The Interplay of Expansion and Scheduling in PAF

Paul Feautrier, Jean-François Collard,
Michel Barreteau, Denis Barthou,
Albert Cohen and Vincent Lefebvre

PR*i*SM, University of Versailles, 45 avenue des États-Unis, 78035 Versailles, France

Abstract

This article presents an overview of our recent research on automatic parallelization of imperative programs. It describes our researches on the extension of the polytope model to general programs, and the current status of the PAF prototype parallelizer. The main contributions are a general algorithm for single-assignment form transformation, an algorithm to compute parallel schedules, and a clean framework to simultaneously address parallelization by scheduling and expansion of data structures. We also recall general results on our array data-flow analysis technique, and advocate for its use at the core of parallelizing compilers. In addition, we discuss important design issues underlying the project, we compare our work with other compilation frameworks, and we propose several research perspectives for enhancing our parallelization scheme.

1 Introduction

This article presents an overview of our recent research on automatic parallelization of imperative programs. It describes our researches on the extension of the polytope model [30] to general programs, and the current status of the PAF (*Parallélisеur Automatique de Fortran*) prototype parallelizer developed in Versailles.

Until recently, the polytope model has been restricted to affine loop nests, i.e. `do` loops with affine bounds and array subscripts. On this class of *static control* programs, our prototype parallelizer proceeds according to the following compilation chain: First, an *array data-flow analysis* (ADA)—a powerful, element-wise reaching definition analysis on arrays—is applied. Thanks to ADA, the program is converted into single-assignment form, thus removing spurious, memory-based dependences. Then, the computations, ordered by the data-flow graph, are scheduled using a logical, integer-valued time. At each time step, a *front* of computations is executed. The generated code then consists of an outer loop on time values that spawns each front in turn.

This paper shows how the same comprehensive parallelization scheme can apply to general programs. First, we recall our results on FADA—an enhanced ADA—in Section 3. Then, in Section 4 we report our results, first published in French, on an algorithm to convert programs into single-assignment form. We also compare these techniques to recent research on *array static single-assignment* (SSA). Notice that, in our minds, *single-assignment* (SA) is a *property* of a program, whereas SSA is *one* framework to achieve this property. SSA provides an intermediate program representation in single-assignment form, allowing further

compilation techniques to be applied; Depending on the application, the program in single-assignment form is actually generated or not. In addition, we believe that the separation between the array data-flow analysis and the single assignment transformation must be clarified, as opposed to the SSA framework. Indeed, this separation is not simply a matter of religion: We show that—as in many compilers—having clear-cut, distinct phases has several benefits.

However, when the behavior of the program cannot be predicted precisely at compile-time, a conversion to single-assignment form may lead to a lot of run-time overhead. We may thus prefer a *partial expansion*, yielding possibly less parallelism but also less run-time overhead. On the other hand, thanks to the experience gained [45] on the memory reduction of affine programs, we notice that some memory cells can be reused at distinct times to store different values (thus breaking the SA property) without losing parallelism. We show how parallelism and expansion of data structures are intermingled, and why recent works of ours happen to be two faces of the same coin, two instances of the same problem. The resolution of the full problem, however, will be left for future work.

Therefore, in addition to providing a comprehensive overview of recent extensions of the polytope model, this paper makes the following contributions:

- We show why, to achieve array single-assignment, the separation of array data-flow analysis and array expansion is important.
- We give a general algorithm to convert programs to single-assignment form.
- We provide formal techniques to schedule general programs, perhaps with speculative execution.
- We introduce a clean framework to simultaneously address parallelization by scheduling and expansion of data structures.

2 Definitions and Overview of the Compilation Chain

Figure 1 shows the different techniques we present in this paper along with their interactions and related articles.

2.1 An Abstract View of Parallelization

In this section, we present a general framework for automatic parallelization and make no limitation on the input programs—neither on control nor on data structures.

Let us consider the set Ω of all operations in the program, and let $\mathcal{W} \subseteq \Omega$ be the set of all writes. Every operation may perform read or write accesses to memory: Let f be the function mapping operations to the memory cells they *write* into.

