Probabilistic Data Flow System with Two-Edge Profiling *†

Eduard Mehofer

Institute for Software Technology and Parallel Systems, University of Vienna, Austria
mehofer@par.univie.ac.at

Bernhard Scholz

Institute for Computer Languages, Vienna University of Technology, Austria
scholz@complang.tuwien.ac.at

Abstract

Traditionally optimization is done statically independent of actual execution environments. For generating highly optimized code, however, runtime information can be used to adapt a program to different environments. In probabilistic data flow systems runtime information on representative input data is exploited to compute the probability with what data flow facts may hold. Probabilistic data flow analysis can guide advanced optimizing transformations in order to improve the running time of programs. In comparison classical data flow analysis does not take runtime information into account. All paths are equally weighted irrespectively whether they are never, heavily, or rarely executed.

In this paper we present the best solution what we can theoretically obtain for probabilistic data flow problems and compare it with the state-of-the-art one-edge approach. We show that the differences can be considerable and improvements are crucial. However, the theoretically best solution is too expensive in general and feasible approaches are required. In the sequel we develop an efficient approach which employs two-edge profiling and classical data flow analysis. We show that the results of the two-edge approach are significantly better than the state-of-the-art one-edge approach.

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

†Accepted for publication at ACM Sigplan Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo'00), Boston, Jan. 2000 (in conjunction with POPL'00).

1 Introduction

Usually, classical data flow analysis is done statically without utilizing runtime information. All paths are equally weighted irrespectively whether they are never, heavily, or rarely executed. In contrast probabilistic data flow analysis takes runtime information into account by using edge probabilities to distinguish between frequently and rarely executed branches. Probabilistic data flow systems propagate data flow facts along paths of the control flow graph weighting them with edge probabilities. The resulting solution gives us the *probabilities* with what data flow facts may hold true during execution at some program point.

Probabilistic data flow analysis can guide advanced optimizing transformations in order to improve the running time of programs. Examples for optimizations which utilize runtime information are register allocation [6], code motion [3, 4], and redundant computations [2].

Figure 1 depicts the probabilistic optimization framework of an optimizing compiler with a profiling feedback loop. The profiler is responsible for producing profile information based on the execution environment. The profile information is passed over to the probabilistic optimizer. Control flow analysis constructs the control flow graph annotated with edge probabilities. Based on the annotated control flow graph and data flow equations, the probabilistic data flow analysis computes a probabilistic solution of the data flow problem. Sophisticated transformations can rely on that probabilistic solution to generate highly optimized code. For changing execution environment the the target code is adapted by the

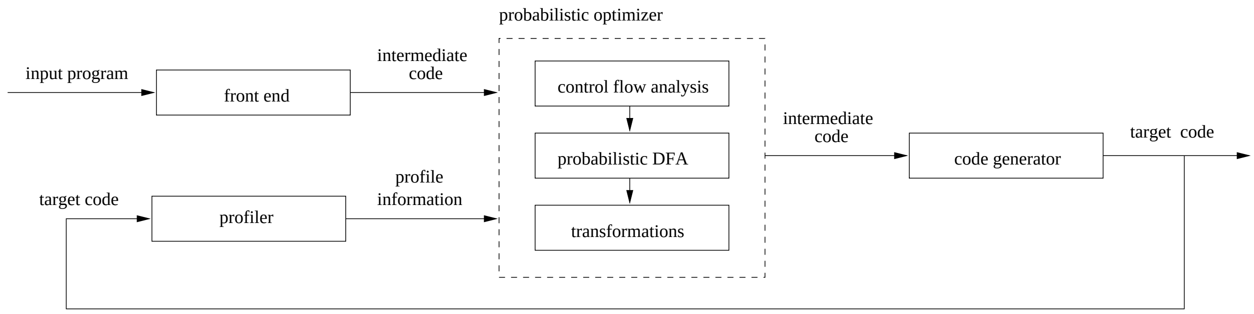

Figure 1: Probabilistic optimization framework.

profiling feedback loop.

A probabilistic data flow framework has been presented by Ramalingam [7]. In his approach it is assumed that a particular branch is independent of the execution history, which is obviously not the case in reality. In order to discuss the accuracy of his solution we define the theoretically best solution what we can achieve and present an approach to calculate that solution. We show that the differences can be considerable and improvements are of crucial importance. However, the approach for calculating the theoretically best solution is too expensive in general and, hence, not feasible in practice.

Besides the computation of the theoretically best solution, a central part in this paper is the extension of the *one-edge approach* presented in [7] to the *two-edge approach*, which takes care of execution history by using two-edge probabilities instead of one-edge probabilities. We specify the equation system for the two-edge approach. Then, we discuss the results of the presented approaches in the context of our running example.

