrates that the probabilistic points-to analysis can identify the points-to relationships with high or low probabilities with very high accuracy.

Table 4 lists the distributions of probabilities of all points-to relationships estimated by points-to analysis techniques and profiled at runtime. For most of the benchmarks, the probability distributions of PPA-P are the same as the profiled probability distributions. It shows that the probabilities estimated by the probabilistic points-to analysis are very accurate.

program	analysis method	0% ~ 10%	10% ~ 20%	20% ~ 30%	30% ~ 40%	40% ~ 50%	50% ~ 60%	60% ~ 70%	70% ~ 80%	80% ~ 90%	90% ~ 100%
reverse(m)	Runtime	16.7%	0	0	0	0	0	0	0	0	83.3%
	PPA-P	16.7%	0	0	0	0	0	0	0	0	83.3%
	PPA-S	0	0	0	0	0	33.3%	0	0	0	66.7%
	TPA	0	0	0	0	0	0	0	0	0	100%
reverse(e)	Runtime	19%	0	0	0	9.5%	19%	0	0	0	52.5%
	PPA-P	19%	0	0	0	9.5%	9.5%	0	0	0	62%
	PPA-S	0	14.3%	14.3%	0	0	0	0	14.3%	14.3%	42.8%
	TPA	0	0	0	0	0	0	0	0	0	100%
hash(m)	Runtime	39.1%	0	0	0	0	0	0	0	0	60.9%
	PPA-P	39.1%	0	0	0	0	0	0	0	0	60.9%
	PPA-S	13%	0	0	26.1%	0	0	26.1%	0	0	34.8%
	TPA	0	0	0	0	0	0	0	0	0	100%
misr(m)	Runtime	40%	0	0	0	0	0	0	0	0	60%
	PPA-P	40%	0	0	0	0	0	0	0	0	60%
	PPA-S	40%	0	0	0	0	0	0	0	40%	20%
	TPA	0	0	0	0	0	0	0	0	0	100%
queens(e)	Runtime	42.9%	0	0	0	0	0	0	0	0	57.1%
	PPA-P	42.9%	0	0	0	0	0	0	0	0	57.1%
	PPA-S	0	0	0	0	85.7%	0	0	0	0	14.3%
	TPA	0	0	0	0	0	0	0	0	0	100%
cq(m)	Runtime	22.2%	0	0	0	0	0	0	0	0	77.8%
	PPA-P	22.2%	0	0	0	0	0	0	0	0	77.8%
	PPA-S	22.2%	0	0	0	0	0	0	0	22.2%	55.6%
	TPA	0	0	0	0	0	0	0	0	0	100%
20000801-2.c(m)	Runtime	33.3%	0	0	0	66.7%	0	0	0	0	0
	PPA-P	0	0	33.3%	33.3%	33.4%	0	0	0	0	0
	PPA-S	100%	0	0	0	0	0	0	0	0	0
	TPA	0	0	0	0	0	0	0	0	0	100%
990127-1.c(m1)	Runtime	0	14.3%	0	0	0	0	0	14.3%	0	71.4%
	PPA-P	0	0	14.3%	0	0	0	0	14.3%	0	71.4%
	PPA-S	0	0	0	14.3%	0	0	14.3%	0	0	71.4%
	TPA	0	0	0	0	0	0	0	0	0	100%
990127-1.c(m2)	Runtime	0	0	14.3%	0	0	0	0	14.3%	0	71.4%
	PPA-P	0	0	14.3%	0	0	0	0	14.3%	0	71.4%
	PPA-S	0	0	0	0	28.6%	0	0	0	0	71.4%
	TPA	0	0	0	0	0	0	0	0	0	100%

Table 4. Distributions of probabilities of Points-to Relationships

3.3 Discussion

PPA-P can accurately estimate probabilities of points-to relations of most selected procedures in the benchmark programs with errors less than 1%. However, the errors of the programs *reverse* and *20000108-2.c* are quite significant compared to the errors of the other programs. The reason is that the current implementation can not handle heap and recursive data structures properly. Heap locations are named after the program points where they are allocated. This naming scheme can not provide enough information for probabilistic points-to analysis to make accurate estimations. It will be improved in the future implementation.