Reaching Definition Analysis: Compute for each operation u, the set $\sigma(u)$ of definitions reaching u. To stress the point that we deal with run-time instances of statements (i.e. operations), the elements of $\sigma(u)$ are called the *sources* of u. In general, the set $\sigma(u)$ has to be a conservative approximation. Notice that, when the array data-flow analysis is exact (for instance, in the case of affine loop nests) then $\sigma(u)$ is a singleton. (See Section 3).

Conversion to Single-Assignment: (See Section 4.) Define a *new* function f' mapping operations to the memory cells they write into, such that: $\forall u, v \in \mathcal{W}, u \neq v \Rightarrow f'(u) \neq f'(v)$. An overview of the expansion algorithm is:

1. First create a data structure `D` such that there is a bijection (a one-to-one mapping) from Ω to the elements of `D`.
2. In each assignment, the left-hand side is replaced with `D [` $f'(u)$ `]`.
3. Each read (e.g., in right-hand side of assignments) is replaced with
   - `D [` $f'(\sigma(u))$ `]`, if $\sigma(u)$ is a singleton.
   - $\phi(\sigma(u))$ otherwise. As in the SSA framework, the ϕ-function restores the flow of data: It picks at run-time the value computed by the actual source in $\sigma(u)$.

Scheduling: We then schedule all operations, i.e. we find a mapping θ from Ω to logical integer dates such that all operations scheduled at the same date can execute simultaneously, in the same "front": $\mathcal{F}(t) = \{u \in \Omega \mid \theta(u) = t\}$. Let $L(\theta)$ be the *latency* of schedule θ, defined as $L(\theta) = \max_{u \in \Omega} \theta(u) - \min_{u \in \Omega} \theta(u)$. Intuitively, the latency captures the execution time (in logical time units) of the parallelized program. Minimizing the latency is an important issue in computing a "good" schedule, but it is of course not the only one when dealing with distributed memory machines. (See Section 5.)

Code Generation: At last, we scan the operations in Ω according to the (partial) order given by the schedule. If $\min_{u \in \Omega} \theta(u) = 0$, then the abstract parallel code is:

```
do t = 0 to L(θ)
  doall F(t)
  synchronize
end do
```

It is no big surprise that this abstract parallelization scheme will be especially suitable for loop nests and arrays. Indeed, the practical techniques we present in this paper are devoted to such kind of programs (see 2.2). Nevertheless this abstract view can be adapted to more general programs and structures:

- This parallelization scheme can be applied to any control structure: we "only" have to name each operation in Ω in a unique way, then craft an element-wise reaching-definition analysis on Ω, and build a (set of) data structures in bijection with Ω. Consider for instance the case of recursive programs: We have to label each node in the call tree, which is a simple process [13, 32]. Then, we need to extract the flow of data, which is a relation on $\Omega \times \Omega$; In the world of formal languages, we have a *transduction* [13].
- To apply this scheme to any data structure, we need to build new classes of functions f and modify the data-flow analysis accordingly. Once again, the transduction framework provides a simple way to express our problems [32].

In both cases, data structure expansion and scheduling algorithms are likely to be completely new techniques. Notice that a straightforward application of the steps above leads to several interesting issues. Converting to single-assignment requires a lot of memory, which can be

reduced when the schedule shows that two values have completely separate "life times"; In this case, the two memory cells storing each value can be *folded* into one; This single cell will store both values in turn (see Section 6.2). On the other hand, converting to single-assignment may add run-time overhead when some execution support (the ϕ functions) is needed to restore the flow of data; Then, one may prefer a partial expansion of data structures, implying more dependences and possibly less parallelism extraction; This is the purpose of the maximal static expansion (MSE), cf. Section 6.1. In conclusion, we see that scheduling and expansion are tightly intermingled. In the beginning of Section 6 we discuss these issues in more details.

