Dataflow Frequency Analysis based on Whole Program Paths

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

Eduard Mehofer
Institute for Software Science
University of Vienna
Vienna, Austria
mehofer@par.univie.ac.at

Abstract

Efficient use of machine resources in high-performance computer systems requires highly optimizing compilers with sophisticated analyses. Static analysis often fails to identify frequently executed portions of a program which are the places where optimizations achieve the greatest benefit.

This paper introduces a novel data flow frequency analysis framework that computes the frequency with which a data flow fact will hold at some program point based on profiling information. Several approaches which approximate the frequencies based on k-edge profiling have been presented. However, no feasible approach for obtaining the accurate solution exists so far. Recently, efficient techniques for recording whole program paths (WPPs) have been developed. Our approach for computing data flow frequencies results in an accurate solution and utilizes WPPs to obtain the solution in reasonable time. In our experiments we show that the execution time of WPP-based frequency analysis is in case of the SPEC benchmark suite only a fraction of the overall compilation time.

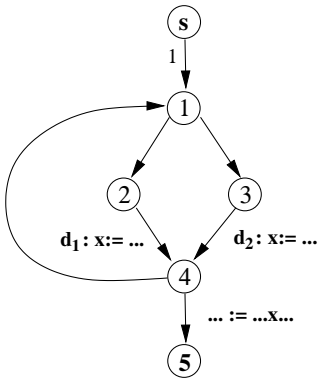
1. Introduction

Program efficiency is a key issue for high performance systems justifying advanced optimization techniques like feedback directed compilation. Runtime information generated by profile runs is provided to the compiler for exploiting the dynamic behavior of a program during the optimization process. In this paper we introduce a novel approach for data flow frequency analysis based on whole program path profiling.

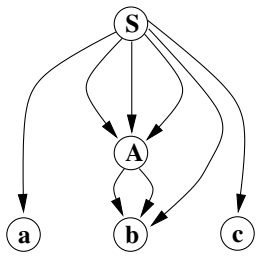
Data flow frequency analysis computes frequencies of data flow facts at some program points based on profiling information. Several approaches have been developed so far which compute an approximation of data flow frequencies [7, 6, 11]. In [11] probabilities are propagated through the control-flow graph without taking execution history into account. In order to get an idea of the precision of this approach, we proposed an *abstract run* [6] which accurately computes data flow frequencies. Experiments with the SPEC benchmark suite show that the deviations between the approach presented in [11] and the accurate results can be considerable and improvements are needed. Since the computational complexity of the abstract run is proportional to the length of a control flow trace, it is not feasible in practice. In [7] we presented a probabilistic data flow framework which achieves more accurate results by taking execution history into account by inspecting intervals of k-edges instead of treating each edge separately. However, the result is still an approximation of the best solution.

So far profiling algorithms were based on edge profiling [1] or profiling of intraprocedural, acyclic paths [2] only. In [2] it has been shown that profiling of acyclic paths can be done efficiently and takes about twice the time of edge profiling. An interprocedural extension of acyclic path profiling [9] results in longer paths, but paths still do not cross loop boundaries. Recently, it has been shown that a complete control flow trace, called *whole program path* (WPP) [5], can be efficiently recorded by employing the SEQUITUR compression algorithm [10]. In this way control flow traces which would be hundreds of MBytes can be compressed to tens of MBytes (cf. [5]). The feasibility of whole program traces opens new possibilities for data flow frequency analysis as well.

In this paper we present a framework for accurately computing data flow frequencies by operating on WPPs [5]. The feedback directed optimization process consists of three steps whereby the first two steps are required to generate WPP files as described in [5]. The data-flow frequency analysis presented in this paper is performed in step three.



(a) Control flow graph



(b) WPP graph for context-free grammar

Figure 1. Example.

1. *Instrumentation:* An instrumented executable is created with additional code for WPP profiling.
2. *Instrumented Execution:* The instrumented code is executed with a representative input data set for generating the WPP file.
3. *Frequency Analysis:* Based on the WPP profile information data-flow frequencies are computed.

In the experimental section we demonstrate that this approach is applicable even for larger applications. Thus for the first time accurate results can be obtained in reasonable time. The computational complexity is proportional to the WPP size instead of the length of the control flow trace.

The paper is organized as follows. Section 2 gives an overview of our approach and illustrates the required steps with an example. Section 3 contains the basic notions used throughout the paper. In Section 4 we develop our approach step by step. Section 5 presents the results of our experiments with the SPEC benchmark suite. Finally, we discuss related work in Section 6 and draw our conclusions in Section 7.

2. Approach

In this section we give an overview of our approach and explain the basic ideas with an example. Consider the control flow graph (CFG) shown in Figure 1(a) with basic blocks mapped to edges rather than nodes. The CFG consists of a branching statement inside a loop with two assignments d_1 and d_2 on edges $\mathbf{2} \rightarrow \mathbf{4}$ and $\mathbf{3} \rightarrow \mathbf{4}$, respectively. For sake of simplicity consider the reaching definitions problem [4].

