Interprocedural Probabilistic Pointer Analysis

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

[†]Department of Computer Science
National Tsing Hua University
Hsinchu 300
Taiwan

[‡]Department of Computer Science
National Taiwan Ocean University
Keelung 202
Taiwan

[§]Microprocessor Research Lab
Intel Corporation
Santa Clara, CA 95052
U.S.A.

Abstract

When performing aggressive optimizations and parallelization to exploit features of advanced architectures, optimizing and parallelizing compilers need to quantitatively assess the profitability of any transformations in order to achieve high performance. Useful optimizations and parallelization can be performed if it is known that certain points-to relationships would hold with high or low probabilities. For instance, if the probabilities are low, a compiler could transform programs to perform data speculation or partition iterations into threads in speculative multithreading, or it would avoid conducting code specialization. Consequently, it is essential for compilers to incorporate pointer analysis techniques that can estimate the possibility for every points-to relationship that it would hold during the execution. However, conventional pointer analysis techniques do not provide such quantitative descriptions and thus hinder compilers from more aggressive optimizations, such as thread partitioning in speculative multithreading, data speculations, code specialization, etc. This paper addresses this issue by proposing a probabilistic points-to analysis technique to compute the probability of every points-to relationship at each program point. A context-sensitive interprocedural algorithm has been implemented based on the iterative data flow analysis framework, and been incorporated into SUIF and MachSUIF. Experimental results show this technique can estimate the probabilities of points-to relationships in benchmark programs with reasonable small errors, about 4.6% in average. Furthermore, the current implementation cannot disambiguate heap and array elements. The errors will be further significantly reduced when the future implementation incorporates techniques to disambiguate heap and array elements.

Corresponding Author:

Jenq Kuen Lee
Department of Computer Science
National Tsing Hua University
Hsinchu 300
Taiwan
Email: jklee@cs.nthu.edu.tw
Tel: +886-3-5715131 ext. 3519
Fax: +886-3-5723694

I. Introduction

There have been considerable efforts on pointer analysis by researchers [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. Some of them have proposed various algorithms to perform points-to analysis, i.e. to compute all possible points-to relationships at every program point. They categorize points-to relationships into two classes: *definitely*-points-to relationships, which hold for all executions, and *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, code specialization etc., and thus has hindered compilers from more aggressive optimizations. *Possibly*-points-to relationships cannot 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.

When performing aggressive optimizations and parallelization to exploit features of advanced architectures, optimizing and parallelizing compilers need to quantitatively assess the profitability of any transformations in order to achieve high performance. The probabilistic points-to analysis technique is designed to provide the quantitative information to let the compilers formulate the cost functions of transformations, many of which will show a profit when certain points-to relationships hold with high or low probabilities. Therefore, compilers can determine whether it is beneficial to perform optimizations and parallelization if they can differentiate the points-to relationships within these ranges from the rest. Experimental results show there are many opportunities for optimizations and parallelization since over 80% of the points-to relationships in the benchmark programs hold within the probability ranges 0%$\sim$10% or 90%$\sim$100%.

One application is to guide data speculation on advanced architectures. For example, IA-64 [13], which relies on static scheduling, provides 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 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 [14]. 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; }
   c = ...;     /* st */
   ... = *p     /* ld */
}
```

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 if-clause. After the if-clause, a store to c is followed by a load from ⋆p. Assume that the load is on a critical path, and hence a compiler wants to schedule the load before the store. However, since ⋆p may alias with c, a compiler would not be able to do so without a support like data speculation (alternatively comparing the base addresses of the load and store or 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.

Probabilistic pointer analysis is also important to thread partitioning in speculative multithreading [15], [16], [17], where potentially dependent threads from a single application are running on parallel hardware. Although the hardware ensures correctness through recovery when the dependences between threads actually occur at run time, the key to achieve high performance under this model is to have minimal dependence violations during the execution. A compiler has the challenging task to estimate the likelihood of dependences so that it can maximize the number of threads for parallel execution but minimize the chance of dependence violations between threads. Therefore, probabilistic pointer analysis is an important technology to guide thread partitioning for general applications in the speculative multithreading model. It has been shown that a compiler can achieve speedups by executing speculative threads when the possibilities of conflicts are low and can avoid slowdown by turning off thread speculation if the possibilities are high [18].

Probabilistic pointer analysis can also be useful in code specialization [19]. Many program variables are observed to be ”quasi-invariant” at run time, where the distribution of the values is skewed with a small number of values occurring most of the time. Knowledge of such frequently occurring values can be exploited by a compiler to generate code that optimizes for the common cases without sacrificing the ability to handle the general case. People have resorted to value or expression profiling to recognize such quasi-invariant behavior. Probabilistic pointer analysis can eliminate the need of some instrumentation points in the expensive value profiling when the recognition of quasi-invariance is hindered by memory aliasing. In addition to the above applications, probabilistic pointer analysis can facilitate optimizations for pointer-based objects on distributed shared memory parallel machines and compiler optimizations with memory hierarchies as well.

This paper presents a probabilistic points-to analysis approach as the groundwork for the aggressive optimizations and parallelization mentioned above. This approach enhances the existing pointer analysis techniques by giving quantitative descriptions which represent the probabilities that points-to relationships might hold. A context-sensitive interprocedural algorithm has been developed based on the iterative data flow analysis framework, and been implemented by incorporating SUIF [20] and MachSUIF [21]. Experimental results show this technique can estimate the probabilities of points-to relationships in benchmark programs with reasonably small errors, about 4.6% in average. Furthermore, the weighted average errors based on execution frequencies are 3.68%.

The remainder of the paper is organized as follows. Section II specifies the problem and Section III presents the algorithm for probabilistic points-to analysis to solve the problem. Section IV extends the algorithm to handle interprocedural analysis. Experimental results will be presented in Section V and the related work is compared in Section VI. Section VII concludes this paper.

II. Background

A. Problem Specifications

The goal of probabilistic points-to analysis is to compute the probability of each points-to relationship that might hold at every program point. For each points-to relationship, say that p points to v (denoted as a tuple $\langle p, v\rangle$), it computes the probability that pointer p points to v at every program point s during the program execution. In other words, a *probability function* $\mathcal{P}(s, \langle p, v\rangle)$ is computed for each points-to relationship $\langle p, v\rangle$ at every program point s by the following equation

$$\mathcal{P}(s, \langle p, v\rangle) \overset{def}{=} \frac{E(s, \langle p, v\rangle)}{E(s)}$$

where $E(s)$ is the number of times s is expected to be visited during program execution and $E(s, \langle p, v\rangle)$ denotes the number of times the points-to relationship $\langle p, v\rangle$ holds at s [22]. When $\mathcal{P}(s, \langle p, v\rangle)$ is equal to 1, the points-to relationship $\langle p, v\rangle$ always holds every time the program point s is visited. On the other hand, if it is equal to 0, then p will never point to v at s.

Consequently, if the probability is between 0 and 1, p might point to v at some instances when the program control reaches s, while p might not point to v at other instances.

Points-to relationships at every program point can be visualized by constructing a *points-to graph* [5], [8], where each points-to relationship is represented by a directed edge from one node to another node. Consequently, computing the probability function for every points-to relationship is equivalent to assigning a probability to the corresponding edge of the graph. Furthermore, all the possible values of the probability function for each probabilistic points-to relationship will be the real numbers ranging from 0 to 1, according to the above equation.

The probability function can be overloaded to compute the possibilities for the set of points-to relationships at every program point, if the set is represented by a vector. Specifically, if A is the set of points-to relationships at s, the probability function for A at s will be

$$\begin{aligned}\mathcal{P}(s,\, A) &\overset{\mathit{def}}{=} \{\mathcal{P}(s,\, \langle p,\, v\rangle) \,|\, \langle p,\, v\rangle \in A\}\\ &= \{\frac{E(s,\, \langle p,\, v\rangle)}{E(s)} \,|\, \langle p,\, v\rangle \in A\}\end{aligned}$$

Such an overloaded probability function returns a vector, the ith element of which contains the result of the probability function for the ith points-to relationship in A.

B. Location Sets and Program Representations

Memory locations are represented by *location sets* [11], each of which is a triple of the form $(b,\, f,\, s)$ where the base b is the name for a block of memory, f is an offset within that block, and s is the stride. A location $(b,\, f,\, s)$ represents the set of locations $\{f + i \times s \,|\, i \in N\}$ within block b. For instance, $(p,\, 0,\, 0)$ denotes a variable p, $(r,\, f,\, 0)$ refers to a field f in a record r, and $(a,\, 0,\, w)$ represents a set of array elements $a[i]$ (w is the size of each element).

Programs will be represented by control flow graphs (CFGs) whose edges are labeled with a static assigned execution frequency [22] or an actual frequency from profiling. These frequencies on the CFG edges can be easily converted into branching probabilities of conditionals and loops, and vice versa [23]. In addition,an empty node will be added at the entry of every loop as the *header* node.

C. Normalization Assumptions

Programs will be normalized such that each pointer assignment statement is one of the four basic pointer assignment statements listed in the following table [8]:

Address-of Assignment	$p = \&q$
Copy Assignment	$p = q$
Load Assignment	$p = \star q$
Store Assignment	$\star p = q$

In addition, every of the first three basic pointer assignments, i.e. statements with the form $p = \cdots$, will be preceded by a *nullifying assignment* of the form $p = nil$. Similarly, every store assignment statement will be preceded immediately by an *indirect nullifying assignment* with the form $\star p = nil$.

