A Novel Probabilistic Data Flow Framework * **

Eduard Mehofer[1] and Bernhard Scholz[2]

[1] Institute for Software Science
University of Vienna, Austria
mehofer@par.univie.ac.at
[2] Institute of Computer Languages
Vienna University of Technology, Austria
scholz@complang.tuwien.ac.at

Abstract. Classical data flow analysis determines whether a data flow fact may hold or does not hold at some program point. Probabilistic data flow systems compute a range, i.e. a probability, with which a data flow fact will hold at some program point. In this paper we develop a novel, practicable framework for probabilistic data flow problems. In contrast to other approaches, we utilize execution history for calculating the probabilities of data flow facts. In this way we achieve significantly better results. Effectiveness and efficiency of our approach are shown by compiling and running the SPECint95 benchmark suite.

1 Introduction

Classical data flow analysis determines whether a data flow fact may hold or does not hold at some program point. For generating highly optimized code, however, it is often necessary to know the *probability* with which a data flow fact will hold during program execution (cf. [10, 11]). In probabilistic data flow systems control flow graphs annotated with edge probabilities are employed to compute the probabilities of data flow facts. Usually, edge probabilities are determined by means of profile runs based on representative input data sets. These probabilities denote heavily and rarely executed branches and are used to weight data flow facts when propagating them through the control flow graph.

Consider the example shown in Fig. 1 to discuss classical and probabilistic data flow analysis. For the sake of simplicity we have chosen as data flow problem the reaching definitions problem [8]. The control flow graph G of our running example consists of two subsequent branching statements inside a loop and four definitions d_1 to d_4. Variable X is defined at edges $\mathbf{2} \to \mathbf{4}$ and $\mathbf{5} \to \mathbf{7}$ by d_1 and d_3. Similarly, variable Y is assigned a value at edges $\mathbf{3} \to \mathbf{4}$ and $\mathbf{6} \to \mathbf{7}$ by d_2 and d_4. Classical reaching definition analysis yields that definitions d_1 to d_4

* This research is partially supported by the Austrian Science Fund as part of Aurora Project "Languages and Compilers for Scientific Computation" under Contract SFB-011.

** Accepted for publication at International Conference on Compiler Construction (CC'01), Genova, Italy, April 2001, LNCS, ©Springer-Verlag.

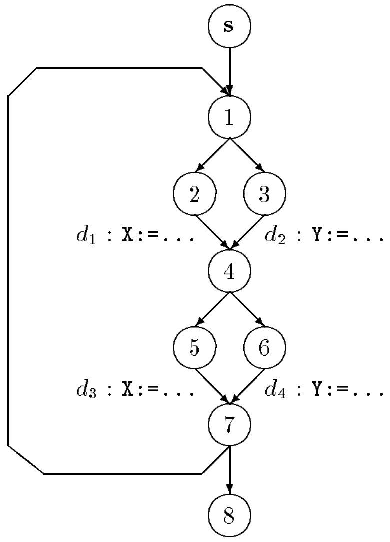

Fig. 1. Running example.

may reach nodes **1** to **8**. The solution is a conservative approximation valid for all possible program runs. However, when we consider specific program runs, we can compute a numerical value denoting the probability with which a definition actually may reach a node during execution. Ramalingam [13] presented a data flow framework which computes the probability of data flow facts, once every edge in the control flow graph has been annotated with a probability. In order to get an idea of the precision of his results, we defined in [9] the best solution $\mathcal{S}_{best}$ that one can theoretically obtain and compared both. We showed that the differences between the theoretically best solution and Ramalingam's solution can be considerable and improvements are necessary. However, the computation of the theoretically best solution is too expensive in general and, hence, not feasible in practice. The modifications of the equation system described in [9] resulted in some improvements, but there is still potential for further improvements left.

Two reasons are responsible for the deviations between the theoretically best solution and Ramalingam's approach. On the one hand, program paths are reduced to edge probabilities. On the other hand, it is an execution history independent approach, i.e. it is assumed that particular branches are independent of execution history, which obviously is not true in reality. Since edge probabilities are indispensable to get an efficient handle on the problem, we focus on execution history in order to get better results. Consider a program run for our example in Fig. 1 that performs 10 iterations. At the beginning the left branches **[1,2,4,5,7]** are executed and in the last iteration the right branches **[1,3,4,6,7,8]** are taken. Without execution history edges $\mathbf{4} \rightarrow \mathbf{5}$ and $\mathbf{4} \rightarrow \mathbf{6}$ are dealt with independently of the incoming edges $\mathbf{2} \rightarrow \mathbf{4}$ and $\mathbf{3} \rightarrow \mathbf{4}$. However by correlating outgoing edges with incoming ones, it can be recognized that paths **[2,4,6]** and **[3,4,5]** are never taken. Hence, d_3 cannot reach node **5**, since d_3 is killed on edge $\mathbf{2} \rightarrow \mathbf{4}$. Similarly, it can be detected that d_4 cannot reach node **6**. Finally, since the loop is exited via path **[6,7,8]**, it can be determined that definition d_2 cannot reach node **8**.

