A Probabilistic Pointer Analysis for Speculative Optimizations

Jeff Da Silva and J. Gregory Steffan

Department of Electrical and Computer Engineering
University of Toronto
{dasilva,steffan}@eecg.toronto.edu

Abstract

Pointer analysis is a critical compiler analysis used to disambiguate the indirect memory references that result from the use of pointers and pointer-based data structures. A conventional pointer analysis deduces for every pair of pointers, at any program point, whether a points-to relation between them (i) *definitely* exists, (ii) *definitely does not* exist, or (iii) *maybe* exists. Many compiler optimizations rely on accurate pointer analysis, and to ensure correctness cannot optimize in the *maybe* case. In contrast, recently-proposed *speculative optimizations* can aggressively exploit the *maybe* case, especially if the likelihood that two pointers alias can be quantified. This paper proposes a *Probabilistic Pointer Analysis* (PPA) algorithm that statically predicts the probability of each points-to relation at every program point. Building on simple control-flow edge profiling, our analysis is both one-level context and flow sensitive—yet can still scale to large programs including the SPEC 2000 integer benchmark suite. The key to our approach is to compute points-to probabilities through the use of linear transfer functions that are efficiently encoded as sparse matrices. We demonstrate that our analysis can provide accurate probabilities, even without edge-profile information. We also find that—even without considering probability information—our analysis provides an accurate approach to performing pointer analysis.

Categories and Subject Descriptors D.3.4 [*Processors*]: Compilers

General Terms Algorithms, Performance

Keywords Dependence analysis, Pointer analysis, Speculative optimization

1. Introduction

Pointers are powerful constructs in C and other similar programming languages that enable programmers to implement complex data structures. However, pointer values are often ambiguous at compile time, complicating program analyses and impeding optimization by forcing the compiler to be conservative. Many pointer analyses have been proposed which attempt to minimize pointer ambiguity and enable compiler optimization in the presence of pointers [1, 2, 8, 12, 20, 25, 26, 30–32]. In general, the design of a pointer analysis algorithm is quite challenging, with many options that trade accuracy for space/time complexity. For example, the most accurate algorithms often cannot scale in time and space to accommodate large programs [12], although some progress has been made recently using *binary decision diagrams* [2, 30, 32].

The fact that memory references often remain ambiguous even after performing a thorough pointer analysis has motivated a class of compiler-based optimizations called *speculative optimizations*. A speculative optimization typically involves a code transformation that allows ambiguous memory references to be scheduled in a potentially unsafe order, and requires a recovery mechanism to ensure program correctness in the case where the memory references were indeed dependent. For example, EPIC instruction sets (eg., Intel's IA64) provide hardware support that allows the compiler to schedule a load ahead of a potentially-dependent store, and to specify recovery code that is executed in the event that the execution is unsafe [11, 18]. Proposed speculative optimizations that allow the compiler to exploit this new hardware support include speculative dead store elimination, speculative redundancy elimination, speculative copy propagation, and speculative code scheduling [7,21,22].

More aggressive hardware-supported techniques, such as thread-level speculation [15,19,24,27] and transactional programming [14] allow the speculative parallelization of sequential programs through hardware support for tracking dependences between speculative threads, buffering speculative modifications, and recovering from failed speculation. Unfortunately, to drive the decision of when to speculate many of these techniques rely on extensive data dependence profile information which is expensive to obtain and often unavailable. Hence we are motivated to investigate compile-time techniques—to take a fresh look at pointer analysis with speculative optimizations in mind.

1.1 Probabilistic Pointer Analysis

A conventional pointer analysis deduces for every pair of pointers, at any program point, whether a points-to relation between them (i) *definitely* exists, (ii) *definitely does not* exist, or (iii) *maybe* exists. Typically, a large majority of points-to relations are categorized as *maybe*, especially if a fast-but-inaccurate approach is used. Unfortunately, many optimizations must treat the *maybe* case the same as the *definitely* case to ensure correctness. However, speculative optimizations can capitalize on the *maybe* case—especially if we can quantify the likelihood that two pointers alias.

We propose a *Probabilistic Pointer Analysis* (PPA) for which we have the following three goals: (i) to accurately predict the probability of each points-to relation at every program point, without the need for detailed data dependence profile information; (ii) to scale to the SPEC 2000 integer benchmark suite; and (iii) to understand potential trade-offs between scalability and accuracy. We have developed a PPA infrastructure, called *Linear One-Level Interprocedural Probabilistic Points-to* (LOLIPoP), based an innovative algorithm that is both scalable and accurate: building on simple edge (control-flow) profiling, our analysis is both one-level context

ASPLOS'06 October 21–25, 2006, San Jose, California, USA.

and flow sensitive, yet can still scale to large programs. The key to our approach is to compute points-to probabilities through the use of linear transfer functions that are efficiently encoded as sparse matrices. LOLIPoP is very flexible, allowing us to explore the scalability/accuracy trade-off space.

1.2 Related Work

Pointer analysis is a well-researched problem and many algorithms have been proposed, yet no single approach has emerged as the preferred choice [17]. A universal solution to pointer analysis is prevented by the large trade-off between precision and scalability, motivating different pointer analyses for different applications [16]. The most accurate algorithms are both context-sensitive and control-flow-sensitive (CSFS) [12, 20, 31]; however, it has yet to be demonstrated whether these approaches can scale to large programs. Context-insensitive control-flow-insensitive (CIFI) algorithms [1, 26] can scale almost linearly to large programs, but these approaches are regarded as overly conservative and may impede aggressive compiler optimization. Several approaches balance this trade-off well by providing an appropriate combination of precision and scalability [2, 8, 30, 32]. However, studies suggest that a more accurate pointer analysis does not necessarily provide increased optimization opportunities [10, 17] because ambiguous pointers will persist. Hence speculative optimizations have recently been proposed to bridge this persistent disparity between safety and optimization. However, none of these conventional pointer analyses quantify the probability of potential points-to relations.

Several compiler analyses that support speculative optimizations have been proposed. In earlier work, Ramalingam [23] proposed a generic dataflow frequency analysis framework that is able to propagate edge frequencies interprocedurally, for use in optimizations where such frequency information is beneficial. Ju et al. [6] presented a probabilistic memory disambiguation framework that quantifies the likelihood that two array references alias by analyzing the array subscripts—but this approach is not applicable to pointers. More recently, Chen et al. [4, 5] developed an intuitive CSFS probabilistic point-to analysis algorithm. Their algorithm is based on an iterative data flow analysis framework, which is modified such that probabilistic information is additionally propagated. Their experimental results show that their technique can estimate the probabilities of points-to relations in benchmark programs with reasonable accuracy, although they model the heap as a single location set and the benchmarks studied are relatively small. Furthermore, their interprocedural approach is based on Emami's algorithm [12] and is therefore not expected to scale to large programs. Fernandez and Espasa [13] proposed a pointer analysis algorithm that targets speculation by relaxing analysis safety. The key insight is that such unsafe analysis results are acceptable because the speculative optimization framework can tolerate them, converting a safety concern into a performance concern. Finally, Bhowmik and Franklin [3] present a similar unsafe approach that uses linear transfer functions to achieve scalability. Unfortunately, neither of these last two approaches provide the probability information needed to compute cost/benefit trade-offs for speculative optimizations.