Moreover, let us consider a specific program run π_r which takes 8 times the left branch **[1,2,4]** and terminates with the right branch **[1,3,4,5]**. Hence, we get the *frequencies* that definition d_1 reaches node **2** seven times while definition d_2 never reaches node **2**, whereas the use of variable **x** at edge $\mathbf{4} \rightarrow \mathbf{5}$ is reached by definition d_2 only.

Recently, approaches have been developed to compress the program path and to make a complete control flow trace feasible [5, 13]. The approach of Larus integrates path profiling [2] and compression algorithm SEQUITUR [10] to reduce the size of a complete program trace. In the remainder of this chapter we will illustrate those techniques with an example and show how the structure of SEQUITUR can be used to compute data flow frequencies efficiently.

Path profiling techniques like [2] capture only acyclic paths which end at loop or procedure boundaries. In our example three acyclic paths are executed: path **a: [s,1,2,4]**, path **b: [1,2,4]**, and path **c: [1,3,4,5]**. Thus program run π_r can be represented by sequence **(a b b b b b b b c)**. SEQUITUR compresses the sequence online and constructs a context-free grammar which generates the sequence of acyclic paths. Note that the non-recursive grammar has only one production per non-terminal and therefore the only word generated by the grammar is the sequence. Larus [5] made slightly modifications to the SEQUITUR algorithm and named it SEQUITUR(1). The SEQUITUR(1) grammar for program run π_r is denoted by the tuple (NT, T, P, S) with NT denoting the set of non-terminals, T the set of terminals, P the set of productions, and S the start symbol. Then we can formally describe the grammar as $NT = \{S, A\}$, $T = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ and P consists of

$$S \rightarrow \mathbf{a}\ A\ A\ A\ \mathbf{b}\ \mathbf{c}$$
$$A \rightarrow \mathbf{b}\ \mathbf{b}$$

Larus defines a WPP as a directed acyclic graph (DAG) representation of the context-free grammar. The WPP for the example is shown in Figure 1(b). Our algorithm uses

the hierarchical structure and works bottom-up starting with the terminal symbols **a**, **b**, and **c**. The terminal symbols represent acyclic paths consisting of CFG nodes and data flow functions which model the data flow effects of basic blocks.

For bi-distributive problems (see Section 3) data flow functions of basic blocks can be composed to one transition function f which describes the data flow effect of the whole path. Consider again our running example and path **b: [1,2,4]**. The transition function of path **b**, denoted by f_b, can be composed of $f_4 \circ f_2 \circ f_1$ with f_1, f_2, f_4 representing the data flow effects of basic blocks **1**, **2**, and **4**, respectively. Transition function f is represented as matrix f^A. The matrix representation is a necessary prerequisite for computing the frequency information of control flow nodes.

For control flow nodes frequency information is determined by *frequency matrices* which have to be computed for all symbols in the SEQUITUR grammar and which is a compressed representation of frequency information for a grammar symbol. The notion of frequency matrices allows the computation of frequencies for non-terminals by inspecting the right-hand side of a production only. E.g. for our running example, the computation of frequency information for non-terminal A is based on the frequency matrix and transition function of symbol **b**. The composition is performed by the matrix calculus which is further explained with the running example below when the algorithm is introduced.

Thus, let us summarize our approach again. Traversing the WPP bottom-up we determine the transition and frequency matrices for terminals **a**, **b**, and **c** first. If all transition and frequency matrices of a grammar symbol in the DAG are available on the outgoing edges, the transition and frequency matrices of the grammar symbol are computed. In our example the transition and frequency matrix of non-terminal A is calculated next followed by non-terminal S. Once start symbol S of the context-free grammar is reached, the data flow frequencies of a control flow node are available.

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 $u \to v \in E$ represent basic blocks of instructions and model the non-deterministic branching structure of G. *Start node* $\mathbf{s}$ and *end node* $\mathbf{e}$ are assumed to be free of incoming and outgoing edges, respectively. A path π of length k is a finite sequence $[u_1, u_2, \ldots, u_k]$ with $k \geq 1$, $u_i \in N$ for $1 \leq i \leq k$ and for all $i \in \{1, \ldots, k-1\}$, $u_i \to u_{i+1} \in E$. A *program run* π_r is a path, which starts with node $\mathbf{s}$ and ends in node $\mathbf{e}$. The set of all prefixes (sub-paths) of a path π starting from the first node of π and ending in node v is denoted by $\mathbf{Prefix}(\pi, v)$.

A monotone data flow analysis problem [4] is a tuple $DFA = (L, \wedge, F, c, G, M)$, where L is a bounded semilattice 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, and $M : E \to F$ is a map from G's edges to data flow functions.

We extend function M to map a path $\pi = [u_1, u_2, \ldots, u_k]$ to a function of F.

$$M(\pi) = \begin{cases} M(u_{k-1} \to u_k) \circ \ldots \\ \qquad \circ M(u_1 \to u_2), & \text{if } \pi \text{ not empty} \\ i, & \text{otherwise} \end{cases} \quad (1)$$

where i is the identity function. Given a path π, we define $\mathbf{state}(\pi)$ to be $M(\pi)(c)$.