III. Probabilistic Points-to Analysis

The conventional points-to analysis can be formulated as a data flow framework [2], [24], [25]. The data flow framework includes *transfer functions*, which formulate the effect of statements on points-to relationships. Suppose the sets of points-to relationships at the program points right before and after S, i.e. S_{in} and S_{out}, are IN_S and OUT_S, respectively. Then the effect of S on points-to relationships can be represented by the transfer function F_S:

$$\mathit{OUT}_S = F_S\,(\mathit{IN}_S)$$

The probabilistic points-to analysis can be formulated as a data flow framework as well. If the sets IN_S and OUT_S are represented by vectors, the vector of probability functions of the points-to relationships in OUT_S can be computed by an overloaded transfer function F_S:

$$\begin{aligned} \mathcal{P}(S_{out},\, \mathit{OUT}_S) &= F_S(\mathcal{P}(S_{in},\, \mathit{IN}_S)) \\ &= F_S(\{\mathcal{P}(S_{in},\, \langle p,\, v\rangle) \mid \langle p,\, v\rangle \in \mathit{IN}_S\}) \end{aligned}$$

F_S returns a vector with the ith element representing the probability function of the ith points-to relationship in OUT_S.

A. Basic Pointer Assignment Statements

Figure 1 summarizes the process of computing the set of points-to relationships OUT_S at the end of every basic pointer assignment statement S by the conventional points-to analysis techniques [5], [8], [11]. Every points-to relationship is associated with an attribute rel, which can be either $true$ or $false$, to specify that the relationship is either a definitely-points-to relationship or possibly-points-to relationship. The transfer functions in Figure 1 are presented under the assumption of normalization. That is, every pointer assignment statement will be preceded by a nullifying assignment, e.g. $S1 : p = q$ will be transformed into the following two contiguous statements $S1' : p = nil$; $S1 : p = q$. Consequently, the set KILL_{S1} of the following data flow computation

$$\mathit{OUT}_{S1} = F_{S1}(\mathit{IN}_{S1}) = \mathit{GEN}_{S1} \cup (\mathit{IN}_{S1} - \mathit{KILL}_{S1})$$

will be handled by $S1'$ while GEN_{S1} is generated by $S1$.

S	$\mathit{OUT}_S = F_S(\mathit{IN}_S)$
$p = \&q$	$\mathit{IN}_S \cup \{(\langle p,\, q\rangle,\, true)\}$
$p = q$	$\mathit{IN}_S \cup \{(\langle p,\, v\rangle,\, rel) \mid (\langle q,\, v\rangle,\, rel) \in \mathit{IN}_S\} \quad rel \in \{true,\, false\}$
$p = \star q$	$\mathit{IN}_S \cup \{(\langle p,\, v\rangle, \bigvee_x (rel_1^x \wedge rel_2^x)) \mid \forall_x \, ((\langle q,\, x\rangle,\, rel_1^x),\, (\langle x,\, v\rangle,\, rel_2^x) \in \mathit{IN}_S)\}$
$\star x = q$	$\mathit{IN}_S \cup \{(\langle p,\, v\rangle,\, rel_1 \wedge rel_2) \mid (\langle x,\, p\rangle,\, rel_1),\, (\langle q,\, v\rangle,\, rel_2) \in \mathit{IN}_S\}$
$p = nil$	$\mathit{IN}_S - \{(\langle p,\, v\rangle,\, rel) \in \mathit{IN}_S\}$
$\star x = nil$	$\mathit{IN}_S - \{(\langle p,\, v\rangle,\, rel) \mid (\langle x,\, p\rangle,\, rel),\, (\langle p,\, v\rangle,\, rel) \in \mathit{IN}_S\}$
	$\cup\ \{(\langle p,\, v\rangle,\, false) \mid (\langle x,\, p\rangle,\, false),\, (\langle p,\, v\rangle,\, rel) \in \mathit{IN}_S\}$

Fig. 1. Computing the Set of Points-to Relationships

In contrast to simply associating an attribute to distinguish definitely-points-to relationships from possibly-points-to relationships, the probabilistic points-to analysis computes a probability function for every points-to relationship $\langle p,\, v\rangle$ at each program point s to estimate the possibility that $\langle p,\, v\rangle$ would hold every time s is visited at runtime. Figure 2 presents the formula to compute the probability function $\mathcal{P}(S_{out},\, \langle p,\, v\rangle)$ of every points-to relationship $\langle p,\, v\rangle \in \mathit{OUT}_S$ at exit of statement S. Note that the table only shows the probability functions of the points-to relationships that are affected by S. The results of the probability functions at S_{out} for the points-to relationships that are not affected by S will be the same as those at S_{in}. That is, if $\langle x,\, y\rangle$ is not influenced by S, then $\mathcal{P}(S_{out},\, \langle x,\, y\rangle) = \mathcal{P}(S_{in},\, \langle x,\, y\rangle)$.

S	$\mathcal{P}(S_{out},\, \langle p,\, v\rangle)$
$p = \&q$	$\begin{cases} 1 & q \equiv v \\ 0 & \textit{otherwise} \end{cases}$
$p = q$	$\mathcal{P}(S_{in},\, \langle q,\, v\rangle)$
$p = \star q$	$\sum_x \mathcal{P}(S_{in},\, \langle q,\, x\rangle) \times \mathcal{P}(S_{in},\, \langle x,\, v\rangle)$
$\star x = q$	$\mathcal{P}(S_{in},\, \langle x,\, p\rangle) \times \mathcal{P}(S_{in},\, \langle q,\, v\rangle) + \mathcal{P}(S_{in},\, \langle p,\, v\rangle)$
$p = \mathit{nil}$	0
$\star x = \mathit{nil}$	$(1 - \mathcal{P}(S_{in},\, \langle x,\, p\rangle)) \times \mathcal{P}(S_{in},\, \langle p,\, v\rangle)$

Fig. 2. Computing the Probability Functions

1) Address-of Assignment $S\colon\ p = \&q$: Statement $S\colon\ p = \&q$ introduces a definitely-points-to relationship $\langle p,\, q\rangle$, i.e. $\langle p,\, q\rangle$ will hold at S_{out} with a probability of 1. Consequently, the probability function $\mathcal{P}(S_{out},\, \langle p,\, v\rangle)$ will be 1 if $q \equiv v$, and 0 otherwise.

2) Copy Assignment $S:\ p = q$: According to Figure 1, a copy assignment $S:\ p = q$ will generate new points-to relationships of p by copying all the points-to relationships of q. Therefore, if $\langle q,\, v\rangle \in \mathit{IN}_S$ with a probability function $\mathcal{P}(S_{in},\, \langle q,\, v\rangle)$, then the probability function of $\langle p,\, v\rangle$ will be $\mathcal{P}(S_{in},\, \langle q,\, v\rangle)$.

3) Load Assignment $S\colon\ p = \star q$: Pointer p will point to v after S is executed when q points to some other pointer x and x points to v before S is executed. As a result, the probability that p points to v at S_{out} will be $P_1 \times P_2$ if q points to x and x points to v with probabilities of P_1 and P_2, respectively. Since there might be more than one such pointer x at S_{in}, the probability function of $\langle p,\, v\rangle$ at S_{out} should be $\sum_x \mathcal{P}(S_{in},\, \langle q,\, x\rangle) \times \mathcal{P}(S_{in},\, \langle x,\, v\rangle)$.

4) Store Assignment $S:\ \star x = q$: Pointer p will point to v at S_{out} when x points to p and q points to v before S is executed. Another possibility would be that p points to v before S.

Therefore, the probability function of $\langle p, v\rangle$ at S_{out} will be $\mathcal{P}(S_{in}, \langle x, p\rangle) \times \mathcal{P}(S_{in}, \langle q, v\rangle) + \mathcal{P}(S_{in}, \langle p, v\rangle)$.

5) Nullifying Assignment $S\colon\ p = nil$: A nullifying statement $S\colon\ p = nil$ kills all points-to relationships of p, and hence p will not point to v after S is executed. That is, $\mathcal{P}(S_{out}, \langle p, v\rangle) = 0$.

6) Indirect Nullifying Assignment $S\colon\ \star x = nil$: If x points to p at S_{in} with the probability function $\mathcal{P}(S_{in}, \langle x, p\rangle) = 1$, then S will kill all points-to relationships of p. On the other hand, if $\mathcal{P}(S_{in}, \langle x, p\rangle) = 0$, no points-to relationships of p will be killed. In other words, the possibility of any points-to relationships of p will be reduced by a factor of $1-\mathcal{P}(S_{in}, \langle x, p\rangle)$. Consequently, if p points to v with a probability function $\mathcal{P}(S_{in}, \langle p, v\rangle)$ at S_{in}, the probability function at S_{out} will be $(1 - \mathcal{P}(S_{in}, \langle x, p\rangle)) \times \mathcal{P}(S_{in}, \langle p, v\rangle)$.

B. 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 probability functions at join nodes. Suppose the probability functions of the points-to relationship $\langle p, v\rangle$ at the program points $B1_{out}$ and $B2_{out}$ right after $B1$ and $B2$ in the control flow graph shown in Figure 3 are $\mathcal{P}(B1_{out}, \langle p, v\rangle)$ and $\mathcal{P}(B2_{out}, \langle p, v\rangle)$, respectively. Then the possibility function of the points-to relationship $\langle p, v\rangle$ at the join node will be

$$\begin{aligned} &\mathcal{P}(\textit{Join}_{in}, \langle p, v\rangle) \\ &\quad = \mathcal{P}(B1_{out}, \langle p, v\rangle) \sqcap \mathcal{P}(B2_{out}, \langle p, v\rangle) \\ &\quad \overset{def}{=\!=} \frac{\mathcal{P}(B1_{out}, \langle p, v\rangle) \times E(B1) + \mathcal{P}(B2_{out}, \langle p, v\rangle) \times E(B2)}{E(B1) + E(B2)} \end{aligned}$$

where $E(B1)$ and $E(B2)$ are the numbers of times $B1$ and $B2$ are expected to be visited during program execution.