2.2 But a Down-to-Earth Program Model

The concrete parallelization techniques presented in this paper applies to the following program model: The authorized control structures are the sequence, `do` (a.k.a. `for`) loops, `while` loops, and conditionals. Obviously, some `while` loops can be transformed into `do` loops. We will suppose here that these simplifications have been performed, when possible, by a previous phase of the compiler. `goto`s are forbidden because they can be eliminated by well known algorithms [3, 2], at the cost of some code duplication in the rare cases where the control graph is not reducible [1]. Procedure calls are not allowed. The problem of *interprocedural parallelization* has been widely studied [41, 18, 38, 10] in the non-recursive case, and we believe such methods can be readily included in our framework.

The only data structures are scalar types and arrays of scalars. Array subscripts are unrestricted.

Since our framework is based on the polytope model, our main interest is to abstract values and operation domains in terms of affine relations. When dealing with complex programs (non-affine array subscripts, dynamic control flow), we may have to cope with *non-linear constraints*: These are equations or inequalities which depend on variables other than loop counters and symbolic constants, and/or are non-linearly dependent on loop counters and structure parameters. For example, non-linear constraints may come from predicates of `if` or `while` constructs or from array subscripts. Obviously, some non-linear constraints can be removed by replacing some variables by their expression in terms of loop counters and symbolic constants (induction variable detection and forward substitution). However, Section 6.2 in this paper is restricted to *affine loop nests*. They are built of: Assignment statements and `do` loops; Scalars and arrays of arbitrary dimensions; Loop bounds and array subscripts restricted to affine functions in the loop counters and symbolic constants.

2.3 Definitons and Notations

The k-th entry of vector $\vec{x}$ is denoted by $\vec{x}[k]$. The sub-vector built from components k to l is written as: $\vec{x}[k..l]$. If $k > l$, then this vector is by convention the vector of dimension 0, which is written $[]$.

Furthermore, $\ll$ denotes the strict lexicographic order on integral vectors. When clear from the context, "max" denotes "$\max_{\ll}$", i.e. the maximum operator according to the $\ll$ order. An instance of statement S is denoted by $\langle S, \vec{x} \rangle$, where $\vec{x}$, the iteration vector of S, is the vector built from the counters of loops surrounding S—including `while` loops—from outside inward.

Let S be a statement in the program. Because of the "surrounding" control structures, S

executes several times. Our program analysis and transformation techniques should be able to distinguish between these successive instances. Each run-time instance of a statement is called an *operation*. Let $\mathrm{Dom}(S)$ be the set of all the instances of S. Let Ω denotes the set of operation spawned by the program, and $\prec$ is the execution order on it. If $u, v \in \Omega$, then $u \prec v$ (read "u before v") means that operation v does not begin executing until u has terminated. A more precise definition of $\prec$ will be given later.

The special name $\perp$ indicates that the array cell under study is not modified by S. A coherent way of thinking about $\perp$ is to consider it as the name of an operation which is executed once before all other operations of the program: $\forall u \in \Omega, \perp \prec u$. In the following, $\perp$ will be used to denote, also, an undefined vector.

