# Compiler Support for Speculative Multithreading Architecture with Probabilistic Points-to Analysis[*]

Peng-Sheng Chen[†] Ming-Yu Hung[†] 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

Speculative multithreading (SpMT) architecture can exploit thread-level parallelism that cannot be identified statically. Speedup can be obtained by speculatively executing threads in parallel that are extracted from a sequential program. However, performance degradation might happen if the threads are highly dependent, since a recovery mechanism will be activated when a speculative thread executes incorrectly and such a recovery action usually incurs a very high penalty. Therefore, it is essential for SpMT to quantify the degree of dependences and to turn off speculation if the degree of dependences passes certain thresholds. This paper presents a technique that quantitatively computes dependences between loop iterations and such information can be used to determine if loop iterations can be executed in parallel by speculative threads. This technique can be broken into two steps. First probabilistic points-to analysis is performed to estimate the probabilities of points-to relationships in case there are pointer references in programs, and then the degree of dependences between loop iterations is computed quantitatively. Preliminary experimental results show compiler-directed thread-level speculation based on the information gathered by this technique can achieve significant performance improvement on SpMT.

## Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—*Compilers, Optimization*; B.1.4 [**Hardware**]: Microprogram Design Aids—*Languages and compilers*

## General Terms

Algorithms Design Languages Measurement Performance Theory

## Keywords

Speculative multithreading, dependence analysis, probabilistic points-to analysis, parallelization

## 1. INTRODUCTION

Speculative multithreading (SpMT) architecture has been proposed by researchers to dynamically exploit parallelism in an application [11, 20, 25, 31, 35]. In the SpMT architecture, threads can be extracted from a sequential program and speculatively executed in parallel. When the execution of parallel threads violates a data dependence specified by the sequential code, a recovery mechanism will be activated to ensure the correct sequential semantics. Furthermore, SpMT can utilize parallelism among noncontiguous regions of a program. As a result, more parallelism can be exploited than the parallelism that the compiler can usually identify statically.

Although the SpMT architecture can automatically exploit thread-level parallelism and handle recovery if misspeculation happens, compilers play an important role in achieving maximal performance. The reason is because every recovery action incurs hefty penalty and the performance improvement gained by speculative parallelization might be nullified by the recover overheads. Compilers can avoid such performance degradation by analyzing the possibilities of conflicts between speculative threads and turning off speculation if the possibilities are over certain thresholds. Therefore, it is necessary for SpMT to incorporate a compiler that can compute quantitatively the possibilities of data and control dependences among speculation candidates in a program before execution. The goal of this work is to develop the essential analysis techniques for the SpMT compiler to compute the possibilities of dependences between speculation candidates.

Dependences between speculation candidates, such as different loop iterations or non-overlapped code regions, can be computed by comparing the read and write references between them. However, if the program contains pointer references, the possibilities of conflicts can not be computed since conventional pointer analysis techniques do not provide quantitative descriptions to tell how likely the pointers are aliased [2, 6, 8, 9, 10, 21, 27, 28, 29, 32, 38, 39,

[*]The work was supported in part by NSC under grant no. NSC91-2213-E007-021, NSC91-2215-E007-035, NSC90-2213-E-019-016, NSC91-2213-E-019-001, NSF under grant. no NSF-CCR-0096383, MOE research excellent project under grant no. 89-E-FA04-1-4, and MOEA research project under grant no. 91-EC-17-A-03-S1-0002.

*PPoPP'03,* June 11–13, 2003, San Diego, California, USA.

40]. These techniques only classify points-to relationships into *definitely*-points-to relationships, which hold for all executions, and *possibly*-points-to relationships, which might hold for some executions. *Possibly*-points-to relationships cannot tell how likely the conditions will hold for the executions, and consequently the compiler has to make a conservative guess and assume the conditions hold for all executions. This paper addresses this issue by presenting a *probabilistic points-to analysis* approach to give a quantitative description for each points-to relationship to represent the probability that it holds.

Once the probability of every points-to relationship is computed, the quantitative computation of dependences between speculation candidates can be proceeded. The results will be used to guide the thread speculation in order to reduce the impact of recovery penalties. Preliminary experimental results show compiler-directed thread-level speculation based on the information gathered by this technique can achieve significant performance improvement on SpMT.

The rest of this paper is organized as follows. Section 2 provides a description on SpMT and its cost model. Section 3 describes the probabilistic data flow analysis framework for the probabilistic points-to analysis. Section 4 details how to compute data dependence probability and the issues on applying of this information on SpMT. Experimental results will be presented in Section 5 and the related work is compared in Section 6. Section 7 summarizes this paper.

## 2. SPECULATIVE MULTITHREADING

In a speculative multithreading (SpMT) model [11, 20, 25, 31, 35] threads are extracted from sequential codes and speculatively executed in parallel without violating the sequential program semantics. If there is a violation of dependence, the hardware must ensure that illegal status will be fixed and the mis-speculated thread will re-execute with proper data. A compiler-guided speculation can reduce the possibilities of mis-speculations, and thus result in better performance.

### 2.1 Architecture and Simulator

This work uses an execution driven simulator *SIMCA* to model the SpMT architecture. *SIMCA* is developed by ARCTiC Lab at University of Minnesota. It is based on the SimpleScalar simulator [7]. *SIMCA* simulates the hardware component interaction in the superthreaded architecture [35, 36]. The superthreaded architecture combines compiler-directed thread-level speculation of control-dependence with runtime verification of data dependence. The execution of a thread in the superthreaded model is partitioned into several stages. Figure 1 shows the stages of a thread pipelining model.

**Continuation Stage**

This stage is to compute recurrence variables, such as loop index variables. These variables are forwarded to the next thread before the next thread is activated. The end of the continuation stage is a fork instruction that spawns the next thread.

**Target-Store-Address-Generation (TSAG) Stage**

This stage stores the data addresses that will be later used by other threads. The hardware will resolve data dependences during program execution.

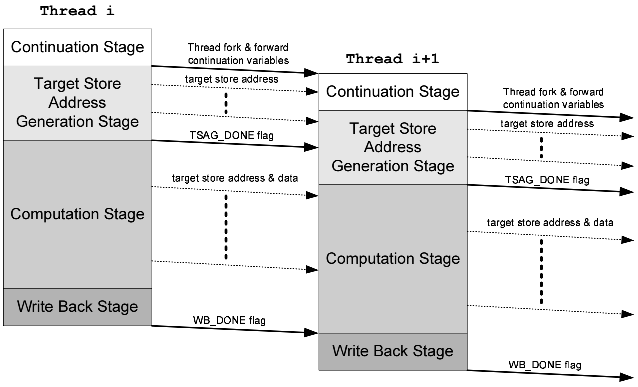

Thread i
Continuation Stage
Target Store Address Generation Stage
Computation Stage
Write Back Stage
Thread fork & forward continuation variables
target store address
TSAG_DONE flag
target store address & data
WB_DONE flag
Thread i+1
Continuation Stage
Target Store Address Generation Stage
Computation Stage
Write Back Stage
Thread fork & forward continuation variables
target store address
TSAG_DONE flag
target store address & data
WB_DONE flag

**Figure 1: Thread Pipelining Execution Model**

**Computation Stage**

The is the parallel computation stage for threads. The *abort_future* instruction can be used to explicitly abort successor threads in case of mis-speculation.

**Write-Back Stage**

The thread completes its execution by writing all of he data from its memory buffer into memory. To maintain the correct memory state, the threads perform their write-back stages in their original order.

In this environment, thread and data speculations can be accomplished by the *abort_future* instruction. When dependence is detected at runtime, it will discard the thread and invoke recovery mechanisms.