In this paper we present a novel probabilistic data flow analysis framework (PDFA) which realizes an *execution history based approach*, i.e. the execution history is taken into account during the propagation of the probabilities through the control flow graph. Our approach is unique in utilizing execution history. We show that in this way significantly better results can be achieved with nearly the same computational effort as other approaches.

The paper is organized as follows. In Section 2 we describe the basic notions required to present our approach. In Section 3 we outline the basic ideas behind $\mathcal{S}_{best}$ and Ramalingam's history-independent approach and compare the results obtained by both approaches for our running example. Our novel approach is developed in Section 4. In Section 5 we compare the probabilistic results of the individual approaches for the SPECint95 benchmark suite and present time measurements. Related work is surveyed in Section 6 and, finally, we draw our conclusions in Section 7.

2 Preliminaries

Programs are represented by *directed flow graphs* $G = (N, E, \mathbf{s}, \mathbf{e})$, with node set N and edge set $E \subseteq N \times N$. Edges $m \rightarrow n \in E$ represent basic blocks of instructions and model the nondeterministic branching structure of G. *Start node* $\mathbf{s}$ and *end node* $\mathbf{e}$ are assumed to be free of incoming and outgoing edges, respectively. An element π of the set of paths Π of length k is a finite sequence $\pi = [n_1, n_2, \ldots, n_k]$ with $k \geq 1$, $n_i \in N$ for $1 \leq i \leq k$ and for all $i \in \{1, \ldots, k-1\}$, $n_i \rightarrow n_{i+1} \in E$.

A *program run* π_r is a path, which starts with node $\mathbf{s}$ and ends in node $\mathbf{e}$. The set of all *immediate predecessors* of a node n is denoted by $pred(n) = \{\, m \mid (m, n) \in E \,\}$. Function **occurs** : $(N \cup \Pi) \times \Pi \rightarrow \mathbb{N}_0$ denotes the number of occurrences of a node/subpath in a path.

As usual, a monotone data flow analysis problem is a tuple $DFA = (L, \wedge, F, c, G, M)$, where L is a bounded semilattice with meet operation $\wedge$, $F \subseteq L \rightarrow L$ is a monotone function space associated with L, $c \in L$ are the "data flow facts" associated with start node $\mathbf{s}$, $G = (N, E, \mathbf{s}, \mathbf{e})$ is a control flow graph, and $M : E \rightarrow F$ is a map from G's edges to data flow functions.

For bitvector problems the semilattice L is a powerset 2^D of finite set D. An element χ in 2^D represents a function from D to $\{0, 1\}$. $\chi(d)$ is 1, if d is element of χ, 0 otherwise.

3 Abstract Run and Ramalingam's Approach

Abstract Run. The theoretical best solution $\mathcal{S}_{best}$ can be determined by an abstract run [9]. The abstract run computes (1) the frequency $C(u)$ with which node u occurs in program run π_r and (2) the frequency $C(u,d)$ with which data fact d is true in node u for program run π_r. While $C(u)$ can be determined very easily, the computation of $C(u,d)$ is based on *monotone data flow problems* [8]: Whenever a node is reached, an associated function which describes the effect of that node on the data flow information is executed (for the details on computing $C(u)$ and $C(u,d)$ rf. to [9]).

Definition 1

$$\mathcal{S}_{best}(u,d) = \begin{cases} \frac{C(u,d)}{C(u)} & \textit{if } C(u) \neq 0, \\ 0 & \textit{otherwise.} \end{cases} \tag{1}$$

The definition above combines both frequencies. If frequency $C(u)$ of node u is zero, then $\mathcal{S}_{best}(u,d)$ is zero as well.

$\mathcal{S}_{best}(u,d)$	d_1	d_2	d_3	d_4	$C(u)$
s	0	0	0	0	1
1	0	0	0.9	0	10
2	0	0	0.889	0	9
3	0	0	1	0	1
4	0.9	0.1	0.1	0	10
5	1	0	0	0	9
6	0	1	1	0	1
7	0	0	1	0.1	10
8	0	0	1	1	1

Table 1. Running example: Result of Abstract Run.