1.3 Contributions

This paper makes the following three contributions:

1. a novel algorithm for pointer analysis based on sparse transformation matrices;
2. an accurate, context-sensitive and flow-sensitive pointer analysis that scales to SPEC integer benchmarks;
3. a method for computing points-to probabilities that does not rely on dependence-profiling, and can optionally use edge-profiling.

```
         int x, y;
         int *a, *b;                        void f() {
                                                int *tmp;
         void main() {                 S9:      tmp = b;
S1:          a = &x;                   S10:     b = a;
S2:          b = &y;                   S11:     a = tmp;
S3:          if(...)                   S12:     g();
S4:              f();                       }

S5:          while(...) {                   void g() {
S6:              g();                  S13:     if(...)
S7:              ... = *b;             S14:         a = &x;
S8:              *a = ...;                  }
             }
         }
```

Figure 1. In this sample program, global pointers a and b are assigned in various ways. The `if` statement at $S3$ is taken with probability 0.9, the `if` statement at $S13$ is taken with a probability of 0.01, and the `while` loop at $S5$ iterates exactly 100 times.

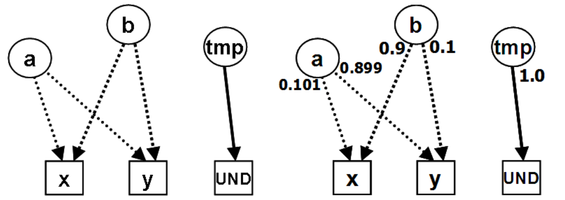
2. A Scalable PPA Algorithm

This section describes our PPA algorithm in detail. We begin by showing an example program. We then give an overview of our PPA algorithm, followed by the matrix framework that it is built on. Finally, we describe the bottom-up and top-down analyses that our algorithm is composed of.

2.1 Example Program

For the remainder of this section, the example program in Figure 1 will be used to illustrate the operation of our algorithm. In this sample program there are three pointer variables (a, b, and tmp) and two variables that can be pointed at (x and y), each of which is allocated a unique location set. We assume that edge profiling indicates that the `if` statement at $S3$ is taken with probability 0.9, the `if` statement at $S13$ is taken with a probability of 0.01, and the `while` loop at $S5$ iterates exactly 100 times (recall that our algorithm can proceed using heuristics in the absence of profile feedback). Initially a and b are assigned to the addresses of x and y respectively. The function $f()$ is then potentially called, which effectively swaps the pointer values of a and b and then calls the function $g()$. The function $g()$ potentially assigns the pointer a to the address of x, depending on the outcome of the `if` statement at $S13$ which is taken 1% of the time.

For an optimizing compiler to safely target the loop at $S5$, accurate knowledge about the dereferences at $S7$ and $S8$ is required. If both instructions always dereference the same location (i.e., $*a == *b$), the dereferences can be replaced by a single temporary variable. Conversely, if the dereference targets are always different and also loop invariant then the corresponding dereference operations can be hoisted out of the loop. If the compiler cannot prove either case to be true (which is often the result in practice because of the difficulties associated with pointer analysis) then it must be conservative and refrain from optimization. In this particular example, neither optimization is possible; however, it is possible to perform either optimization speculatively so long as the optimized code is guarded with a check and recovery mechanism [22]. To decide whether a speculative optimization is desirable for the code involving the points-to relations at $S7$ and $S8$, we first require the corresponding probabilities.



(a) Conventional points-to graph.

(b) Probabilistic points-to graph

Figure 2. A points-to graph and the corresponding probabilistic points-to graph, associated with the program point after $S4$ and initially into $S5$ (P_{S5}) in the example found in Figure 1. A dotted arrow indicates a *maybe* points-to relation whereas a solid arrow denotes a *definite* points-to relation. UND is a special location set used as the sink target for when a pointer's points-to target is undefined.

2.2 Algorithm Overview

The main objective of our probabilistic pointer analysis is to compute, at every program point s the probability that any pointer α points to any addressable memory location β. More precisely, given every possible points-to relation $\langle\alpha, *\beta\rangle$, the analysis must solve for the probability function $\rho(s, \langle\alpha, *\beta\rangle)$ for all program points s. The expected probability is defined by the following equation

$$\rho(s, \langle\alpha, *\beta\rangle) = \frac{E(s, \langle\alpha, *\beta\rangle)}{E(s)} \tag{1}$$

where $E(s)$ is the expected execution count associated with program point s and $E(s, \langle\alpha, *\beta\rangle)$ is the expected frequency for which the the points-to relation $\langle\alpha, *\beta\rangle$ holds dynamically at the program point s [4]. Intuitively, the probability is largely determined by the control-flow path and the procedure calling context to the program point s—hence an approach that is both control-flow and context sensitive[1] will produce the most probabilistically accurate results.

To perform pointer analysis, we must first construct an abstract representation of addressable memory called a *static memory model*. For our PPA algorithm the static memory model is composed of *location sets* [31].[2] A location set can represent one or more real memory locations, and can be classified as a pointer, pointer-target, or both. A location set only tracks its approximate size and the approximate number of pointers it represents, allowing us to abstract away the complexities of aggregate data structures and arrays of pointers. For example, fields within a C struct can either be merged into one location set, or else treated as separate location sets. Such options give us the flexibility to explore the accuracy/complexity trade-off space without modifying the underlying algorithm. We also define a special location set called UND as the sink target for when a pointer's points-to target is undefined.

Since a location set can be a pointer, a location set can point to another location set. Such a relation is called a *points-to* relation, and the set of all such relations is called a *points-to graph*: a directed graph whose vertices represent location sets. Specifically, a directed edge from vertex α to vertex β indicates that the pointer α may point to the target β. In a flow-sensitive analysis where statement order is considered when computing the points-to graph, every point in the program may have a unique points-to graph. Our PPA algorithm computes a probabilistic points-to graph which simply annotates each edge of a regular points-to graph with a weight representing the probability that the points-to relation will hold. Figure 2(a) shows an example points-to graph based on the example code given in Figure 1, and Figure 2(b) gives the corresponding annotated probabilistic points-to graph. It is important to note that the sum of all outgoing edges for a given vertex is always equal to one, to satisfy Equation 1.

To perform an inter-procedural context-sensitive analysis and to compute probability values for points-to relations, our PPA algorithm also requires as input an *interprocedural control flow graph* (ICFG) that is decorated with expected runtime frequency; we explain the construction of the ICFG in greater detail later in Section 3. The ICFG is a representation of the entire program that contains the control flow graphs for all procedures, connected by the overall call graph. Furthermore, all control-flow and invocation edges in the ICFG are weighted with their expected runtime frequency. These edge weights can be obtained through the use of simple edge profiling (eg., the output from `gprof`) or by static estimation based on simple heuristics.