### 2.2 Cost Model

Once loop iterations are distributed into threads, a compiler must decide if it is profitable to execute these threads speculatively. The following cost model can be used by the compiler to make this decision:

$$L_o > L_s + \sum_{set} \left(V_{freq}(set) \times \left(O_r^{set} + L_c^{set}\right)\right) \qquad (1)$$

where *set* represents each violation relationship, $L_o$ is the execution time of original codes, $L_s$ is the execution time of speculative threads without recovery codes, $V_{freq}(set)$ is the violation frequency for *set*, $O_r^{set}$ is the overhead of *set* to invoke recovery codes, and $L_c^{set}$ is the time needed to actually execute recovery codes of *set*. For simplicity, this paper conservatively assumes each violation relationship is independent.

Since multithreading execution is out-of-order, a data dependence between speculated threads does not always lead to dependence violation for each execution. Suppose $E$ represents the violation ratio when the dependence exists. The violation frequency $V_{freq}(set)$ can be computed as the product of $E$ and the probability of data dependence between threads for *set*, $P_{dep}(set)$. For each *set*, the difference in $O_r^{set}$ and $L_c^{set}$ is very small so that it can be ignored. Consequently, Equation (1) can be written as follows.

$$\begin{aligned} L_o &> L_s + \sum_{set} \left(P_{dep}(set) \times E \times \left(O_r^{set} + L_c^{set}\right)\right) \\ &\simeq L_s + E \times (O_r + L_c) \times \sum_{set} P_{dep}(set) \qquad (2) \end{aligned}$$

$E$ can be viewed as constant and evaluated by experimental results, while $\sum P_{dep}(set)$ will be the dominating factor can be computed using the results of the probabilistic points-to analysis.

# 3. PROBABILISTIC POINTS-TO ANALYSIS

## 3.1 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

$$\mathcal{P}(s, \langle p, v\rangle) \overset{\text{def}}{=} \begin{cases} \frac{E(s, \langle p, v\rangle)}{E(s)} & \text{if } E(s) \neq 0 \\ 0 & \text{otherwise} \end{cases} \qquad (3)$$

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 $s$ [26].

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

$$\mathcal{P}(s, A) \overset{\text{def}}{=} \{\mathcal{P}(s, \langle p, v\rangle) \mid \langle p, v\rangle \in A\}$$

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

### Program Representations and Normalization

Programs will be represented by control flow graphs (CFGs) whose edges are labeled with a static assigned execution frequency [26, 37] 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 *join* node of each conditional.

Programs will be normalized such that each pointer assignment statement is one of the four basic pointer assignment statements listed in the following table [28]:

| | |
|---|---|
| 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 $\star p = nil$.

## 3.2 Algorithm Outline

The conventional points-to analysis can be formulated as a data flow framework [6, 12, 24]. 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 $F_S$:

$$OUT_S = F_S\,(IN_S)$$

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 $F_S$:

$$\begin{aligned} \mathcal{P}(S_{out}, OUT_S) &= F_S(\mathcal{P}(S_{in}, IN_S)) \\ &= \{F_S(\mathcal{P}(S_{in}, \langle p, v\rangle)) \mid \langle p, v\rangle \in IN_S\} \end{aligned}$$

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

### 3.2.1 Basic Pointer Assignment Statements

Figure 2 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 [10, 28, 38]. 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.

| $S$ | $OUT_S = F_S(IN_S)$ |
|---|---|
| $p = \&q$ | $IN_S \cup \{(\langle p, q\rangle, true)\}$ |
| $p = q$ | $IN_S \cup \{(\langle p, v\rangle, rel) \mid (\langle q, v\rangle, rel) \in IN_S\}$ |
| $p = \star q$ | $IN_S \cup \{(\langle p, v\rangle, \bigvee_x (rel_1^x \wedge rel_2^x)) \mid$<br>$\forall_x\, ((\langle q, x\rangle, rel_1^x), (\langle x, v\rangle, rel_2^x) \in IN_S)\}$ |
| $\star x = q$ | $IN_S \cup \{(\langle p, v\rangle, rel_1 \wedge rel_2) \mid$<br>$(\langle x, p\rangle, rel_1), (\langle q, v\rangle, rel_2) \in IN_S\}$ |
| $p = nil$ | $IN_S - \{(\langle p, v\rangle, rel) \in IN_S\}$ |
| $\star x = nil$ | $IN_S - \{(\langle p, v\rangle, rel) \mid (\langle x, p\rangle, rel), (\langle p, v\rangle, rel) \in IN_S\}$<br>$\cup\ \{(\langle p, v\rangle, false) \mid$<br>$(\langle x, p\rangle, false), (\langle p, v\rangle, rel) \in IN_S\}$ |

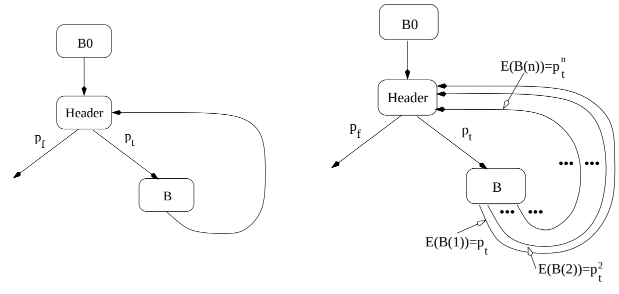
**Figure 2: Computing the Set of Points-to Relationships**

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 3 presents the formula to compute the probability function $\mathcal{P}(S_{out}, \langle p, v\rangle)$ of every points-to relationship $\langle p, v\rangle \in OUT_S$ 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 $\mathcal{P}(S_{out}, \langle x, y\rangle) = \mathcal{P}(S_{in}, \langle x, y\rangle)$.

### 3.2.2 Meet Operator $\sqcap$

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

**Figure 3: Computing the Probability Functions**



(a) Actual CFG

(b) Imaginary CFG

**Figure 4: Loops**

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 $B2_{out}$ after $B1$ and $B2$ are $\mathcal{P}(B1_{out},\,\langle p,\,v\rangle)$ and $\mathcal{P}(B2_{out},\,\langle p,\,v\rangle)$, respectively. Then the probability function of the points-to relationship $\langle p,\,v\rangle$ at the join node will be

$$\begin{aligned} &\mathcal{P}(Join_{in},\,\langle p,\,v\rangle) \\ &\quad = \mathcal{P}(B1_{out},\,\langle p,\,v\rangle) \sqcap \mathcal{P}(B2_{out},\,\langle p,\,v\rangle) \\ &\quad \stackrel{\text{def}}{=} \frac{\mathcal{P}(B1_{out},\,\langle p,\,v\rangle) \times E(B1) + \mathcal{P}(B2_{out},\,\langle p,\,v\rangle) \times E(B2)}{E(B1) + E(B2)} \end{aligned}$$

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 relationships:

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

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

### 3.2.3 Conditionals

The most commonly used conditionals is the *if-then-else* construct. Suppose $OUT_{Then}$ and $OUT_{Else}$ 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 probability 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:

$$\mathcal{P}(Join_{in},\,\langle p,\,v\rangle) = \mathcal{P}(Then_{out},\,\langle p,\,v\rangle) \sqcap \mathcal{P}(Else_{out},\,\langle p,\,v\rangle)$$

### 3.2.4 Loops

Since a loop can iterate an arbitrary number of times, it can be imagined as if there were an unbounded 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 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 $i$th iteration is $p_t^i$, i.e. $E(B[i]) = p_t^i$ where $B[i]$ denotes $B$ at $i$th 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