4 Related Work

There have been considerable efforts on pointer analysis by researchers [1, 4, 6, 7, 8, 12, 17, 18, 20, 22, 24, 25]. The proposed techniques compute at program points either aliases or points-to relationships. They categorize aliases or points-to relationships into two classes: *must* aliases or *definitely*-points-to relationships, which hold for executions, and *may*-aliases or *possibly*-points-to relationships, which hold for at least one execution. However, they can not tell which *may*-aliases or possibly-points-to relationships hold for the most of executions and which for only few executions. Such information is crucial for compilers to determine if certain optimizations and transformations will be beneficial. The probabilistic points-to analysis approach proposed in this paper is the first algorithm to compute such information.

The most closely related work is the *data flow frequency analysis* proposed by Ramalingam [15]. It provides a theoretical foundation for data flow frequency analysis, which computes at program points the expected number of times that certain conditions might hold. The probabilistic points-to analysis approach proposed in this paper is built upon the probabilistic data flow analysis framework, which is adapted from Ramalingam's data flow frequency analysis. However, this paper focuses on points-to analysis, which is a complicated issue because of the dynamic associations property of pointers. Extra cares are needed for probabilistic points-to analysis with the recent establishments for foundations of probabilistic data flow equations. Furthermore, this technique solves the probabilistic data flow analysis problem on CFGs, eliminating the overhead of generating the *exploded graphs* [16].

In the work related to data speculations for modern computer architectures, such as IA-64 [5], Ju et al. [11] gives a probabilistic memory disambiguation approach for array analysis and optimizations. However, the problem remains open for pointer-induced memory references. This work tries to provide a solution to fill-in the open areas. In the work related to compiler optimizations for pointer-based programs on distributed shared-memory parallel machines, affinity analysis [3] and data distribution analysis [13] are currently able to estimate which processor an object is resided in. For programs with pointer usages, a pointer will be pointing to a set of objects with may-aliases. In this case, our analyzer can be integrated with the conventional affinity analyzer, and the integrated scheme can calculate the amortized amount of objects a processor owns for a task execution. Thus it will help program optimizations.

References

1. Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*, Columbus, Ohio, August 1995.
2. Tim Callahan and John Wawrzynek. Simple profiling system for suif. In *Proceedings of the First SUIF Compiler Workshop*, January 1996.
3. M. C. Carlisle and A. Rogers. Software caching and computation migration in olden. In *Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, pages 29–39, July 1995.
4. Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, South Carolina, January 1993.
5. Intel Corporation. *IA-64 Application Developer's Architecture Guide*. 1999.
6. Manuvir Das. Unification-based pointer analysis with directional assignments. *SIGPLAN Notices*, 35(5):35–46, May 2000. *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*.
7. Alain Deutsch. Interprocedural May-Alias analysis for pointers: Beyond k-limiting. *SIGPLAN Notices*, 29(6):230–241, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
8. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural Points-to analysis in the presence of function pointers. *SIGPLAN Notices*, 29(6):242–256, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
9. The Stanford SUIF Compiler Group. The suif library. Technical report, Stanford University, 1995.
10. M.S. Hecht. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.

11. R. D.C. Ju, J.-F. Collard, and K. Oukbir. Probabilistic memory disambiguation and its application to data speculation. In *Proceedings of the 3rd Workshop on Interaction between Compilers and Computer Architecture*, Oct 1998.
12. William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *SIGPLAN Notices*, 27(7):235–248, July 1992. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*.
13. Jenq Kuen Lee, Dan Ho, and Yue-Chee Chuang. Data distribution analysis and optimization for pointer-based distrib uted programs. In *Proceedings of the 26th International Conference on Parallel Proces sing (ICPP)*, Bloomingdale, IL, August 1997.
14. Steven S. Muchnick. *Advanced Compiler Design & Implementation*. Morgen Kaufmann, 1997.
15. G. Ramalingam. Data flow frequency analysis. *SIGPLAN Notices*, 31(5):267–277, May 1996. *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*.
16. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, San Francisco, California, January 1995.
17. Erik Ruf. Context-insensitive alias analysis reconsidered. *SIGPLAN Notices*, 30(6):13–22, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.
18. Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. *SIGPLAN Notices*, 34(5):77–90, May 1999. *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*.
19. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.
20. Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of POPL '97: 24nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997.
21. Michael D. Smith. The suif machine library. Technical report, Division of of Engineering and Applied Science, Harvard University, March 1998.
22. Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of POPL '96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41, St. Petersburg Beach, Florida, January 1996.
23. Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. *SIGPLAN Notices*, 29(6):85–96, June 1994. *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*.
24. Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for C programs. *SIGPLAN Notices*, 30(6):1–12, June 1995. *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation*.
25. Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. *SIGPLAN Notices*, 34(5):91–103, May 1999. *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*.