Because our analysis is a control flow-sensitive analysis, at every point s the program is said to have a probabilistic points-to graph denoted P_s. Given a second point in the program $s\prime$ such that a forward path from s to $s\prime$ exists, the probabilistic points-to graph $P_{s\prime}$ can be computed using a transfer function that represents the changes in the points-to graph that occur on the path from s to $s\prime$, as formulated by

$$P_{s\prime} = f_{s\rightarrow s\prime}(P_s). \tag{2}$$

2.3 Matrix-Based Analysis Framework

As opposed to conventional pointer analyses which are set-based and can use analysis frameworks composed of bit vectors or BDDs, a PPA requires the ability to track floating-point values for the probabilities. Conveniently, the probabilistic points-to graph and transfer functions can quite naturally be encoded as matrices, although the matrix formulation in itself is not fundamental to the idea of PPA. The matrix framework is a simple alternative to propagating frequencies in an iterative data flow framework [23]. We choose matrices for several reasons: (i) matrices are easy to reason about, (ii) they have many convenient and well-understood properties, and (iii) optimized implementations are readily available. Our algorithm can now build on two fundamental matrices: a probabilistic points-to matrix P, and a *linear* transformation matrix T. Thus we have the fundamental PPA equation

$$P_{out} = T_{in\rightarrow out} \times P_{in}. \tag{3}$$

One key to the scalability of our algorithm is the fact that the transformation matrix is linear, allowing us to compute the probabilistic points-to graph at any point in the program by simply performing matrix multiplication—we do not require the traditional data flow framework used by other flow-sensitive approaches [4].

2.3.1 Points-to Matrix

We encode a probabilistic points-to graph using an $N \times M$ points-to matrix where M is the number of location set vertices that can be pointed at, and N is the number of pointer location sets plus the number of target location sets—therefore the vertices that act both as pointers and pointee-targets have two matrix row entries and are hence counted twice. The following equation formally defines the points-to matrix format:

[1] A *context-sensitive* pointer analysis distinguishes between the different calling contexts of a procedure, and a *control-flow-sensitive* pointer analysis takes into account the order in which statements are executed within a procedure.

[2] Note that our algorithm itself is not necessarily dependent on this location-set-based model.

$$P_s = \begin{bmatrix} p_{1,1} & \cdots & p_{1,M} \\ p_{2,1} & \cdots & p_{2,M} \\ \vdots & \ddots & \vdots \\ p_{N,1} & \cdots & p_{N,M} \end{bmatrix}$$

$$p_{i,j} = \begin{cases} \rho(s, \langle i', j' \rangle) & i \leq N - M \\ 1 & i > N - M \text{ and } i = j + (N - M) \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

The rows 1 to $N - M$ are reserved for the pointer locations sets and the rows $N - M + 1$ to N are reserved for the target location sets. To determine the row associated with a given pointer or pointee variable, the $\texttt{row\_id}(\alpha)$ function is used. Given a pointer variable α, the function $\texttt{row\_id}(\alpha)$ is equal to the matrix row mapped to the pointer α and the function $\texttt{row\_id}(\&\alpha)$ is equal to the matrix row mapped to the address of α. For a points-to matrix, pointer-target location sets are mapped to their corresponding column number by computing $\texttt{row\_id}(\&\alpha) - (N - M)$. The inner matrix spanning rows 1 to $N - M$ fully describes the probabilistic points-to graph; the other inner matrix spanning rows $N - M + 1$ to N is the identity matrix, and is only included to satisfy the fundamental PPA equation. Finally, but crucially, the matrix is maintained such that every row within the matrix sums to one—allowing us to treat a row in the matrix as a probability vector $\overrightarrow{P}$.

Example Consider the points-to graph depicted in Figure 2. We assume that $\texttt{row\_id}(a)$ = 1, $\texttt{row\_id}(b)$ = 2,
$\texttt{row\_id}(tmp)$ = 3, $\texttt{row\_id}(\&x)$ = 4, $\texttt{row\_id}(\&y)$ = 5,
and $\texttt{row\_id}(\texttt{UND})$ = 6. We also assume that the columns correspond to x, y, and UND respectively. This produces the corresponding points-to matrix:

$$P_{S5} = \begin{array}{r} \\ a \\ b \\ tmp \\ x \\ y \\ \texttt{UND} \end{array} \!\!\begin{array}{c} \begin{array}{ccc} x & y & \texttt{UND} \end{array} \\ \begin{bmatrix} 0.101 & 0.889 & 0 \\ 0.9 & 0.1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

2.3.2 Transformation Matrix

A transfer function for a given points-to matrix is encoded using an $N \times N$ transformation matrix, where N is the number of pointer location sets plus the number of target location sets. Each row and column is mapped to a specific location set using the equivalent $\texttt{row\_id}(\alpha)$ function. Transformation matrices are also maintained such that the values in every row always sum to one. Given any possible instruction or series of instructions, there exists a transformation matrix that satisfies Equation 3. If a statement has no effect on the probabilistic points-to graph, then the corresponding transformation matrix is simply the identity matrix. The following sections describe how transformation matrices are computed.

2.4 Representing Assignment Instructions

In any C program there are four basic ways to assign a value to a pointer, creating a points-to relation:

1. address assignment: $a = \&b$;
2. pointer assignment: $a = b$;
3. load assignment: $a = *b$;
4. store assignment: $*a = b$.

For each of the four cases there exists a corresponding transformation matrix. Types (1) and (2) generate a safe transformation matrix, whereas types (3) and (4) are modeled using a one-level unsafe transformation. The dereferenced target that is introduced in type (3) or (4) is modeled as a shadow variable and any ensuing shadow variable aliasing is ignored, which is of course unsafe. If desired, safety can be added by incorporating an additional lightweight alias analysis (as will be discussed further in Section 2.8). For each of the four cases, a transformation matrix is computed using the following equation:

$$T_{[\alpha=\beta,p]} = \begin{bmatrix} t_{1,1} & t_{1,2} & \cdots & t_{1,N} \\ t_{2,1} & t_{2,2} & \cdots & t_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ t_{N,1} & t_{N,2} & \cdots & t_{N,N} \end{bmatrix}$$

$$t_{i,j} = \begin{cases} p & i = \texttt{row\_id}(\alpha) \text{ and } j = \texttt{row\_id}(\beta) \\ 1 - p & i = j = \texttt{row\_id}(\alpha) \\ 1 & i = j \text{ and } i \neq \texttt{row\_id}(\alpha) \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

In this equation, α represents the pointer location set on the left side of the assignment and β denotes the location set (pointer or target) on the right side of the assignment. The probability value p represents the appropriate probability for the transformation and it is equal to 1 divided by the approximate number of pointers represented by the pointer location set α as defined in Equation 6. A pointer location set can represent multiple pointers. Such a location is required to represent the following cases: (1) an array of pointers; (2) pointers within recursive data structures are statically modeled as a single location set; (3) pointers within `C structs` are merged when a field-insensitive approach is utilized; and (4) shadow variable aliasing (as described in Section 2.8). A heuristic is used to approximate how many pointers are represented by a given pointer location set if this information cannot be determined statically. A probability of $p = 1$ is equivalent to a *strong update* used in a traditional flow-sensitive pointer analysis, whereas a probability of $p < 1$ is representative of a *weak update*.

$$p = \frac{1}{\text{approx \# of pointers in } \alpha} \tag{6}$$

It is important to note that the transformation matrix used for pointer assignment instructions is simply the identity matrix, with the exception of one row that represents the left side of the assignment.