Table 1 summarizes the results of our running example with program run π_r. Columns correspond to definitions (d_1 to d_4), and rows to nodes of the control flow graph. If $\mathcal{S}_{best}(u,d)$ is 0, definition d does not reach node u (impossible event). If $\mathcal{S}_{best}(u,d)$ is 1, definition d reaches node u each time (certain event). Any other numerical value in the range between 0 and 1 represents the probability for definition d reaching node u. E.g. consider $\mathcal{S}_{best}(\mathbf{4},d_1)$ and $\mathcal{S}_{best}(\mathbf{4},d_2)$. Node **4** occurs in program run π_r 10 times. Hence, the denominator of Equation 1 is 10 in both cases. To determine the number of times d_1 and d_2 reach node **4**, we trace definitions d_1, d_2 in execution path π_r. Since edge $\mathbf{2} \rightarrow \mathbf{4}$ is executed 9 times, d_1 reaches node **4** at least 9 times. Definition d_1 is killed on edge $\mathbf{5} \rightarrow \mathbf{7}$ and in the last iteration node **4** is entered through edge $\mathbf{3} \rightarrow \mathbf{4}$. Therefore, definition d_1 does not hold true in node **4** for the last iteration. Hence, the number of times d_1 reaches node **4** is 9 and $\mathcal{S}_{best}(\mathbf{4},d_1) = 9/10$. Similarly, since edge $\mathbf{3} \rightarrow \mathbf{4}$ is executed once and d_2 does not reach node **4** via edge $\mathbf{2} \rightarrow \mathbf{4}$, $\mathcal{S}_{best}(\mathbf{4}, d_2) = 1/10$. Note that classical reaching definitions analysis yields that every definition can reach each node except the start node. Nevertheless, an abstract run is not a viable approach. The main drawback stems from the tremendous size of program path π_r and the resulting execution time.

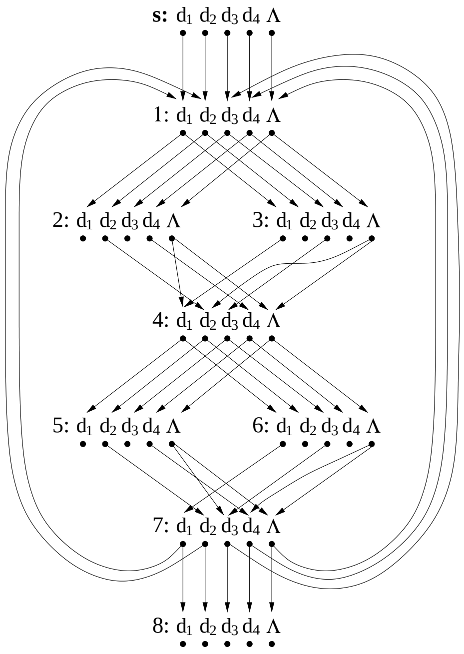

Fig. 2. Exploded flow graph of running example.

Ramalingam's Approach. Ramalingam [13] presents a framework for finite bi-distributive subset problems which estimates how often or with which probability a fact holds true during program execution. It is based on exploded control flow graphs introduced by Reps et al. [14] and Markov chains (with minor changes). Fig. 2 depicts the exploded control flow graph (*ECFG*) of our running example. The exploded control flow graph has $N \times D_\Lambda$ nodes, where D_Λ denotes data fact set D extended by symbol Λ. Edges of the extended control flow graph are derived from the representation relation (see [14]). Based on the *ECFG*, a linear equation system in $\mathbb{R}^+$ solves the expected frequencies which are used to compute the probabilities: $y(v, d)$ denotes the expected frequency for fact d to hold true at node v, and $y(v, \Lambda)$ gives the expected number of times that node v is executed. Thus $Prob(v, d) = y(v, d)/y(v, \Lambda)$ yields the probability for fact d to hold true in node v. The linear equation system is given as follows.

Ramalingam's Equation System:

$y(\mathbf{s}, \Lambda) = 1$

for all d in D:

$y(\mathbf{s}, d) = c(d)$

for all v in $N \setminus \{\mathbf{s}\}$: for all δ in D_Λ:

$$y(v, \delta) = \sum_{(u,\delta') \in \mathbf{pred}(v,\delta)} p(u, v) * y(u, \delta')$$

where $\mathbf{pred}(v, \delta)$ denotes the set of predecessors of node (v, δ) in the *ECFG* ($v \in N$, $\delta \in D_\Lambda$) and $p(u, v)$ denotes the probability that execution will follow edge $u \rightarrow v$ once u has been reached.

Table 2 lists the results of that approach for our running example. In comparison to Table 1, we can see deviations due to the reduction of the entire path to simple edge probabilities and the assumption that an incoming edge is independent from an outgoing edge. E.g. consider the probability of definition d_4 to reach node **6**. Although the abstract run yields probability 0, the result of Ramalingam's approach is 0.299, since it cannot be recognized that node **6** is always entered via path **[3,4,6]**.

$Prob(u, \delta)$	d_1	d_2	d_3	d_4	Λ
s	0	0	0	0	1
1	0.082	0.299	0.817	0.332	10
2	0.082	0.299	0.817	0.332	9
3	0.082	0.299	0.817	0.332	1
4	0.908	0.369	0.082	0.299	10
5	0.908	0.369	0.082	0.299	9
6	0.908	0.369	0.082	0.299	1
7	0.091	0.332	0.908	0.369	10
8	0.091	0.332	0.908	0.369	1

Table 2. Running example: Result of the history-independent approach.