For *bi-distributive data flow analysis problems* semilattice L is a powerset 2^D of finite set $D = \{d_1, \ldots, d_n\}$ and the meet operator is the set-union operator. Note that problems with the set-intersection as meet operator are to be solved by its dual problem [12].

Definition 1 *For a bi-distributive problem all transition functions $f \in F$ distribute over union operator ($\cup$) and intersection operator ($\cap$).*

$$\forall X, Y \in 2^D : f(X \cup Y) = f(X) \cup f(Y) \quad (2)$$

$$\forall X, Y \in 2^D : f(X \cap Y) = f(X) \cap f(Y) \quad (3)$$

A transition function $f \in F$ of a bi-distributive problem can be represented by its representation function $f^r : D_\Lambda \to 2^{D_\Lambda}$ where $D_\Lambda = D \cup \{\Lambda\}$ and Λ is an artificial data flow fact which holds in a state of a node if the node is reachable from start node $\mathbf{s}$.

Definition 2 *The representation function f^r of a transition function $f \in F$ is defined as follows:*

$$\begin{array}{rrcl} & f^r(\Lambda) & = & f(\emptyset) \cup \{\Lambda\} \\ \forall d \in D : & f^r(d) & = & f(\{d\}) - f(\emptyset) \end{array} \quad (4)$$

The notions of a representation function f^r are given in [11] and it is an equivalent representation of the representation relation as introduced in [12].

Lemma 1 *Every function $f \in F$ is completely determined by its representation function.*

$$f(X) = [f^r(\Lambda) - \{\Lambda\}] \cup \bigcup_{d \in X} f^r(d) \quad (5)$$

Lemma 2 *For all functions $f \in F$ of a bi-distributive data-flow analysis problem following must hold:*

$$\forall d_i \neq d_j \in D : f^r(d_i) \cap f^r(d_j) = \emptyset \quad (6)$$

Above lemma essentially says that the maps of the representation relation are disjunct.

4. Computation of Dataflow Frequencies

Bi-distributive problems feature certain algebraic properties for representing data flow functions (i.e. transition functions) as matrices and states as vectors. This isomorphic relation to matrix calculus is heavily exploited in order to efficiently compute data flow frequencies for a given program run.

Vector Representation. An alternative representation of element X in 2^D is a vector $\vec{x}$ of numbers in $\{0,1\}$ of length $n+1$ where n is the number of data flow facts in D. More formally, we define a mapping between sub-sets of 2^D and vectors. The function $\varphi : 2^D \rightarrow \{0,1\}^{n+1}$ maps a subset $X \in 2^D$ to vector $\vec{x}$

$$x_i = \begin{cases} 1, & d_i \in X, \text{for all } i, 1 \leq i \leq n \\ 1, & i = n+1 \\ 0, & \text{otherwise} \end{cases} \tag{7}$$

where x_i denotes the ith element of vector $\vec{x}$. An vector element is set to one if the corresponding data flow fact is in set X otherwise zero. Element x_{n+1} represents Λ and is set to one.

Consider all sub-paths(prefixes) π in the program-path starting from start node $\mathbf{s}$ to node v. Recall that **state**(π) denotes the state at v after execution along the subpath π.

Definition 3

$$\vec{y}_{\pi_r}(v) = \sum_{\pi \in \textbf{Prefix}(\pi_r, v)} \varphi(\textbf{state}(\pi)) \tag{8}$$

The equation above computes the frequencies of data flow facts and the execution frequency of node v (i.e. the frequency of symbol Λ). The states of all paths from start node $\mathbf{s}$ to node v are transformed to vectors and then summed up in order to compute the frequencies of data flow facts. A sub-path contributes to a frequency of a data flow fact if the data flow fact is in **state**(π) of the sub-path π. For all data flow facts which hold in the state the corresponding vector elements in $\varphi(\textbf{state}(\pi))$ are set to one and the corresponding data flow frequencies are incremented by one. Note that the definition above is computeable, however, the computation time is polynomial with the length of the program path. For long program paths the computation time can be tremendously slow and better techniques for computing data flow frequencies are required.

Matrix Calculus. A bi-distributive function $f \in F$ can expressed as a $(n+1) \times (n+1)$ matrix of $\{0,1\}$ numbers as already proposed in [11].

Definition 4 *Matrix f^A of a transition function $f \in F$ is a $(n+1) \times (n+1)$ matrix whose elements are given as follows*

$$a_{ij} = \begin{cases} 1, & d_i \in f^r(d_j) \\ 0, & \textit{otherwise} \end{cases} \tag{9}$$

where $i, j \in \{1, \ldots, n+1\}$.

Note that data flow fact d_{n+1} is equal to Λ. The ijth element of the matrix is set to one if element d_i is in the map of the representation function $f^r(d_j)$ – otherwise zero.

