Probabilistic Communication Optimizations and Parallelization for Distributed-Memory Systems *†

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

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

Abstract

In high-performance systems execution time is of crucial importance justifying advanced optimization techniques. Traditionally, optimization is based on static program analysis. The quality of program optimizations, however, can be substantially improved by utilizing runtime information. Probabilistic data-flow frameworks compute the probability with what data-flow facts may hold at some program point based on representative profile runs. Advanced optimizations can use this information in order to produce highly efficient code. In this paper we introduce a novel optimization technique in the context of High Performance Fortran (HPF) that is based on probabilistic data-flow information. We consider statically undefined attributes which play an important role for parallelization and compute for those attributes the probabilities to hold some specific value during runtime. For the most probable attribute values highly-optimized, specialized code is generated. In this way significantly better performance results can be achieved. The implementation of our optimization is done in the context of VFC, a source-to-source parallelizing compiler for HPF/F90.

1. Introduction

In high-performance systems execution time is of crucial importance justifying advanced optimization techniques like feedback-oriented compilation. Traditionally optimization is done statically independent of actual execution runs. Runtime information, however, can be exploited to generate highly efficient code. Program runs with representative input data sets are used to gather profile information which is passed to the compiler in a profiling feedback loop. The more expensive compile/execute cycle is acceptable for many developers, if good performance results can be achieved in this way.

Our approach is based on *probabilistic data-flow systems*. Classical data-flow analysis computes whether a data-flow fact may hold or will not hold at some program point, weighting all paths equally, irrespectively whether they are never, heavily, or rarely executed. In contrast probabilistic data-flow analysis takes profile information into account to distinguish between frequently and rarely executed paths. The resulting solution gives us the *probabilities* with what data-flow facts may hold true during execution at some program point. This is achieved by weighting data-flow facts with edge probabilities during propagation through the control-flow graph.

In this paper we present an optimization technique based on probabilistic data-flow information in the context of High Performance Fortran [7], a data-parallel extension of Fortran 90 supporting a globally shared address space. The main task of an HPF compiler on a distributed-memory system is (1) distributing the data across the memories of the processing nodes, (2) arranging parallel execution of statements, and (3) inserting communication whenever some data located on another processing node is required. The optimization developed here affects communication as well as parallelization. Consider an assignment statement of the form $A(i) = A(i + k) + B(i)$ inside a loop with loop variable i. If the distributions of arrays A and B as well as the value of variable k are statically defined, the compiler can generate a highly-optimized, specialized code. However, if arrays A and B are dynamically distributed and the value of variable k cannot be determined statically, compilation gets complicated resulting in potentially inefficient code. Classical data-flow analysis is inappropriate for handling such situations efficiently. Probabilistic data-flow analysis, how-

*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 PDP'2001, Mantova, Italy, February 2001.

ever, gives us exactly the information that is needed for an efficient compilation. Specialized code can be generated for distributions with highest probability resulting in an excellent code size – performance ratio.

The paper is organized as follows. In Section 2 we motivate the idea of our optimization with an illustrating example. Basic notions are introduced in Section 3. In Section 4 an overview of probabilistic optimization frameworks is given. Section 5 presents our optimization algorithm. Implementation and experimental results of our approach for the HPF/F90 compiler *VFC* [2, 3] are reported in Section 6. Related work is surveyed in Section 7 and, finally, we draw our conclusions and discuss future work in Section 8.

2. Motivation

Consider the HPF code shown in Figure 1(a). Arrays A and B are declared to be dynamically distributed with the keyword `dynamic` (line (2)). Whereas for statically distributed arrays the association of a distribution with an array remains invariant for the whole lifetime of the array once it has been established, the distribution of dynamically distributed arrays may be changed. Such changes of distributions are shown in Figure 1(a) in the program portion from line (4) to line (14). Depending on some conditions different distributions are assigned to arrays A and B by executing the HPF directive `redistribute`. In addition, variable k gets some value assigned on lines (16) to (21). Finally, the computational loop is shown from line (23) to line (26).