Example The transformation matrices corresponding to the pointer assignment statements $S1$ and $S10$ from Figure 1 are:

$$T_{S1} = T_{[a=\&x,1.0]} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$T_{S10} = T_{[b=a,1.0]} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

To compute the points-to matrix at $S2$ we use T_{S1} and the fundamental PPA equation as follows:

$$P_{S2} = T_{S1} \cdot P_{S1}$$

$$\begin{array}{r} \\ a \\ b \\ tmp \\ x \\ y \\ \texttt{UND} \end{array} \!\!\begin{array}{c} \begin{array}{ccc} x & y & \texttt{UND} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The resulting points-to matrix at $S2$ shows that the points-to relation $\langle a, \&x \rangle$ exists with probability 1, while all other pointers (b and tmp) are undefined.

2.5 Representing Basic Blocks

For a basic block with a series of instructions $S1 \dots Sn$ whose individual transformation matrices correspond to $T_1 \dots T_n$, we can construct a single transformation matrix that summarizes the entire basic block using the following:

$$T_{bb} = T_n \cdot \dots \cdot T_2 \cdot T_1 \tag{7}$$

Therefore, given any points-to matrix at the inbound edge of a basic block, the points-to matrix at the outbound edge can be computed simply by performing the appropriate matrix multiplications. Note that the construction of a transformation matrix is a backward-flow analysis: to solve for the transformation matrix that summarizes an intraprocedural path from s to $s\prime$, the analysis starts at $s\prime$ and traverses backwards until it reaches s.

Example The basic block that contains statements $S1$ and $S2$ from Figure 1 can be summarized as:

$$T_{bb(S1-S2)} = T_{S2} \cdot T_{S1}$$

$$\begin{bmatrix} 0&0&0&1&0&0\\ 0&0&0&0&1&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&0&0&0&1&0\\ 0&0&0&0&0&1 \end{bmatrix} = \begin{bmatrix} 1&0&0&0&0&0\\ 0&0&0&0&1&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&0&0&0&1&0\\ 0&0&0&0&0&1 \end{bmatrix} \begin{bmatrix} 0&0&0&1&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&0&0&0&1&0\\ 0&0&0&0&0&1 \end{bmatrix}$$

Assume that we are given the points-to matrix for the start of the basic block at $S1$ in Figure 1; also assume that all pointers are undefined at that point. The points-to matrix at the end of the basic block (i.e., at $S3$) can be computed as follows:

$$P_{S3} = T_{bb(S1-S2)} \cdot P_{S1} = \begin{array}{c} \\ a \\ b \\ tmp \\ x \\ y \\ \texttt{UND} \end{array} \begin{array}{c} \begin{array}{ccc} x & y & \texttt{UND} \end{array} \\ \begin{bmatrix} 1&0&0\\ 0&1&0\\ 0&0&1\\ 1&0&0\\ 0&1&0\\ 0&0&1 \end{bmatrix} \end{array}$$

The resulting points-to matrix indicates that at $S3$ the points-to relations $\langle a, \&x \rangle$ and $\langle b, \&y \rangle$ exist with a probability of 1.

2.6 Representing Control Flow

The main objective of this matrix-based PPA framework is to summarize large regions of code using a single transformation matrix. To summarize beyond a single basic block, our transformation matrices must be able to represent control flow. Recall that the construction of a transformation matrix proceeds backwards, from $s\prime$ backwards to s. When the start of a basic block is encountered during this backwards analysis, the analysis must categorize all of that basic block's incoming edges. For now we consider the following three non-trivial cases for each edge:

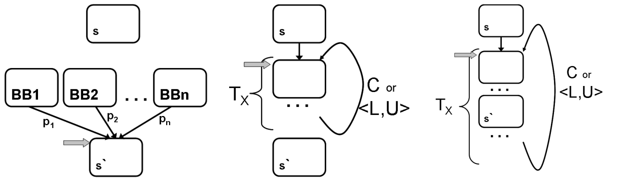
1. the edge is a forward edge (Figure 3(a));
2. the edge is a back-edge and $s\prime$ is *outside* the region that the back-edge spans (Figure 3(b));
3. the edge is a back-edge and $s\prime$ is *within* the region that the back-edge spans (Figure 3(c)).

The following considers each case in greater detail.

2.6.1 Forward Edges

When there exists a single incoming forward edge from another basic block the transformation matrix that results is simply the product of the transformation matrices for the current basic block and the incoming basic block. When there are exactly two incoming, forward edges from basic blocks γ and δ, we compute the transformation matrix as follows:

$$T_{if/else} = p \cdot T_\gamma + q \cdot T_\delta \tag{8}$$



(a) Forward edge (b) Back-edge, $s\prime$ outside (c) Back-edge, $s\prime$ inside

Figure 3. Control flow possibilities

T_γ and T_δ represent the transformation matrices from the program point s to the end of each basic block γ and δ respectively. The scalar probability p represents the fan-in probability from basic block γ, and q represents the fan-in probability from basic block δ, and we require that p and q sum to 1. This situation of two forward incoming edges typically arises from the use of `if`/`else` statements. In the more general case, we compute the transformation matrix as follows:

$$T_{cond} = p_i \sum T_i \tag{9}$$

This equation is simply a generalized version of Equation 8 with the added constraint that $\sum p_i = 1$.

Example From Figure 1 the function $g()$ can be fully summarized using Equation 9:

$$T_{g()} = 0.01 \cdot T_{S14} + 0.99 \cdot I = \begin{bmatrix} 0.99&0&0&0.01&0&0\\ 0&1&0&0&0&0\\ 0&0&1&0&0&0\\ 0&0&0&1&0&0\\ 0&0&0&0&1&0\\ 0&0&0&0&0&1 \end{bmatrix}$$

The identity matrix I is weighted with a probability 0.99 since there is no `else` condition. Recall that the `if` statement at $S13$ is taken 1% of the time. This matrix indicates that after the function $g()$ executes, a has a 1% chance of pointing at x and a 99% chance of remaining unchanged.

2.6.2 Back-Edge with $s\prime$ Outside it's Region

When a back-edge is encountered and $s\prime$ is *outside* the region that the back-edge spans, we can think of the desired transformation matrix as similar to that for a fully-unrolled version of the loop—eg., the same transformation matrix multiplied with itself as many times as the trip-count for the loop. In the case where the loop trip-count is constant, we can model the back-edge through simple exponentiation of the transformation matrix. Assuming that T_x is the transformation matrix of the loop body, and C is the constant loop trip-count value, we can model this type of back-edge with the following:

$$T_{loop} = {T_x}^C \tag{10}$$

When the loop trip-count is not a constant, we estimate the transformation matrix by computing the distributed average of all possible unrollings for the loop. Assuming that the back-edge is annotated with a lower-bound trip-count value of L and an upper-bound value of U, the desired transformation matrix can be computed efficiently as the geometric series averaged from L to U:

$$T_{loop} = \frac{1}{U - L + 1} \sum_{L}^{U} {T_x}^i \tag{11}$$

Example Consider the `while` loop found at statement $S5$ in Figure 1. The transformation matrix for the path from $S5$ to $S8$ is:

$$T_{S5\to S8} = (T_{S6\to S8})^{100} = (T_{g()})^{100} = \begin{bmatrix} 0.37 & 0 & 0 & 0.63 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

This matrix indicates that after the loop fully iterates, a has a 63% chance of pointing at x and a 37% chance of remaining unchanged.