Lemma 3 *The matrix calculus is an isomorphic representation of transition functions $f \in F$.*

$$\forall X \in 2^D : \varphi(f(X)) = f^A \cdot \varphi(X) \tag{10}$$

The above lemma essentially says that the following diagram commutes:

$$\begin{array}{ccc} 2^D & \xrightarrow{\varphi} & \{0,1\}^{n+1} \\ {\scriptstyle f}\downarrow & & {\scriptstyle f^A}\downarrow \\ 2^D & \xrightarrow{\varphi} & \{0,1\}^{n+1} \end{array}$$

Example. In Section 1 our example has three bi-distributive functions. First, function $M(\mathbf{2} \rightarrow \mathbf{4}) = f_1$ defines d_1 and kills d_2. Second, function $M(\mathbf{3} \rightarrow \mathbf{4}) = f_2$ defines d_2 and kills d_1. Third, the identity function i for all other mappings.

As described in Section 3, the bi-distributive functions can be fully determined by their representation functions (see Lemma 1) that are a compact description of the functions itself.

d	i	$f_1^r(d)$	$f_2^r(d)$
d_1	$\{d_1\}$	$\emptyset$	$\emptyset$
d_2	$\{d_2\}$	$\emptyset$	$\emptyset$
Λ	$\{\Lambda\}$	$\{\Lambda, d_1\}$	$\{\Lambda, d_2\}$

In the next step we transform the representation functions to matrices as described in Definition 4.

$$i^A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, f_1^A = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}, f_2^A = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

Now, $f_2(\{d_1, d_2\})$ can be transformed to a matrix-vector multiplication as stated in Lemma 3.

$$f_2^A \cdot \varphi(\{d_1, d_2, \Lambda\}) = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \end{pmatrix}$$

The result of the matrix-vector multiplication is a vector $(0,1,1)$ which is the representation of data flow set $\{d_2, \Lambda\}$ since the second and third element of the vector are set to one. It is easy to see that for all possible data flow sets of our example the matrix-vector multiplication is an equivalent representation of the data flow function.

Frequency Matrix. For exploiting the WPP data structure of a compressed program path we introduce the concept of frequency matrix. In Definition 3 we have a finite sum of vectors which can be rewritten to a matrix-vector multiplication based on Lemma 3. This new representation of the data-flow frequencies allows a composition of non-terminals in the sequitur grammar.

Definition 5

$$\mathcal{F}_\pi(v) = \sum_{[u_1,\ldots,u_k=v]\in \textbf{Prefix}(\pi,v)} M([u_1,\ldots,u_k])^A \tag{11}$$

A frequency matrix $\mathcal{F}_\pi(v)$ is defined for a path π and a node v of the control flow graph. It contains the whole data-flow frequency information for node v.

Lemma 4

$$\vec{y}_{\pi_r}(v) = \mathcal{F}_{\pi_r}(v)\varphi(c) \tag{12}$$

The lemma above says that data-flow frequency information is the matrix-vector multiplication of the frequency matrix and the data-flow information associated with the start node s.

Frequency Matrices of SEQUITUR Symbols. Data flow frequencies are computed based on the compressed program path π_r represented as SEQUITUR grammar [10]. The terminals of the grammar correspond to acyclic paths $t : [u_1,\ldots,u_k]$ where $t \in T$ is the terminal and $[u_1,\ldots,u_k]$ is the node sequence of the acyclic path. For every non-terminal symbol $nt \in NT$ there is only one production $nt \to X_1X_2\ldots X_k$ where X_i $(1 \le i \le k)$ is either a non-terminal or a terminal symbol. The SEQUITUR grammar does not contain any recursive nonterminals and therefore it is possible to represent the grammar as a DAG (see Section 1).

For computing the frequency information of a node we exploit the DAG structure of the compressed program path. In the first phase of the algorithm frequency matrices and transition functions of terminal symbols are computed. The transition functions of terminal symbols are used in the second phase within which the algorithm computes frequency matrices for non-terminals by composing frequency matrices and transition functions from the right-hand side of the production. The composition can only be performed when all symbols on the right-hand side are already computed. To meet this criterion the computation of non-terminals must be in a reverse topological order. For obtaining data-flow frequencies of a node the frequency matrix of the start symbol S has to be multiplied by the data flow facts associated with the start node s (cf. Lemma 4).

Lemma 5 *The frequency matrices of symbols in a* SEQUITUR *grammar are given as follows:*

$$\mathcal{F}_t(v) = \begin{cases} M([u_1,\ldots,v])^A, & v \in [u_1,\ldots,u_k] \\ 0_{n+1}, & \textit{otherwise} \end{cases} \tag{13}$$

$$\mathcal{F}_{nt}(v) = \mathcal{F}_{X_1}(v) + \mathcal{F}_{X_2}(v)M(X_1)^A + \ldots + \mathcal{F}_{X_k}(v)M(X_1\ldots X_{k-1})^A \tag{14}$$

The frequency matrix of terminal symbol t is computed by the matrix representation of the transition function starting from node u_1 to node v of the control flow graph. If node v does not occur in the acyclic path of terminal t, then the frequency matrix is a zero matrix, i.e. node v is not executed in path t. Frequency matrix $\mathcal{F}_{nt}$ of non-terminal nt is composed out of the symbols on the right-hand side of the production. The composition is a sum of matrix-multiplications consisting of frequency matrix of symbol X_i and the transition function of the path constituted by $X_1\ldots X_{i-1}$.