The paper is organized as follows. In Section 2 we intuitively describe the problem and motivate our approach. In Section 3 we describe basic notions used in this paper. In Section 4 we compute $\mathcal{S}_{best}$ based on abstract execution. The results of the one-edge approach are presented for our running example in Section 5. The two-edge approach is developed in Section 6, and the results of the three approaches are discussed there. Related work is surveyed in Section 7 and, finally, we draw our conclusions in Section 8.

2 Motivation

In this section we intuitively explain the theoretically best solution, show the importance of considering execution history, and illustrate the essence of our approach. For the sake of demonstration, we have chosen the classical reaching definitions problem (see [5]). Figure 2 shows the control flow graph of our running example consisting of two subsequent branching statements inside a loop. The example has four definitions (d_1 to d_4) for two variable X and Y. Variable X is defined at edges $\mathbf{2} \to \mathbf{4}$ by d_1 and $\mathbf{5} \to \mathbf{7}$ by d_3. Variable Y is defined at edges $\mathbf{3} \to \mathbf{4}$ by d_2 and edge $\mathbf{6} \to \mathbf{7}$ by d_4. Let us assume that the example executes 10 times the loop with firstly taking 9 times left branch **[1,2,4,5,7]**, and terminating by taking right branch **[1,3,4,6,7,8]**.

The classical reaching definitions problem computes that all definitions reach nodes **1** to **8**. This solution is a conservative approximation which is valid for all program runs. Additional runtime information can refine the results according to a specific execution environment. Rather than considering the binary information whether a definition may or will not reach a node, a numerical value denotes *with what probability* a definition will reach a node.

A probabilistic data flow framework which computes the probability of data flow facts has been presented by Ramalingam [7]. In order to discuss the accuracy of his solution, we have to raise the following question: *What is the "best" solution we can theoretically obtain?* Consider our running example of Figure 2 and the reaching definitions problem. Obvi-

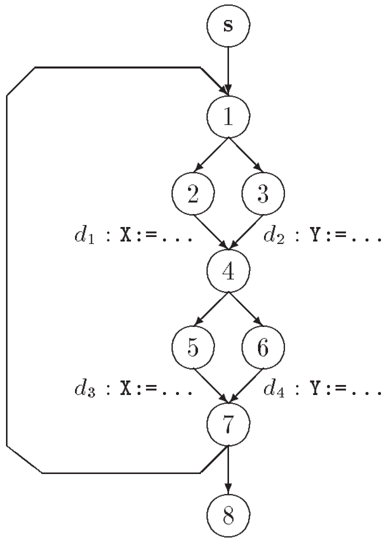

Figure 2: Running example.

ously, the "best" solution $\mathcal{S}_{best}(d_i, n)$ which denotes the probability for a definition d_i to reach a control flow node n with respect to a program run π_r is given by the ratio:

$$\mathcal{S}_{best}(d_i, n) = \frac{nr.\ of\ times\ d_i\ reaches\ n}{nr.\ of\ times\ n\ occurs\ in\ \pi_r} \tag{1}$$

E.g. consider $\mathcal{S}_{best}(d_1, \mathbf{4})$ and $\mathcal{S}_{best}(d_2, \mathbf{4})$. 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 reaches node **4**, we trace definitions d_1, d_2 in execution path π_r. Since edge $\mathbf{2} \to \mathbf{4}$ is executed 9 times, d_1 reaches node **4** at least 9 times. In the last loop iteration node **4** is entered through edge $\mathbf{3} \to \mathbf{4}$ and previously definition d_1 has been killed on edge $\mathbf{5} \to \mathbf{7}$. Therefore, definition d_1 does not hold in node **4** for the last iteration. Hence, the number of times d_1 reaches node **4** is 9 and $\mathcal{S}_{best}(d_1, \mathbf{4}) = 9/10$. Similarly, since edge $\mathbf{3} \to \mathbf{4}$ is executed once and d_2 does not reach node **4** via edge $\mathbf{2} \to \mathbf{4}$, $\mathcal{S}_{best}(d_2, \mathbf{4}) = 1/10$.

The starting point of our work is $\mathcal{S}_{best}$. We present an approach to compute $\mathcal{S}_{best}$, however, this approach is too expensive and not feasible in practice. Nevertheless, it enables us to compare the results of $\mathcal{S}_{best}$ with the results achieved in [7]. Since the differences can be considerable, better solutions are of crucial importance. In fact, there are two reasons which are responsible for the deviations. On the one hand, a program path is reduced to edge probabilities. On the other hand, it is assumed that a particular branch is independent of the execution history, which obviously is not the case in reality. The usage of edge probabilities is indispensable to get an efficient handle on the problem. However, there exists a potential for improvements by taking the execution history into account.