An highly-optimized compilation of the i-loop (line 24-26) can be done if the distributions of arrays A and B are statically defined. If analysis yields single distributions for both arrays, the same methods as for statically distributed arrays can be applied. Otherwise the compiler has two options (see [15]): A generic code can be generated which works for any distribution, or specialized versions of the loop can be generated, one for each possible distribution. The disadvantage of a generic code is that it is usually less efficient. Further block-cyclic distributions comprise only block and cyclic distributions but not indirect distributions. On the other hand, generating specialized versions of loops may result in exponential code growth as stressed in [15]. In our case arrays A and B can be distributed in a block or cyclic manner which yields four combinations even for that simple example.

As a consequence, precise analysis is of crucial importance in order to prevent code explosion and to achieve nevertheless good performance. Classical data-flow analysis yields for Figure 1 (a) that both arrays can be distributed in a block or cyclic manner. Unfortunately, this information is too coarse-grained and cannot be used to restrict the number of specialized versions. Probabilistic data-flow analysis, however, gives us the handle to distinguish between profitable and non-profitable specialized versions. Assume that a probabilistic framework yields that array A and B are cyclically distributed with a probability of 90% or 60%, respectively. Then it makes sense to handle cyclic/cyclic and cyclic/block distributions for arrays A/B specifically as shown in Figure 1 (b). The generic code of line (9) is applied to all remaining cases which ensures an excellent code size – performance ratio.

However, an efficient compilation of the i-loop does not only depend on the distribution of arrays A and B. Of course, if arrays A and B have the same distribution, the owner-computes-strategy assures that no communication between both arrays has to be performed. Additionally, the value of variable k is of particular interest. Variable k does not only have an impact on communication, but also on parallelization. If the value of k is zero, the loop has a loop independent anti-dependence relation, otherwise it has a loop-carried relation. If k is greater than zero, it is a loop-carried anti-dependence relation, and if k is less than zero, it is a loop-carried flow-dependence relation. Without any information about the value of k, the compiler has to deal with all these situations.

Assume that probabilistic data-flow analysis yields that variable k has value 4 with 80%, otherwise the value is unknown. The value of 4 allows a very efficient compilation, since no flow-dependence relation exists. Moreover, if the program is executed on four processors, no communication is required in case of a cyclic distribution of array A. The generated code is shown in Figure 1 (b). The cases $(cyclic, cyclic, 4)$ and $(cyclic, block, 4)$ denoting the distributions of arrays A and B and the value of variable k are handled specifically. This two cases are selected based on the results of the probabilistic data-flow system and will most likely occur at runtime. Otherwise, the less efficient generic code will be executed, which, however, is only seldom the case limiting in this way substantial adverse performance effects.

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 $x \to y \in E$ 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. The set of all *immediate predecessors* of a node n is $pred(n) = \{\, m \mid (m, n) \in E \,\}$.

A monotonic data-flow analysis problem is a tuple *DFA* $= (L, \wedge, F, c, G, M)$, where L is a bounded semilattice with meet operation $\wedge$, $F \subseteq L \to L$ is a monotone function space associated with L, $c \in L$ are the "data-flow

```
 (1) real A(N),B(N)
 (2) !HPF$ dynamic::A,B
 (3) ...
 (4) !  assign distributions
 (5) if (...)  then
 (6)    !HPF$ redistribute A(cyclic)
 (7)    if (...)  then
 (8)       !HPF$ redistribute B(cyclic)
 (9)    else
(10)       !HPF$ redistribute B(block)
(11)    end if
(12) else
(13)    !HPF$ redistribute A(block)
(14) end if
(15) ...
(16) !  assign some value to k
(17) if (...)
(18)    k = 4
(19) else
(20)    call f(k,l,m)
(21) end if
(22) ...
(23) !  computation
(24) do i=1,N
(25)    A(i) = A(i+k) + B(i)
(26) end do
```

(a) HPF source code

```
 (1) ...
 (2) !  code for loop (24)-(26)
 (3) if (A=cyclic ∧ B=cyclic ∧ k=4) then
 (4)    [code for (cyclic,cyclic,4)]
 (5) else
 (6)    if (A=cyclic ∧ B=block ∧ k=4) then
 (7)       [code for (cyclic,block,4)]
 (8)    else
 (9)       [generic code for (*,*,*)]
(10)    end if
(11) end if
(12) ...
```

(b) Generated message passing code

Figure 1. Illustrating example.

facts" associated with start node s, and $M : E \rightarrow F$ is a map from G's edges to data-flow functions.