2.6.3 Back-Edge with $s\prime$ Inside it's Region

The final case occurs when the edge is a back-edge and $s\prime$ is *inside* the region that the back-edge spans, as shown in Figure 3(c). Since $s\prime$ is within the loop, the points-to relations at $s\prime$ may change for each iteration of that loop. In this case we compute the desired transformation matrix using a geometric series such that U is the maximum trip-count value:

$$T_{loop} = \frac{1}{U} \sum_{0}^{U-1} {T_x}^i \qquad (12)$$

Example In Figure 1 this scenario occurs if we require the transformation matrix for the path $S1$ to $S7$, since in this case $s\prime$ is $S7$, which is within the `while` loop at $S5$. The required transformation matrix can be computed as follows:

$$T_{S1\to S7} = T_{g()} \cdot \tfrac{1}{100} \textstyle\sum_{0}^{99} (T_{S6\to S8})^i \cdot T_{S1\to S5}$$

2.7 Bottom Up and Top Down Analyses

We have so far described a method for computing probabilistic points-to information across basic blocks, and hence within a procedure. To achieve an accurate program analysis we need a method for propagating a points-to matrix inter-procedurally. To ensure that our analysis can scale to large programs, we have designed a method for inter-procedural propagation such that each procedure is visited no more than a constant number of times.

We begin by computing a transformation matrix for every procedure through a reverse topological traversal of the call graph (eg., a bottom-up (BU) pass). Recursive edges in the call-graph are *weakened*[3] and procedures are then analyzed iteratively for a fixed number of times to ensure that an accurate transformation matrix is computed. The result of the bottom-up pass is a linear transformation matrix that probabilistically summarizes the behavior of each procedure.

In the second phase of the analysis, we initialize a points-to matrix by (i) computing the result of all statically defined pointer assignments, and (ii) setting all other pointers to point at the *undefined* location set (`UND`). We then propagate the points-to matrix throughout the entire program using a forward topological traversal of the call-graph (eg., a top-down (TD) pass). When a load or store instruction is reached, the probability vector for that dereference is retrieved from the appropriate row in the matrix. When a call instruction is reached we store the points-to matrix at that point for future use. Finally, we compute the initial points-to matrix into every procedure as the weighted average of all incoming points-to matrices that were previously stored.

Example The call graph for our example dictates that in the bottom-up phase of the analysis the procedure-level transformation matrices are computed in the following order: $T_{g()}$, $T_{f()}$, and then $T_{main()}$. This is intuitively necessary since $T_{f()}$ requires $T_{g()}$; and $T_{main()}$ requires both $T_{f()}$ and $T_{g()}$. The algorithm then proceeds into the top-down phase which visits the procedures in the reverse order. Initially, the following input points-to matrix into `main()` is used since there are no static declarations:

$$P_{main()\_in} = \begin{array}{r} \\ a \\ b \\ tmp \\ x \\ y \\ \texttt{UND} \end{array} \begin{array}{c} \begin{array}{ccc} x & y & \texttt{UND} \end{array} \\ \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

The algorithm propagates and updates this matrix forward until a pointer dereference or a procedure call instruction is reached. At $S4$, the points-to matrix P_{S4} is cached so that when the procedure `f()` is analyzed the points-to matrix representing the initial state of `f()` will be available.

$$P_{f()\_in} = P_{S4} = \begin{array}{r} \\ a \\ b \\ tmp \\ x \\ y \\ \texttt{UND} \end{array} \begin{array}{c} \begin{array}{ccc} x & y & \texttt{UND} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

Similarly, the points-to matrices P_{S6} and P_{S12} are also stored to analyze procedure `g()` during the top-down phase. These matrices are later merged using a weighted sum based on the fan-in frequencies from their respective callee procedures:

$$P_{g()\_in} = 100 \times P_{S6} + P_{S12}$$

2.8 Safety

At this point our algorithm does not necessarily compute a safe result in all circumstances: for certain code, the theoretical points-to transfer function may contain non-linear side effects which are not captured by our linear transformation matrix encoding. Non-linear effects occur when multiple levels of dereferencing are used to assign to pointers: (i) load assignment: $a = *b$; (ii) store assignment: $*a = b$.

We handle multi-level dereferencing by instantiating *shadow variable* pointer location sets, a technique that is similar to invisible variables [12]. These shadow variable location sets can potentially alias with other pointer location sets and cause unsafe behavior if ignored. To handle these non-linearities safely, we use a lightweight, context-insensitive and flow insensitive unification-based alias analysis [26] to precompute any shadow variable aliasing that can potentially occur. We assume that aliasing between location sets occurs with a probability that is inversely-proportional to the size of the aliasing set. All transformations involving shadow variables that alias are extended to handle these non-linearities in a safe manner. However, it is key to note that since we are supporting speculative optimizations, we also have the option of ignoring safety.

3. The LOLIPoP PPA Infrastructure

Figure 4 shows a block diagram of the LOLIPoP infrastructure, which operates on `C` source code. We have also developed a lightweight edge-profiler which instruments `C` source code to track control flow edges, invocation edges, and indirect function call targets. The foundation of LOLIPoP is a SUIF 1.3 compiler pass [29]. The pass begins by building an Interprocedural Control Flow Graph (ICFG) that is annotated with any available profile information. The static memory model (SMM) is then built by extracting location sets from SUIF's symbol tables. For this paper, all memory allocation call sites are treated as a single unique location set, and all aggregate structures and arrays are merged into a single location set each. Recursive data structures are merged into one data structure after one level of recursion. We then perform the bottom-up (BU) and top-down (TD) phases of the analysis as described in Section 2.7. We exploit the sparse matrix library available through MATLAB's runtime library to perform all matrix operations.

[3] Weakening means that we iteratively tag edges within SCCs [28] as 'weakened' and then ignore them for the purpose of topological traversal, since such edges are recursion-causing invocation edges.

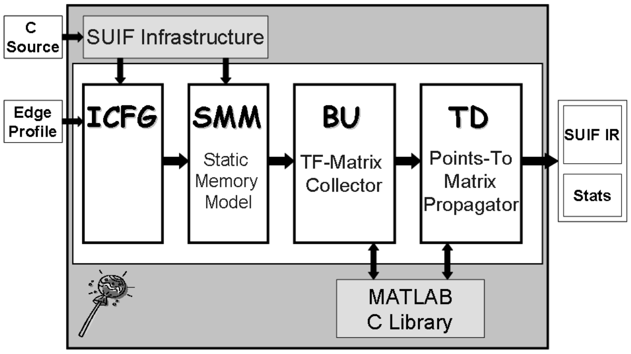


Figure 4. The LOLIPoP infrastructure.