Similarly, the meet operator $\sqcap$ can be overloaded to handle sets of probabilistic points-to

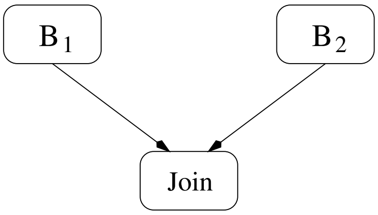


Fig. 3. Meet Operation

relationships:

$$\begin{aligned}
&\mathcal{P}(\mathit{Join}_{in},\, \mathit{IN}_{\mathit{Join}}) \\
&\quad = \mathcal{P}(B1_{out},\, \mathit{OUT}_{B1}) \sqcap \mathcal{P}(B1_{out},\, \mathit{OUT}_{B2}) \\
&\quad = \{\mathcal{P}(B1_{out},\, \langle p,\, v\rangle) \sqcap \mathcal{P}(B2_{out},\, \langle p,\, v\rangle) \mid \langle p,\, v\rangle \in \mathit{IN}_{\mathit{Join}}\}
\end{aligned}$$

where OUT_{B1}, OUT_{B2}, and $\mathit{IN}_{\mathit{Join}}$ are the sets of points-to relationships at program points $B1_{out}$, $B2_{out}$ and Join_{in} respectively, and $\mathit{Join}_{in} = \mathit{OUT}_{B1} \cup \mathit{OUT}_{B2}$.

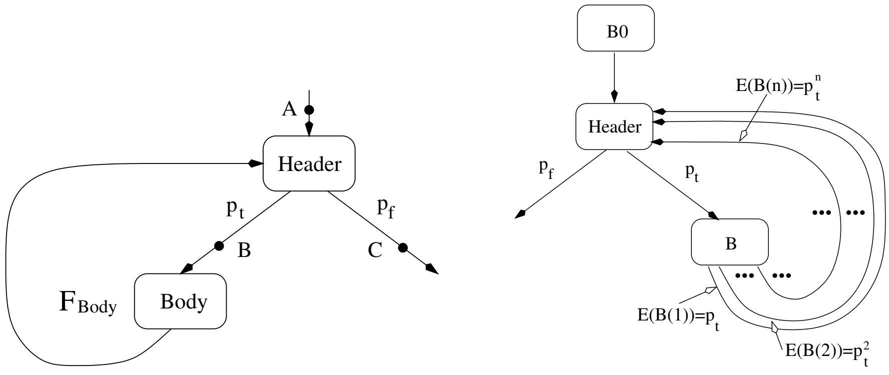
C. Conditionals

The most commonly used conditionals is the *if-then-else* construct. Suppose $\mathit{OUT}_{\mathit{Then}}$ and $\mathit{OUT}_{\mathit{Else}}$ are the sets of points-to relationships at the exit points Then_{out} and $Else_{out}$ of *Then* and *Else* branches respectively, while p_t and p_f are the branching probabilities of *Then* and *Else* branches respectively and $p_t + p_f = 1$. Then the possibility function of the points-to relationship $\langle p,\, v\rangle$ at the merge point Join_{in} of the *Then* and $Else$ branches can be computed by the meet operation:

$$\begin{aligned}
\mathcal{P}(\mathit{Join}_{in},\, \langle p,\, v\rangle) &= \mathcal{P}(\mathit{Then}_{out},\, \langle p,\, v\rangle) \sqcap \mathcal{P}(\mathit{Else}_{out},\, \langle p,\, v\rangle) \\
&= p_t \times \mathcal{P}(\mathit{Then}_{out},\, \langle p,\, v\rangle) + p_f \times \mathcal{P}(\mathit{Then}_{out},\, \langle p,\, v\rangle)
\end{aligned}$$

D. Loops

Since a loop can iterate an arbitrary number of times, it can be imaged as if there were an infinite number of outgoing edges leaving from the exit point of the loop body and then joining the header node. Specifically, the back edge of the loop shown in Figure 4(a) in fact represents



(a) CFG of A Loop (b) Imaginary CFG

Fig. 4. Loops

the infinitive number of out-edges of the loop body B, as shown in Figure 4(b). Furthermore, if the branching probability of entering the loop is p_t, then the expected frequency that B will be visited at the ith iteration is p_t^i, i.e. $E(B[i]) = p_t^i$ where $B[i]$ denotes B at the ith iteration. Therefore, the vector of probability functions for the set of points-to relationships IN_{Header} at at the entry of the header node will be

$$\begin{aligned}
&\mathcal{P}(\mathit{Header}_{in}, \mathit{IN}_{Header}) \\
&= \mathcal{P}(B0_{out}, \mathit{OUT}_{B0}) \sqcap \mathcal{P}(B[1]_{out}, \mathit{OUT}_{B[1]}) \sqcap \\
&\quad \mathcal{P}(B[2]_{out}, \mathit{OUT}_{B[2]}) \sqcap \cdots \sqcap \mathcal{P}(B[n]_{out}, \mathit{OUT}_{B[n]}) \sqcap \cdots \\
&= \mathcal{P}(B0_{out}, \mathit{OUT}_{B0}) \sqcap (\prod_{i=1}^{\infty} \mathcal{P}(B[i]_{out}, \mathit{OUT}_{B[i]})) \\
&\sim \mathcal{P}(B0_{out}, \mathit{OUT}_{B0}) \sqcap \mathcal{P}(B_{out}, \mathit{OUT}_{B})
\end{aligned}$$

The approximation is made because the probabilistic points-to analysis is implemented based on an iterative data flow framework. In other words, the set of points-to relationships OUT_B at B_{out} is the result of meet (union) operations on all the points-to relationships that can reach B_{out} until OUT_B converges, i.e.

$$\begin{aligned}
\mathit{OUT}_B &= \mathit{OUT}_B[1] \sqcap \mathit{OUT}_B[2] \sqcap \cdots \sqcap \mathit{OUT}_B[n] \sqcap \cdots \\
&= \mathit{OUT}_B[1] \cup \mathit{OUT}_B[2] \cup \cdots \cup \mathit{OUT}_B[n] \cup \cdots
\end{aligned}$$

Furthermore, the expected frequency of B_{OUT} is the sum of the expected frequencies of $B_{OUT}[i]$ being visited at all iterations, i.e. $E(B_{OUT}) = \sum_1^\infty E(B[i]_{OUT})$. Therefore, $E(B_{OUT}) = p_t + p_t^2 + \cdots + p_t^n + \cdots = p_t/(1 - p_t)$, if there are no statements branching out the loop.

In order to find a solution of the above equation, a symbolic probability will be assigned to each probability function as its result at the entry of the header node. Since the probability functions $\mathcal{P}(B_{out},\, \boldsymbol{OUT}_B)$ will be computed by the transfer function F_B with the vector $\mathcal{P}(B_{in},\, \boldsymbol{IN}_B)$ as its argument, the vector returned by $\mathcal{P}(B_{out},\, \boldsymbol{OUT}_B)$ will be functions of these symbolic probabilities. As a result, the equation $\mathcal{P}(\boldsymbol{Header}_{in},\, \boldsymbol{IN}_{Header}) = \mathcal{P}(B0_{out},\, \boldsymbol{OUT}_{B0}) \sqcap \mathcal{P}(B_{out},\, \boldsymbol{OUT}_B)$ is in fact an equation system, and the values of symbolic probabilities can be computed by solving the equation system.

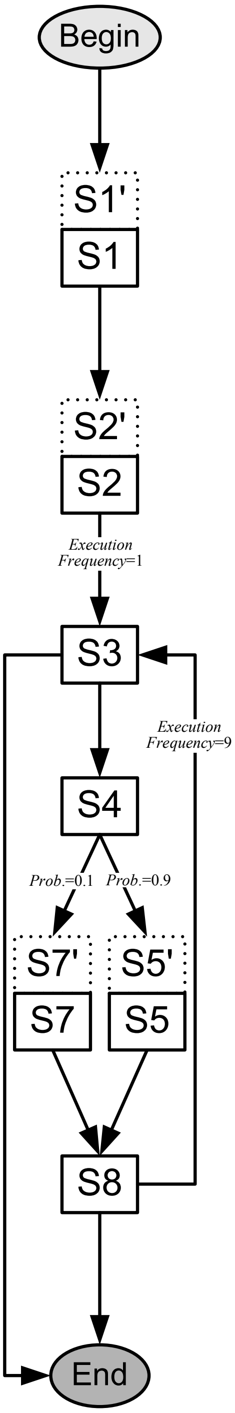
Consider a very simple loop with only one points-to relationship $\langle p,\, v\rangle$. A symbolic probability P is introduced at the header entry, i.e. $\mathcal{P}(\boldsymbol{Header}_{in},\, \langle p,\, v\rangle) = P$, and hence $\mathcal{P}(B_{in},\, \langle p,\, v\rangle) = P$. Suppose $\mathcal{P}(B0_{out},\, \langle p,\, v\rangle) = 1,\ E(B) = 10,\ and\ \mathcal{P}(B_{out},\, \langle p,\, v\rangle) = F_S(\mathcal{P}(B_{in},\, \langle p,\, v\rangle)) = 0.9P$. Then the symbolic probability P can be solved:

$$\begin{aligned} \mathcal{P}(\boldsymbol{Header}_{in},\, \boldsymbol{IN}_{Header}) &= \mathcal{P}(B0_{out},\, \boldsymbol{OUT}_{B0}) \sqcap \mathcal{P}(B_{out},\, \boldsymbol{OUT}_B) \\ P &= (1 \times 1 + 10 \times 0.9P) \,/\, (1 + 10) \\ P &= 0.5 \end{aligned}$$

E. Example

Since the probabilistic points-to analysis is implemented based on an iterative data flow framework, the solution of the equation system will be solved when the sets in a loop are converged. Consider the program and its corresponding CFG shown in Figure 5. Every points-to relationship $\langle p,\, v\rangle$ at each program point s is associated with its probability function, and hence is denoted by a tuple with the form $[\langle p,\, v\rangle,\, \mathcal{P}(s,\, \langle p,\, v\rangle)]$. The tuple can be called as a *probabilistic points-to relationship*.