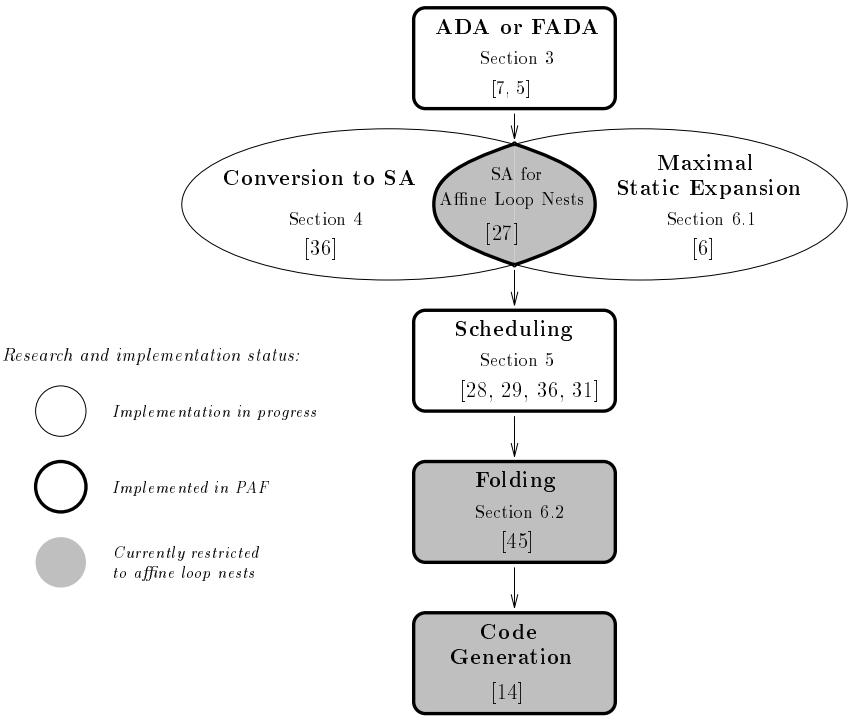

Figure 1: Parallelization framework for unrestricted loop nests over arrays

Figure 1 describes the parallelization process for this program model. The first phase consists in an element-wise array data-flow analysis of the program. Then, we convert the program into single-assignment, i.e., for each assignment S, we create a data structure $\mathtt{D}_S$ isomorphic to $\mathrm{Dom}(S)$. If the program is a nest of loops, then $\mathtt{D}_S$ is an array whose dimension is the number of loops surrounding S. At last, we compute a schedule for every operation, possibly fold the data structures and generate the corresponding parallel code.

3 Fuzzy Array Data-Flow Analysis

This section is an overview of our array data-flow analysis; The interested reader may refer to [7, 5] for details.

3.1 Problem Statement

The basic problem of reaching-definition analysis is to find, for a given read, the write which produced the read value. (The analysis of reaching definition is usually considered as a special case of data-flow analysis, but we make no difference in this paper.) This defining write is called the *source*.

Sources in array data-flow analysis (a.k.a. value-based dependences [52]) are more precise than usual reaching definitions because, on the one hand, they capture data flow array-element-wise and, on the other hand, they describe data flow on an operation-per-operation basis. For instance, a reference to `A[i]` is lexically different from a reference `A[j]`, and two separate analyses are usually done [8]. In ADA, however, both refer to the same value if $i = j$. Of course, only appropriate tools for integer linear programming [26, 50] make this analysis possible.

3.2 Definitions

Formally, let us consider a *textual* reference r and an instance u of this reference. Let c be the memory cell read by r in u. The source is an operation v satisfying three conditions: First, it writes into c; Second, it executes before u; And third, it is such that no operation executing between v and u also writes into c.

The *depth* of a construct is the number of surrounding `do` and `while` loops of this construct. The counter of a loop at depth k is thus component $k+1$ of the iteration vector. Let $\langle R, \vec{y} \rangle$ be an operation performing some read in an element `A[`$g(\vec{y})$`]` of an array `A`, and let $\langle S, \vec{x} \rangle$ be an operation that writes it with subscripts `A[`$g(\vec{y})$`]`. Let N_{SR} be the number of loops surrounding *both* S and R; And let $\lhd$ be the textual order of the program—$S \lhd R$ iff S occurs before R in the program text. The sequential execution order $\prec$ can be formally defined as

$$\langle S, \vec{x} \rangle \prec \langle R, \vec{y} \rangle \iff \bigvee_{p=0}^{N_{SR}} \langle S, \vec{x} \rangle \prec_p \langle R, \vec{y} \rangle,$$

$$\text{where } 0 \leq p < N_{SR} : \langle S, \vec{x} \rangle \prec_p \langle R, \vec{y} \rangle \iff (\vec{x}[1..p] = \vec{y}[1..p]) \wedge (\vec{x}[p+1] < \vec{y}[p+1]),$$

$$\text{and } \langle S, \vec{x} \rangle \prec_{N_{SR}} \langle R, \vec{y} \rangle \iff \vec{x}[1..N_{SR}] = \vec{y}[1..N_{SR}] \wedge S \lhd R.$$

3.3 Bases of our Technique

As soon as our program model includes conditionals, `while` loops, and `do` loops with non-linear bounds, we have to consider a set $\mathrm{Dom}(S))$ defined by non-linear constraints.