We consider execution history by using *two-edge probabilities* instead of *one-edge probabilities*. E.g. consider again our running example: Whereas in the *one-edge approach* presented in [7] edges $\mathbf{4} \to \mathbf{5}$ and $\mathbf{4} \to \mathbf{6}$ are dealt with independently of the incoming edges $\mathbf{2} \to \mathbf{4}$ and $\mathbf{3} \to \mathbf{4}$, our *two-edge approach* recognizes 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 $\mathbf{2} \to \mathbf{4}$. Similarly, we find that d_4 cannot reach node **6**. Additionally, since the loop is exited via path **[6,7,8]**, in the two-edge approach it can be determined that definition d_2 cannot reach node **8**. In this paper we develop the two-edge approach, present the results for our running example, and compare it with the solution $\mathcal{S}_{best}$ and the one-edge approach.

3 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 $(m, n) \in E$ represent basic blocks of instructions and model the nondeterministic branching structure of G. *Start node* **s** and *end node* **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 $k \geq 0$ and $n_i \in N$ for all i, $1 \leq i \leq k$ iff for all $i \in \{1, \ldots, k-1\}$, $n_i \to n_{i+1}$ is in E.

A *program run* π_r is a path, which starts with node **s** and ends in node **e**. $\mathbf{Paths}(u, v)$ denotes the set of all paths from u to v. The set of all *immediate successors* and *immediate predecessors* of a node n are $succ(n) = \{\, m \mid (n, m) \in E \,\}$ and $pred(n) = \{\, m \mid (m, n) \in E \,\}$. The function **occurs** : $(N \cup \Pi) \times \Pi \to \mathbb{N}_0$ counts the number of occurrence of a node/sub-path in a path.

Definition 1 *A monotone data flow analysis problem is a tuple $DFA = (L, \wedge, F, c, G, M)$, where*

- *L is a bounded semi-lattice with meet operation $\wedge$.*
- *$F \subseteq L \to L$ 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 with start node $\mathbf{s}$ and end node $\mathbf{e}$.*
- *$M : E \to F$ is a map from G's edges to data flow functions.*

Function M is to be extended to path $\pi = [n_1, n_2, \ldots, n_k]$ where $M(\pi)$ is defined as $M(\pi) = M(n_{k-1} \to n_k) \circ M(n_{k-2} \to n_{k-1}) \ldots M(n_1 \to n_2)$. If path π is empty ($\pi = [\,]$), then $M([\,])$ is the identity function, $\lambda x.x$. Given path π from the start node $\mathbf{s}$ to some node u, we define $\mathbf{state}(\pi)$ to be $M(\pi)(c)$.

Definition 2 *The meet-over-all-path (MOP) solution $S(u)$ for a data flow analysis problem is given as follows:*

$$S(u) = \bigwedge_{\pi \in \mathbf{Paths}(\mathbf{s}, u)} \mathbf{state}(\pi) \tag{2}$$

For bit-vector problems the semi-lattice 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.

4 Abstract Run

In Section 2 an intuitive notion of the "best" solution $\mathcal{S}_{best}$ has been given. Based on the monotone data flow analysis problem *DFA* (see Definition 1) we can describe $\mathcal{S}_{best}$ as abstract run. An abstract run computes (1) the frequencies with what nodes occur in program run π_r and (2) the frequencies with what data facts hold in nodes. More formally, the frequency $C(u)$ of node u is defined as the number of times u occurs in π_r,

$$C(u) = \mathbf{occurs}(u, \pi_r). \tag{3}$$

The frequency $C(u, d)$ denotes the number of times data fact d holds true at u for program run π_r. Function $C(u, d)$ can be described as a sum over a recursively defined *solution sequence* of program run $\pi_r = [\mathbf{s}, n_2, \ldots, n_{k-1}, \mathbf{e}]$. The first element $X_0^{(\mathbf{s})}$ in solution sequence $[X_0^{(\mathbf{s})}, X_1^{(n_2)}, \ldots, X_{k-1}^{(n_{k-1})}, X_k^{(\mathbf{e})}]$ are data facts c, which are associated with start node $\mathbf{s}$. To abstractly execute an edge $e_i = n_i \to n_{i+1}$ $(1 \le i < k)$ in π_r, we use the recurrence $X_i^{(n_{i+1})} = M(n_i \to n_{i+1})(X_{i-1}^{(n_i)})$. Then we have,

$$C(u, d) = \sum_{\substack{0 \le i \le k, \\ n_i = u}} X_i^{(n_i)}(d). \tag{4}$$

In Equation 4 we increment $C(u, d)$ whenever d holds in u. Note that $X_i^{(n_i)}$ is element of 2^D and $X_i^{(n_i)}(d)$ returns 1 if d is in set $X_i^{(n_i)}$, 0 otherwise.