Example. In the following we discuss the computation of frequency matrices for terminal symbol **b** and non-terminal symbol A of control flow node **4** in our running example. Before computing non-terminal A we need to compute terminal **b** since terminal **b** occurs on the right-hand side of the production in A.

Terminal **b** represents an acyclic path from node **1** to node **4** via **2**. The mapping function of path **[1,2,4]** is composed by the function product of the identity function and f_1. Therefore, the frequency matrix $\mathcal{F}_b(4)$ is given by the matrix representation of the transition function of f_1.

$$\mathcal{F}_b(4) = M([1,2,4])^A = (f_1 \circ i)^A = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The frequency matrix of nonterminal $A \to$ **b b** is computed by following sum:

$$\mathcal{F}_A(4) = \mathcal{F}_b(4) + \mathcal{F}_b(4)M(b)^A = \begin{pmatrix} 0 & 0 & 2 \\ 0 & 0 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

The nonterminal A describes the path **b b**. Since the subpath **b** is executed twice the frequency matrix of A is the frequency matrix of **b** multiplied by 2.

Algorithm. In Figure 2 the algorithm for computing the data-flow frequency is given. The algorithm computes for all nodes in the control flow graph the data-flow frequencies $\vec{y}(v)$ as shown in the main procedure. The computation is divided in two phases. In the first phase the terminals of the SEQUITUR grammar are computed (shown in procedure

ComputeTerminal). In the second phase non-terminals are composed in reverse topological order by procedure *ComputeNonterminal*. Finally, data-flow frequency of node v is determined by the matrix-vector multiplication of the frequency matrix $\mathcal{F}_S(v)$ and data-flow facts c associated with the start node $\mathbf{s}$.

In the best case the size of the DAG grows in logarithmic order to the length of the program run [5]. The computation of the frequency matrix of a node is proportional to the size of the DAG (and not longer proportional to the length of the program run) since for a node v every grammar symbol is only computed once. We compute the data-flow frequency information for all nodes separately and therefore our algorithm exhibits a computational complexity of $O(|N||\sigma|n^3)$ where $|\sigma|$ is the size of the SEQUITUR grammar, $|N|$ the number of nodes in the control flow graph, and n the number of data flow facts. The additional factor $(n+1)^3$ stems from the matrix-multiplication which can be further reduced to $O(n^2)$ if the sparsity of transition functions is exploited.

1: **procedure** ComputeTerminal($v, t: [u_1, \ldots, u_k]$)
2: **begin**
3: $M(t)^A := M([u_1, \ldots, u_k])^A$;
4: **if** $v \in [u_1, \ldots, u_k]$ **then**
5: $\mathcal{F}_t(v) = M([u_1, \ldots, v])^A$;
6: **else**
7: $\mathcal{F}_t(v) := 0_{n+1}$;
8: **endif**
9: **end**
10: **procedure** ComputeNonterminal($v, nt \rightarrow X_1 X_2 \ldots X_k$)
11: **begin**
12: $\mathcal{F}_{nt}(v) := 0_{n+1}$;
13: $M(nt)^A := I_{n+1}$;
14: **for** $i := 1$ **to** k **do**
15: $\mathcal{F}_{nt}(v) := \mathcal{F}_{nt}(v) + \mathcal{F}_{X_i}(v) M(nt)^A$;
16: $M(nt)^A := M(X_i)^A M(nt)^A$;
17: **endfor**
18: **end**
19: **procedure** Main
20: **begin**
21: **forall** $v \in N$ **do**
22: **forall** $t \in T$ **do**
23: ComputeTerminal(v, t);
24: **endfor**
25: **forall** $nt \in NT$ in reverse topological order **do**
26: ComputeNonterminal(v, nt);
27: **endfor**
28: $\vec{y}_{\pi_r}(v) = \mathcal{F}_S(v)\varphi(c)$;
29: **endfor**
30: **end**