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

In order to find a solution of the above equation, a symbolic probability will be assigned to each probability function as its value at the entry of the header node. Since the probability functions $\mathcal{P}(B_{out},\,OUT_B)$ will be computed by the transfer function $F_B$ with the vector $\mathcal{P}(B_{in},\,IN_B)$ as its argument, the vector returned by $\mathcal{P}(B_{out},\,OUT_B)$ will be functions of these symbolic probabilities. As a result, the equation $\mathcal{P}(Header_{in},\,IN_{Header}) = \mathcal{P}(B0_{out},\,OUT_{B0}) \sqcap \mathcal{P}(B_{out},\,OUT_B)$ is in fact a linear system, and the values of symbolic probabilities can be computed by solving the linear 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. $\mathcal{P}(Header_{in},\,\langle p,\,v\rangle) = P$, 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* $\mathcal{P}(B_{out},\,\langle p,\,v\rangle) = F_B(\mathcal{P}(B_{in},\,\langle p,\,v\rangle)) = 0.9P$. Then the symbolic probability $P$ can be solved:

$$\begin{aligned} \mathcal{P}(Header_{in},\,IN_{Header}) &= \mathcal{P}(B0_{out},\,OUT_{B0}) \sqcap \\ &\quad\ \mathcal{P}(B_{out},\,OUT_B) \\ P &= (1 \times 1 + 10 \times 0.9P) \,/\, (1 + 10) \\ P &= 0.5 \end{aligned}$$

## 3.3 Interprocedural Analysis

### 3.3.1 Algorithm

The algorithm for interprocedural probabilistic points-to analysis is developed based on the algorithm developed by Emami et al. [10]. 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 [10, 28].

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. [10] and Wilson and Lam [38] 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 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 [28, 38], 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.

### 3.3.2 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 for recursive procedures. 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 linear system for the meet operation on the incoming-edges to the entry node.

## 3.4 Example

Consider the example shown in Figure 5. Assume the branching probabilities of the two $if$ branches are both 0.9. The program calls a recursive function $func$ after creating the points-to relationships $\langle p,\, v\rangle$ and $\langle q,\, v\rangle$ by $S1$ and $S2$, respectively. Figure 5 depicts the iterations performed by the interprocedural analysis to reach a fixed point. Both the sets of points-to relationships at the entry and exit of every statement $Si$, e.g. $IN_{Si}$ and $OUT_{Si}$, are shown. However, in order to save space, only the $OUT_{Si}$ sets of some statements are displayed since these statements pass the $IN_{Si}$ set directly to the $OUT_{Si}$ set.

After the mapping process at call site $S3$ and introducing symbolic probabilities at the function entry, the set $IN_{S11}$ contains the tuples $[\langle x,\, x_1\rangle,\, P_1]$, $[\langle y,\, y_1\rangle,\, P_2]$, $[\langle x_1,\, x_2\rangle,\, P_3]$, and $[\langle y_1,\, y_2\rangle,\, P_4]$ at iteration 1. These tuples will be propagated and transformed by statements, and the set of the probabilistic points-to relationships that reach $S17_{in}$, consists of tuples $[\langle x, x_1\rangle, P_1]$, $[\langle y, y_1\rangle, P_2]$, $[\langle x_1, x_2\rangle, (1-0.9P_1)P_3]$, $[\langle y_1, y_2\rangle, (1-0.1P_2)P_4]$, $[\langle x_1, y_2\rangle, 0.9P_1P_2P_4]$, and $[\langle y_1, x_2\rangle, 0.1P_1P_2P_3]$. Since the effect of the recursive function $func$ will not known until the iterations reach a fixed point, symbolic probabilities are introduced at the end of call site $S17_{out}$.

At iteration 2, the symbolic probabilities can be resolved since the points-to relationships of $x$ and $y$ are not modified by any statements in $func$, and the result is $P_1 = P_2 = 1$. Furthermore, two new points-to relationships $\langle x_1,\, y_2\rangle$ and $\langle y_1,\, x_2\rangle$ are merged into the set $S11_{in}$, and hence two more symbolic probabilities $P_5$ and $P_6$ and the set $IN_{S11}$ will be comprised of tuples $\{[\langle x,\, x_1\rangle,\, 1]$, $[\langle y,\, y_1\rangle,\, 1]$, $[\langle x_1,\, x_2\rangle,\, P_3]$, $[\langle y_1,\, y_2\rangle,\, P_4]$, $[\langle x_1,\, y_2\rangle,\, P_5]$, and $[\langle y_1,\, x_2\rangle,\, P_6]$. The tuples will be transformed by statements and reach the end of $func$ with different probability functions, i.e. they are $[\langle x,\, x_1\rangle,\, 1]$, $[\langle y,\, y_1\rangle,\, 1]$, $[\langle x_1,\, x_2\rangle,\, 0.1(0.9P_6 + 0.1P_3) + 0.9P'_3]$, $[\langle y_1,\, y_2\rangle,\, 0.1(0.9P_4 + 0.1P_5) + 0.9P'_4]$, $[\langle x_1,\, y_2\rangle,\, 0.1(0.9P_4 + 0.1P_5) + 0.9P'_5]$, and $[\langle y_1,\, x_2\rangle,\, 0.1(0.9P_6 + 0.1P_3) + 0.9P'_6]$.

All the sets of probabilistic points-to relationships will converge at iteration 3, since no new tuples will be generated. Now the symbolic probabilities can be computed by solving the following linear system that is obtained from the equation $\mathcal{P}(S11_{in}, IN_{S11}) = \mathcal{P}(S3_{in}, IN_{S3}) \sqcap \mathcal{P}(S17_{in}, IN_{S17})$:

$$\begin{aligned}
P_3 &= (1 + 9 \times (0.1P_3 + 0.9P_6))/(1 + 9)\\
P_4 &= (1 + 9 \times (0.9P_4 + 0.1P_5))/(1 + 9)\\
P_5 &= (9 \times (0.9P_4 + 0.1P_5))/(1 + 9)\\
P_6 &= (9 \times (0.1P_3 + 0.9P_6))/(1 + 9)
\end{aligned}$$

As a result, the set of probabilistic points-to relationships $IN_{S11}$ at the entry of recursive function $func$ contains $[\langle x, x_1\rangle, 1]$, $[\langle y, y_1\rangle, 1]$, $[\langle x_1, x_2\rangle, 0.19]$, $[\langle y_1, y_2\rangle, 0.91]$, $[\langle x_1, y_2\rangle, 0.81]$, and $[\langle y_1, x_2\rangle, 0.09]$. Furthermore, the probabilities of points-to relationships at the end of $func$ can be determined by solving the following equations

$$\begin{aligned}
P'_3 &= 0.1(0.9P_6 + 0.1P_3) + 0.9P'_3\\
P'_4 &= 0.1(0.9P_4 + 0.1P_5) + 0.9P'_4\\
P'_5 &= 0.1(0.9P_4 + 0.1P_5) + 0.9P'_5\\
P'_6 &= 0.1(0.9P_6 + 0.1P_3) + 0.9P'_6
\end{aligned}$$

The result will be $P'_3 = P'_6 = 0.9$ and $P'_4 = P'_5 = 0.1$.

