Applications of Fuzzy Array Dataflow Analysis

Denis Barthou, Jean-François Collard and Paul Feautrier

Laboratoire PRiSM, Université de Versailles-S^{t}Quentin
45, avenue des États-Unis, 78035 Versailles, FRANCE

Abstract. Array dataflow analysis can be exact in the general case when it involves only affine constraints on loop counters. This paper first presents an iterative method in the framework of Fuzzy Array Dataflow Analysis and then describes applications of fuzzy analysis on some usual techniques in compilation and parallelization.

1 Introduction

The performances of a compiler rely on its capacity to find in the source program the information it needs to optimize code generation or exhibit parallelism. Detailed information is provided by methods such as *Array Dataflow Analysis* [4, 7] designed to compute, for every array cell value read in a right-hand side expression the very operation which produced it. However few methods can handle non-static programs. For programs using `if`, `while` loops or non-affine array subscripts, no exact information can be hoped for in the general case. The purpose of this paper is twofold: describe an iterative method gathering partial information that can be used in the framework of the *Fuzzy Array Dataflow Analysis* (FADA)[3] and present some applications of this technique such as program checking, parallelization and minimal memory expansion.

2 From Exact to Fuzzy Array Dataflow Analysis

The basic problem of array dataflow analysis is, given an operation $\langle \mathtt{R}, \boldsymbol{y} \rangle$ called the "sink", which is an iteration of a statement R whose iteration domain is $\mathbf{I}(\mathtt{R})$, and an element $\mathtt{a}(\boldsymbol{g}(\boldsymbol{y}))$ of an array a which is read by $\langle \mathtt{R}, \boldsymbol{y} \rangle$ to find the "source" of $\mathtt{a}(\boldsymbol{g}(\boldsymbol{y}))$ in $\langle \mathtt{R}, \boldsymbol{y} \rangle$. The source is an operation $\sigma(\langle \mathtt{R}, \boldsymbol{y} \rangle)$ which writes into $\mathtt{a}(\boldsymbol{g}(\boldsymbol{y}))$, which is executed before $\langle \mathtt{R}, \boldsymbol{y} \rangle$ and such that no operation which executes between $\sigma(\langle \mathtt{R}, \boldsymbol{y} \rangle)$ and $\langle \mathtt{R}, \boldsymbol{y} \rangle$ also writes into $\mathtt{a}(\boldsymbol{g}(\boldsymbol{y}))$. The computation of the source is in two steps: first compute the source for each statement, known as the direct dependence since [2], then combine these sources in the expression of $\sigma(\langle \mathtt{R}, \boldsymbol{y} \rangle)$, as detailed in [4]. Suppose that we are investigating source candidates from a statement S: $\langle \mathtt{S}, \boldsymbol{x} \rangle$, writing into array a at subscripts $\boldsymbol{f}(\boldsymbol{x})$. The candidate source has to verify the following constraints:

- Existence predicate: $\langle \mathtt{S}, \boldsymbol{x} \rangle$ is a valid operation: $\boldsymbol{x} \in \mathbf{I}(\mathtt{S})$.
- Subscript equation: $\langle \mathtt{S}, \boldsymbol{x} \rangle$ and $\langle \mathtt{R}, \boldsymbol{y} \rangle$, access the same array cell: $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{y})$,
- Sequencing condition: $\langle \mathtt{S}, \boldsymbol{x} \rangle$ is executed before $\langle \mathtt{R}, \boldsymbol{y} \rangle$: $\langle \mathtt{S}, \boldsymbol{x} \rangle \prec \langle \mathtt{R}, \boldsymbol{y} \rangle$,

- Environment: sources have to be computed under the hypothesis that $\langle \mathtt{R}, \boldsymbol{y} \rangle$ is a valid operation, i.e. $\boldsymbol{y} \in \mathbf{I}(\mathtt{R})$.

The direct dependence is then given by $\langle \mathtt{S}, \boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y}) \rangle$ where $\boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y}) = \max_{\ll} \{\boldsymbol{x} | \, \boldsymbol{x} \in \mathbf{I}(\mathtt{S}), \boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{g}(\boldsymbol{y}), \langle \mathtt{S}, \boldsymbol{x} \rangle \prec \langle \mathtt{R}, \boldsymbol{y} \rangle\}$ and where $\ll$ represents the lexicographic order.