Definition 3

$$\mathcal{S}_{best}(u, d) = \begin{cases} \frac{C(u,d)}{C(u)}, & \textit{if } C(u) \neq 0 \\ 0, & \textit{otherwise} \end{cases} \tag{5}$$

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. Let us consider our running example and program run π_r from Section 2. The first element for start node $\mathbf{s}$ of the recursively defined solution sequence is empty ($X_0^{(\mathbf{s})} = \emptyset$). In the next step we abstractly execute edge $\mathbf{s} \to \mathbf{1}$. On this transition no reaching definition is killed or generated and, therefore, $X_1^{(1)}$ is empty as well. The same is valid for edge $\mathbf{1} \to \mathbf{2}$ ($X_2^{(2)} = \emptyset$). Since d_1 is generated on edge $\mathbf{2} \to \mathbf{4}$, $X_3^{(4)} = \{d_1\}$. We continue to compute the solution sequence until the last edge of program run π_r is abstractly executed. Now, we can sum up definition d over all elements of the solution sequence, which are associated with node u. If data fact d holds, $C(u, d)$ is incremented by one. E.g., $X_k^{(4)}(d_1)$ is equal one for $k \in \{3, 8, 13, 18, 23, 28, 33, 38, 43\}$. For k equal to 48, d does not hold due to the kill of definition d_1 in preceding edge $\mathbf{5} \to \mathbf{7}$ and the transition $\mathbf{1} \to \mathbf{3}$ instead of $\mathbf{1} \to \mathbf{2}$.

Table 1 summarizes the results of our running example. A column corresponds to a definition (d_1 to d_4). Each row corresponds to a node in the control

$\mathcal{S}_{best}(u,d)$	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: Abstract Run

flow graph. Definition d cannot reach node u if numerical value of $\mathcal{S}_{best}(u,d)$ is zero (impossible event). If $\mathcal{S}_{best}(u,d)$ is 1 definition d reaches node u every time (certain event). Any other numerical value in the range between 0 and 1 represents the likelihood of definition d_i to reach the node. E.g., the abstract run computes a probability of 0.9 for definition d_3 in node **1**.

Note that classical reaching definitions analysis (see Definition 2) 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.

5 One-Edge Approach

Ramalingam [7] provides a one-edge framework for finite bi-distributive subset problems to estimate how often or with what probability a fact holds true during program execution. The estimation is based on one-edge probabilities. Similar to the meet-over-all-path solution (see Definition 2) a sum-over-all-path solution was introduced to mathematically describe a probabilistic data flow problem. Note that for modeling the problem Markov chains (with minor changes) are employed. Ramalingam has shown that the sum-over-all-path solution can be computed by a linear equation system, which is derived from the *exploded control flow graph* (ECFG) introduced by Reps et al. [8].

To compute the one-edge probabilities, the following runtime information is required: (1) frequencies of nodes and (2) frequencies of edges. Then, the one-edge probability $p(u \to v)$ which denotes the probability that edge $u \to v$ is executed once node u has been reached, is given by

$$p(u \to v) = \frac{\mathbf{occurs}(u \to v, \pi_r)}{\mathbf{occurs}(u, \pi_r)}. \qquad (6)$$

The frequency of edge $u \to v$ is divided by the frequency of node u. Consequently, the sum over all probabilities of outgoing edges must be 1,

$$\sum_{v \in \mathbf{succ}(u)} p(u \to v) = 1. \qquad (7)$$

Definition 4 *The sum-over-all path is defined as follows,*

$$E(u) = \sum_{\pi \in \mathbf{Paths}(\mathbf{s}, u)} p(\pi), \qquad (8)$$

$$E(u,d) = \sum_{\substack{\pi \in \mathbf{Paths}(\mathbf{s}, u) \\ \mathbf{state}(\pi)(d) = 1}} p(\pi), \qquad (9)$$

where $p(\pi)$ denotes the probability of path π and is the product of all one-edge probabilities in path π.

$E(u)$ is the expected number of times node u is executed and is expressed as a sum over probabilities of paths from the start node **s** to node u. The definition of $E(u,d)$ differs from $E(u)$ such that paths from **s** to u, where data fact d does not hold true, are excluded from the sum. Combining both numbers give the probability $prob(u,d) = E(u,d)/E(u)$ that d holds true in u. Note that in this probabilistic path model it is assumed that an outgoing edge of a node is independent of incoming edges.