Table 3 lists the absolute deviations as a percentage. Except for start node **s** the data facts deviate in a range between $\pm 0.8\%$ and $\pm 91.9\%$. In the following section we present a novel approach which takes execution history into account and yields in this way significantly better results.

4 Two-Edge Approach

The main idea of our two-edge approach is to relate outgoing edges with incoming ones. Instead of propagating information through nodes the two-edge approach carries data flow information along edges in order to take execution history

$\Delta\%$	d_1	d_2	d_3	d_4
s	0.0	0.0	0.0	0.0
1	8.2	29.9	-8.3	33.2
2	8.2	29.9	-7.2	33.2
3	8.2	29.9	-18.3	33.2
4	0.8	26.9	-1.8	29.9
5	-9.2	36.9	8.2	29.9
6	90.8	-63.1	-91.8	29.9
7	9.1	33.2	-9.2	26.9
8	9.1	33.2	-9.2	-63.1

Table 3. Comparison of the History-Independent Approach vs. Abstract Run (0% means that there is no deviation; maximum deviations are ±100%).

into account. This is achieved by relating unknowns of the equation system to *ECFG* edges. Let $\widehat{y}(\mathbf{v} \rightarrow \mathbf{w}, d)$ denote the expected frequency for fact d to hold true at node $\mathbf{w}$ under the condition that edge $\mathbf{v} \rightarrow \mathbf{w}$ has been taken, and let $\widehat{y}(\mathbf{v} \rightarrow \mathbf{w}, \Lambda)$ denote the expected number of times that edge $\mathbf{v} \rightarrow \mathbf{w}$ is executed.

Further, $p(u, v, w)$ denotes the probability that execution will follow edge $\mathbf{v} \rightarrow \mathbf{w}$ once it reaches edge $\mathbf{u} \rightarrow \mathbf{v}$. Consequently, the sum of the probabilities for all outgoing edges of edge $\mathbf{u} \rightarrow \mathbf{v}$ must be either one or zero[1]. Path profiling techniques necessary to compute $\mathbf{occurs}([\mathbf{u,v,w}], \pi_r)$ are discussed in detail in [19].

$$p(u, v, w) = \begin{cases} \frac{\mathbf{occurs}([\mathbf{u,v,w}], \pi_r)}{\mathbf{occurs}(\mathbf{u} \rightarrow \mathbf{v}, \pi_r)} & \text{if } \mathbf{occurs}(\mathbf{u} \rightarrow \mathbf{v}, \pi_r) \neq 0, \\ 0 & \text{otherwise.} \end{cases} \tag{2}$$

Next we introduce a function **In** to determine the set of preceding ingoing edges of an edge. The function is derived from the exploded CFG. Fig. 3 depicts the relation between an edge $\mathbf{v} \rightarrow \mathbf{w}$ with data fact δ and its preceding node $\mathbf{u}$ with data fact δ''.

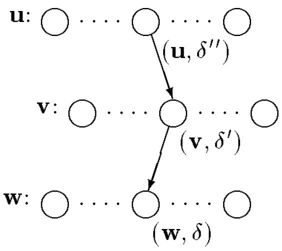

Fig. 3. Graphical visualization of function **In**

[1] If there are no outgoing edges for edge $\mathbf{u} \rightarrow \mathbf{v}$.

Hence, **In** is given as follows,

$$\mathbf{In}(\mathbf{v} \to \mathbf{w}, \delta) = \{(u, \delta') | (u, \delta'') \to (v, \delta') \to (w, \delta) \in \Pi_{ECFG}\}\,. \tag{3}$$

where Π_{ECFG} represents the set of paths in the exploded control flow graph.

Since we need a "root" edge, we introduce an artificial edge $\epsilon \to \mathbf{s}$ with pseudo node ϵ. This artificial edge does not have any predecessor edge and the outgoing edges of node $\mathbf{s}$ are successor edges. Note that $pred(\mathbf{s}) = \{\epsilon\}$. We associate the initial values $c(d)$ of the data flow analysis problem with the data facts of edge $\epsilon \to \mathbf{s}$. We further extend the probability $p(\epsilon, \mathbf{s}, v)$: For node $\mathbf{v}$, $p(\epsilon, \mathbf{s}, v)$ is either one (the program executes node v immediately after node $\mathbf{s}$) or the probability is zero (another subsequent node of node $\mathbf{s}$ was taken.)

Finally, we can describe the linear equation system for a general data flow problem by the following formula. Once the equation system of edge unknowns has been solved, the expected frequencies of data facts are determined by summing up the unknowns of the incoming edges as shown in Equation 7.