| Program | $IN_{Si}/OUT_{Si}$ (Iteration 1) | $IN_{Si}/OUT_{Si}$ (Iteration 2) |
|---|---|---|
| $S : p = \&v;$ | - | - |
|  | $[\langle p, v\rangle, 1]$ | $[\langle p, v\rangle, 1]$ |
| $S2 : q = \&u;$ | $[\langle p, v\rangle, 1]$ | $[\langle p, v\rangle, 1]$ |
|  | $[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$ | $[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$ |
| $S3 : func(\&p, \&q);$ | $[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$ | $[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$ |
|  | - | $[\langle p, v\rangle, 1]\,[\langle q, u\rangle, 1]$ |
| $S11 : func(x, y)\{$<br>$int \star\star x;$<br>$int \star\star y;$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$ |
| $S12 : \quad if(...)$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$ |
| $S13 : \quad t = \star y;$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$ |
|  | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle t, y_2\rangle, P_2P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$<br>$[\langle t, y_2\rangle, P_4]\,[\langle t, x_2\rangle, P_6]$ |
| $S13' : \quad \star\, x = t;$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle t, y_2\rangle, P_2P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$<br>$[\langle t, y_2\rangle, P_4]\,[\langle t, x_2\rangle, P_6]$ |
|  | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, (1 - P_1)P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_1P_2P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle y_1, y_2\rangle, P_4]\,[\langle y_1, x_2\rangle, P_6]$<br>$[\langle x_1, y_2\rangle, P_4]\,[\langle x_1, x_2\rangle, P_6]$ |
| $S14 : \quad else$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$ |
| $S15 : \quad t = \star x;$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$ |
|  | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle t, x_2\rangle, P_1P_3]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$<br>$[\langle t, x_2\rangle, P_3]\,[\langle t, y_2\rangle, P_5]$ |
| $S15' : \quad \star\, y = t;$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle t, x_2\rangle, P_1P_3]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_4]$<br>$[\langle x_1, y_2\rangle, P_5]\,[\langle y_1, x_2\rangle, P_6]$<br>$[\langle t, x_2\rangle, P_3]\,[\langle t, y_2\rangle, P_5]$ |
|  | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, (1 - P_2)P_4]$<br>$[\langle y_1, x_2\rangle, P_1P_2P_3]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3]\,[\langle x_1, y_2\rangle, P_5]$<br>$[\langle y_1, x_2\rangle, P_3]\,[\langle y_1, y_2\rangle, P_5]$ |
| $S16 : \quad if(...)$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, 0.9(1 - P_1)P_3 + 0.1P_3]$<br>$[\langle y_1, y_2\rangle, 0.1(1 - P_2)P_4 + 0.9P_4]$<br>$[\langle x_1, y_2\rangle, 0.9P_1P_2P_4]$<br>$[\langle y_1, x_2\rangle, 0.1P_1P_2P_3]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, 0.9P_6 + 0.1P_3]$<br>$[\langle y_1, y_2\rangle, 0.9P_4 + 0.1P5]$<br>$[\langle x_1, y_2\rangle, 0.9P_4 + 0.1P_5]$<br>$[\langle y_1, x_2\rangle, 0.9P_6 + 0.1P_3]$ |
| $S17 : \quad func(x, y)$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, (1 - 0.9P_1)P_3]$<br>$[\langle y_1, y_2\rangle, (1 - 0.1P_2)P_4]$<br>$[\langle x_1, y_2\rangle, 0.9P_1P_2P_4]$<br>$[\langle y_1, x_2\rangle, 0.1P_1P_2P_3]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, 0.9P_6 + 0.1P_3]$<br>$[\langle y_1, y_2\rangle, 0.9P_4 + 0.1P_5]$<br>$[\langle x_1, y_2\rangle, 0.9P_4 + 0.1P_5]$<br>$[\langle y_1, x_2\rangle, 0.9P_6 + 0.1P_3]$ |
|  | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, P_3']\;\;[\langle y_1, y_2\rangle, P_4']$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, P_3']\;\;[\langle y_1, y_2\rangle, P_4']$<br>$[\langle x_1, y_2\rangle, P_5']\,[\langle y_1, x_2\rangle, P_6']$ |
| $S18 :\}$ | $[\langle x, x_1\rangle, P_1]\,[\langle y, y_1\rangle, P_2]$<br>$[\langle x_1, x_2\rangle, 0.1(1 - 0.9P_1)P_3 + 0.9P_3']$<br>$[\langle y_1, y_2\rangle, 0.1(1 - 0.1P_2)P_4 + 0.9P_4']$<br>$[\langle x_1, y_2\rangle, 0.09P_1P_2P_4]$<br>$[\langle y_1, x_2\rangle, 0.01P_1P_2P_3]$ | $[\langle x, x_1\rangle, 1]\,[\langle y, y_1\rangle, 1]$<br>$[\langle x_1, x_2\rangle, 0.1(0.9P_6 + 0.1P_3) + 0.9P_3']$<br>$[\langle y_1, y_2\rangle, 0.1(0.9P_4 + 0.1P_5) + 0.9P_4']$<br>$[\langle x_1, y_2\rangle, 0.1(0.9P_4 + 0.1P_5) + 0.9P_5']$<br>$[\langle y_1, x_2\rangle, 0.1(0.9P_6 + 0.1P_3) + 0.9P_6']$ |

**Figure 5: Interprocedural Analysis Example**

## 3.5 Naming Heap Objects

Heap objects are modeled as array elements. For every `malloc(n)` statement, an array with a unspecified number of elements, each with the size `n`, will be created. Each heap object allocated from the same `malloc` statement will be assigned a unique symbolic index, which can be formulated based on the enclosing loop iterations and calling contexts. Consider the following EM3D code fragment that builds lists of E and H nodes:

```
main( ) {
  elist = make_list(N);              // S1
  hlist = make_list(N);              // S2
  make_neighbor_list(elist, hlist); // S3
  make_neighbor_list(hlist, elist); // S4
  for (...) {
    compute(elist);                  // S5
    compute(hlist);                  // S6
  }
}
Node *make_list(int size) {
  list = null;
  for (int i = 0; i < size; i++) {
    p = malloc(Node); // S7 => MallocS7[ ] is
                      // introduced, i.e.
                      // p=&MallocS7[symbolic_index]
    p.next = list;
    list = p;
  }
  return list;
}
```

A symbolic index `main@s1:i` will be associated with the heap object that is allocated by `S7` at the ith iteration of the loop in `make_list` called by `S1` in `main`. In other words, `S7` is equivalent to `p = &MallocS7[main@s1:i]`, where `main@s1` denotes the calling context and `i` represents the loop iteration. Similarly, `S7` can be viewed as `p=&MallocS7[main@s2:i]` when `make_list` is called by `S2`.

The points-to relationships that are generated after S1 is executed include $[\langle\texttt{elist}, \&\texttt{MallocS7}[\texttt{main@S1:N-1}]\rangle, 1]$, $[\langle\texttt{MallocS7}[\texttt{main@S1:i}]\texttt{.next}, \&\texttt{MallocS7}[\texttt{main@S1:i-1}]\rangle, 1]$, $1 \le \texttt{i} \le \texttt{N} - 1$. Similarly, S2 introduces the following points-to relationships: $[\langle\texttt{hlist}, \&\texttt{MallocS7}[\texttt{main@S2:N-1}]\rangle, 1]$, $[\langle\texttt{MallocS7}[\texttt{main@S2:i}]\texttt{.next}, \&\texttt{MallocS7}[\texttt{main@S2:i-1}]\rangle, 1]$, $1 \le \texttt{i} \le \texttt{N} - 1$. This example shows the symbols `main@S1` and `main@S2` denote the calling contexts and consequently a compiler can recognize that `elist` and `hlist` point to different portions of the virtual array `MallocS7`.