The equation system for solving $E(u)$ and $E(u,d)$ is derived from the exploded control flow graph (*ECFG*). The *ECFG* has $N \times D_\Lambda$ nodes, where D_Λ denotes data fact set D extended by symbol Λ. The symbol Λ can be seen as a data fact, that always holds true when a node is reached. A node in the original control flow graph is represented by $|D_\Lambda|$ nodes in *ECFG*. The edges of the extended control flow graph are computed by the representation relation (see [8]). Note that in the *ECFG* the transfer-functions are inlined such that the DFA problem is reduced to a graph-theoretical problem without having

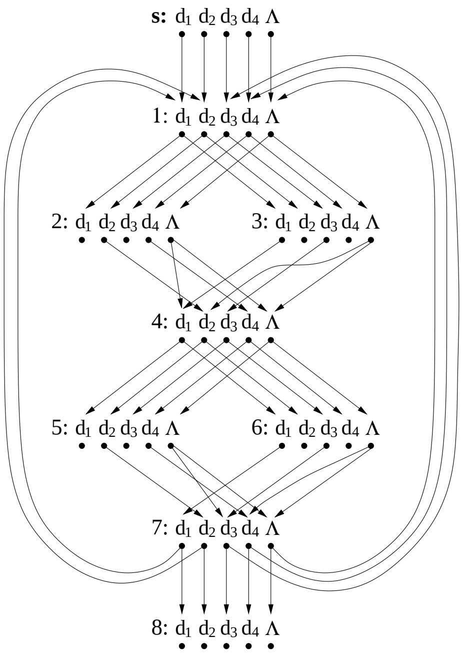

Figure 3: Exploded flow graph of running example.

a monotone function space F. Figure 3 depicts the exploded control flow graph (*ECFG*) of our running example. We can see that edges with identity functions as like as edge $\mathbf{1} \rightarrow \mathbf{2}$ are straight forward connections. Nontrivial transfer functions such as $\mathbf{2} \rightarrow \mathbf{4}$ are represented in *ECFG* such that data facts which are not destroyed are still connected. Those data facts, which are generated, are directly connected to Λ. Data facts, which are destroyed, remain unconnected.

Based on the *ECFG*, a linear equation system in $\mathbb{R}^+$ solves the sum-over-all-paths problem. The linear equation system is given as follows.

Ramalingam's One-Edge Equation System:

$$y(\mathbf{s}, \Lambda) = 1 \tag{10}$$

for all d in D:

$$y(\mathbf{s}, d) = c(d) \tag{11}$$

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

$$y(v, \delta) = \sum_{(u,\delta') \in \mathbf{pred}(v,\delta)} p(u \rightarrow v, \delta) * y(u, \delta') \tag{12}$$

where $\mathbf{pred}(v, \delta)$ denotes the set of predecessors of node (v, δ) in the *ECFG* ($v \in N$, $\delta \in D_\Lambda$).

Equation 10 sets Λ of start node **s** to 1. Equation 11 initializes data fact unknowns of start node **s** with either zero or one depending on whether data facts hold true in c. The last equation 12 describes the relation for unknowns, which does not belong to start node **s**. The left-hand side of the equation is a weighted sum over the unknowns of preceding nodes in *ECFG*. The weights are determined by the one-edge probabilities. Note that if there is no predecessor for a node in *ECFG*, then the sum is zero.

Ramalingam has shown that the solution of unknown $y(u, d)$ is equal to $E(u, d)$ and $y(u, \Lambda)$ is equal to $E(u)$ (for details see [7]).

Table 2 lists the solution 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 of 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 the one-edge approach is 0.299, since the one-edge approach does not recognize that node **6** is always entered via path **[3,4,6]**. This drawback leads us to the two-edge approach which is described in the following section.

$prob(u,\delta)$	d_1	d_2	d_3	d_4	$y(u,\Lambda)$
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: One-Edge Approach

6 Two-Edge Approach

The two-edge approach uses two-edge probabilities and the classical MOP solution (see Def. 2) to achieve better results than the one-edge approach. Without changing the structure of the equation system the weights of the unknowns are modified. First, let us introduce two-edge probability $p(u,v,w)$ which denotes the probability that edge $v \to w$ is executed once node v has been reached via edge $u \to v$.

$$p(u,v,w) = \frac{\mathbf{occurs}([u,v,w],\pi_r)}{\mathbf{occurs}(v,\pi_r)}$$

Note that the sum of the two-edge probabilities $p(u,v,w)$ over the predecessors of v is equal to the one-edge probability $p(v,w)$.

$$p(v,w) = \sum_{u \in \mathbf{pred}(v)} p(u,v,w)$$

In the following we will illustrate equations of unknowns for both approaches and discuss the differences. Note that the equations contain two functions RD and TRANSP. $\mathrm{RD}(n,d_i)$ is 1, if definition d_i may reach node n and 0 otherwise (i.e. RD denotes the solution of the classical reaching definitions problem), and $\mathrm{TRANSP}([u,v,w],d_i)$ is 1, if path $[u,v,w]$ is transparent for d_i (i.e. d_i is not killed along that path) and 0 otherwise. In both examples probabilities of the one-edge approach are overrated compared with the abstract run.