Two-Edge Equation System

$$\widehat{y}(\epsilon \to \mathbf{s}, \Lambda) = 1 \tag{4}$$

for all d in D:

$$\widehat{y}(\epsilon \to \mathbf{s}, d) = c(d) \tag{5}$$

for all $\mathbf{v} \to \mathbf{w}$ in E: for all δ in D_Λ:

$$\widehat{y}(\mathbf{v} \to \mathbf{w}, \delta) = \sum_{(u,\delta') \in \mathbf{In}(\mathbf{v} \to \mathbf{w}, \delta)} p(u, v, w) * \widehat{y}(\mathbf{u} \to \mathbf{v}, \delta') \tag{6}$$

for all w in N: for all δ in D_Λ:

$$\widehat{y}(w, \delta) = \sum_{u \in \mathbf{pred}(v)} \widehat{y}(\mathbf{v} \to \mathbf{w}, \delta) \tag{7}$$

The two-edge approach generates a set of very simple linear equations. Consequently, any of the standard algorithms for solving linear algebraic equations can be used. Most of these algorithms have a worst case complexity of $O(n^3)$ where n is the number of unknowns in the equation system. In the equation system of the two-edge approach there exist $(|E| + 1) \times |D_\Lambda|$ unknowns. Clearly, Ramalingam's approach has less unknowns $|N| \times |D_\Lambda|$ due to the fact that the probabilities are related to nodes rather than edges. But the effort can be reduced by solving the equation system in two steps. In the first step we only solve Λ unknowns, which only depend on Λ unknowns itself. In the second step we solve data fact unknowns, which depend on Λ and data fact unknowns. Due to the first step Λ unknowns become constants for the second step.

Standard algorithms for solving linear equation system are usually to inefficient because they fail to utilize the extreme sparsity of the CFG. For our purpose, we can adapt various elimination methods [16, 1, 17, 8] (a good survey

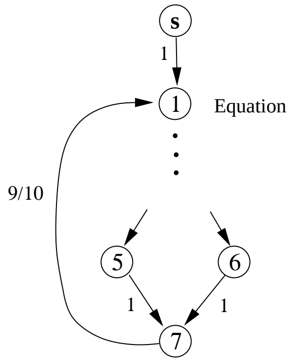

Fig. 4. Subgraph of Fig. 1 for node **1** annotated with edge probabilities.

can be found in [15]). These algorithms are often linear or almost linear in size of the graph.

Clearly, the two-edge approach can be extended to a three-edge, four-edge, or k-edge approach accordingly resulting in better probabilistic results. However, the time required to solve the system of equations will increase as well. As shown in our experimental section the two-edge approach yields for the SPECint95 benchmark suite very precise results. Hence, we believe that the two-edge approach is a good compromise between complexity and required precision.

In the following we present the differences between Ramalingam's and our two-edge approach for our running example. We illustrate the differences by discussing equations of data flow facts. In the first case Ramalingam's approach overestimates the probabilities and in the second case underestimates them.

Case 1. Consider Fig. 4 and the equation at node **1** for definition d_1. Since the edge probability of $\mathbf{7} \rightarrow \mathbf{1}$ equals $9/10$, the equations of Ramalingam's approach are given as follows:

$$
\begin{aligned}
y(1, d_1) &= 9/10 * y(7, d_1) + 1 * y(\mathbf{s}, d_1) \\
y(7, d_1) &= y(5, d_1) + y(6, d_1)
\end{aligned}
$$

Since $y(\mathbf{s}, d_1)$ is initialized to zero, $y(1, d_1)$ depends solely on $y(7, d_1)$. Further, definition d_1 is killed on edge $\mathbf{5} \rightarrow \mathbf{7}$ which results in $y(5, d_1)$ to be zero and, hence, $y(7, d_1)$ depends in turn solely on $y(6, d_1)$. Thus the value of $y(6, d_1)$ is propagated to $y(1, d_1)$, although the path **[6,7,1]** is never executed. As a consequence, the value of of $y(1, d_1)$ is too high compared with the result of the abstract run.

Since in the two-edge approach the unknowns are related to edges rather than nodes, the expected frequency of a data fact at a node is defined by the sum of unknowns of all incoming edges:

$$
\widehat{y}(1, d_1) = \widehat{y}(\mathbf{s} \rightarrow \mathbf{1}, d_1) + \widehat{y}(\mathbf{7} \rightarrow \mathbf{1}, d_1)
$$

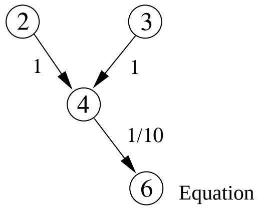

Fig. 5. Subgraph of Fig. 1 for node **6** annotated with edge probabilities.