Further consider the code fragment in EM3D that constructs the neighbor lists of both lists:

```
make_neighbor_lists(list1, list2) {
  for (p = list1; p; p = p.next) {
    p.nlist = malloc(NEIGH);  // S8
    for (int i = 0; i < NEIGH; i++)
      p.nlist[i] = &list2[random()];
  }
}
```

New points-to relationships generated by this function after the call site `S3` will be $[\langle\texttt{MallocS7}[\texttt{main@S1:N-i-1}]\texttt{.nlist}, \&\texttt{MallocS8}[\texttt{main@S3:i}]\rangle, 1]$, and $[\langle\texttt{MallocS8}[\texttt{main@S3:i}][\texttt{j}], \&\texttt{MallocS7}[\texttt{main@S2:?}]\rangle, 1]$, $0 \le \texttt{i} \le \texttt{N} - 1$, $0 \le \texttt{j} \le \texttt{NEIGH} - 1$. Similarly, S4 introduces points-to relationships $[\langle\texttt{MallocS7}[\texttt{main@S2:N-i-1}]\texttt{.nlist}, \&\texttt{MallocS8}[\texttt{main@S4:i}]\rangle, 1]$, and $[\langle\texttt{MallocS8}[\texttt{main@S4:i}][\texttt{j}], \&\texttt{MallocS7}[\texttt{main@S1:?}]\rangle, 1]$, $0 \le \texttt{i} \le \texttt{N} - 1$, $0 \le \texttt{j} \le \texttt{NEIGH} - 1$. Question marks are used since the indexes are generated by a random function. However, it should not hurt since it gives the compiler enough information for dependence analysis.

# 4. DATA DEPENDENCE PROBABILITY

This section shows how to compute the probabilities of data dependences using the probabilistic points-to analysis ($PPA$) information.

## Definition

Consider the memory object $p$ referenced at program point $S_1$, denoted as $p_{S_1}$, and memory object $q$ referenced at program point $S_2$, denoted as $q_{S_2}$. The probability $\mathcal{P}_{S_1\,\delta\,S_2}$ that $S_2$ depends on $S_1$ due to a flow dependence from $p_{S_1}$ to $q_{S_2}$ is defined as follows:

$$\mathcal{P}_{S_1\,\delta\,S_2} \overset{\text{def}}{=} \frac{E(S_2,\, p_{S_1}\;\delta\, q_{S_2})}{E(S_2)} \tag{4}$$

where $E(S_2)$ is the number of times $S_2$ is executed during execution and $E(S_2,\, p_{S_1}\;\delta\, q_{S_2})$ is the number of times the flow dependence relationship between $p_{S_1}$ and $q_{S_2}$ holds.

A flow dependence relationship $S_1\,\delta\,S_2$ exists when the value of $p_{S_1}$ defined at $S_1$ flows to $S_2$ and is referenced by $q_{S_2}$. Since memory objects can be defined through a variable or a pointer, $p$ may not be defined every time $S_1$ is visited during the execution and furthermore the value of $p$ may be modified by statements between $S_1$ and $S_2$. A flow dependence relationship is generated only when $p$ is defined at $S_1$ reaches $S_2$, $q$ is referenced at $S_2$, $p$ and $q$ are aliases at $S_2$. Therefore, let $\mathcal{P}^{S_2}_{DEF}(p_{S_1})$ be the value of $p$ defined at $S_1$ reaches $S_2$, $\mathcal{P}_{REF}(q_{S_2})$ be the probability that $q$ is referenced at $S_2$ and $\mathcal{P}_{alias}(S_2, (p_{S_1}, q_{S_2}))$ be the probability that $p$ and $q$ are aliases at $S_2$, the above equation can be computed as follows:

$$\begin{aligned}
&\mathcal{P}_{S_1\,\delta\,S_2}\\
&= \frac{\sum_{q_{S_2}} \mathcal{P}^{S_2}_{DEF}(p_{S_1}) \times \mathcal{REF}(q_{S_2})}{E(S_2)} \times \mathcal{P}_{alias}(S_2, (p_{S_1}, q_{S_2}))\\
&= \mathcal{P}^{S_2}_{DEF}(p_{S_1}) \times \frac{\sum_{q_{S_2}} \mathcal{REF}(q_{S_2})}{E(S_2)} \times \mathcal{P}_{alias}(S_2, (p_{S_1}, q_{S_2}))\\
&= \mathcal{P}^{S_2}_{DEF}(p_{S_1}) \times \mathcal{P}_{REF}(q_{S_2}) \times \mathcal{P}_{alias}(S_2, (p_{S_1}, q_{S_2}))
\end{aligned}$$

$\mathcal{P}^{S_2}_{DEF}(p_{S_1})$ and $\mathcal{P}_{REF}(q_{S_2})$ will be computed by probabilistic data flow framework presented by Ramalingam [26]. The complex part $\mathcal{P}_{alias}(S_2, (p_{S_1}, q_{S_2}))$ will be evaluated by the probabilistic points-to analysis. Similar formulas will be derived for anti-dependence or output dependence relationships. For conciseness, only flow dependence relationships will be presented in the rest of section.

### *Example 1*

Consider the following program fragment.

```
S1:  if (...)
S2:     p = &M
S3:  M = ...
     ...
S5:  ... = *p;
```

If $p$ points to $M$ at $S_5$, then there is a flow dependence from $M$ at $S_3$ to $*p$ at $S_5$. Assume $\mathcal{P}_{alias}(S_5, (M_{S_3},\, *p_{S_5}))$ computed by PPA is 0.8. Since both $\mathcal{P}^{S_5}_{DEF}(M,\, S_3)$ and $\mathcal{P}_{REF}(*p,\, S_5)$ are both equal to 1, The probability of the flow dependence relationship $S_3\,\delta\,S_5$ exists can be computed as follows:

$$\begin{aligned}
\mathcal{P}_{S_3\,\delta\,S_5} &= \mathcal{P}^{S_5}_{DEF}(M,\, S_3) \times \mathcal{P}_{REF}(*p,\, S_5) \times\\
&\quad\ \mathcal{P}_{alias}(S_5, (M_{S_3},\, *p_{S_5}))\\
&= 1 \times 1 \times 0.8\\
&= 0.8
\end{aligned}$$

### *Example 2*

Consider another example that introduces pointer-induced loop-carried data dependences.

```
S1:  while (...) {
S2:     p = &K;
S3:     if (...)
S4:       K = ...
        ...
S5:     ... = *q
        ...
S6:     if (...)
S7:       q = p;
     }
```

If p and q are aliased, there will be a flow dependence relationship between $S_4$ and $S_5$. Assume the probabilities that the statement $S_4$ and $S_7$ enclosed by $IF$-construct will be executed are 0.8 and 0.2 respectively, and assume the probability the condition of the $WHILE$-construct will be true at $S_1$ is 0.9. Then the probability of the flow dependence relationship $S_4\,\delta\,S_5$ can be computed as follows:

$$\begin{aligned}
\mathcal{P}_{S_4\,\delta\,S_5} &= \mathcal{P}^{S_5}_{DEF}(K,\, S_4) \times \mathcal{P}_{REF}(*q,\, S_5) \times\\
&\quad\ \mathcal{P}_{alias}(S_5, (K_{S_4},\, *q_{S_5}))\\
&= 0.8 \times 1 \times \mathcal{P}_{alias}(S_5, (K_{S_4},\, *q_{S_5}))\\
&= 0.8 \times \mathcal{P}_{Points-to}(S_5, \langle q,\, K\rangle)\\
&= 0.8 \times \frac{0.2 \times 9}{10}\\
&= 0.144
\end{aligned}$$

### *Example 3*

Consider the `compute` function in EM3D.

```
compute(list) {
  for (p = list; p; p = p.next) {
    for (i = 0; i < NEIGH; i++) {
      q = p.nlist[i];
      p.value -= q.coff * q.value;
    }
  }
}
```

Since the compiler can tell $p$ at different iterations points to different heap objects and $q$ points to a different region of the virtual array `MallocS7` (see Section 3.5), the probability of aliases are 0 and hence there are no dependences.