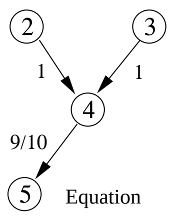

Figure 4: Subgraph of Figure 2 for node 5 annotated with edge probabilities.

First Example. Figure 4 shows a fragment of the control flow graph of our running example. Let us consider data flow equation at node **5** for definition d_3. Since the edge probability of $\mathbf{4} \to \mathbf{5}$ is given by 9/10, we have for the one-edge approach the following equation:

$$y(\mathbf{5},d_3) = 9/10 * y(\mathbf{4},d_3).$$

Note that definition d_3 can only reach node **4** via edge $\mathbf{3} \to \mathbf{4}$, since d_3 is killed on edge $\mathbf{2} \to \mathbf{4}$; furthermore path **[3,4,5]** is never taken. However, these facts are not reflected by the one-edge approach. While the abstract run results in probability 0 for d_3 reaching node **5**, the one-edge approach propagates the value of node **4** with a probability of 9/10 which is overrated and results in a probability for d_3 reaching node **5** of 0.082.

The two-edge equation is given as follows,

$$\begin{aligned}\widetilde{y}(5,d_3) = {} & [p(\mathbf{2},\mathbf{4},\mathbf{5}) * \mathrm{RD}(\mathbf{2},d_3) * \\ & \mathrm{TRANSP}([\mathbf{2{,}4{,}5}],d_3) + \\ & p(\mathbf{3},\mathbf{4},\mathbf{5}) * \mathrm{RD}(\mathbf{3},d_3) * \\ & \mathrm{TRANSP}([\mathbf{3{,}4{,}5}],d_3)] * \widetilde{y}(4,d_3)\end{aligned}$$

The weight of $\widetilde{y}(\mathbf{4},d_3)$ in equation for unknown $\widetilde{y}(\mathbf{5},d_3)$ is a sum of two probabilities. Both probabilities contribute to the sum if and only if RD and $TRANSP$ are one. Since

$$\mathrm{TRANSP}([\mathbf{2{,}4{,}5}],d_3) = 0, \text{ and}$$
$$p(\mathbf{3},\mathbf{4},\mathbf{5}) = 0,$$

both weights are reduced to zero and we obtain

$$\widetilde{y}(\mathbf{5},d_3) = 0 * \widetilde{y}(\mathbf{4},d_3) = 0,$$

which corresponds to the result of our abstract run.

Second Example. Let us consider definition d_1 in **1**. Figure 5 depicts the fragment of the control flow graph annotated with one-edge probabilities. Since the edge probability of $\mathbf{7} \to \mathbf{1}$ is given by 9/10, the equation of the one-edge approach is given by

$$y(\mathbf{1},d_1) = 9/10 * y(\mathbf{7},d_1) + 1 * y(\mathbf{s},d_1).$$

Again, the probability of 9/10 is overrated. The two-edge approach splits the factor for unknown $\widetilde{y}(\mathbf{7},d_1)$ depending on the incoming edges. For $\widetilde{y}(\mathbf{s},d_1)$ there is only one factor due to the fact that there is no preceding edge for start node **s**.

$$\begin{aligned}\widetilde{y}(\mathbf{1},d_1) = {} & [p(\mathbf{5},\mathbf{7},\mathbf{1}) * \mathrm{RD}(\mathbf{5},d_1) * \\ & \mathrm{TRANSP}([\mathbf{5{,}7{,}1}],d_1) + \\ & p(\mathbf{6},\mathbf{7},\mathbf{1}) * \mathrm{RD}(\mathbf{6},d_1) * \\ & \mathrm{TRANSP}([\mathbf{6{,}7{,}1}],d_1)] * \widetilde{y}(\mathbf{7},d_1) + \\ & 1 * \widetilde{y}(\mathbf{s},d_1)\end{aligned}$$

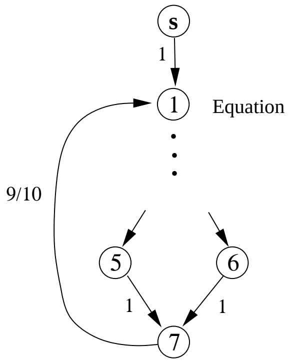

Figure 5: Subgraph of Figure 2 for node **1** annotated with edge probabilities.