As soon as the program model includes conditionals, **while** loops or non-affine **do** loop bounds or subscripts, the existence predicate and subscript equation may contain non-linear terms and the exact computation of $\boldsymbol{K}_{\mathtt{S}}$ cannot be achieved in the general case. However, linear relations may be found between constraints in order to compute the smallest set of all the exact sources for any shape of the non-linear constraints verifying these relations. To reach this goal, a solution is to make the source depend on parameters representing the non-linear terms. Pugh and Wonnacott [7] proposed to keep the parametric expression of the non-linear functions in the source when they depend only on $\boldsymbol{y}$. Given a statement S, they may be represented by the set of vectors $D_{\mathtt{S}}(\boldsymbol{y})$ for which they are verified, called *parameter domain*[1]. Note that the dimension $M_{\mathtt{S}}$ of the vectors of $D_{\mathtt{S}}(\boldsymbol{y})$ is lower or equal to the dimension of the iteration vector of S. The expression of $\boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y})$ is $\max L_{\mathtt{S}}(\boldsymbol{y}) \cap \{\boldsymbol{x} | \, \boldsymbol{x}[1..M_{\mathtt{S}}] \in D_{\mathtt{S}}(\boldsymbol{y})\}$ where $L_{\mathtt{S}}$ is the set of vectors verifying all linear constraints. If $\boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y})$ is defined, there exists a vector $\boldsymbol{\beta}_{\mathtt{S}}(\boldsymbol{y})$ called parameter of the maximum such that $\boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y}) = \max L_{\mathtt{S}} \cap \{\boldsymbol{x} | \, \boldsymbol{x}[1..M_{\mathtt{S}}] = \boldsymbol{\beta}_{\mathtt{S}}(\boldsymbol{y})\}$. Hence the source can be computed as a function of the parameters of the maximum of all direct dependences. We have shown that for any property $\mathcal{P}$ that is a relation of inclusion between union or intersection of parameter domains and linearly defined sets, the set of the parameters of the maximum corresponding to all the parameter domains verifying $\mathcal{P}$ is defined by linear constraints and is therefore computable [1]. The aim then is to find some properties on the parameter domains. This can be done by an algorithm based on the abstract symbolic tree of the program [3] and more precise relations may be found by analyzing the expressions of the non-linear constraints.

3 Iterative Analysis

The purpose of the iterative analysis is to find relations between the non-linear constraints coming from different statements so as to compare parameter domains. Given two constraints that are the same function but appear at different places in the program, we can say that they have the same value if the variables they use are the same and have the same values. As a variable has the same value in two operations if it has the same source, the equality of the values of constraints may be proved in some cases by a dataflow analysis. Since this dataflow analysis can be fuzzy, the method can then be applied once more and eventually the fuzziness will be reduced by successive analyses. More formally, given two statements S and S′ writing into array a, we will suppose that only one non-linear constraint appears in the computation of $\boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y})$ and $\boldsymbol{K}_{\mathtt{S}'}(\boldsymbol{y})$. Let c and c' be the non-linear constraints respectively involved in $\boldsymbol{K}_{\mathtt{S}}(\boldsymbol{y})$ and $\boldsymbol{K}_{\mathtt{S}'}(\boldsymbol{y})$, appearing in statements T and T′ .

- Partial equality: the constraints c and c' are the same, use the same variables and a dataflow analysis shows that these variables have the same sources in both operations in a context C that is defined by linear inequalities. The relation is $D_{\mathtt{S}} \cap C = D_{\mathtt{S}'} \cap C$.
- Image of a parameter domain: the constraints c and c' are the same, use the same variables and the sources of the variables of c at operation $\langle \mathtt{T}, \boldsymbol{x} \rangle$ are the same as the sources of the variables of c' at operation $\langle \mathtt{T}', f(\boldsymbol{x}) \rangle$, with f an affine function w.r.t. the iteration vector. The relation is $f(D_{\mathtt{S}}) = D_{\mathtt{S}'}$.

These relations can be generalized to any number of statements and non-linear constraints. The reader is referred to [1] for technical details.

4 Applications

We present thereafter the application of FADA to variable initialization checking and code parallelization.