The set of probabilistic points-to relationships $\boldsymbol{OUT}_{S2}$ before entering the loop contains $[\langle p,\, v\rangle,\, 1]$



	Iteration 1	Iteration 2	Iteration 3
Statement	IN_{Si}	IN_{Si}	IN_{Si}
	OUT_{Si}	OUT_{Si}	OUT_{Si}
$S1'$: $p = nil$;	-	-	-
	-	-	-
$S1$: $p = \&v$;	-	-	-
	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$
$S2'$: $q = nil$	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$
	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$
$S2$: $q = \&u$;	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$	$[\langle p, v\rangle, 1]$
	$[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$	$[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$	$[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$
$S3$: $while(...)$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$
	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$
$S4$: $if(...)$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$
	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$
$S5'$: $p = nil$;	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$
	$[\langle q, u\rangle, P_2]$	$[\langle q, u\rangle, P_2]\,[\langle q, v\rangle, P_4]$	$[\langle q, u\rangle, P_2]\,[\langle q, v\rangle, P_4]$
$S5$: $p = q$;	$[\langle q, u\rangle, P_2]$	$[\langle q, u\rangle, P_2]\,[\langle q, v\rangle, P_4]$	$[\langle q, u\rangle, P_2]\,[\langle q, v\rangle, P_4]$
	$[\langle p, u\rangle, P_2]\,[\langle q, u\rangle, P_2]$	$[\langle p, u\rangle, P_2]\,[\langle p, v\rangle, P_4]$ $[\langle q, u\rangle, P_2]\,[\langle q, v\rangle, P_4]$	$[\langle p, u\rangle, P_2]\,[\langle p, v\rangle, P_4]$ $[\langle q, u\rangle, P_2]\,[\langle q, v\rangle, P_4]$
$S6$: $else$			
$S7'$: $q = nil$;	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$	$[\langle p, v\rangle, P_1]\,[\langle q, u\rangle, P_2]$ $[\langle p, u\rangle, P_3]\,[\langle q, v\rangle, P_4]$
	$[\langle p, v\rangle, P_1]$	$[\langle p, v\rangle, P_1]\,[\langle p, u\rangle, P_3]$	$[\langle p, v\rangle, P_1]\,[\langle p, u\rangle, P_3]$
$S7$: $q = p$;	$[\langle p, v\rangle, P_1]$	$[\langle p, v\rangle, P_1]\,[\langle p, u\rangle, P_3]$	$[\langle p, v\rangle, P_1]\,[\langle p, u\rangle, P_3]$
	$[\langle p, v\rangle, P_1]\,[\langle q, v\rangle, P_1]$	$[\langle p, v\rangle, P_1]\,[\langle p, u\rangle, P_3]$ $[\langle q, v\rangle, P_1]\,[\langle q, u\rangle, P_3]$	$[\langle p, v\rangle, P_1]\,[\langle p, u\rangle, P_3]$ $[\langle q, v\rangle, P_1]\,[\langle q, u\rangle, P_3]$
$S8$:	$[\langle p, v\rangle, 0.1P_1]$ $[\langle q, v\rangle, 0.1P_1]$ $[\langle p, u\rangle, 0.9P_2]$ $[\langle q, u\rangle, 0.9P_2]$	$[\langle p, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle q, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle p, u\rangle, 0.9P_2 + 0.1P_3]$ $[\langle q, u\rangle, 0.9P_2 + 0.1P_3]$	$[\langle p, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle q, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle p, u\rangle, 0.9P_2 + 0.1P_3]$ $[\langle q, u\rangle, 0.9P_2 + 0.1P_3]$
	$[\langle p, v\rangle, 0.1P_1]$ $[\langle q, v\rangle, 0.1P_1]$ $[\langle p, u\rangle, 0.9P_2]$ $[\langle q, u\rangle, 0.9P_2]$	$[\langle p, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle q, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle p, u\rangle, 0.9P_2 + 0.1P_3]$ $[\langle q, u\rangle, 0.9P_2 + 0.1P_3]$	$[\langle p, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle q, v\rangle, 0.1P_1 + 0.9P_4]$ $[\langle p, u\rangle, 0.9P_2 + 0.1P_3]$ $[\langle q, u\rangle, 0.9P_2 + 0.1P_3]$

Fig. 5. Example

and $[\langle q,\, u\rangle,\, 1]$. Consequently, symbolic probabilities P_1 and P_2 are introduced at the entry of the loop, and hence the set IN_{S3} consists of $[\langle p,\, v\rangle,\, P_1]$ and $[\langle q,\, u\rangle,\, P_2]$. Statement $S5\colon\ p = q;$ of the if branch kills the $[\langle p,\, v\rangle,\, P_1]$ tuple and creates a new tuple $[\langle p,\, u\rangle,\, P_2]$, while statement $S7\colon\ q = p;$ of the $else$ branch removes the $[\langle q,\, u\rangle,\, P_2]$ tuple and introduces $[\langle q,\, v\rangle,\, P_1]$ to the set OUT_{S7}. Assume the branching probabilities of the if and $else$ branches are 0.9 and 0.1 respectively, as shown in the CFG. Merging the two branches, the set of probabilistic points-to relationships OUT_{S8} at the exit point of the loop will be comprised of the tuples $[\langle p,\, v\rangle,\, 0.1P_1]$, $[\langle q,\, v\rangle,\, 0.1P_1]$, $[\langle p,\, u\rangle,\, 0.9P_2]$, and $[\langle q,\, u\rangle,\, 0.9P_2]$ after iteration 1.

Two symbolic probabilities P_3 and P_4 are introduced in the set of probabilistic points-to relationships IN_{S3} at iteration 2, since two more points-to relationships are now merged into the loop entry. After iteration 2 is executed, the set OUT_{S8} will contain $[\langle p,\, v\rangle,\, 0.1P_1 + 0.9P_4]$, $[\langle q,\, v\rangle,\, 0.1P_1 + 0.9P_4]$, $[\langle q,\, u\rangle,\, 0.9P_2 + 0.1P_3]$, and $[\langle p,\, u\rangle,\, 0.9P_2 + 0.1P_3]$.

No more symbolic probabilities are declared when iteration 3 starts, and no new tuples are created by any statements at iteration 3. That is, all the sets of probabilistic points-to relationships converge after 3 iterations of execution. Now the equation system can be solved to obtain the values of the symbolic probabilities. Suppose the loop body is executed 9 times. $\mathcal{P}(S3_{in},\, \mathit{IN}_{S3}) = \mathcal{P}(B2_{out},\, \mathit{OUT}_{B2}) \sqcap \mathcal{P}(B8_{out},\, \mathit{OUT}_{S8})$ represents the following equation system:

$$\begin{aligned}
P_1 &= (1 + 9 \times (0.1P_1 + 0.9P_4))/(1+9) \\
P_2 &= (1 + 9 \times (0.9P_2 + 0.1P_3))/(1+9) \\
P_3 &= (9 \times (0.9P_2 + 0.1P_3))/(1+9) \\
P_4 &= (9 \times (0.1P_1 + 0.9P_4))/(1+9)
\end{aligned}$$

The values of the symbolic probabilities will be $P_1 = 0.19$, $P_2 = 0.91$, $P_3 = 0.81$, and $P_4 = 0.09$ after the equation system is solved. In other words, the probabilistic points-to relationships in IN_{S3} at loop entry will be $[\langle p,\, v\rangle,\, 0.19]$, $[\langle q,\, u\rangle,\, 0.91]$, $[\langle p,\, u\rangle,\, 0.81]$, and $[\langle q,\, v\rangle,\, 0.09]$.

F. Analysis

The unique feature that distinguishes the probabilistic points-to analysis from the conventional points-to analysis techniques is that the probabilistic points-to analysis associates a probability function for every points-to relationship. In other words, the probabilistic points-to analysis basically computes the set of points-to relationships at every program point following the same set of the rules conducted by most existing conventional points-to analysis techniques [2], [5], [8], [11], as outlined in Section III-A. The difference is that the probabilistic points-to analysis will go one step further — computing the probability function of every points-to relationships. Therefore, the probabilistic points-to analysis can be divided into two phases: identifying the sets of points-to relationships and computing the probability function of every points-to relationship.

1) Termination: The first phase of the probabilistic points-to analysis is simply the conventional points-to analysis. As mentioned by the previous work of other researchers, the points-to analysis can be implemented by applying the well-known iterative or interval-based techniques [2], [5], [6], [8], [11]. Since the number of pointer variables are finite, the number of relationships will be finite. The only exception is the heap objects, which might be infinite. However, all points-to analysis techniques employ ways to represent the heap objects with finite representations, e.g. a single *heap* object [5], imposing k-limit rules [6], location sets [11], or sparse graphs [2]. Consequently, the number of points-to relationships is finite and hence the process will definitely terminate.

The second phase of the points-to analysis is to compute the probability function of every points-to relationship when the sets of points-to relationships converge. As described in Section III, a symbolic probability will be introduced for every points-to analysis at the merging points with back edges, and the results of probability function will be represented by an equation system of these symbolic probabilities. Therefore, this process can be performed by one iteration. In addition, there are several open-source solvers, e.g. GNU Scientific library (*GSL*), that can efficiently solve the symbolic equations to obtain the values represented by these symbolic probabilities.

2) Complexity: The time and space complexities of the first phase have be described and proved by previous work of other researchers [2], [5], [6], [8], [11]. As presented in Section III-F.1, the second phase takes only one iteration. Therefore, the time complexity of formulating the equation system in the second phase will be $O(N \times V)$, where N is the maximum number of points-to relationships and V is the number of nodes on the CFG.