Since the transparency condition does not hold on the path **[5,7,1]** and the probability $p(\mathbf{6},\mathbf{7},\mathbf{1})$ is zero,

$$\mathrm{TRANSP}(\mathbf{[5,7,1]}, d_1) = 0, \text{ and}$$
$$p(\mathbf{6},\mathbf{7},\mathbf{1}) = 0$$

we can reduce the equation of $\widetilde{y}(\mathbf{1}, d_1)$ to,

$$\widetilde{y}(\mathbf{1}, d_1) = 1\ \widetilde{y}(\mathbf{s}, d_1) = 0,$$

which perfectly matches the result of the abstract run.

Equation System. In the following an equation system for a general probabilistic data flow problem is given. Although the weightings of the unknowns can differ in the two-edge approach, the structure of the equation system is identical to the one-edge approach.

Two-Edge Equation System:

$$\widetilde{y}(\mathbf{s}, \Lambda) = 1 \tag{13}$$

for all d in D:

$$\widetilde{y}(\mathbf{s}, d) = c(d) \tag{14}$$

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

$$\widetilde{y}(w, \delta) = \sum_{(v,\delta')\in\mathbf{pred}(w,\delta)} p(v \to w, \delta) * \widetilde{y}(v, \delta') \tag{15}$$

where

for all $u \to v$ in E:

$$p(v \to w, \Lambda) = p(v \to w) \tag{16}$$

for all $u \to v$ in E: for all d in D:

$$p(v \to w, d) = \sum_{\substack{u \in \mathbf{pred}(v),\\ S(u)(d) = 1,\\ d \in M([u,v,w])(\{d\})}} p(u, v, w) \tag{17}$$

$S(u)(d)$ denotes the classical MOP data flow solution as given in Definition 2, and $d \in M(u \to v \to w)(\{d\})$ specifies the transparency condition.

Equations 13, 14, and 15 describe the equation system similar to the one-edge approach. In our approach the weightings of the unknowns in Equation 15 are determined by a function over edge $u \to w$ and data fact δ. Equation 16 and 17 define the weighting function $p(v \to w, \delta)$. $p(v \to w, \delta)$ is equal to the one-edge probability iff δ is Λ. For a data fact $d \in D$ we sum up the two-edge probabilities multiplied by the classical solution and the transparency condition. The classical solution ($S(u)(d) = 1$) can only deliver zero or one. If it is zero, the corresponding two-edge probability vanishes. The same is valid for the transparency condition $d \in M([u, v, w])(\{d\})$. Note that if the transparency condition and the classical solution hold for all incoming edges the probability $p(v \to w, \delta)$ is equal to $p(v \to w)$. It is important to stress that for immediate successors of start node $\mathbf{s}$ there does not exist a two-edge probability $p([u \to \mathbf{s} \to w])$ due to the fact that there is no predecessors of the start node. In this case we introduce an artificial node ε, which is an "immediate predecessors" of start node $\mathbf{s}$. The two-edge probability $p(\varepsilon \to \mathbf{s} \to w)$ is given by the one-edge probability $p(\mathbf{s} \to w)$ for an immediate successor w of start node $\mathbf{s}$.

Sequence of Nodes. Our two-edge approach is superior to the one-edge approach for joining and branching nodes. Analysis, performed at statement-level, may lead to sub-graphs where the nodes have exactly one predecessor and successor node. Such sequences of nodes can cause a loss of accuracy. Basic blocks comprise sequences of nodes, which results in better probabilistic data flow solutions.

Discussion. For the two-edge approach the results of our running example are shown in Table 3. To compare the improved results with Ramalingam's approach, Table 4 lists the deviations of Ramalingam's approach vs. abstract run and two-edge approach vs. abstract run for data facts d_1 to d_4. The rows correspond to control flow graph nodes and are grouped in a pair of two. The first line represents deviations of Ramalingam's approach vs. abstract run – the second line deviations of our two-edge approach vs. abstract run. All numbers are given in percent. Zero percent means that there is no deviation. Maximum deviations are ±100%.