# 5. EXPERIMENTS

This section first compares the estimated probabilities of all points-to relationships by PPA with the probabilities gathered at runtime to show the accuracy of PPA. Then compiler-directed speculation will be performed on an SpMT simulator to demonstrate the impact of performance with the incorporation of dependence analysis and PPA.

## 5.1 PPA

A prototype compiler has been implemented upon the SUIF system [34] and CFG library of MachSUIF [30] to perform the interprocedural probabilistic points-to analysis.

The routine in the SPAN compiler to associate variables with location sets is integrated in the compiler as well [28]. In addition, CAS (Computer Algebra System) GiNaC and GNU Scientific library GSL are used to process symbolic and mathematical computation. 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. 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, as shown in Figure 6.

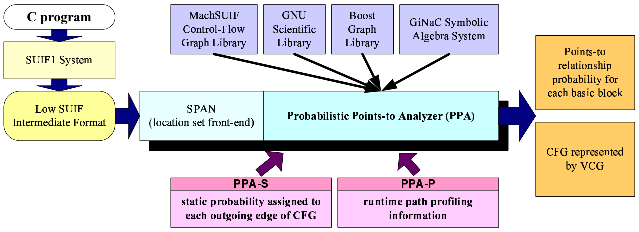

C program
SUIF1 System
Low SUIF Intermediate Format
MachSUIF Control-Flow Graph Library
GNU Scientific Library
Boost Graph Library
GiNaC Symbolic Algebra System
SPAN (location set front-end)
Probabilistic Points-to Analyzer (PPA)
Points-to relationship probability for each basic block
CFG represented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

**Figure 6: Prototype Implementation**

| Program | Description |
|---|---|
| *990127-1* | Test program from gcc-3.0.1 testsuite. |
| *shuffle* | The program tests a random number generator using a card shuffling procedure. (netlib.org) |
| *20000801-2* | Test program from gcc-3.0.1 testsuite |
| *fir2dim* | DSPstone filter benchmark. |
| *misr* | A program create and use link list. (McGill) |
| *fft* | An FFT test program. (netlib.org) |
| *dhrystone* | The dhrystone benchmark v2.1. |
| *clinpack* | This is the Linpack program (floating-point) rewritten by C. (netlib.org) |
| *alvinn* | This program trains a neural network called ALVINN using back propagation. (SPEC92) |
| *queens* | A program that finds solutions to the eight-queens chess problem. (netlib.org) |
| *power* | The Power Pricing benchmark. (Olden) |

**Table 1: Benchmark programs for PPA verification**

| 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 2: Statically Assigned Probabilities**

Several applications have been chosen as the benchmarks, as listed in Table 1. These benchmark programs have been executed to gather the detailed points-to information 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, as listed in Table 2, and the probabilistic points-to analysis algorithm will be executed based on these edge probabilities.
- Probabilistic points-to analysis based on path profiling information (PPA-P)
  A profiling tool has been built upon SUIF to gather the execution frequency of every edge in CFG, and probabilistic points-to analysis will be performed based on the path 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* $\xi$

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

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

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

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

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. The average error of estimated probabilities by PPA-S compared to the runtime frequencies is about 21.70%. With the aid of edge profiling information, PPA-P reduces the average error down to 2.68%. The figures show the probabilistic points-to analysis approach can estimate the likelihood that each points-to relationship would hold with relatively small errors.

This result is significant since most compiler optimizations rely on the ability to determine if points-to relationships hold with high or low probabilities. Let $\textit{Points-to}_{Runtime}(l\% \sim h\%)$ be the set of points-to relationships with runtime-profiled probabilities within the range $l\%{\sim}h\%$. In addition, let $\textit{Points-to}_{PPA}(l\%{\sim}h\%)$ 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

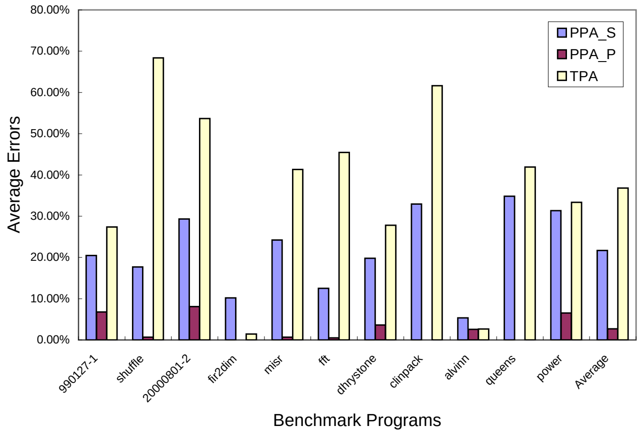


**Figure 7: Average Errors**

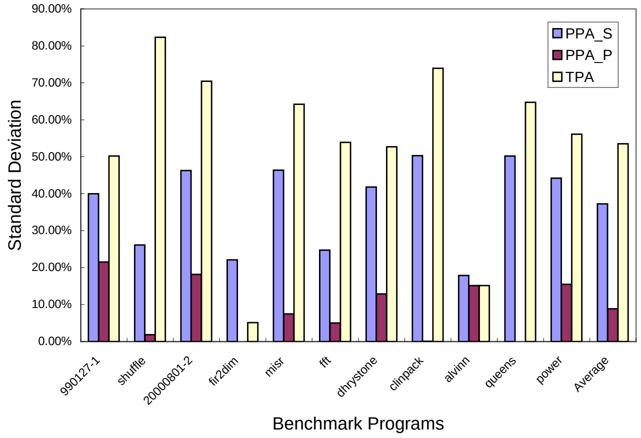


**Figure 8: Standard Deviation**

*Points-to*$_{Runtime}$($l$%∼$h$%). Then the *accuracy within the probability range l%∼h%* of PPA is defined as the ratio of the size of *Points-to*$_{PPA}$($l$%∼$h$%) over the size of *Points-to*$_{Runtime}$($l$%∼$h$%), i.e.

$$accuracy_{PPA}(l\%\sim h\%) = \frac{Points\text{-}to_{PPA}(l\%\sim h\%)}{Points\text{-}to_{Runtime}(l\%\sim h\%)}$$

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-P in the probability range 0%∼10% is 91.89%, while the accuracy of PPA-P in the range 90%∼100% is 96.48%, respectively.