The space complexity of the second phase is $O(N^2 \times V)$, since an equation system of symbolic probabilities will be computed at each CFG node and the equation system can be represented by a matrix. However, every matrix will be sparse because in most programs every location is pointed to by close to one pointer [5], [26].

IV. Interprocedural Analysis

A. Algorithm

The algorithm for interprocedural probabilistic points-to analysis is developed based on the algorithm developed by Emami et al. [5]. At each call site, points-to relationships are mapped from actual parameters to formal parameters by the algorithm, and the results are unmapped back to the variables in the caller once the called function is analyzed. During the mapping process, symbolic names (or *ghost* location sets) will be declared to represent variables outside the scope of the called functions, i.e. *invisible* variables [5], [8].

Instead of constructing an invocation graph, this algorithm implements a call stack to keep track of procedure invocations. The node for an invoked procedure will be pushed into the stack and popped out of the stack when the invocation ends. Therefore, the contents of the call stack represent the nodes on the paths from the root of the invocation graph to the currently active procedure. Furthermore, the contents of the call stack determine the calling context of the procedure currently being analyzed. If a cycle is created by recursive invocations, an approximation similar to that done by Emami et al. [5] and Wilson and Lam [11] will be applied.

A symbolic probability will be assigned to every points-to relationship at the entry of a procedure as the value of its probability function. The intraprocedural algorithm outlined in the previous section can then be applied to compute the probability function of every points-to relationship at the end of the procedure. The transition of the probability functions from the procedure entry to the procedure exit represents the effects of the procedure. In other words, the transformations can be viewed as the transfer function of the procedure under the context. Like the analysis done by other researchers [8], [11], the transfer function will be cached to avoid duplicate computations. If the procedure is invoked with the same set of points-to relationships, maybe with different probability functions, the transfer function can be used to compute the results by substituting the symbolic probabilities with the values of the probability functions.

B. Handling Recursive Procedures

In addition to the symbolic probability that will be declared for every points-to relationship at the procedure entry as done for nonrecursive procedures, one matching symbolic probability will be declared for the points-to relationship at the end of the procedure. The reason to introduce a new set of symbolic probabilities is because it is not possible to compute the probability functions directly at the end of a recursive procedure. This set of probabilistic points-to relationships will be served as the transfer function of the recursive procedure at the current stage.

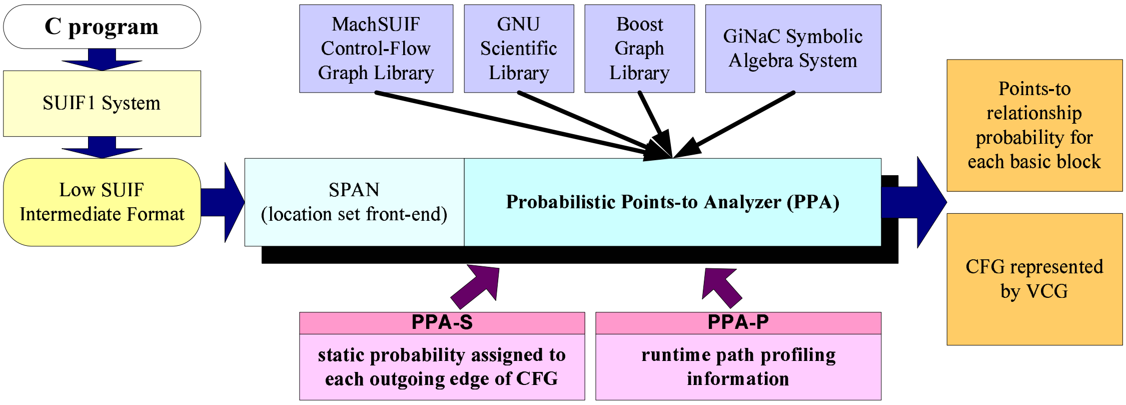
When a recursive invocation is encountered, the current transfer function, i.e. the set of points-to relationships with symbolic probabilities at the end of the procedure, will be used to compute the *OUT* set of the invocation statement. If new points-to relationships are merged to the entry of the recursive procedure at later iterations, a pair of symbolic probabilities will be declared for every new points-to relationship, one for the procedure entry and one for the procedure exit. Furthermore, the new points-to relationships at the end of procedure will be included as part of the transfer function. The process will be repeated until none of sets change.

Once the sets converge, the symbolic probabilities at the procedure entry can be obtained by solving the equation system for the meet operation on the incoming-edges to the entry node.

V. Experimental Results

A. Platform and Benchmarks

A prototype compiler has been implemented upon the SUIF system [20] and CFG library of MachSUIF [21] to perform the interprocedural probabilistic points-to analysis. The routines in the SPAN compiler to associate variables with location sets have been integrated in the compiler as well [8]. In addition, CAS (Computer Algebra System) *GiNaC* [27] and GNU Scientific library *GSL* have been incorporated to process symbolic and mathematical computations. Figure 6 shows the structure of the prototype compiler.



C program
SUIF1 System
Low SUIF Intermediate Format
MachSUIF Control-Flow Graph Library
GNU Scientific Library
Boost Graph Library
GiNaC Symbolic Algebra System
SPAN (location set front-end)
Probabilistic Points-to Analyzer (PPA)
Points-to relationship probability for each basic block
CFG represented by VCG
PPA-S
static probability assigned to each outgoing edge of CFG
PPA-P
runtime path profiling information

Fig. 6. Structure of Prototype Compiler

Programs are first transformed from the high-SUIF format to the low-SUIF format by SUIF and represented by CFGs using the CFG library of MachSUIF. All the variables on the CFG nodes will be associated with location sets by the SPAN routine. The compiler will then traverse the CFGs to compute the probability function of every probabilistic points-to relationship at each program point. This section will present the preliminary experimental results of this implementation.

Several applications have been chosen as the benchmarks, as listed in Table I. The fourth column contains the number of lines in the source and header files reported by the Unix utility *wc*. The fifth column reports the number of CFG nodes which are computed by MachSUIF CFG library. The sixth column reports the number of user-defined functions. These benchmark

Program	Description	Source	LOC	CFG Nodes	Functions
990127-1	Test program from gcc-3.0.1 testsuite.	GCC	31	23	1
shuffle	The program tests a random number generator using a card shuffling procedure.	netlib.org	713	64	4
20000801-2	Test program from gcc-3.0.1 testsuite	GCC	40	26	4
fir2dim	DSPstone filter benchmark.	DSPStone	152	46	2
misr	A program create and use link list.	McGill	276	142	5
fft	An FFT test program.	netlib.org	963	132	7
dhrystone	The dhrystone benchmark v2.1.	netlib.org	1443	182	12
clinpack	This is the Linpack program (floating-point) rewritten by C.	netlib.org	1385	405	12
alvinn	This program trains a neural network called ALVINN using back propagation.	SPEC92	272	125	8
queens	A program that finds solutions to the eight-queens chess problem.	netlib.org	363	132	3
treeadd	This program adds the numbers stored at each node of a binary tree.	Olden	202	36	4
power	The Power Pricing benchmark.	Olden	818	194	17
hash	A program builds a hash table.	McGill	257	68	5

TABLE I
BENCHMARK PROGRAMS

outgoing edge	assigned probability
Fall-through	1
IF (taken)	$p_t = 0.5$
IF (not taken)	$p_f = 0.5$
Loop-back edge	$p_t = 0.9$
Loop-exit edge	$p_f = 0.1$

TABLE II
PROBABILITY OF OUTGOING EDGE STATICALLY ASSIGNED.

programs will then be executed to gather the detailed points-to information at runtime. The runtime results will be compared with the following three variations of probabilistic points-to analysis:

- Probabilistic points-to analysis based on static probabilities (PPA-S)

A probability will be assigned to each outgoing edge of CFG, as listed in Table II, and the probabilistic points-to analysis algorithm will be executed based on these edge probabilities.

- Probabilistic points-to analysis based on profiling information (PPA-P)

  We have developed a profiling tool by SUIF to gather the execution frequency of every edge in CFG, and probabilistic points-to analysis will be performed based on the profiling information to compute the probabilities of points-to relationships in selected benchmarks.

- Traditional points-to analysis (TPA)

  The probability of each points-to relationship is assumed to be 1.

The discrepancy of the estimated probability for every points-to relationship by these points-to analysis methods from the probability observed at runtime, i.e. $|P_{estimated} - P_{runtime}|$, will be measured at the end of each basic block of all procedures. The accuracy of these variations of probabilistic points-to analysis will be quantified by averaging all the discrepancies gathered at all basic blocks to obtain the *average error*

$$\xi = \frac{\sum_{i=1}^{n} |P_{estimated}(i) - P_{runtime}(i)|}{n}$$

The precision of probabilistic points-to analysis will be quantified by computing variances gathered at all basic blocs to obtain the *standard deviation*

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (P_{estimated}(i) - P_{runtime}(i))^2}{n}}$$

where $P_{estimated}(i)$ is the estimated probability of the i_{th} points-to relationship, and $P_{runtime}(i)$ is the runtime profiled probability of the i_{th} points-to relationship.

In addition, weighted version of average errors will be computed to take into the frequencies of executions into account

$$\xi_w = \frac{\sum_{i=1}^{n} |P_{estimated}(i) - P_{runtime}(i)| \times w_i}{\sum_{i=1}^{n} w_i}$$

where w_n is the frequencies of points-to relationship i.

B. Results

Figure 7 and Figure 8 show the average errors and standard deviation of estimated probabilities of points-to relationships by these methods compared to the profiled probabilities at runtime, respectively. Table III summarizes the average errors and standard deviation in Figure 7 and Figure 8 in a tabular format. The numbers in the figures and table show our probabilistic points-to analysis approach can estimate the likelihood that 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 24%. With the aid of edge profiling information, PPA-P reduces the average error down to 4.6%. Furthermore, Figure 9 demonstrates the weighted average errors ξ_w of PPA-P are even smaller (3.68%) when taking the execution frequencies into account.