Note that $\widehat{y}(\mathbf{s} \rightarrow \mathbf{1}, d_1)$ is 0 because there is no reaching definition in start node **s**. Only the second part of the sum needs to be considered.

$$\begin{aligned} \widehat{y}(\mathbf{7} \rightarrow \mathbf{1}, d_1) = p(5,7,1) * \widehat{y}(\mathbf{5} \rightarrow \mathbf{7}, d_1) + \\ p(6,7,1) * \widehat{y}(\mathbf{6} \rightarrow \mathbf{7}, d_1) \end{aligned}$$

The first part of the sum reduces to 0 since definition d_1 is killed on edge $\mathbf{5} \rightarrow \mathbf{7}$. The second part of the sum is 0 as well due to the fact that probability $p(6,7,1)$ is zero (path **[6,7,1]** is never taken). We obtain $\widehat{y}(1, d_1) = 0$, which perfectly matches the result of the abstract run.

Case 2. Consider Fig. 5 and the data flow equation at node **6** for definition d_2. Since the edge probability of $\mathbf{4} \rightarrow \mathbf{6}$ is given by $1/10$, we have for Ramalingam's approach the following equation:

$$y(6, d_2) = 1/10 * y(4, d_2)$$

Note that on edge $\mathbf{3} \rightarrow \mathbf{4}$ variable Y is defined by d_2, whereas edge $\mathbf{2} \rightarrow \mathbf{4}$ is transparent for variable Y. Hence, definition d_2 can reach node **6** also via path **[2,4,6]** depending on execution history. Since execution history is not taken into account, this results in a value less than probability 1 of the abstract run.

Again, for the two-edge approach, we are relating the unknowns to edges resulting in $\widehat{y}(6, d_2) = \widehat{y}(\mathbf{4} \rightarrow \mathbf{6}, d_2)$, since node **4** is the only predecessor. By weighting the incoming edges of node **4** with their probabilities we get:

$$\begin{aligned} \widehat{y}(\mathbf{4} \rightarrow \mathbf{6}, d_2) = p(2,4,6) * \widehat{y}(\mathbf{2} \rightarrow \mathbf{4}, d_2) + \\ p(3,4,6) * \widehat{y}(\mathbf{3} \rightarrow \mathbf{4}, d_2). \end{aligned}$$

Note that probability $p(2,4,6)$ is zero and the first part of the sum vanishes since path **[2,4,6]** is never taken. Probability $p(3,4,6)$ is 1 according to Equation 2 since path **[3,4,6]** is always executed when edge $\mathbf{3} \rightarrow \mathbf{4}$ is reached. Therefore we obtain

$$\widehat{y}(\mathbf{4} \rightarrow \mathbf{6}, d_2) = \widehat{y}(\mathbf{3} \rightarrow \mathbf{4}, d_2)$$

Definition d_2 is executed on edge $\mathbf{3} \rightarrow \mathbf{4}$ and, hence, $\widehat{y}(\mathbf{3} \rightarrow \mathbf{4}, d_2)$ is equal to $\widehat{y}(\mathbf{3} \rightarrow \mathbf{4}, \Lambda)$. Since $\widehat{y}(\mathbf{3} \rightarrow \mathbf{4}, \Lambda)$ denotes the number of times edge $\mathbf{3} \rightarrow \mathbf{4}$ occurs in the program run, $\widehat{y}(\mathbf{3} \to \mathbf{4}, \Lambda)$ equals 1. Finally, we substitute backwards and gain $\widehat{y}(6, d_2) = 1$ which perfectly matches the result of the abstract run.

It is important to stress that our method, which is based on relating outgoing edges to incoming ones, can trace rather complicated flow graph paths. E.g. we get that definition d_3 reaches node **6** each time (i.e. probability equals 1): Thus our method finds out that edge $\mathbf{5} \to \mathbf{7}$ has been executed prior to node **6** in one of the previous loop iterations and that definition d_3 has not been killed by taking edge $\mathbf{2} \to \mathbf{4}$ before execution arrives at node **6**.

For the running example the solution of our two-edge approach is identical to the best solution $\mathcal{S}_{best}$ of the abstract run as shown in Table 1, whereas Ramalingam's approach significantly deviates from the abstract run as shown in Table 3.

5 Experimental Results

In our experiments we address two issues. First, we show that for the two-edge approach the analysis results are significantly better. We illustrate this by analyzing the SPECint95 benchmark suite. Second, we demonstrate that probabilistic data flow analysis frameworks are viable even for larger programs, since in most cases the original equation system can be reduced considerably and the remaining equations are usually trivial ones with a constant or only one variable on the right-hand side.

The compilation platform for our experiments is GNU gcc. We integrated abstract run, Ramalingam's one-edge approach, and the two-edge approach. To evaluate Ramalingam's approach and the two-edge approach we have chosen SPECint95 as benchmark suite and the reaching definitions problem as reference data flow problem. The profile information has been generated by running the training set of SPECint95. Of course, the same training set has been used for the abstract run as well.