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

## 5.2 Applications on SpMT

### 5.2.1 Simulation

The *SIMCA* simulator has been used to evaluate the performance on SpMT of several benchmark applications. *SIMCA* is developed by ARCTiC Lab at University of Minnesota. It is based on the SimpleScalar simulator, sim-outorder. *SIMCA* simulates the hardware component interaction in the superthreaded architecture [35, 36]. This is used as our simulator platform for experiments.

| Probability Range | PPA-S | PPA-P | PPA-S | PPA-P |
|---|---|---|---|---|
| 0%∼10% | 6.51% | 91.89% | 12.49% | 93.30% |
| 10%∼20% | 20.00% | 60.00% | | |
| 20%∼30% | 25.00% | 60.00% | 33.33% | 61.90% |
| 30%∼40% | 0.00% | 100.00% | | |
| 40%∼50% | 56.86% | 100.00% | 93.91% | 98.38% |
| 50%∼60% | 97.96% | 73.76% | | |
| 60%∼70% | 0.00% | 0.00% | 22.73% | 100.00% |
| 70%∼80% | 22.73% | 100.00% | | |
| 80%∼90% | 0.00% | 29.41% | 84.71% | 96.98% |
| 90%∼100% | 85.05% | 96.48% | | |

**Table 3: Accuracy of Estimated Probabilities**

In this work, speculation is handled by *abort_future* instruction of superthreaded model and the software recovery mechanisms. For not losing generality, the TSAG stage will not be used to simplify the impact for performances. By this policy, the behavior of the probabilistic analysis information on thread speculation can be evaluated. The configuration of the simulator is shown in table 4.

| Simulator Configuration | |
|---|---|
| num of thread units | 2 and 4 |
| memory buffer size/units | 256 and 128 bytes |
| comm. units | 8 entries |
| mem-buffer write port | 2 |
| comm.-units to mem-buffer port | 2 |
| fork delay | 4 cycles |
| cache memory | 2-level cache hierarchy,<br>total delay 6 cycles |

**Table 4: Configuration of *SIMCA* Simulator**

### 5.2.2 Experimental Results

First, the relation between data dependence probability and program execution is evaluated by assuming a flow dependence exists between loop iterations with different dependence probabilities. A skeleton of evaluated code fragment is shown as Figure 9. Figure 10 shows the results under sequential, 2-thread, and 4-thread models.

The execution times of 2-thread and 4-thread systems are close to the sequential execution, when the probabilities of dependences are about 50% and 60%, respectively. This value can be used to compute $E$ of equation (2) in section 2.2. The figure also shows if the probability is high, speculative mechanism will in fact cause performance degradation. Therefore a compiler must be able to determine if speculative execution is profitable. A cost model is constructed for the simulation configuration based on the above observation, and the compiler will determine when to turn on or off the speculation mechanism.

In order to evaluate the effectiveness of compiler-directed speculation, several kernel loops are selected from programs and listed in table 5 In the *malloc* program, a free-list with

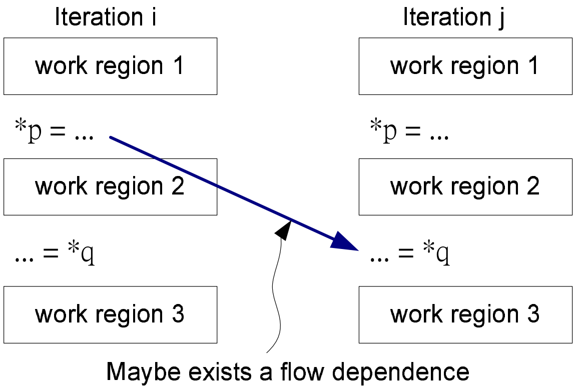


**Figure 9: The skeleton of code fragment**

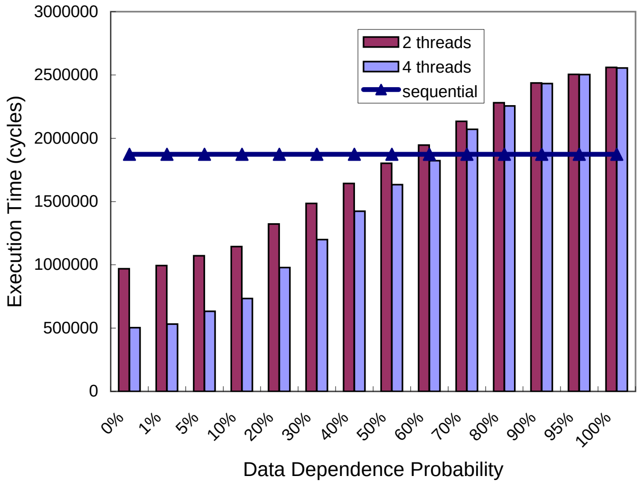


**Figure 10: Data dependence probabilities vs. Execution times**

| Program | Description |
|---|---|
| *990127-1* | Test program from gcc-3.0.1 testsuite. |
| *shuffle* | The program tests a random number generator using a card shuffling procedure. (netlib.org) |
| *em3d* | A program that operates on an irregular linked data structure. |
| *data retrieval* | A program simulates the behavior that retrieves a node from a set of nodes and modifies the node's content. (handcoding) [1] |
| *malloc* | A storage allocation program from [18]. |
| *map* | A C version of C++ STL map operation. [33] |

**Table 5: Benchmark Programs**

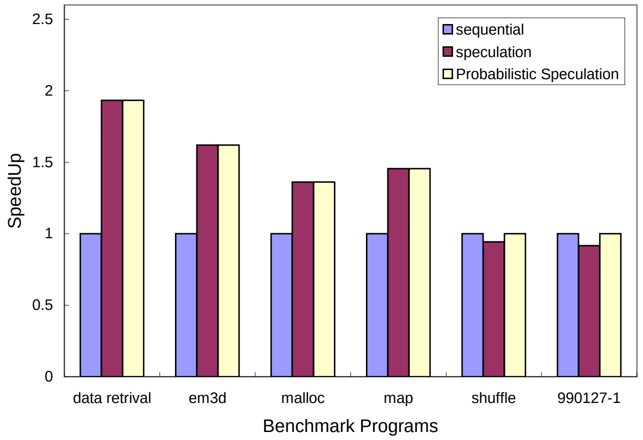


**Figure 11: Speedups on 2-thread System**

1000 nodes is created and 1000 times of allocating memory operation are performed, with randomly generated sizes ranging from 20 to 60. In the *map* program, a string-to-integer C++ STL map operation is executed. First, 500 elements generated randomly in the range from 1 to 500 are inserted into a double linked-list. Then, 3000 times of map operation will be done and the key of each map operation is generated randomly with the same range.

The compiler will adopt one of the following three strategies:

**Sequential Execution** The compiler will turn off the speculation mechanism when it identifies a *may*-dependence. The loop iterations will be executed sequentially.

**Speculation** The speculation mechanism will be turned on. The loop iterations will be executed by speculated threads.

**Probabilistic Speculation** The compiler will analyze the probabilities of dependences using the PPA information. It then decides whether to turn on the speculation mechanism or not based on the cost model.

Figure 11 shows the comparison of execution speedup between different strategies with two thread units. The programs *em3d* and *data retrieval* have low probabilities, so it is better to use speculated threads. For programs *malloc* and *map*, they do the table lookup operations from a pointer-linked list. In most of cases, the operations are almost independent between list nodes and hence probability of conflicts is low. Consequently, the compiler turns on the speculation mechanism for these programs and achieves speedup on 2 threads.

On the other hand, the programs *shuffle* and *990127-1* exhibit high probabilities between loop iterations. Therefore the benefits from multithreading execution will be nullified by mis-speculative penalty and consequently sequential execution will be a better choice. This figure shows that the probabilistic speculation strategy uses the data dependence probability to choose the best strategy for speculation, and hence it always achieves performance improvement.

## 6. RELATED WORK

There have been considerable efforts on pointer analysis by researchers [2, 6, 8, 9, 10, 21, 27, 28, 29, 32, 38, 39, 40]. 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 proposed in our research work computes key information for optimizations on speculative multi-threading environments.

In the work related to speculative multithreading (SpMT) model, research work can be seen in [11, 20, 25, 31, 35]. We employ the superthreaded architecture [35, 36] as our experimental platforms. In the work related to data speculations for modern computer architectures, such as IA-64 [16, 19], Ju et al. [17] 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 [22] 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. General *data flow frequency analysis* is proposed by Ramalingam [26]. 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.