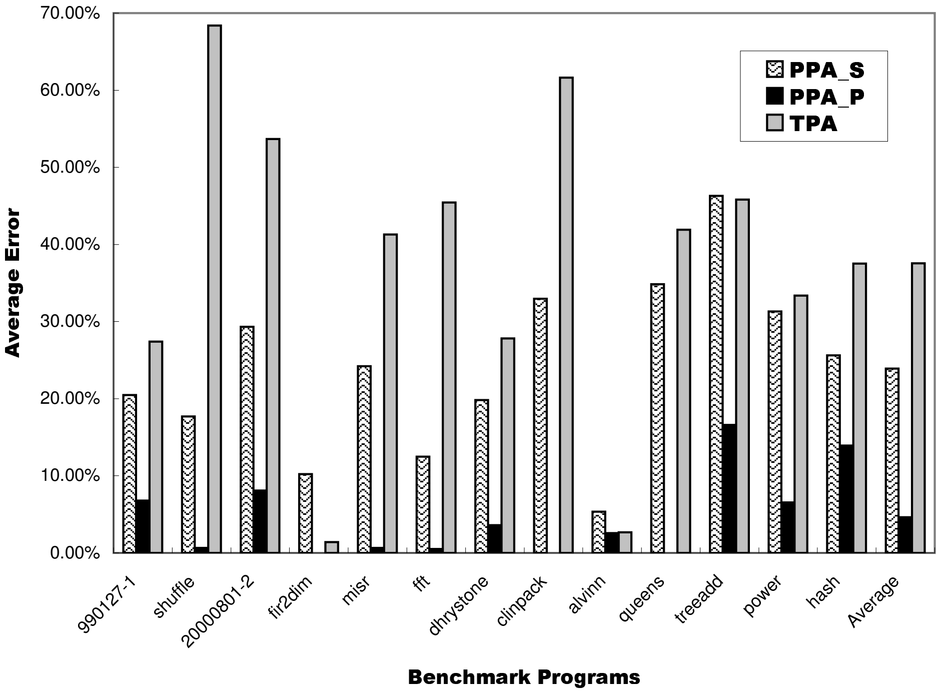


Fig. 7. Average Errors

This result is significant since many compiler optimizations rely on the ability to determine if points-to relationships hold with high or low probabilities. Two statistics will be measured

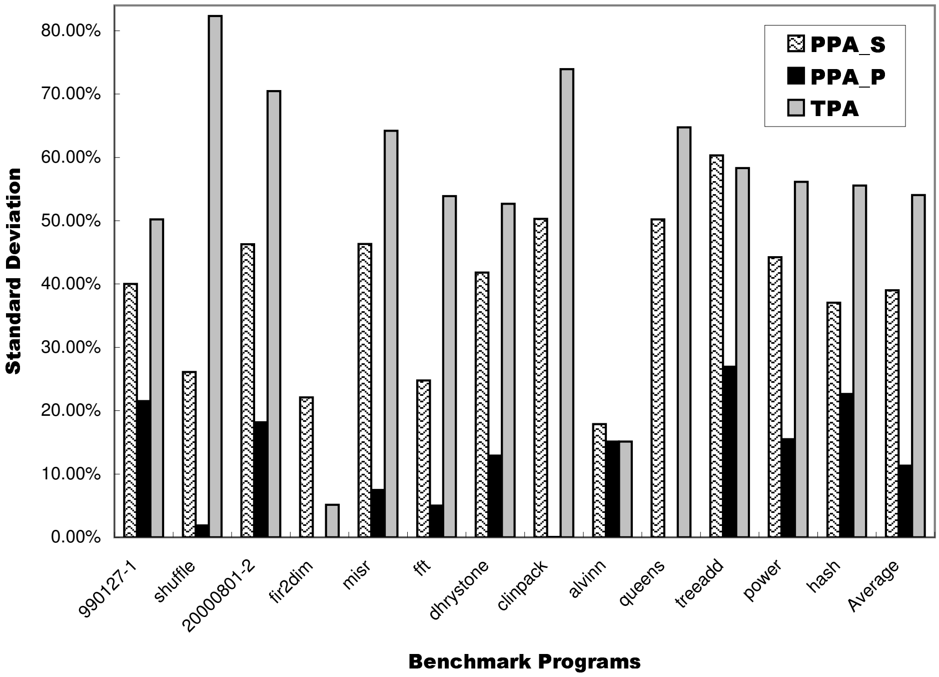


Fig. 8. Standard Deviation

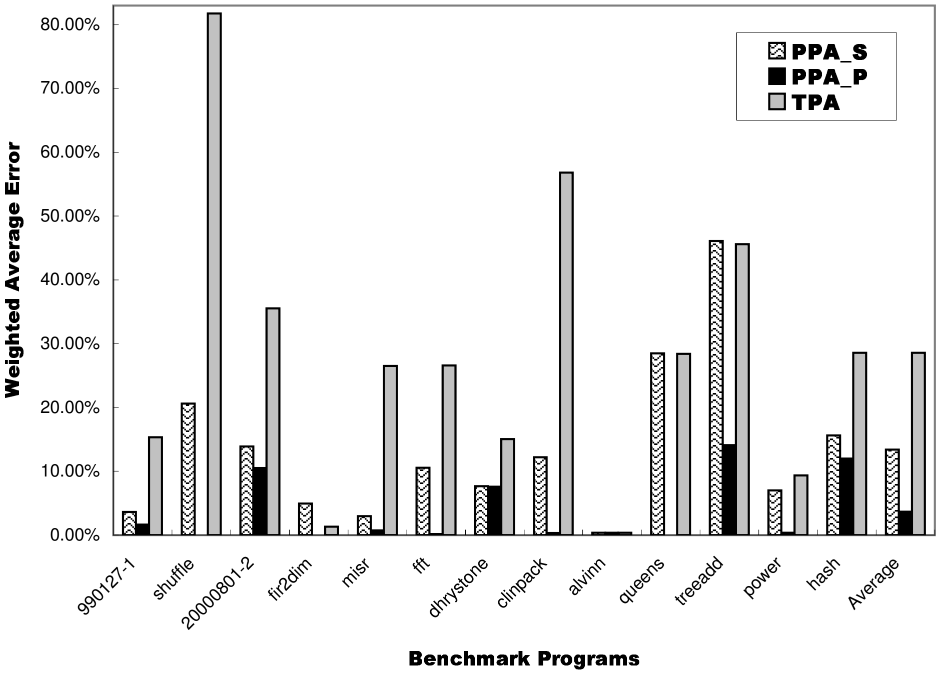


Fig. 9. Weighted Average Errors

	Average Errors			Standard Deviation			Weighted Average Errors		
Programs	PPA-S	PPA-P	TPA	PPA-S	PPA-P	TPA	PPA-S	PPA-P	TPA
99012701	20.49%	6.79%	27.39%	40.00%	21.51%	50.20%	3.64%	1.63%	15.35%
shuffle	17.70%	0.66%	68.39%	26.12%	1.88%	82.23%	20.62%	0.00%	81.77%
20000801-2	29.33%	8.09%	53.68%	46.26%	18.19%	70.45%	13.90%	10.53%	35.53%
fir2dim	10.21%	0.00%	1.42%	22.08%	0.00%	5.13%	4.94%	0.00%	1.31%
misr	24.23%	0.66%	41.31%	46.34%	7.46%	64.19%	2.97%	0.74%	26.50%
fft	12.50%	0.51%	45.45%	24.76%	5.03%	53.89%	10.55%	0.16%	26.59%
dhrystone	19.81%	3.61%	27.80%	41.81%	12.92%	52.68%	7.65%	7.59%	15.03%
clinpack	32.95%	0.01%	61.61%	50.30%	0.06%	73.94%	12.20%	0.32%	56.83%
alvinn	5.36%	2.57%	2.66%	17.89%	15.13%	15.14%	0.38%	0.37%	0.37%
queens	34.85%	0.00%	41.93%	50.22%	0.00%	64.72%	28.49%	0.00%	28.42%
treeadd	46.30%	16.60%	45.83%	60.31%	26.92%	58.32%	46.09%	14.10%	45.61%
power	31.32%	6.55%	33.37%	44.22%	15.50%	56.12%	7.00%	0.38%	9.34%
hash	25.61%	13.94%	37.51%	37.05%	22.64%	55.57%	15.65%	12.01%	28.56%
Overall	23.90%	4.61%	37.57%	39.03%	11.33%	54.05%	13.39%	3.68%	28.55%

TABLE III

AVERAGE ERRORS AND STANDARD DEVIATION

Probability Range	PPA-S	PPA-P	PPA-S	PPA-P
0%∼10%	6.75%	90.49%	12.85%	91.83%
10%∼20%	16.67%	50.00%		
20%∼30%	30.56%	33.33%	45.83%	29.17%
30%∼40%	0.00%	8.33%		
40%∼50%	50.00%	79.69%	90.72%	95.16%
50%∼60%	96.25%	72.63%		
60%∼70%	45.45%	0.00%	42.86%	44.90%
70%∼80%	13.16%	57.89%		
80%∼90%	0.00%	29.41%	83.40%	96.28%
90%∼100%	83.70%	95.80%		

TABLE IV

ACCURACY OF PPA ESTIMATED PROBABILITIES

to demonstrate this analysis can achieve high accuracy and precision in identifying points-to relationships with high or low probabilities.