For bitvector problems the semi-lattice L is a powerset 2^D of finite set D. Bi-distributive bitvector problems require that all functions in function space F distribute over both set union and set intersection.

A probabilistic data-flow analysis(PDFA) problem is a tuple (*DFA*,*RT*) where *DFA* is a bi-distributive bitvector problem and *RT* represents runtime information gained by profiling. A PDFA framework computes the number of times a node x is expected to execute, denoted by $E(x, \Lambda)$, and the number of times fact d holds true at program point x, denoted by $E(x, d)$.

4. Probabilistic Optimization Frameworks

Probabilistic program optimization consists of probabilistic data-flow analysis followed by an optimizing transformation which takes the probabilistic data-flow results into account. This structure is reflected by probabilistic optimization frameworks.

Figure 2 depicts a probabilistic optimization framework for a compiler with a profiling feedback loop. The profiler is responsible for producing profile information based on the execution environment. This 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 on data-flow equations, probabilistic data-flow analysis computes a probabilistic solution of the data-flow problem. Sophisticated transformations can utilize that solution for generating highly efficient code. For changing execution environments the target code is adapted by the profiling feedback loop.

Ramalingam [12] developed 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 accuracy of his results, we defined in [9] the "best" solution what we can theoretically obtain and showed that the differences can be considerable. 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

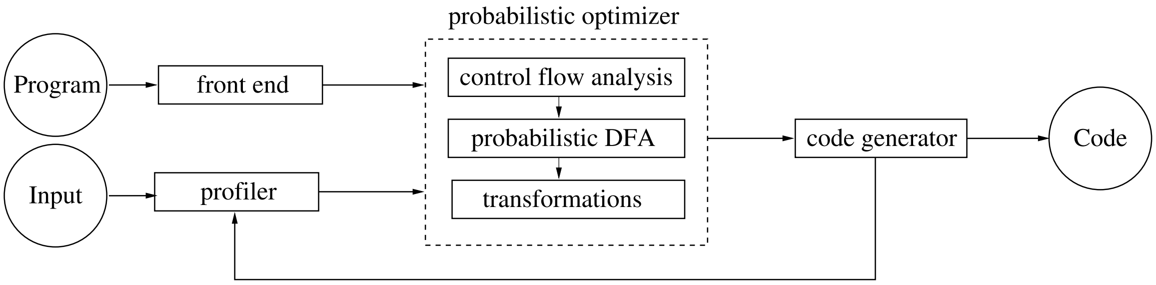

Figure 2. Probabilistic optimization framework.

execution history, which is obviously not the case in reality. While the usage of edge probabilities is indispensable to get an efficient handle on the problem, there exists a potential for improvements by taking execution history into account. In [9] we developed a novel approach which models execution history by altering the data-flow equation system getting in this way significantly better results. As a consequence, optimizations utilizing probabilistic data-flow information are more effective as well.

5. Probabilistic Compilation of Array Accesses

The goal of our algorithm is twofold. First, it is of paramount importance to improve performance of the program by specializing code. Second, the code size should be kept as small as possible. Both goals are diametric and can be controlled by a threshold. By tweaking the threshold we either improve the performance or we reduce code size whereas, still, the most likely data distributions are taken for loop specializations.

Conventional data-flow analysis deduces program information in order to determine what facts may or will not hold during the execution of a program. In contrary probabilistic data-flow analysis frameworks (cf. [9, 12]) take runtime information into account. By considering runtime information PDFA frameworks can compute *probabilities* of data-flow facts. To guide our optimization we compute information with what probability distributions of dynamically distributed arrays reach a loop. Based on this information we can selectively produce code for the most likely occurring distributions. Other distributions are covered by a generic version of the loop.