Table 1. Benchmark inputs used.

Benchmark		`Train` **Input**	`Ref` **Input**
SPECint2000	BZIP2	default	default
	CRAFTY	default	default
	GAP	default	default
	GCC	default	expr.i
	GZIP	default	default
	MCF	default	default
	PARSER	default	default
	PERLBMK	diffmail.pl	diffmail.pl
	TWOLF	default	default
	VORTEX	default	bendian1.raw
	VPR	default	default
SPECint95	COMPRESS	default	reduced to 5.6MB
	GO	default	9stone21.in
	IJPEG	default	vigo.ppm
	LI	default	default
	M88KSIM	default	default
	PERL	jumble.pl	primes.pl

For experiments when edge-profile information is not available, we use the following three compile-time heuristics: (i) we assume that fan-in probability is a uniform distribution between all incoming branches; (ii) when the upper and lower bound for the trip count of a loop cannot be determined through a simple inspection of the loop bounds, we assume that the lower bound is zero and the upper bound is ten[4]; and (iii) we assume that fan-in call graph invocation edge counts have a uniform distribution between all callee procedures. We do not use a heuristic for indirect function call targets and therefore require the edge-profiler to always provide that information.

Finally, we put a great deal of effort into optimizing the execution of LOLIPoP. In particular we exploit sparse matrices, we compress matrices before exponentiating, we perform aggressive memoization of matrix results at the intra-procedural level, and we use an efficient implementation for computing the geometric series for Equations 11 and 12 [9].

4. Evaluating LOLIPoP

In this section we evaluate LOLIPoP's running-time, the accuracy of the pointer analysis results in a conventional sense, and the accuracy of the computed probabilities.

4.1 Experimental Framework

The following describes our framework for evaluating LOLIPoP. All timing measurements were obtained using a 3GHz Pentium IV with 1GB of RAM. We report results for all of the SPECint95 and SPECint2000 benchmarks except for the following: 252.EON, which is written in C++ and not handled by our compiler; 126.GCC, which is similar to 176.GCC; and 147.VORTEX, which is identical to 255.VORTEX. Table 1 describes both the `ref` and `train` inputs used for each benchmark.

Table 2. LOLIPoP measurements, including lines-of-code (LOC) and transformation matrix size N for each benchmark, as well as the running times for both the unsafe and safe analyses. The time taken to obtain points-to profile information at runtime is included for comparison.

Benchmark		LOC	Matrix Size	Running Time (min:sec)		
				Unsafe	Safe	Profile
SPECint2000	BZIP2	4686	251	0:0.3	0:0.3	13:34
	CRAFTY	21297	1917	0:5.5	0:5.5	14:47
	GAP	71766	25882	54:56	83:38	55:56
	GCC	22225	42109	309:40	N/A	39:58*
	GZIP	8616	563	0:0.71	0:0.77	3:48
	MCF	2429	354	0:0.39	0:0.61	19:46
	PARSER	11402	2732	0:30.7	0:50.0	84:52
	PERLBMK	85221	20922	44:15	89:43	N/A
	TWOLF	20469	2611	0:16.6	0:20.6	N/A
	VORTEX	67225	11018	3:59	4:56	0:0.7*
	VPR	17750	1976	0:9.3	0:10.3	197:0
SPECint95	COMPRESS	1955	97	0:0.1	0:0.1	1:55
	GO	29283	651	0:2.9	0:3	5:58
	IJPEG	31457	4491	0:23.4	0:24.9	7:12
	PERL	4686	5395	5:3	7:49	8:37
	LI	27144	3868	0:28.8	0:59.15	72:5
	M88KSIM	19933	1932	0:4.9	0:5.24	0:0.2

*Note that for GCC and VORTEX the profiler is significantly faster because a reduced ref input set was used to make the analysis tractable (see Table 1).

4.2 Analysis Running-Time

LOLIPoP meets our objective of scaling to the SPECint95 and SPECint2000 benchmark suites. Table 2 shows the running-times for both the safe (where shadow variable alias analysis is performed) and unsafe analyses. Each running-time includes the bottom-up and top-down phases of the overall analysis, but ignores the time taken to build the interprocedural control flow graph and static memory model—but note that this time is negligible in most cases. The runtimes range from less than a second for four benchmarks up to 5 hours for most challenging benchmark (GCC). These results are promising especially given that this is an academic prototype implementation of PPA.

For speculative optimizations, the alternative to pointer analysis is to use a points-to profiler which instruments the source code to extract points-to frequency information at runtime; hence for comparison purposes we have implemented a points-to profiler that instruments the source code so that the memory addresses of all location sets can be tracked and queried at runtime. This involves two key instrumentation techniques: (1) overloading library function calls to `alloc` and `free` to track heap location set addresses; and (2) storing the addresses of every stack variable (whose address is taken) when pushed on the program's runtime stack. Every pointer dereference is also instrumented with a wrapper function that enables the points-to profiler to determine which location set is being referenced during the dereference operation. The points-to profiler queries the set of available addresses to determine which runtime points-to relation is currently applicable to that address, and the count of this points-to relation is then incremented. It is important to understand that a points-to profiler is only able to track points-to relations that actually occur at runtime, and hence has fewer possibilities to track than a static points-to analysis. In summary, points-to profiling is a computationally intense analysis and furthermore has no ability to provide safety in the resulting alias information it provides.

[4] Fully exploring the impact of these default loop bounds is beyond the scope of this work.

Table 3. LOLIPoP Measurements.

Benchmark	Avg. Dereference Size			Avg. Maximum Certainty
	Safe	$p > 0.001$	Unsafe	
BZIP2	1.00	1.00	1.00	1.00
COMPRESS	1.080	1.08	1.08	0.96
CRAFTY	1.830	1.40	1.83	0.95
GAP	143.84	77.61	6.21	0.78
GCC	N/A	N/A	2.64	0.96
GO	3.290	2.15	3.29	0.94
GZIP	1.41	1.31	1.45	0.90
IJPEG	6.740	2.46	1.33	0.90
LI	80.10	14.70	4.34	0.76
M88KSIM	1.82	1.66	1.84	0.83
MCF	1.51	1.51	1.51	0.92
PARSER	42.52	2.09	3.23	0.97
PERLBMK	18.48	5.45	3.10	0.79
PERL	88.98	8.40	35.35	0.87
TWOLF	1.26	1.25	1.19	0.90
VORTEX	6.06	3.61	6.13	0.91
VPR	1.18	1.10	1.09	0.95

The results in Table 2 show that in most cases, our analysis approach is much faster than the profiler. In fact, for two cases (PERLBMK and TWOLF) the profiler did not terminate after two weeks of execution. It is also important to note that the profiling approach is very dependent on the input set used, both in the results it provides and on the profiler's runtime. For GCC and VORTEX reduced reference input sets were used to ensure a tractable profiler time (described in Table 1)—in these cases the profiler outperforms LOLIPoP because LOLIPoP must analyze the entire program while the profiler only analyzes the subset of code that is exercised by the reduced input set. For the more challenging benchmarks (GAP, GCC, and PERLBMK), there is a significant increase in running-time to compute safe results—i.e., to handle pointer assignments with multiple levels of dereferencing as described in Section 2.8.