- The accuracy percentage in runtime information

  Assume $\textit{Points-to}_{Runtime}(l\%{\sim}h\%)$ be the set of points-to relationships with runtime-profiled

Probability Range	PPA-S	PPA-P	PPA-S	PPA-P
0%∼10%	56.00%	96.03%	56.28%	94.95%
10%∼20%	1.10%	10.34%		
20%∼30%	19.30%	26.09%	9.36%	23.73%
30%∼40%	0.00%	7.69%		
40%∼50%	27.12%	28.49%	45.67%	88.17%
50%∼60%	47.96%	86.81%		
60%∼70%	9.26%	0.00%	30.43%	36.67%
70%∼80%	33.33%	44.00%		
80%∼90%	0.00%	17.86%	83.03%	98.01%
90%∼100%	83.95%	98.08%		

TABLE V
CONFIDENCE OF PPA ESTIMATED PROBABILITIES

probabilities within the range $l\%{\sim}h\%$. *Points-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 from $l\%$ to $h\%$ and are also in the set *Points-to*$_{Runtime}(l\%{\sim}h\%)$. Then the *accuracy percentage within the probability range* $l\%{\sim}h\%$ of PPA is defined as the ratio of the size of *Points-to*$_{PPA}(l\%{\sim}h\%)$ over the size of *Points-to*$_{Runtime}(l\%{\sim}h\%)$, i.e.

$$|\textit{Points-to}_{PPA}(l\% \sim h\%)| / \, |\textit{Points-to}_{Runtime}(l\% \sim h\%)|$$

Table IV presents the accuracy of PPA-S and PPA-P within different probability ranges based on the above definition. The first section of Table IV shows the accuracy percentage of PPA-S and PPA-P in the probability range 0%∼10% is 6.75% and 90.49% respectively, while the accuracy of both PPA-S and PPA-P in the range 90%∼100% is 83.70% and 95.80%, respectively.

- The confidence percentage in PPA information

  Let *Points-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 from $l\%$ to $h\%$. *Points-to*$_{Runtime}(1\%{\sim}h\%)$ be the set of points-to relationships with runtime-profiled probabilities within the range $l\%{\sim}h\%$ and are also in the set *Points-to*$_{PPA}(l\%{\sim}h\%)$. The *confidence percentage within the proba-*

bility range $l\%{\sim}h\%$ of PPA is defined as the ratio of the size of *Points-to*$_{Runtime}(l\%{\sim}h\%)$ over the size of *Points-to*$_{PPA}(l\%{\sim}h\%)$, i.e.

$$|\textit{Points-to}_{Runtime}(l\% \sim h\%)|/\,|\textit{Points-to}_{PPA}(l\% \sim h\%)|$$

Table V presents these information within different probability ranges based on the above definition.

This result demonstrates that the probabilistic points-to analysis can identify the points-to relationships with high or low probabilities with very high accuracy.

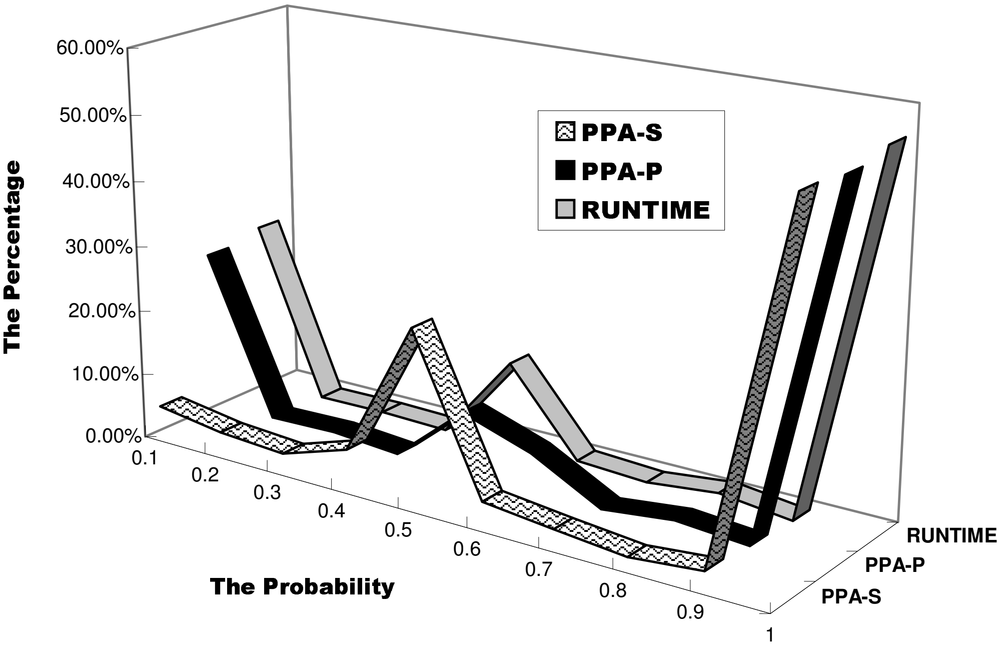


Fig. 10. Distributions of Points-to Relationship Probabilities

Figure 10 and Table VI list 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 very close to the profiled probability distributions. The high or low probabilities occupy most of points-to relationships and these points-to relationships with high or low probabilities are the most important parts in compiler optimizations. Such characteristic is important to compiler optimizations.

program	analysis method	0% ≀ 10%	10% ≀ 20%	20% ≀ 30%	30% ≀ 40%	40% ≀ 50%	50% ≀ 60%	60% ≀ 70%	70% ≀ 80%	80% ≀ 90%	90% ≀ 100%
990127-1	Runtime	22.36%	1.86%	3.11%	0	0	0	0	4.97%	2.48%	65.22%
	PPA-P	16.77%	0	9.94%	0	0	0	1.86%	8.07%	1.86%	61.49%
	PPA-S	0.62%	0	3.11%	8.70%	2.48%	1.86%	6.83%	3.11%	0.62%	72.67%
shuffle	Runtime	69.05%	0	0	0	0	0	0	0	0	30.95%
	PPA-P	69.05%	0	0	0	0	0	0	0	0	30.95%
	PPA-S	16.67%	14.29%	11.90%	0	26.19%	0	0	0	0	30.95%
20000801-2	Runtime	45.59%	0	0	0	16.18%	0	0	0	0	38.24%
	PPA-P	33.82%	0	11.76%	0	20.59%	0	0	0	0	33.82%
	PPA-S	0	11.76%	20.59%	0	2.94%	13.24%	0	5.88%	0	45.59%
fir2dim	Runtime	0	0	0	0	0	0	0	3.72%	0	96.28%
	PPA-P	0	0	0	0	0	0	0	3.72%	0	96.28%
	PPA-S	0	0	0	0	17.72%	0	0	0	3.72%	78.56%
misr	Runtime	41.42%	0	0	0	0	0	0	0	0	58.58%
	PPA-P	42.01%	0	0	0	0	0	0	0	0	57.99%
	PPA-S	17.16%	0	3.55%	0	5.33%	0	0	0	0	73.96%
fft	Runtime	12.63%	0	0	0	65.66%	0	0	0	0	21.72%
	PPA-P	13.64%	0	0	0	27.78%	36.87%	0	0	0	21.72%
	PPA-S	1.01%	0	1.52%	0	88.38%	0	0	0	0	9.09%
dhrystone	Runtime	27.65%	0	0	0.22%	0	0	0	0	0	72.12%
	PPA-P	21.90%	0.22%	0	0.22%	3.98%	3.76%	0	0	0	69.91%
	PPA-S	7.30%	0.22%	0	0	2.43%	2.43%	0	0.22%	3.54%	83.85%
clinpack	Runtime	47.83%	0	0	0	23.98%	3.60%	0	0	0	24.60%
	PPA-P	47.83%	0	0	0	23.98%	3.60%	0	0	0	24.60%
	PPA-S	0	0	0	9.69%	42.73%	4.22%	0	0	0	43.35%
alvinn	Runtime	2.28%	0	0	0	0	0	0	0	0	97.72%
	PPA-P	0	0	0	0	0	0	0	0	0	100.00%
	PPA-S	0	0	0	0	3.99%	0	0	0	3.99%	92.02%
queens	Runtime	41.89%	0	0	0	0	0	0	0	0	58.11%
	PPA-P	41.89%	0	0	0	0	0	0	0	0	58.11%
	PPA-S	0	0	0	0	38.51%	0	0	0	0	61.49%
treeadd	Runtime	20.51%	2.56%	0	0	46.15%	0	0	0	0	30.77%
	PPA-P	10.26%	0	12.82%	25.64%	15.38%	0	12.82%	0	0	23.08%
	PPA-S	51.28%	10.26%	0	0	7.69%	7.69%	0	0	10.26%	12.82%
power	Runtime	28.87%	4.47%	0	0	0	0	0	4.47%	0.69%	61.51%
	PPA-P	23.37%	9.28%	0	0	0	3.78%	0.69%	8.93%	8.25%	45.70%
	PPA-S	11.68%	17.18%	0	15.81%	11.68%	0	10.31%	0	0.34%	32.99%
hash	Runtime	23.74%	0	7.31%	5.02%	1.83%	0.91%	5.02%	7.31%	0	48.86%
	PPA-P	30.59%	1.37%	5.48%	0.91%	0.91%	0.91%	0	0	0	55.25%
	PPA-S	3.65%	5.02%	9.59%	18.26%	5.02%	16.44%	5.94%	2.28%	0.46%	33.33%
Overall	Runtime	26.13%	0.47%	0.58%	0.33%	13.51%	0.86%	0.31%	1.50%	0.17%	56.14%
	PPA-P	24.71%	0.86%	1.14%	0.36%	9.53%	5.98%	0.28%	1.56%	0.75%	54.84%
	PPA-S	3.78%	2.22%	1.58%	4.95%	25.88%	2.67%	1.50%	0.42%	1.50%	55.50%

TABLE VI

PROBABILITY DISTRIBUTIONS OF POINTS-TO RELATIONSHIPS

C. Discussion

PPA-P can estimate the probabilities of points-to relationships at the end of every basic block in all procedures in the benchmark programs with small errors. The only exceptions are the programs *treeadd* and *hash*, which have average errors of 16.60% and 13.94% respectively even with the PPA-P technique. The main reason is that the current implementation cannot disambiguate heap and array elements. Hence, it cannot correctly estimate the probabilities of points-to relationships in the programs with linked-list structures. The other source of discrepancy is the edge profiling technique used in this implementation to gather the execution frequencies. Intuitively, path profiling should provide more accurate information than edge profiling. However, it is proved that edge profiling is as excellent as path profiling for most programs, even for programs in SPEC95 with complex conditional control flow [28]. Therefore, profiling is not critical as handling heap and array elements, and hence extension for the points-to relationship representation will be incorporated to disambiguate heap and array elements.