4.1 Variable Initialization Checking

In a correct program, all variables are initialized before they are used. Verifying this by a dataflow analysis can help to check the correctness of the program or validate some properties on non-linear constraints. When the analysis is fuzzy, the condition for which the source of the value of a does not come from S is a conjunction of affine constraints on $\boldsymbol{y}$ and $\boldsymbol{\beta}_{\mathtt{S}}$. Let $q(\boldsymbol{y})$ and $r(\boldsymbol{y}, \boldsymbol{\beta}_{\mathtt{S}})$ be the predicates forming this condition. When the source comes from S, $\forall \boldsymbol{y} \in \mathbf{I}(\mathtt{R})$ s.t. $q(\boldsymbol{y})$ then $r(\boldsymbol{y}, \boldsymbol{\beta}_{\mathtt{S}}) = \mathit{false}$. According to the definition of the parameter of the maximum, this is equivalent to: $\forall \boldsymbol{y} \in \mathbf{I}(\mathtt{R})$ s.t. $q(\boldsymbol{y}) = \mathit{true}, \exists \boldsymbol{x}$ s.t. $(r(\boldsymbol{y}, \boldsymbol{x}) = \mathit{false}) \wedge (c(\boldsymbol{y}, \boldsymbol{x}) = \mathit{true})$ where c is the non-linear constraint involved. This condition can be generalized to any number of direct dependences and non-linear constraints. Checking the condition can be left to the programmer or submitted to an assertion generator.

4.2 Code Parallelization

There are two basic techniques for extracting parallelism from a dependence graph: one consists in computing a schedule, the other one in computing a placement.

Fuzzy Scheduling We must guarantee that: $\theta(\mathtt{S}, \boldsymbol{x}) + 1 \leq \theta(\mathtt{R}, \boldsymbol{y})$. In the result of the corresponding FADA, $\boldsymbol{x}$ is an affine function ϕ of $\boldsymbol{x}$ and of parameters $\boldsymbol{\beta}_{\mathtt{S}}$ which must satisfy a set of affine predicates $P(\boldsymbol{x})$. We may refine the above inequality into $\boldsymbol{y} \in \mathcal{I}(\mathtt{R}), \boldsymbol{\alpha} \in P(\boldsymbol{y}) \Rightarrow \theta(\mathtt{S}, \phi(\boldsymbol{y}, \boldsymbol{\alpha})) + 1 \leq \theta(\mathtt{R}, \boldsymbol{y})$. Suppose we have expressed the schedule θ as an affine form with unknown coefficients. Since everything is affine, we are in a position to apply Farkas lemma; the result is a set of linear equations in the coefficients of the schedule and new positive unknowns, the *Farkas multipliers*. These equations may be solved as in [5].

Memory Expansion In order to take into account memory based dependences in the above schedule, a solution is to find the minimal memory expansion which is consistent with this schedule. The method presented by Lefebvre [6] can be used in the present case with little or no modification. Indeed, it is obvious that, even in the case of dynamic control structures and non-linear arrays, we may still compute an ordinary dependence graph. In the case of FADA, the shape of the source is exactly the same as in the exact analysis case, hence the same algorithms apply. In some cases, parameters will disappear, for instance, when expansion of a scalar has been deemed unnecessary. When a parameter is actually needed, its value must be recorded when the corresponding control operations are executed. If speculation has been used, this means that a read operation may not be executed before the results of the controlling operations are known. This is a new constraint which has to be taken into account when computing the schedule.

5 Conclusion

Many applications in the compilation and parallelization field take advantage of our technique, with little change in their algorithms. The Fuzzy Array Dataflow Analysis extends the scope of variable initialization checking, code parallelization to some programs with dynamic control structures. Moreover, even a fuzzy result can give enough information for a significant improvement of the output of these techniques. Further developments on the combination of compilation and parallelization methods with fuzzy analysis will be the subject of future work.

References

1. Denis Barthou, Jean-François Collard, and Paul Feautrier. Fuzzy array dataflow analysis. Technical Report 95/33, PRiSM Laboratory, 1995.
2. Thomas Brandes. The importance of direct dependences for automatic parallelization. In *ACM Int. Conf. on Supercomputing*, St Malo, France, July 1988.
3. J.-F. Collard, D. Barthou, and P. Feautrier. Fuzzy array dataflow analysis. In *Proc. of 5th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.
4. Paul Feautrier. Dataflow analysis of scalar and array references. *Int. J. of Parallel Programming*, 20(1):23–53, February 1991.
5. Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.
6. Vincent Lefebvre. Gestion de la mémoire dans les programmes parallèles. In *8eme rencontres francophones du parallélisme*, pages 149–152, May 1996.
7. William Pugh and David Wonnacott. An exact method for analysis of value-based array data dependences. In *Lecture Notes in Computer Science 768: Sixth Annual Workshop on Programming Languages and Compilers*, Portland, OR, August 1993. Springer-Verlag.