Our research work in [15] pioneers the research efforts in giving quantitative descriptions for pointer-based aliases analysis. We draw an analogy here for the distance between probabilistic point-to analysis and general probabilistic data flow analysis. Conventional aliases analysis is still quite an active research item even though the deterministic data flow equations are well established much earlier. The probabilistic points-to analysis is complicated due to the dynamic association property of pointers. Our research work in [15] only gives intra-procedural cases. In this work, we give an methodology useful for inter-procedural cases. We report experimental results for an extensive set of applications to see the accuracy of our probabilistic point-to analysis. In addition, we give the first research work, to our best knowledge, to estimate the effects of the application of probabilistic point-to analysis for speculative multi-threading environments. This work is also a part of our efforts in our research group to develop compiler toolkits [4, 5, 13, 14, 23, 41] for high-performance and low-power micro-processors.

## 7. CONCLUSION

With the increased design of speculation mechanisms in advanced microprocessors, the ability for compilers to perform optimizations on speculations of advanced architectures becomes important. In this research work, we presented probabilistic point-to analysis framework and experiments which can take advantages of speculative multi-threading facilities provided by architectures. In our experiments, a family of important applications can be speeded up with our analysis. Considering a large depository of pointer-based objects, the program hopes to find objects meeting certain properties and constraints. During the iterations, if the object is not found, it will go for next iterations. If the pointer-based object is found to meet the criteria, house keeping work and update were done and dependence exists. Probabilistic loop-carried dependence induced by pointers were found in this family of applications. We give examples of data retrieval, malloc, and map operators in our experimental section for such cases. With the fundamental designs of table lookup in data structures and the applications of data mining, we feel this will be important for applications. Probabilistic point-to analysis will also be important for data speculations and code specializations on advanced architectures. We are in the process of investigating applications of probabilistic point-to analysis in those areas.

## 8. REFERENCES

[1] A. Berson, S. Smith, and K. Thearling. *Building Data Mining Applications for CRM.* McGraw-Hill, 1999.

[2] 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.

[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] R.-G. Chang, T.-R. Chuang, and J. K. Lee. Efficient support of parallel sparse computation for array intrinsic functions of Fortran 90. In *Conference Proceedings of the 1998 International Conference on Supercomputing*, pages 45–52, Melbourne, Australia, July 13–17, 1998. ACM SIGARCH.

[5] R.-G. Chang, J.-S. Li, J. K. Lee, and T.-R. Chuang. Probabilistic inference schemes for sparsity structures of fortran 90 array intrinsics. In *2001 International Conference on Parallel Processing (ICPP '01*, pages 61–68, Washington - Brussels - Tokyo, Sept. 2001. IEEE.

[6] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM Press, 1993.

[7] B. D. and T. Austin. *The SimpleScalar Tool Set, Version 3.0.* Unversity of Wisconsin Madison Computer Science Department.

[8] 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)*, volume 35.5 of *ACM Sigplan Notices*, pages 35–46, N.Y., June 18–21 2000. ACM Press.

[9] A. 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.

[10] M. Emami, R. Ghiya, and L. 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.

[11] L. Hammond, B. Hubbert, M. Siu, M. Prabhu, M. Chen, and K. Olukotun. The stanford hydra cmp.

*IEEE MICRO Magazine*, March-April 2000.

[12] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York, 1 edition, 1977.

[13] G.-H. Hwang, J. K. Lee, and R. D.-C. Ju. A function-composition approach to synthesize Fortran 90 array operations. *Journal of Parallel and Distributed Computing*, 54(1):1–47, 10 Oct. 1998.

[14] G.-H. Hwang, J. K. Lee, and R. D.-C. Ju. Array operation synthesis to optimize HPF programs on distributed memory machines. *Journal of Parallel and Distributed Computing*, 61(4):467–500, Apr. 2001.

[15] 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.

[16] Intel Corporation. *IA-64 Application Developer's Architecture Guide*, 1999.

[17] D.-C. R. Ju, J.-F. Collard, and K. Oukbir. Probabilistic memory disambiguation and its application to data speculation. In G. Lee and P.-C. Yew, editors, *Third Workshop on Interaction between Compilers and Computer Architectures (INTERACT-3)*, San Jose, CA, Oct. 1998.

[18] B. W. Kernighan and D. M. Ritchie. *The C programming language, Second Edition.* Prentice Hall, 1988.

[19] R. Krishnaiyer, D. Kulkarni, D. M. Lavery, W. Li, C.-C. Lim, J. Ng, and D. C. Sehr. An advanced optimizer for the ia-64 architecture. *IEEE Micro*, 20(6):60–68, November/December 2000.

[20] V. Krishnan and J. Torrellas. A chip-multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers*, 48(9):866–880, 1999.

[21] W. Landi and B. 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.

[22] J. K. Lee, D. Ho, and Y.-C. Chuang. Data distribution analysis and optimization for pointer-based distributed programs. In *Proceedings of the 1997 International Conference on Parallel Processing (ICPP '97)*, pages 56–63, Washington - Brussels - Tokyo, Aug. 1997. IEEE Computer Society Press.

[23] Y.-J. Lin, Y.-S. Hwang, and J. K. Lee. Compiler optimizations with dsp-specific semantic descriptions. In *Proceedings of the 2002 International Workshop on Languages and Compilers for Parallel Computing*, July 2002.

[24] S. S. Muchnick. *Advanced compiler design and implementation.* Morgan Kaufmann Publishers, 2929 Campus Drive, Suite 260, San Mateo, CA 94403, USA, 1997.

[25] J. Oplinger, D. Heine, S.-W. Liao, B. A. Nayfeh, M. S. Lam, and K. Olukotun. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Stanford University, February 1997.

[26] G. Ramalingam. Data flow frequency analysis. In *Proceedings of the ACM SIGPLAN '96 conference on Programming language design and implementation*, pages 267–277. ACM Press, 1996.

[27] E. 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.*

[28] R. Rugina and M. Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, pages 77–90. ACM Press, 1999.

[29] 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*, pages 1–14, Paris, France, Jan. 1997.

[30] M. D. Smith. *The SUIF Machine Library.* Division of of Engineering and Applied Science, Harvard University, March 1998.

[31] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar. Multiscalar processors. In *25 Years ISCA: Retrospectives and Reprints*, pages 521–532, 1998.

[32] 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*, pages 32–41, St. Petersburg Beach, Florida, Jan. 1996.

[33] B. Stroustrup. *The C++ programming language.* Addison-Wesley, 1991.

[34] The Stanford SUIF Compiler Group. *The SUIF Library.* Stanford University, 1995.

[35] J.-Y. Tsai, J. Huang, C. Amlo, D. J. Lilja, and P.-C. Yew. The superthreaded processor architecture. *IEEE Transactions on Computers*, 48(9):881–902, 1999.

[36] J.-Y. Tsai, Z. Jiang, and P.-C. Yew. Compiler techniques for the superthreaded architectures. *International Journal of Parallel Programming*, 27(1):1–19, 1999.

[37] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN '94 conference on Programming language design and implementation*, pages 85–96. ACM Press, 1994.

[38] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of the conference on Programming language design and implementation*, pages 1–12. ACM Press, 1995.

[39] P. Wu, P. Feautrier, D. Padua, and Z. Sura. Instance-wise points-to analysis for loop-based dependence testing. In *Proceedings of the 16th international conference on Supercomputing*, pages 262–273. ACM Press, 2002.

[40] S. H. Yong, S. Horwitz, and T. 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.*

[41] Y.-P. You, C.-R. Lee, and J. K. Lee. Compiler analysis and supports for leakage power reduction on microprocessors. In *Proceedings of the 2002 International Workshop on Languages and Compilers for Parallel Computing*, July 2002.