Probabilistic Data Flow Results. In our first experiment we have compared the deviations of the two-edge approach from the abstract run with the deviations of the one-edge approach from the abstract run. For each benchmark program of SPECint95 we added the absolute deviations for each CFG node and each data flow fact. The ratio of the sums of the two-edge approach over the one-edge approach is shown in Fig. 6. The improvement for SPECint95 is in the range of 1.38 for *vortex* up to 9.55 for *li*. The experiment shows that the results of the two-edge approach compared to the one-edge approach are significantly better.

Above we have shown that the results of the two-edge approach are significantly better than the results of the one-edge approach, but maybe both solutions deviate substantially from the theoretically best solution. Hence, we executed the abstract run to get the theoretically best solution and compared it with the two-edge approach. For each benchmark program of SPECint95 we calculated for all CFG nodes and data flow facts the mean of deviations of the probabilities. The results are shown in Fig. 7. The mean of deviations is between

SPECint95	Improvement Ratio
go	1.95
m88ksim	2.26
gcc	2.54
compress	3.28
li	9.55
ijpeg	4.20
perl	2.44
vortex	1.38

Fig. 6. SPECint95: Improvement ratio of two-edge approach compared to Ramalingam's one-edge approach.

SPECint95	Av. Prob Δ	Function hits
go	1.72	42.6%
m88ksim	0.31	87.9%
gcc	0.41	76.8%
compress	0.36	78.9%
li	0.11	94.2%
ijpeg	0.20	92.2%
perl	0.16	89.9%
vortex	0.16	92.4%

Fig. 7. SPECint95: Comparison two-edge approach with abstract run.

0.11% for *li* and 1.72% for *go*. The reason for this excellent result is that usually programs execute only a small fraction of the potential paths. For non-executed CFG nodes the result of the two-edge approach always coincides with the probability of the abstract run (namely zero), since the two-edge probability (Equ. 2) yields zero. Hence, we did a second comparison which is more meaningful. We calculated the percentage of functions for which the two-edge approach coincides with the abstract run. The function hits reaches for the two-edge approach from 42.6% for benchmark *go* up to 94.2% for *li* which is an excellent result as well.

Effort for Solving Linear Algebraic Equation Systems. In general, the worst case complexity for solving linear algebraic equation systems is $O(n^3)$ with n denoting the number of unknowns. The number of unknowns for the original one-edge and two-edge equation system as described in Sections 3 and 4 can be rather big. However, unknowns, which are related to control flow nodes or edges which are not visited during the profile run, can be removed immediately. In the SPECint95 suite only 48.2% of the original unknowns of the one-edge approach have to be computed; for the two-edge approach 40.5% of the original unknowns need to be solved.

Moreover, the structure of the equations for the SPECint95 suite is rather simple. For the one-edge approach 54.3% of the equations are trivial ones with constants on the right-hand side only. Similarly, for the two-edge approach we have a percentage rate of 58.8% equations with constants on the right-hand side. For both approaches only about 1.6% of the equations have more than two variables (up to 150) on the right-hand side. Hence, a linear equation solver for sparse systems is of key importance.

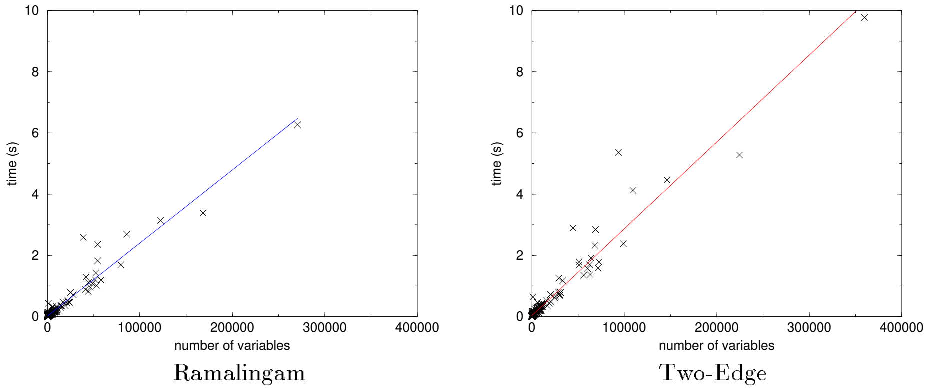

Fig. 8. Time to solve unknowns.