Figure 2. Algorithm

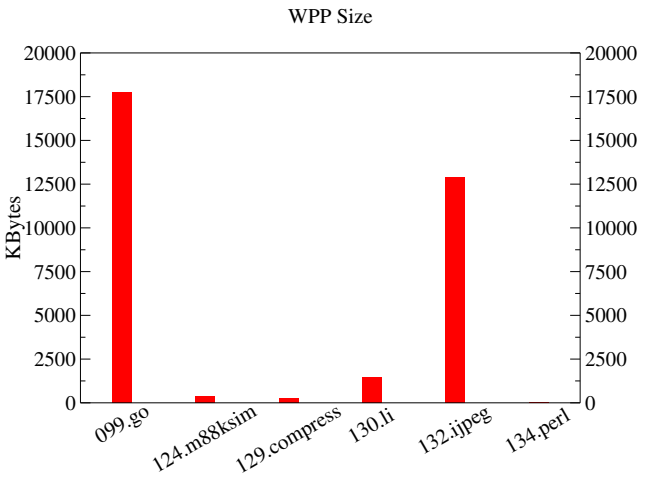
5. Experiments

This section describes the experimental results of our WPP-based data flow frequency analysis framework. As compilation platform GNU gcc has been used. We extended GNU gcc with WPP profiling as described in [5]. Generation of instrumented code is controlled by GNU gcc command line options. Subsequently, the instrumented code is executed and a WPP profile is generated which is an additional input for GNU gcc together with the original source code (feedback compilation loop). The WPP-based data flow frequency analysis framework is controlled by GNU gcc command line options as well. To evaluate our approach we have chosen SPECint95 as benchmark suite and the reaching definitions problem as reference data flow problem. The experiments were conducted on a Sun Ultra Enterprise 450 (UltraSPARC-II 296MHz) with 2560MB RAM.

The size of WPP files is displayed in Figure 3 and ranges from 25 KBytes for benchmark 134.perl up to 17740 KBytes for 099.go. The compile time overhead of our WPP-based frequency analysis is shown in Figure 4. The compile time overhead is at most 33% for benchmark 099.go, 15% for benchmark 132.ijpeg, and between 0.2% and 4% for the remaining benchmarks.

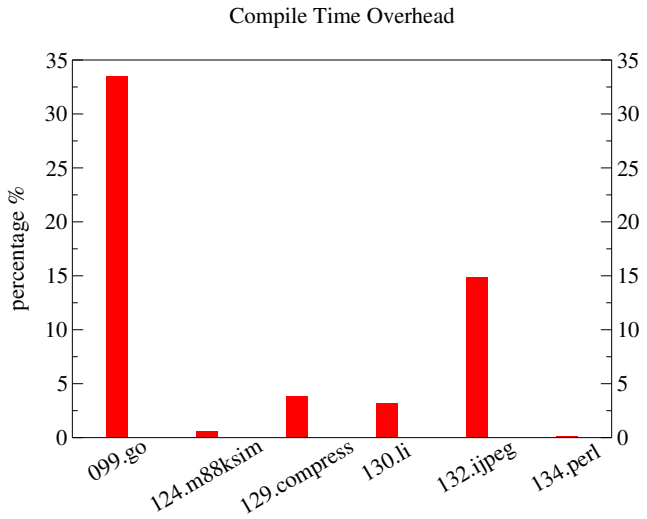
The experiments indicate that the approach is practical even for larger applications. Figures 3 and 4 show that the compile time overhead is roughly proportional to the size of the WPP files.

Our next experiment investigates possibilities for efficient implementation strategies of the frequency module. Note that usually only a small fraction of nodes are actually executed and analyzed. Huge portions of code are never executed and, hence, are not considered in the analysis.



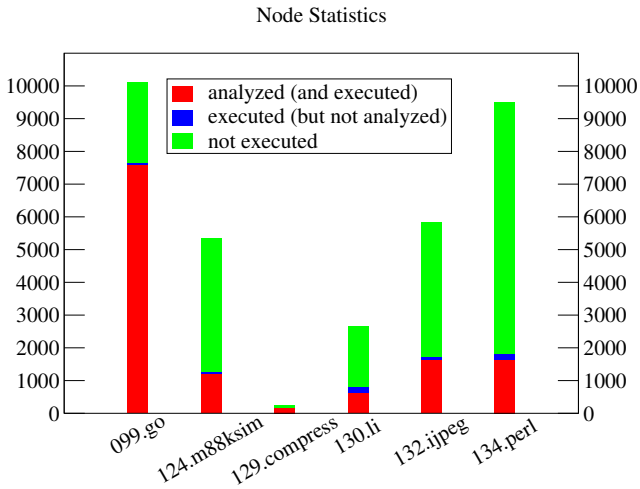
WPP Size of SPECint95 programs in KBytes.

Figure 3. WPP Size.



Compile Time Overhead is the percentage of compile time which is required to compute the WPP based frequencies.

Figure 4. Compile Time Overhead.



Statistics of Nodes in the CFG. A node is either executed and analyzed (definitions exist), or executed and not analyzed (no definitions exist), or not executed.

Figure 5. Node Statistics.

Figure 5 shows the different kind of code portions with a stacked bar chart. We distinguish between nodes which are executed at runtime and contain definitions and, as a consequence, have to be dealt with by the data flow frequency analysis (*executed and analyzed*). Next we consider nodes which are executed at runtime, but do not contain definitions such that those nodes are not dealt with by data flow frequency analysis (*executed and not analyzed*). Finally there are nodes which are not executed and hence not analyzed (*not executed and not analyzed*). First of all note that the fraction of nodes which are executed and not analyzed are neglectable: For all benchmarks the black bar is almost not visible. Except for benchmarks 099.go and 129.compress where the fraction of executed code displayed by dark bars is dominant, the remaining benchmarks show the property that only a small fraction of nodes is actually executed (brighter bars are dominant). Hence, special treatment for non-executed control flow nodes is of paramount importance to obtain efficiency.

The results suggest that accurate data flow frequency analysis is feasible for optimizing compilers. In case of the SPECint95 benchmark suite the overhead of WPP-based frequency analysis is only a fraction of the overall compile time.