4.3 Pointer Analysis Accuracy

The accuracy of a conventional pointer analysis algorithm is typically measured and compared by computing the average cardinality of the target location sets that can be dereferenced across all pointer dereference sites in the program—in short, the average dereference size. Table 3 shows the average dereference sizes for all benchmarks studied, showing the safe result, the result when any points-to relation with a probability less than 0.001 is ignored, and the unsafe result (when shadow variable aliasing is ignored)—average maximum certainty will be discussed later in Section 4.4.1. One very interesting result is that the benchmarks with a relatively large average dereference size for the safe analysis (GAP, LI, PARSER, PERLBMK, PERL) show a dramatic decrease when unlikely points-to relations are ignored (i.e., those for which $p < 0.001$). This result suggests that many points-to relations are unlikely to occur at runtime, underlining the strong potential for speculative optimizations. As expected, a similar result is observed for the unsafe version of the analysis since the safe analysis introduces many inaccuracies through the flow-insensitive, context-insensitive pass that addresses shadow variable aliasing. These inaccuracies create many low probability points-to relations that are unlikely to ever occur at runtime. For example, the safe average dereference size for GAP is relatively high at 143.84, while the unsafe size is only 6.21.

4.4 Probabilistic Accuracy

We now measure the accuracy of the probabilities computed by LOLIPoP by comparing the two probability vectors $\overrightarrow{P}_s$ and $\overrightarrow{P}_d$ at every pointer dereference point. $\overrightarrow{P}_s$, represents the probability vector reported by LOLIPoP—the static probability vector. $\overrightarrow{P}_d$ represents the dynamic probability vector calculated by the points-to profiler. In particular, we want to quantify the accuracy of the probability vectors $\overrightarrow{P_s}$ that are statically computed at every pointer dereference. For comparison, we use the results of the profiler where each benchmark (using it's `ref` input) is instrumented to track—for each pointer dereference location—a frequency vector that indicates the frequency that each location set is the target. Each resulting dynamic frequency vector is then normalized into a dynamic probability vector ($\overrightarrow{P}_d$) so that it may be compared with the corresponding probability points-to relation vector, as described in Equation 1. To compare the two vectors in a meaningful way, we compute the *normalized average Euclidean distance* (NAED) as defined by:

$$\text{NAED} = \frac{1}{\sqrt{2}} \cdot \frac{\sum \|\overrightarrow{P}_s - \overrightarrow{P}_d\|}{\text{(\# pointer dereferences)}} \qquad (13)$$

This metric summarizes the average error uniformly across all probability vectors at every pointer dereference on a scale that ranges from zero to one, where a zero means no discrepancy between dynamic and static vectors, and a one means there is always a contradiction at every dereference.

Figure 5 shows the NAED for the SPECint95 and SPECint2000 benchmarks relative to the dynamic execution on the `ref` input set (results for GAP, PERLBMK, and TWOLF are omitted because their dynamic points-to profile information could not be tractably computed). In the first experiment (*D*) we distribute probability uniformly to every target in the static points-to probability vector $\overrightarrow{P_s}$, making the naive assumption that all targets are equally likely. This experiment is used to quantify the value-added of probability information, and leads to an average NAED across all benchmarks of 0.32 relative to the dynamic result. It is important to notice that for BZIP2, COMPRESS, and GO even the uniform distribution (*D*) is quite accurate.

The second experiment (*Sr*) plots the NAED for the safe analysis using edge-profile information from the `ref` input set. Comparing the static and dynamic results using the same input set allows us to defer the question of how representative the profiling input set is. With LOLIPoP we improve the NAED to an average of 0.25 across all benchmarks, although that average can be misleading. For about half of the benchmarks probability information does not make a large difference (bzip2, compress, go, m88ksim, mcf, vpr), while for the remaining benchmarks probability information significantly improves the NAED. For LI, the probability information slightly increases the NAED. The LI benchmark contains a tremendous amount of shadow variable aliasing—we know this because of the large gap between the safe and unsafe average dereference sizes shown in Figure 3. The spurious points-to relations introduced by the 'safe' analysis appear to corrupt the useful probability information. A similar result would be expected for GAP. Using a more accurate shadow variable analysis pass would help to reduce this effect; applying field-sensitivity would also help because it would drastically reduce the amount of shadow variable aliasing.

The next experiment (*Ur*) shows the NAED for the unsafe analysis, also using edge-profile information from the `ref` input set. Comparing with the safe experiment (*Sr*), surprisingly we see that on average the unsafe result is more accurate (with an NAED of 0.24): this result implies that safety adds many false points-to relations, and can actually be undesirable in the context of speculative optimizations. The exception to this is GZIP, where the NAED deteriorates when transitioning from safe to unsafe. This implies that GZIP frequently utilizes and relies on many levels of pointer dereferencing.

The final two experiments plot the NAED for the unsafe analysis when using the `train` input set (*Ut*), and when using compile-

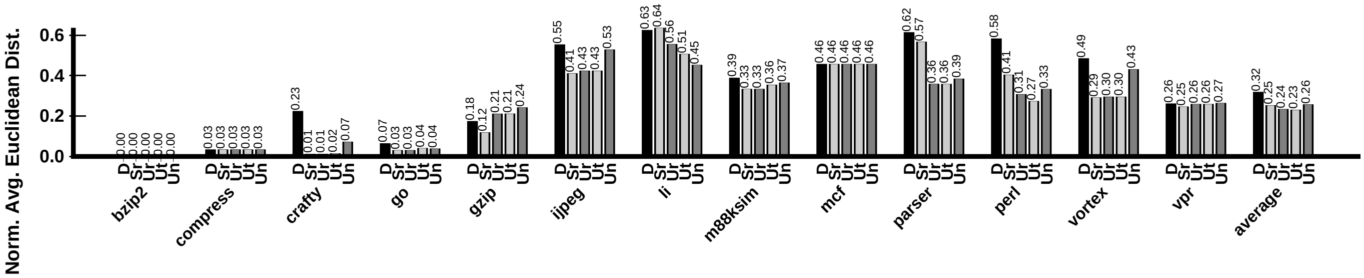


Figure 5. Normalized Average Euclidean Distance (NAED) relative to the dynamic execution on the `ref` input set. *D* is a uniform distribution of points-to probabilities, *Sr* is the safe LOLIPoP result using the `ref` input set, and the *U* bars are the unsafe LOLIPoP result using the `ref` (*Ur*) and `train` (*Ut*) input sets, or instead using compile-time heuristics (*Un*).

time heuristics instead of edge-profile information (*Un*). Surprisingly, the average NAED when using the `train` input set (*Ut*) is slightly more accurate than with the `ref` input set (*Ur*): this indicates that there are aliases which occur rarely during the `ref` execution, which when profiled and fed back into LOLIPoP, become a source of inaccuracy compared to the lesser coverage of the `train` input set. Finally, we see that the unsafe approaches are more accurate with edge-profiling information than when compile-time heuristics are used (*Un*), although even with heuristics LOLIPoP is more accurate than the uniform distribution (*D*).