Although this experiment was carried out using a single set of input data, PPA-P should still be able to accurately estimate the probabilities of points-to relationships even if different input sets are used. There reason is that there is an extremely high correlation for the same programs even with different input data sets [29].

VI. Related Work

There have been considerable efforts on pointer analysis by researchers [1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. 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 all executions, and *may*-aliases or *possibly*-points-to relationships, which might hold for some executions. However, they cannot tell which *may*-aliases or possibly-points-to relationships hold for most executions and which for only few executions. Such information is crucial for compilers to determine if certain optimizations and transformations will be beneficial. To the best of our knowledge, the work by the authors on probabilistic points-to analysis is the first to compute such information.

The most closely related work is the *data flow frequency analysis* proposed by Ramalingam [22]. 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 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. Furthermore, this technique performs the probabilistic points-to analysis on CFGs, eliminating the overhead of generating *exploded graphs* [30].

In the work related to data speculations for modern computer architectures, such as IA-64[13], Ju et al. [14] 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[31] and data distribution analysis[32] 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.

This work presents a major enhancement for pointer analysis to keep up with modern compiler optimizations. Aggressive optimizations, such as thread partitioning in speculative multithreading, data speculations, code specialization, etc, can be performed by compilers to improve performance on advanced architecture once the compilers are able to quantify the likelihood of the dynamic associations of pointers. A preliminary work done by the authors was limited to intraprocedural analysis and based on the interval analysis technique (i.e. the elimination technique), and hence it can not handle programs with irreducible flow graphs [33]. This work is implemented based on an iterative data flow technique, and further extended to perform context-sensitive interprocedural analysis.

In addition, this work has been applied to thread partitioning in the speculative multithreading model. The result shows that a compiler can achieve speedups by executing speculative threads when the possibilities of conflicts are low and can avoid slowdown by turning off thread speculation if the possibilities are high [18].

VII. Conclusions

This paper presented the probabilistic points-to analysis that could provide a quantitative description for each points-to relationship to represent the probability that it would hold. Such a piece of information is essential since optimizing and parallelizing compilers need to formulate the cost functions in order to assess the profitability of any transformations. A context-sensitive interprocedural algorithm has been implemented to compute the probability of every points-to relationship at each program point based on the iterative data flow analysis framework, and been incorporated into SUIF and MachSUIF. Experimental results showed this technique could estimate the probabilities of points-to relationships in benchmark programs with reasonable small errors, about 4.6% in average and 3.7% in weighted average. In addition, experimental results showed there were many opportunities for optimizations and parallelization since over 80% of the points-to relationships in the benchmark programs holded with very high or low probabilities, where useful optimizations and transformations could be performed. This technique has been applied to the speculative multithreading architecture to demonstrate that a compiler can achieve speedups by executing speculative threads when the possibilities are low and can avoid slowdown by turning off thread speculation if the possibilities are high.

References

[1] M. Burke, P. Carini, J.-D. Choi, and M. 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] J.-D. Choi, M. Burke, and P. 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*, Charleston, South Carolina, Jan. 1993, pp. 232–245.

[3] M. Das, “Unification-based pointer analysis with directional assignments.” in *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI-00)*, ser. ACM Sigplan Notices, vol. 35.5. N.Y.: ACM Press, June 18–21 2000, pp. 35–46.

[4] A. Deutsch, “Interprocedural May-Alias analysis for pointers: Beyond k-limiting,” *SIGPLAN Notices*, vol. 29, no. 6, pp. 230–241, June 1994, *Proceedings of the ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation.*

[5] M. Emami, R. Ghiya, and L. J. Hendren, “Context-sensitive interprocedural Points-to analysis in the presence of function pointers,” *SIGPLAN Notices*, vol. 29, no. 6, pp. 242–256, June 1994, *Proceedings of the ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation.*

[6] W. Landi and B. G. Ryder, “A safe approximate algorithm for interprocedural pointer aliasing,” *SIGPLAN Notices*, vol. 27, no. 7, pp. 235–248, July 1992, *Proceedings of the ACM SIGPLAN ’92 Conference on Programming Language Design and Implementation.*

[7] E. Ruf, “Context-insensitive alias analysis reconsidered,” *SIGPLAN Notices*, vol. 30, no. 6, pp. 13–22, June 1995, *Proceedings of the ACM SIGPLAN ’95 Conference on Programming Language Design and Implementation.*

[8] R. Rugina and M. Rinard, “Pointer analysis for multithreaded programs,” *SIGPLAN Notices*, vol. 34, no. 5, pp. 77–90, May 1999, *Proceedings of the ACM SIGPLAN ’99 Conference on Programming Language Design and Implementation*.

[9] M. Shapiro and S. 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*, Paris, France, Jan. 1997, pp. 1–14.

[10] B. Steensgaard, “Points-to analysis in almost linear time,” in *Conference Record of POPL ’96: 23nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, Jan. 1996, pp. 32–41.

[11] R. P. Wilson and M. S. Lam, “Efficient context-sensitive pointer analysis for C programs,” *SIGPLAN Notices*, vol. 30, no. 6, pp. 1–12, June 1995, *Proceedings of the ACM SIGPLAN ’95 Conference on Programming Language Design and Implementation*.

[12] S. H. Yong, S. Horwitz, and T. Reps, “Pointer analysis for programs with structures and casting,” *SIGPLAN Notices*, vol. 34, no. 5, pp. 91–103, May 1999, *Proceedings of the ACM SIGPLAN ’99 Conference on Programming Language Design and Implementation*.

[13] I. Corporation, *IA-64 Application Developer’s Architecture Guide*, 1999.

[14] R. D. 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.

[15] H. Akkary and M. A. Driscol, “A dynamic multithreading processor,” in *Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, 1998, pp. 226–236.

[16] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, “Multiscalar processors,” in *Proceedings of the 22nd annual international symposium on Computer architecture*, 1995, pp. 414–425.

[17] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, “A scalable approach to thread-level speculation,” in *The 27th Annual International Symposium on Computer Architecture*, 2000, pp. 1–12.

[18] P.-S. Chen, M.-Y. Hung, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee, “Compiler support for speculative multithreading architecture with probabilistic points-to analysis,” in *Proceedings of ACM SIGPLAN Conference on Principles and Practice of Parallel Programming*, June 2003, pp. 25–36.

[19] R. Muth, S. Watterson, and S. Debray, “Code specialization based on value profiles,” in *Proceedings of the 7th International Static Analysis Symposium*, 2000, pp. 340–359.

[20] T. S. S. C. Groupd, “The suif library,” Stanford University, Tech. Rep., 1995.

[21] M. D. Smith, “The suif machine library,” Division of of Engineering and Applied Science, Harvard University, Tech. Rep., March 1998.

[22] G. Ramalingam, “Data flow frequency analysis,” *SIGPLAN Notices*, vol. 31, no. 5, pp. 267–277, May 1996, *Proceedings of the ACM SIGPLAN ’96 Conference on Programming Language Design and Implementation*.

[23] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, “Accurate static estimators for program optimization,” *SIGPLAN Notices*, vol. 29, no. 6, pp. 85–96, June 1994, *Proceedings of the ACM SIGPLAN ’94 Conference on Programming Language Design and Implementation*.

[24] M. Hecht, *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.

[25] S. S. Muchnick, *Advanced Compiler Design & Implementation*. Morgen Kaufmann, 1997.

[26] W. Landi, B. G. Ryder, and S. Zhang, “Interprocedural side effect analysis with pointer aliasing,” *SIGPLAN Notices*, vol. 28, no. 6, pp. 56–67, June 1993, *Proceedings of the ACM SIGPLAN ’93 Conference on Programming Language Design and Implementation*.

[27] C. Bauer, A. Frink, and R. Krechel, “Introduction to the GiNaC framework for symbolic computation within the C++ programming language,” *Journal of Symbolic Computation*, vol. 33, pp. 1–12, 2002.

[28] T. Ball, P. Mataga, and M. Sagiv, “Edge profiling versus path profiling: the showdown,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 1998, pp. 134–148.

[29] F. Gabbay and A. Mendelson, “Can program profiling support value prediction?” in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, 1997, pp. 270–280.

[30] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural dataflow analysis via graph reachability,” in *Conference Record of POPL ’95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, San Francisco, California, Jan. 1995, pp. 49–61.

[31] 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*, July 1995, pp. 29–39.

[32] J. K. Lee, D. Ho, and Y. C. Chuang, “Data distribution analysis and optimization for pointer-based distributed programs,” in *Proceedings of the 26th International Conference on Parallel Processing (ICPP)*. Bloomingdale, IL USA: IEEE Computer Society, August 1997, pp. 56–63.

[33] Y.-S. Hwang, P.-S. Chen, J. K. Lee, and R. D.-C. Ju, “Probabilistic points-to analysis,” in *Proceedings of the 2001 International Workshop on Languages and Compilers for Parallel Computing*, August 2001.