6. Related Work

Several approaches for data flow frequency analysis have been proposed. Ramalingam [11] presents a probabilistic data flow analysis (PDFA) framework which computes the probability of a data flow fact to hold at some program point. This approach is based on exploded control flow graphs [12] and Markov-chains. The approach yields an approximate solution which may differ from the accurate values considerably, since execution history is not taken into account. In [7] we improved that approach by utilizing execution history for calculating the probabilities of the data flow facts. For calculating the deviations of the probabilistic approaches from the accurate solution we developed an abstract run [6]. The abstract run accurately calculates the frequencies, however the computational complexity is proportional to the program path length and thus not feasible in practice. In the high performance area optimizations based on data flow frequency analysis are presented in [3, 8].

For recording a complete control flow trace a second approach has been developed called *timestamped whole program path* (TWPP) [13]. While the compression algorithm proposed in [5] makes it difficult to access path traces of a specific function, TWPP utilizes compression techniques which allow an easy access to path traces on function basis. The SEQUITUR algorithm is not employed by TWPP. Recently an algorithm has been proposed which allows to access sub-paths in a WPP as well [14].

7. Conclusion

In this paper we presented a novel data flow frequency analysis framework which is based on WPPs and succeeds in contrast to existing approaches to compute data flow frequencies accurately. The computational complexity of the algorithm is proportional to the WPP size and as a consequence feasible even for larger applications. Thus there

exists a family of data flow frequency analyses of varying accuracy and efficiency according to the needs and requirements of an user. On the one hand highly efficient approximative approaches as presented in [11, 7] can be used at the price of reduced accuracy. On the other hand accurate solutions can be obtained by the approach presented here at the price of increased computational effort. In future research activities we will try to utilize special properties of frequency analysis to reduce the computational complexity further.

References

[1] T. Ball and J. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[2] T. Ball and J. Larus. Efficient path profiling. In *Proc. of the IEEE/ACM International Symposium on Microarchitecture*, pages 46–57, Paris, France, 1996.

[3] S. Benkner, 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.

[4] M. Hecht. *Flow analysis of computer programs*. Programming Language Series. North-Holland, 1977.

[5] J. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation (PLDI)*, pages 259–269, Atlanta, Georgia, May 1999.

[6] 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.

[7] E. Mehofer and B. Scholz. A novel probabilistic data flow framework. In *International Conference on Compiler Construction (CC 2001)*, Lecture Notes in Computer Science (LNCS), Vol. 2027, pages 37 – 51, Genova, Italy, April 2001. Springer.

[8] E. Mehofer and B. Scholz. Probabilistic communication optimizations and parallelization for distributed-memory systems. In *Proc. of PDP 2001*, pages 186–192, Mantova, Italy, February 2001.

[9] D. Melski and T. Reps. Interprocedural path profiling. In *International Conference on Compiler Construction (CC'99)*, Lecture Notes in Computer Science (LNCS), Vol. 1575, pages 47 – 62, Amsterdam, The Netherlands, March 1999. Springer.

[10] C. G. Nevill-Manning and I. H. Witten. Compression and explanation using hierarchical grammars. *The Computer Journal*, 40(2/3), 1997.

[11] 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.

[12] 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.

[13] Y. Zhang and R. Gupta. Timestamped whole program path representation and its applications. In *Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 180–190, Snowbird, Utah, June 2001.

[14] Y. Zhang and R. Gupta. Path matching in compressed control flow traces. In *IEEE Data Compression Conference*, pages 132 – 141, Snowbird, Utah, April 2002.

A. Proofs

Proof 1 *(Proof of Lemma 1) Based on Definition 2 we can deduce following equalities:*

$$f(\{x_i\}) = f^r(x_i) \cup f(\emptyset) \tag{15}$$

$$f(\emptyset) = f^r(\Lambda) - \{\Lambda\} \tag{16}$$

First, we prove that for all $X \in 2^D - \{\emptyset\}$ Lemma 1 holds for all separable functions $f \in F$. Let $X = \{x_1, \ldots, x_k\}$ a non-empty subset.

$$\begin{aligned} f(\{x_1, \ldots, x_k\}) &= f(\{x_1\}) \cup \ldots f(\{x_k\}) \\ &= [f^r(x_1) \cup f(\emptyset)] \cup \ldots \\ &\quad \cup [f^r(x_k) \cup f(\emptyset)] \\ &= [f^r(\Lambda) - \{\Lambda\}] \cup \textstyle\bigcup_{d \in X} f^r(d) \end{aligned} \tag{17}$$

Second, for empty sets the proof is trivial since the second operand of the outer union reduces to $\emptyset$ and Equation 5 reduces to Equation 16. □

Proof 2 *(Proof of Lemma 2) Assume $d_i \neq d_j \in D$ and f is a bi-distributive function. Then, we can rewrite f^r of the left-hand side in Equation 6 by function f.*

$$\begin{aligned} f^r(d_i) \cap f^r(d_i) &= (f(\{d_i\}) - f(\emptyset)) \cap \\ &\qquad (f(\{d_i\}) - f(\emptyset)) \\ &= (f(\{d_i\}) \cap f(\{d_i\})) - f(\emptyset) \end{aligned} \tag{18}$$