We have chosen an elimination framework introduced by Sreedhar et al.[16], which was originally developed to solve classical bit-vector problems on control flow graphs. Especially, for our extremely sparse equation system Sreedhar's framework is well suited. In Fig. 8 a data point represents the number of unknowns for a C-module of SPECint95 and the time in seconds to solve the unknowns. Here, we have measured the graph reduction and propagation of Sreedhar's framework without setting up DJ-graphs[2] and without setting up the equations itself which takes additionally time since profile data must be accessed. The left graph depicts the measurements of Ramalingam's approach – the right graph shows the measurements of the two-edge approach. It is really remarkable that Sreedhar's algorithm nearly works linear on extremely sparse equation systems. The measurements were taken on a Sun Ultra Enterprise 450 (4x UltraSPARC-II 296MHz) with 2560MB RAM.

6 Related Work

Several approaches have been proposed which take advantage of profile information to produce highly efficient code.

Ramalingam [13] presents a generic data flow framework which computes the probability that a data flow fact will hold at some program point for finite

[2] To set up DJ-graphs dominator trees are required. Recently, linear algorithms were introduced[4].

bi-distributive subset problems. The framework is based on exploded control flow graphs introduced by Reps, Horwitz, Sagiv [14] and on Markov-chains. Contrary to our approach, execution history is not taken into account. To our best knowledge we are not aware of any other execution history based approach. Optimizations based on PDFAs are presented in [10, 11].

Alternatively, Ammons and Larus [2] describe an approach to improve the results of data flow analysis by identifying and duplicating hot paths in the program's control flow graph resulting in a so-called hot path graph in which these paths are isolated. Data flow analysis applied to a hot path graph yields more precise data flow information. The goal of this approach differs from our work. We improve the precision of a probabilistic data flow solution and do not modify the control flow graph in order to enable heavily executed code to be highly optimized.

Finally, profile information is used by several researchers for specific optimization problems in order to get better results (e.g. [6, 5, 12, 7, 3, 18]).

7 Conclusion

Probabilistic data flow frameworks set forth new directions in the field of optimization. We presented a novel, practicable probabilistic data flow framework which takes execution history into account by relating outgoing edges to incoming ones. In this way we achieve significantly better results. Practical experiments which have been performed for the SPECint95 benchmark suite showed that the two-edge approach is feasible and the precision of the probabilistic results is sufficient.

References

1. F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, March 1976.
2. G. Ammons and J.R. Larus. Improving data-flow analysis with path profiles. In *Proc. of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI'98)*, pages 72–84, Montreal, Canada, June 1998.
3. R. Bodík, R. Gupta, and M.L. Soffa. Complete removal of redundant expressions. *ACM SIGPLAN Notices*, 33(5):1–14, May 1998.
4. A.L. Buchsbaum, H. Kaplan, A. Rogers, and J.R. Westbrook. Linear-time pointer-machine algorithms for LCAs, MST verification, and dominators. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing (STOC-98)*, pages 279–288, New York, May 23–26 1998. ACM Press.
5. B. Calder and D. Grunwald. Reducing branch costs via branch alignment. *ACM SIGPLAN Notices*, 29(11):242–251, November 1994.
6. J. A. Fisher. Trace scheduling : A technique for global microcode compaction. *IEEE Trans. Comput.*, C-30(7):478–490, 1981.
7. R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial dead code elimination using predication. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'97)*, pages 102–115, San Francisco, California, November 1997.

8. M.S. Hecht. *Flow Analysis of Computer Programs.* Programming Language Series. North-Holland, 1977.
9. E. Mehofer and B. Scholz. Probabilistic data flow system with two-edge profiling. *Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00). ACM SIGPLAN Notices*, 35(7):65 – 72, July 2000.
10. E. Mehofer and B. Scholz. Probabilistic procedure cloning for high-performance systems. In *12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2000)*, Sao Pedro, Brazil, October 2000.
11. E. Mehofer and B. Scholz. Probabilistic communication optimizations and parallelization for distributed-memory systems. In *PDP 2001*, Mantova, Italy, February 2001.
12. T. C. Mowry and C.-K. Luk. Predicting data cache misses in non-numeric applications through correlation profiling. In *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-97)*, pages 314–320, Los Alamitos, December 1–3 1997. IEEE Computer Society.
13. G. Ramalingam. Data flow frequency analysis. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 267–277, Philadephia, Pennsylvania, May 1996.
14. T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61, San Francisco, CA, January 1995.
15. B. G. Ryder and M. C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–315, September 1986.
16. V. C. Sreedhar, G. R. Gao, and Y.-F. Lee. A new framework for elimination-based data flow analysis using DJ graphs. *ACM Transactions on Programming Languages and Systems*, 20(2):388–435, March 1998.
17. R. E. Tarjan. Fast algorithms for solving path problems. *Journal of the ACM*, 28(3):594–614, July 1981.
18. C. Young and M. D. Smith. Better global scheduling using path profiles. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO-98)*, pages 115–126, Los Alamitos, November 30–December 2 1998. IEEE Computer Society.
19. C. Young and M.D. Smith. Static correlated branch prediction. *ACM Transactions on Programming Languages and Systems*, 21(5):1028–1075, September 1999.