More formally, a dynamically distributed array $v \in V$, where V is the set of dynamically distributed arrays $\{v_1, \ldots, v_k\}$, has a data distribution $t \in T$, where T is the set of distributions $T = \{t_1, \ldots, t_n\}$. During runtime HPF redistribution directives may change the distribution of arrays.

To represent possible distributions a bitvector $\mathbf{Rd}(y) \in 2^D$ is associated with each flow graph node y, where

$$D = \underbrace{\{d_{t_1}^{(v_1)}, \ldots, d_{t_n}^{(v_1)}\} \cup \ldots \cup \{d_{t_1}^{(v_k)}, \ldots, d_{t_n}^{(v_k)}\}}_{k}$$

and $d_{t_i}^{(v_j)}$ is the data fact that distribution t_i of distributed array v_j reaches node y. Each bit position in $\mathbf{Rd}(y) \in 2^D$ corresponds to one distribution of an array and a bit indicates that either the distribution may reach or does not reach node y.

We assume also that there are two easily computed (local) functions from the set of edges E to the power set of D. The first one we call **RKill** function. The function $\mathbf{RKill}(x \to y)$ is interpreted as the set of distributions that are killed in edge $x \to y$. Informally, the data fact $d_{t_i}^{(v_j)}$ of distributed array v_j with distribution t_i is in the set $\mathbf{RKill}(x \to y)$ iff a different distribution t_h ($h \neq i$) is defined for variable v_j in edge $x \to y$. The second we call **RDefs** function. If distribution t_i of distributed array v_j is changed within node $x \to y$ and the distribution of variable v_j is not subsequently modified within edge $x \to y$, then data fact $d_{t_i}^{(v_j)}$ is element of $\mathbf{RDefs}(x \to y)$.

Let for each node y be $\mathbf{Rd}(y)$ the distributions that comprises the information whether a distribution of a dynamically distributed array may reach node y or not. The fundamental relationship of our reaching distributions problem that enables us to compute the probabilities of $\mathbf{Rd}(y)$ is given in Figure 3.

Rd1. $\mathbf{Rd}(\mathbf{s}) = c$.

Rd2. For each node y, $y \neq \mathbf{s}$:

$$\mathbf{Rd}(y) = \bigcup_{x \in \mathbf{pred}(y)} ([\mathbf{Rd}(x) - \mathbf{RKill}(x \to y)] \cup \mathbf{RDefs}(x \to y)).$$

Figure 3. Equations for reaching distributions

Based on equations given in Figure 3, a PDFA framework [9] computes for all program points $y \in N$ expected frequencies $E(y, d_{t_i}^{(v_j)})$ of data facts and expected frequencies $E(y, \Lambda)$ of nodes. Both frequencies combined yield the information with what probability a distribution t_i of distributed array v_j may reach node y:

$$P(y, d_{t_i}^{(v_j)}) = \begin{cases} \frac{E(y, d_{t_i}^{(v_j)})}{E(y,\Lambda)}, & \text{iff } E(y,\Lambda) \neq 0 \\ 0, & \text{otherwise} \end{cases} .$$

For each loop in the program the most promising distributions are specialized. Let $\mathit{LVars}(l) = \{v_{i_1}, \ldots, v_{i_m}\} \subseteq V$ be distributed arrays that are accessed inside loop l. The distributions of a loop are denoted by a vector which is element of the cross-product of m-times the set of possible distribution T,

$$\vec{t} = < t_1^{(v_{i_1})}, t_2^{(v_{i_2})}, \ldots, t_m^{(v_{i_m})} > \in \underbrace{T \times T \times \ldots T}_{m},$$

where m is the number of accessed arrays in loop l. Under the assumption that the occurrence of distributions are independent of each other, the probability is given as follows,

$$P(< t_1^{(v_{i_1})}, t_2^{(v_{i_2})}, \ldots, t_m^{(v_{i_m})} >) =$$

$$P(x, d_{t_1}^{(v_{i_1})}) \cdot P(x, d_{t_2}^{(v_{i_2})}) \cdot \ldots \cdot P(x, d_{t_m}^{(v_{i_m})}),$$

where x is the loop header of loop l. We define a threshold th in a range between zero and one. Only those distributions are taken, that are greater than the threshold to the power of m – the number of accessed arrays inside loop l. This threshold parameterizes the optimization. A threshold of zero enables all distributions. In contrary a threshold of one allows only a generic loop. More formally, the set of promising distributions is given below

$$\textbf{Dist} = \{\vec{t} \,|\, \vec{t} \in T^m \wedge P(\vec{t}) \geq th^m\} .$$