$\widetilde{prob}(u,\delta)$	d_1	d_2	d_3	d_4	$\widetilde{y}(u,\Lambda)$
s	0	0	0	0	1
1	0	0.299	0.817	0.332	10
2	0	0.299	0.817	0.332	9
3	0	0.299	0.817	0.332	1
4	0.9	0.369	0.082	0.299	10
5	0.9	0.369	0	0.299	9
6	0.9	0.369	0.081	0	1
7	0.09	0.332	0.908	0.369	10
8	0.09	0	0.908	0.369	1

Table 3: Running example: Two-Edge Approach

Δ%	d_1	d_2	d_3	d_4
s	0.0	0.0	0.0	0.0
	0.0	0.0	0.0	0.0
1	8.2	29.9	-8.3	33.2
	0.0	29.9	-8.3	33.2
2	8.2	29.9	-7.2	33.2
	0.0	29.9	-7.2	33.2
3	8.2	29.9	-18.3	33.2
	0.0	29.9	-18.3	33.2
4	0.8	26.9	-1.8	29.9
	0.0	26.9	-1.8	29.9
5	-9.2	36.9	8.2	29.9
	-10.0	36.9	0.0	29.9
6	90.8	-63.1	-91.8	29.9
	90.0	-63.1	-91.9	0.0
7	9.1	33.2	-9.2	26.9
	9.0	33.2	-9.2	26.9
8	9.1	33.2	-9.2	-63.1
	9.0	0.0	-9.2	-62.1

Table 4: Comparison of One-Edge and Two-Edge Approach

Except for start node **s** Ramalingam's approach deviates in a range between ±0.8 and ±91.9. The perfect matches are highlighted in Table 4. Note that the two-edge approach can perfectly predict 11 (out of 36) data facts with a probability of zero instead of 4 perfectly predicted of the one-edge approach which is a significant improvement in comparison to the one-edge approach and which can be of crucial importance for applications.

7 Related Work

Several approaches have been proposed to improve the efficiency of a program by utilizing profile information.

Ramalingam [7] presents a generic data flow framework which computes the probability that a data flow fact holds or does not hold for finite bi-distributive subset problems. The framework is based on the exploded control flow graph introduced by Reps, Horwitz, Sagiv [8] and on Markov chains. However, execution history is not taken into account. We extend this one-edge approach to a two-edge approach achieving in this way significantly better results. To our best knowledge we are not aware of any approach, which considers execution history.

Proebsting and Fischer [6] introduced a new probabilistic register allocation algorithm for quantifying the costs and benefits of allocating variables to registers so that excellent allocation choices can be made. For their application a probabilistic data flow framework can be used to alleviate the issue of describing and efficiently computing probabilities for heuristics.

Alternatively, Ammons and Larus [1] describe an approach to improve 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 a control flow graph in order to enable heavily executed code to be highly optimized.

Finally, Gupta, Berson, and Fang present algorithms to improve specific optimizations by utilizing profile information, namely partial dead code elimination and partial redundancy elimination. In [3] profile information is used to remove partial dead code along frequently executed paths at the expense of adding additional instructions along infrequently executed ones. Similarly, in [4, 2] profile information is used to remove partial redundant code along heavily executed paths at the expense of introducing additional expression evaluations along less frequently executed paths.

8 Conclusion and Future Work

Based on runtime information the results of data flow problems can be adapted to specific execution environments. Probabilistic data flow frameworks open a wide field for sophisticated optimizations. In this paper we presented a theoretically best probabilistic data flow solution and compared it with the state-of-the-art one-edge approach described in [7]. The differences show that there exists an important potential for improvements. One reason for the deviations between the one-edge approach and the theoretically best solution is the fact that in the one-edge approach it is assumed that the probability of taking a particular branch is independent of the execution history, which is obviously not true. We developed a novel, practicable two-edge approach which takes execution history into account by modifying the weighting of the unknowns based on profile information. The structure of the equation system itself has been retained unchanged from the one-edge approach. The results of our two-edge approach are significantly better than the one-edge approach. However for our running example we still did not succeed to meet the theoretically best solution.

Currently, we try to improve our approach further by changing the structure of the equation system itself in order to model execution history in a better way.

References

[1] G. Ammons and J.R. Larus. Improving dataflow 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.

[2] Rastislav Bodik, Rajiv Gupta, and Mary Lou Soffa. Complete removal of redundant computations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-98)*, volume 33,5, pages 1–14, New York, June 17–19 1998. ACM Press.

[3] 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.

[4] R. Gupta, D. Berson, and J.Z. Fang. Path profile guided partial redundancy elimination using speculation. In *IEEE International Conference on Computer Languages*, pages 230–239, Chicago, Illinois, May 1998.

[5] M.S. Hecht. *Flow Analysis of Computer Programs*. Programming Language Series. North-Holland, 1977.

[6] Todd A. Proebsting and Charles N. Fischer. Probabilistic register allocation. In *Proceedings of the 5th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[7] 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.

[8] 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.