The first possibility is to approximate this domain by a set $\widehat{\mathrm{Dom}}(S)$, obtained by ignoring non-linear constraints. We would like to characterize the subset $\widehat{\mathrm{Q}}_S(\langle R, \vec{y} \rangle)$ of $\widehat{\mathrm{Dom}}(S)$ of candidate sources for $\langle R, \vec{y} \rangle$ "coming" from S. Supposing for the moment that array subscripts are still linear, one may obtain the following result:

$$\widehat{\mathrm{Q}}_S(\langle R, \vec{y} \rangle) = \{\vec{x} \mid \vec{x} \in \widehat{\mathrm{Dom}}(S),\ f(\vec{x}) = g(\vec{y}),\ \langle S, \vec{x} \rangle \prec \langle R, \vec{y} \rangle\}.$$

where $\vec{x} \in \widehat{\mathrm{Dom}}(S)$ is the existence predicate, $f(\vec{x}) = g(\vec{y})$ is the conflict predicate (with f and g affine functions w.r.t. $\vec{x}$ and $\vec{y}$, respectively), and $\langle S, \vec{x} \rangle \prec \langle R, \vec{y} \rangle$ is the sequencing predicate stating that the write is executed before the read. In addition we have the implicit information that $\vec{y} \in \mathrm{Dom}(R)$.

However, we cannot say in general that the source is given by the lexicographic maximum of this set: The non-linear part of $\mathrm{Dom}(S)$ has been ignored. One solution is to take the entire set $\widehat{\mathrm{Q}}_S(\langle R, \vec{y}\rangle)$ as an approximation of the source. But then, and with the exception of very special cases, computing the maximum of approximate sources has no meaning, and the best we can do is to use their union as an approximation.

Can we do better than that? Let us consider an example. Notice first that, for expository reasons, only scalars are considered. The method, however, applies to arrays with any subscript.

```
    program conditional
    do x = 1 to N
       if ... then
S1        s = ...
       else
S2        s = ...
       end if
    end do
R   ... = ... s ...
```

Assuming that $N \geq 1$, what is the source of `s` of statement R from the program `conditional`? We may build an approximate candidate set from S_1 and another one from S_2. Since both are approximate, we cannot do anything beside taking their union, and the result is highly inaccurate.

Another possibility is to partition the set of candidates according to the value x of the loop counter. Let us introduce a new boolean function $b(x)$ which represents the outcome of the test at iteration x. The x-th candidate may be written

$$\tau(x) = \textbf{if } b(x) \textbf{ then } \langle S_1, x\rangle \textbf{ else } \langle S_2, x\rangle.$$

We then have to compute the maximum of all these candidates. It is an easy matter to prove that $x < x' \Rightarrow \tau(x) \prec \tau(x')$, so the source is $\tau(N)$. Since we have no idea of the value of $b(N)$, the best we can do is to say that we have a source *set*, or a *fuzzy* source, which is obtained by taking the union of the two arms of the conditional:

$$\sigma([]) = \{\langle S_1, N\rangle, \langle S_2, N\rangle\}. \tag{1}$$

Notice here the precision we have been able to achieve. However, the technique we have used here is not easily generalized. Another way of obtaining the same result is the following. Let $\mathrm{L} = \{x \,|\, 1 \leq x \leq N\}$. Observe that the candidate set from S_1 (resp. S_2) can be written $\{\langle S_1, x\rangle \,|\, x \in \mathrm{D}_{S_1} \cap \mathrm{L}\}$ (resp. $\{\langle S_2, x\rangle \,|\, x \in \mathrm{D}_{S_2} \cap \mathrm{L}\}$) where $\mathrm{D}_{S_1} = \{x \,|\, b(x) = \textbf{true}\}$ and $\mathrm{D}_{S_2} = \{x \,|\, b(x) = \textbf{false}\}$. Obviously,

$$\mathrm{D}_{S_1} \cap \mathrm{D}_{S_2} = \emptyset, \text{ and } \mathrm{D}_{S_1} \cup \mathrm{D}_{S_2} = \{1, \ldots, N\}.$$

We have to compute $\beta = \max\big(\max(\mathrm{D}_{S_1} \cap \mathrm{L}), \max(\mathrm{D}_{S_2} \cap \mathrm{L})\big)$. As D_{S_1} and D_{S_2} partition $\{1, \ldots, N\}$, it is a general property that

$$\beta = \max \mathrm{L} = N.$$

Moreover, we know that β belongs either to D_{S_1} or D_{S_2} which gives again the result (1).