In Figure 4 an algorithm is outlined in order to calculate the most promising distributions of a loop l. The code of a loop is replaced by several specialized versions. Note that a generic version of a loop is always generated that can handle arbitrary data distributions of the accessed arrays inside the loop. We specialize the loops according to the computed set **Dist** which describes the most likely distributions $\vec{t}$ of the accessed arrays inside the loops. The code transformation template in Figure 5 illustrates how distributions **Dist** $= \{\vec{t}_1, \vec{t}_2, \ldots\}$ are selected and specialized.

Similar to the reaching distribution problem we can employ our probabilistic data-flow analysis framework to compute probabilities of reaching definitions for scalar variables. Here, we consider only those definitions which assign a variable a simple constant and which influence data dependency. The probability formula for reaching distributions and certain scalar variable values is simply extended

function GetPromisingDistributions(l)
    Dist := $\emptyset$;
    m := *LVars*(l);
    foreach $\vec{t}$ in T^m
        if $P(\vec{t}) \geq th^m$ **then**
            Dist := **Dist** $\cup$ $\{ \vec{t} \}$;
        endif
    endfor
    return Dist;
endfunction

Figure 4. Get most promising distributions of a loop l

by multiplying the probabilities of the involved variable values and distributions. Note that the power number of the threshold must be changed accordingly.

if ((v_{i_1} = $t_1^{(1)}$) $\wedge \ldots$ (v_{i_m} = $t_m^{(1)}$)) **then**
    insert code for data distributions $\vec{t}_1$ *of loop l*
else
    if ((v_{i_1} = $t_1^{(2)}$) $\wedge \ldots$ (v_{i_m} = $t_m^{(2)}$)) **then**
        insert code for data distributions $\vec{t}_2$ *of loop l*
        $\vdots$
        else   ! default
            insert generic code version of loop l
end if

Figure 5. Code transformation template

6. Implementation and Experimental Results

The compilation platform for our optimization is VFC [2, 3], a source-to-source parallelizing HPF/F90 compiler. The probabilistic data-flow analysis approach introduced in [9] has been implemented. It differs from other approaches by taking execution history into account. In this way more precise probabilistic results can be achieved (for any details cf. [9]).

Probabilistic analysis is based on exploded control flow graphs introduced in [13] and Markov chains. For solving the linear algebraic equation system we have chosen the elimination framework described in [14] which has been originally developed to solve classical bit-vector problems on control flow graphs. Especially for our extremely sparse equation system the elimination framework is well suited.

Based on the probabilistic data-flow framework we realized probabilistic reaching distributions. In HPF the distribution of an array can be defined in several ways: (1)

with specification directives (ALIGN, DISTRIBUTE), (2) with executable directives (REALIGN, REDISTRIBUTE), or (3) as a result of a procedure call. The different syntactical possibilities to specify distributions as well as the two-level mapping which allows alignments and distributions to be dynamically modified makes analysis complicated. To get a simple handle on this problem we insert distribution/mapping assignments [11, 8] in a pre-pass whenever a distribution is assigned to an array. Thereafter, analysis is applied to those distribution/mapping assignments.

Once probabilistic reaching distributions information has been computed, the algorithm of the subsequent optimizing transformation is rather simple: Whenever the probability of some distribution is beyond a threshold, specialized code is generated for that distribution. Obviously, this transformation will result in a performance improvement if specialized code is more efficient than generic code which is usually the case.

Since the performance improvement can be considerably, above all if data dependency plays a role, we try to figure out the lower bound for performance improvements in our experiments. Consider the following code fragment.