Datafacts d_i and d_j are not equal and f distributes over $\cap$. Therefore, we obtain

$$\begin{aligned} (f(\{d_i\}) \cap f(\{d_i\})) - f(\emptyset) &= f(\{d_i\} \cap \{d_j\}) - f(\emptyset) \\ &= f(\emptyset) - f(\emptyset) \\ &= \emptyset \end{aligned} \tag{19}$$

□

Proof 3 *(Proof of Lemma 3) In the following we show that the matrix-vector multiplication of $\vec{y} = f^A \vec{x}$ is an equivalent representation of $Y = f(X)$ where $X, Y \in D$, $f \in F$, $\vec{x} = \varphi(X)$, and $\vec{y} = \varphi(Y)$. The matrix multiplication can be represented as $n+1$ dot-products of y_i where i is the ith element of $\vec{y}$.*

$$
\begin{aligned}
y_i &= \sum_{1\le j\le n+1} a_{ij}\vec{x}(j) \\
&= \sum_{1\le j\le n+1} \left[\begin{cases} 1, & d_i \in f^r(d_j) \\ 0, & \textit{otherwise} \end{cases} \right] \cdot \\
&\quad \left[\begin{cases} 1, & d_j \in X, \textit{for } 1 \le j \le n \\ 1, & j = n+1 \\ 0, & \textit{otherwise} \end{cases} \right] \\
&= \sum_{1\le j\le n+1} \left[\begin{cases} 1, & d_i \in [f^r(\Lambda) \cup f^r(d_j)] \wedge d_j \in X \\ 0, & \textit{otherwise} \end{cases} \right]
\end{aligned}
$$

The last transformation of the equation holds because all maps of the representation function are disjunct (see Lemma 2). By reversing vector $\vec{y}$ into a set, we obtain following formula

$$Y = [f^r(\Lambda) - \{\Lambda\}] \cup \bigcup_{d\in X} f^r(d)$$

which is an equivalent representation of transition function f. □

Proof 4 *(Proof of Lemma 4)*

$$
\begin{aligned}
\vec{y}_{\pi_r}(v) &= \mathcal{F}_{\pi_r}(v)\varphi(c) \\
&= \left(\sum_{\pi\in\textbf{Prefix}(\pi_r,v)} M(\pi)^A \right) \varphi(c) \\
&= \sum_{\pi\in\textbf{Prefix}(\pi_r,v)} \varphi(M(\pi)(c)) \\
&= \sum_{\pi\in\textbf{Prefix}(\pi_r,v)} \varphi(\textbf{state}(\pi))
\end{aligned}
$$

□

Proof 5 *(Proof of Lemma 5) In the first part of the proof we show that the definition of frequency matrix holds for Equation 13 terminal symbol $t : [u_1, \ldots, u_k]$ holds for definition 5.*

$$
\begin{aligned}
\mathcal{F}_t(v) &= \sum_{\pi\in\textbf{Prefix}(\pi_r,v)} M(\pi)^A \\
&= \begin{cases} M([u_1, \ldots, v])^A, & v \in [u_1, \ldots, u_k] \\ 0_{n+1}, & \textit{otherwise} \end{cases}
\end{aligned}
$$

The sum reduces to one addend since the path of terminal t is acyclic and it may contain v only once. If v is not in path $[u_1, \ldots, u_k]$, the frequency sum reduces to zero.

*In the following we show that the definition of frequency matrix holds for non-terminals $nt \to X_1 \ldots X_k$ as well. We split up the sum into paths which are in the paths of the symbol sequences $X_1,\ X_1X_2, \ldots, X_1X_2 \ldots X_k$. By splitting up the sum we can replace the sums by matrices $\mathcal{F}_{X_i}$ multiplied by the transition function $M(X_1)^A \ldots M(x_{i-1})^A$. The sub-paths of sequences $X_1X_2 \ldots X_i$ are given by set difference $\Delta_{nt}(i, v) =$ **Prefix**$(X_1X_2 \ldots X_i, v) -$ **Prefix**$(X_1X_2 \ldots X_{i-1}, v)$.*

$$
\begin{aligned}
\mathcal{F}_{nt}(v) &= \sum_{\pi\in\textbf{Prefix}(X_1X_2\ldots X_k,v)} M(\pi)(v)^A \\
&= \sum_{\pi\in\textbf{Prefix}(X_1,v)} M(\pi)(v)^A + \sum_{\pi\in\Delta_{nt}(2,v)} M(\pi)(v)^A + \\
&\quad \ldots + \sum_{\pi\in\Delta_{nt}(k,v)} M(\pi)(v)^A \\
&= \mathcal{F}_{X_1}(v) + \mathcal{F}_{X_2}(v)M(X_1)^A + \\
&\quad \mathcal{F}_{X_k}(v)M(X_1 \ldots X_{k-1})^A
\end{aligned}
$$

□