4.4.1 Average Maximum Certainty

To further evaluate the potential utility of the points-to probability information provided by LOLIPoP, we measure *average maximum certainty*:

$$\text{Avg. Max. Certainty} = \frac{\sum(\text{Max Probability Value})}{(\#\text{ pointer dereferences})} \qquad (14)$$

This equation takes the maximum probability value associated with all points-to relations at every pointer dereference and averages these values across all pointer dereference sites. Data speculative optimizations benefit from increased certainty that a given points-to relation exists: since the probabilities across a probability vector sum to one, if there is one large probability it implies that the probabilities for the remaining location sets are small. In other words, the closer the average maximum certainty value is to one, the more potential there is for successful speculative optimization. The average maximum certainty for each SPEC benchmark is given in Table 3, and in general these values are quite high: the average value across all benchmarks is 0.899. This indicates that on average, at any pointer dereference, there is likely only one dominant points-to relation. Therefore a client analysis using LOLIPoP will be very certain of which points-to relation will exist at a given pointer dereference.

5. Conclusions

As speculative optimization becomes a more widespread approach for optimizing and parallelizing code in the presence of ambiguous memory references, we are motivated to predict the likelihood of points-to relations without relying on expensive dependence profiling. We have presented LOLIPoP, a probabilistic pointer analysis algorithm that is one-level context-sensitive and flow-sensitive, yet can still scale to large programs including the SPECint2000 benchmark suite. The key to our approach is to compute points-to probabilities through the use of linear transfer functions that are efficiently encoded as sparse matrices.

We have used LOLIPoP to draw several interesting conclusions. First, we found that—even without considering probability information—LOLIPoP provides an accurate approach to performing pointer analysis. Second, we demonstrated that many points-to relations are unlikely to occur at runtime, underlining the strong potential for speculative optimizations. Third, we found that the unsafe version of our analysis is more probabilistically accurate than the safe version, implying that safety adds many false points-to relations and can actually be undesirable in the context of speculative optimizations. Finally, we demonstrated that LOLIPoP can produce accurate probabilities when using compile-time heuristics instead of edge-profile information.

References

[1] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, May 1994. DIKU report 94/19.

[2] M. Berndl, O. Lhotak, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI '03: Proceedings of the 2003 conference on Programming language design and implementation*, pages 103–114, New York, NY, USA, 2003. ACM Press.

[3] A. Bhowmik and M. Franklin. A fast approximate interprocedural analysis for speculative multithreading compilers. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 32–41, New York, NY, USA, 2003. ACM Press.

[4] 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. *SIGPLAN Not.*, 38(10):25–36, 2003.

[5] P.-S. Chen, Y.-S. Hwang, R. D.-C. Ju, and J. K. Lee. Interprocedural probabilistic pointer analysis. *IEEE Trans. Parallel Distrib. Syst.*, 15(10):893–907, 2004.

[6] R. D. ching Ju, J.-F. Collard, and K. Oukbir. Probabilistic memory disambiguation and its application to data speculation. *SIGARCH Comput. Archit. News*, 27(1):27–30, 1999.

[7] X. Dai, A. Zhai, W.-C. Hsu, and P.-C. Yew. A general compiler framework for speculative optimizations using data speculative code motion. In *CGO*, pages 280–290, 2005.

[8] M. Das, B. Liblit, M. Fahndrich, and J. Rehof. Estimating the impact of scalable pointer analysis on optimization. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 260–278, London, UK, 2001. Springer-Verlag.

[9] J. DaSilva. A probabilistic pointer analysis for speculative optimizations. Master's thesis, University of Toronto, February 2006.

[10] A. Diwan, K. S. McKinley, and J. E. B. Moss. Using types to analyze and optimize object-oriented programs. *ACM Transactions on Programming Languages and Systems*, 23(1):30–72, 2001.

[11] C. Dulong. The ia-64 architecture at work. *Computer*, 31(7):24–32, 1998.

[12] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM Press.

[13] M. Fernandez and R. Espasa. Speculative alias analysis for executable code. In *PACT '02: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques*, pages 222–231, Washington, DC, USA, 2002. IEEE Computer Society.

[14] L. Hammond, B. D. Carlstrom, V. Wong, B. Hertzberg, M. Chen, C. Kozyrakis, and K. Olukotun. Programming with transactional coherence and consistency (tcc). In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1–13. ACM Press, Oct 2004.

[15] L. Hammond, M. Willey, and K. Olukotun. Data speculation support for a chip multiprocessor. In *Proceedings of the Eighth ACM Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[16] M. Hind. Pointer analysis: haven't we solved this problem yet? In *PASTE '01: Proceedings of the 2001 workshop on Program analysis for software tools and engineering*, pages 54–61, New York, NY, USA, 2001. ACM Press.

[17] M. Hind and A. Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM Press.

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

[19] V. Krishnan and J. Torrellas. A chip multiprocessor architecture with speculative multithreading. *IEEE Transactions on Computers, Special Issue on Multithreaded Architecture,* September 1999.

[20] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI)*, volume 27, pages 235–248, New York, NY, 1992. ACM Press.

[21] J. Lin, T. Chen, W.-C. Hsu, and P.-C. Yew. Speculative register promotion using advanced load address table (alat). In *CGO '03: Proceedings of the international symposium on Code generation and optimization*, pages 125–134, Washington, DC, USA, 2003. IEEE Computer Society.

[22] J. Lin, T. Chen, W.-C. Hsu, P.-C. Yew, R. D.-C. Ju, T.-F. Ngai, and S. Chan. A compiler framework for speculative analysis and optimizations. In *PLDI '03: Proceedings of the 2003 conference on Programming language design and implementation*, pages 289–299, New York, NY, USA, 2003. ACM Press.

[23] G. Ramalingam. Data flow frequency analysis. In *PLDI '96: Proceedings of the 1996 conference on Programming language design and implementation*, pages 267–277, New York, NY, USA, 1996. ACM Press.

[24] A. Roth and G. S. Sohi. Speculative data-driven multithreading. In *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, January 2001.

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

[26] B. Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM Press.

[27] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. The stampede approach to thread-level speculation. *ACM Transactions on Computer Systems*, 23(3):253–300, August 2005.

[28] R. Tarjan. Testing flow graph reducibility. In *STOC '73: Proceedings of the fifth annual ACM symposium on Theory of computing*, pages 96–107, New York, NY, USA, 1973. ACM Press.

[29] S. Tjiang, M. Wolf, M. Lam, K. Pieper, and J. Hennessy. *Languages and Compilers for Parallel Computing*, pages 137–151. Springer-Verlag, Berlin, Germany, 1992.

[30] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI '04: Proceedings of the 2004 conference on Programming language design and implementation*, pages 131–144, New York, NY, USA, 2004. ACM Press.

[31] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI '95: Proceedings of the 1995 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1995. ACM Press.

[32] J. Zhu and S. Calman. Symbolic pointer analysis revisited. In *PLDI '04: Proceedings of the 2004 conference on Programming language design and implementation*, pages 145–157, New York, NY, USA, 2004. ACM Press.