HPF code fragment:

```
if (...)  then
    !HPF$ redistribute A(block,*)
    !HPF$ redistribute B(block,*)
    !HPF$ redistribute C(block,*)
...
DO I=1,N
    DO J = 1,N
        A(I,J) = B(I,J) + C(I,J)
    END DO
END DO
```

If the distribution of arrays A, B, and C cannot be statically determined, VFC generates code just before the loop to send/receive potentially non-local data items. As a consequence, global-to-local index conversion has to be applied to access array elements. However, if the compiler knows that arrays A, B, and C are distributed in a row-wise manner, the communication phase prior to the loop is omitted and the data items are accessed directly without global-to-local index conversion. Figure 6 shows the performance improvement achieved by the specialized code version. For the generic code version a row-wise data distribution has been taken at runtime too. We measured the execution time of the i-loop for different problem sizes, i.e. not the code generated prior to the loop. The performance improvement reaches from 39% to 82% and stems from the indexing overhead introduced by the global-to-local conversion only.

N	Performance improvement
128	1.82
256	1.39
512	1.52
1024	1.59

Figure 6. Performance improvement ratio.

7. Related Work

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

Ramalingam [12] 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 [13] and on Markov chains. However, execution history is not taken into account.

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 [4] 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 [5] 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.

In [9] we developed an abstract run to compute the theoretically best solution and showed that the differences between Ramalingam's history-independent approach [12] and the results of the abstract run can be considerably. In the approach presented in [9] we take execution history into account by modifying the equation system presented in [12].

In [10] we developed a novel procedure cloning algorithm which was motivated by a Fortran 90 problem that can cause serious performance losses. While in Fortran 77 the principal mechanism of passing array arguments is call-by-reference, in Fortran 90 a copy-in/copy-out argument transfer strategy may have to be adopted. An actual argument which is stored in a non-contiguous memory area and which is passed to a contiguous dummy array has to be copied in a contiguous temporary array on entry of a procedure and copied out at procedure exit. In order to avoid a copy-in/copy-out argument transfer, procedure cloning can be applied. However, a main problem of procedure cloning is the possibility of code explosion, which can be in the worst case exponential [6]. We tackle this problem by using the results of a probabilistic data-flow system. Based on the solution of a probabilistic data-flow analysis problem the probability of an actual argument to be contiguous or not is calculated and a ranking of potential procedure signatures is determined at a call-site.

8. Conclusion and Future Work

We presented a novel optimization technique which minimizes the communication effort and supports parallelization by specializing code sections. A key issue in restricting program growth is the distinction between profitable and non-profitable specializations. This distinction is done based on probabilistic data-flow information. Specializations are generated only for program configurations which are beyond some threshold. The parameterizable threshold controls program growth. For a given threshold the most profitable specializations are created resulting in an excellent performance – code size ratio.

Probabilistic data-flow frameworks set forth new directions in the field of optimization and we expect that there exists a broad range of applications. Currently, we are working on the one hand on probabilistic data-flow analysis to improve the probabilistic results further. On the other hand, we are investigating optimizing transformations which are based on probabilistic data-flow information.

References

[1] G. Ammons and J. 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.

[2] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1999.

[3] S. Benkner, S. Andel, R. Blasko, P. Brezany, A. Celic, B. Chapman, M. Egg, T. Fahringer, J. Hulman, Y. Hou, E. Kelc, E. Mehofer, H. Moritsch, M. Paul, K. Sanjari, V. Sipkova, B. Velkov, B. Wender, and H. Zima. *Vienna Fortran Compilation System - Version 2.0 - User's Guide*, October 1995.

[4] R. Gupta, D. Berson, and J. 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.

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

[6] M. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston,TX, April 1991.

[7] High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, January 1997. Available via HPFF home page: http://www.crpc.rice.edu/HPFF.

[8] E. Mehofer. *Optimization of Data Remapping in Data-Parallel Languages*. PhD thesis, Vienna University of Technology, April 1998.

[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 (to appear).

[11] E. Mehofer and H. Zima. Distribution assignment placement. Technical Report TR 96-5, Institute for Software Technology and Parallel Systems, University of Vienna, Austria, December 1996.

[12] 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.

[13] 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.

[14] 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, Mar. 1998.

[15] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley, 1996.