To summarize these observations, our method will be to give new names—*parameters*—to the result of maxima calculations in the presence of non-linear terms. These parameters are not arbitrary. As shown in the example, the sets they belong to—the *parameter domains*—are in relation to each other. More generally, one can find relations on non-linear constraints, either by a simple examination of the syntactic structure of the program, or by more sophisticated techniques. These relations imply relations on the parameters, which are then used to increase the accuracy of the source. In some cases, these relations may be so precise as to reduce the fuzzy source to a singleton, thus giving an exact result. See [7, 5] for a formal definition and handling of these parameters.

The source σ as a *quast*, i.e. a many-level conditional in which:

- Predicates are tests for the positiveness of quasi-affine forms [1] in the loop counters, the symbolic constants, and the new parameters (when non-linear constraints are involved in the maximum computation).
- Leaves are either sets of operations whose iteration vector components are again quasi-affine, or $\perp$.

3.4 Improving Accuracy

To improve the accuracy of our analysis, we find properties on non-affine constraints involved in the description of the dependences and integrate them in the data-flow analysis. As shown in the previous example, these properties imply the properties on the parameter domains introduced in our computation.

Several techniques have been proposed to find properties on the variables of the program or on non-affine functions (See [33, 40, 17, 47, 48, 55] for instance). They use very different formalisms and algorithms, such as abstract interpretation or pattern-matching. However, the relations they find can be written as first order formulae of additive arithmetic on the variables and non-affine functions of the program. This general type of property makes the data-flow analysis algorithm independent of the practical technique involved to find properties.

How the properties are taken into account in the analysis is detailed in [7, 5]. The quality of the approximation is defined w.r.t. the ability of the analysis to integrate (fully or partially) these properties. We have shown that in general, the analysis cannot find the smallest set of possible sources [5]. This is due to decidability reasons, but for some kind of properties such as the properties implied by the structure of the program, we have shown that the best approximation could be obtained.

Besides existing symbolic analyses, we propose in the following section a powerful method that uses the results of a data-flow analysis to improve the accuracy of another data-flow analysis.

3.5 Iterative analysis

The key remark in this section is that two values of the same variable at two different steps of the execution are equal if they have the same *source*. Thanks to this remark, we may go

[1] Quasi-affine forms may include integer division.

one step further in data-flow analyses: The result of a first application of the FADA analysis can in turn help a second application in deriving a more precise result.

To see this, suppose that the same array occurs in the left hand side of two statements, with differing variables as subscripts. These variables are supposed not to depend linearly on induction variables. Data-flow analyses do not make assumptions on the values of variables, and therefore are not able to give the exact source. We may, however, try to prove that whatever the values of these variables, these values are equal. As hinted above, we may apply a data-flow analysis on the subscripting variables themselves, thus iterating the overall process of the analysis. Similarly, two constraints that are the same function but appear at different places in the program have the same value if the variables they use are the same and have the same values.

Therefore, the purpose of iterative analysis is to find properties between the non-linear constraints appearing in the existence predicates and in the conflicting access constraints of different write statements. This method may use the results of data-flow analysis on the variables of the non-linear constraints so as to find more accurate relations. As this data-flow analysis can be fuzzy, the method can then be applied once more and eventually the fuzziness will be reduced by successive analyses. This method finds some relations between parameter domains. Refer to [7, 5] for details.

3.6 An Example

Let us consider the program in Figure 2. This program will be used throughout this paper. If P is an intricate predicate, we cannot make any assumption on its outcome. But since T always executes when j equals N, a value read by $\langle S, i, j\rangle$ with $j > N$, is never defined by an operation $\langle S, i', j'\rangle$ with $j' \leq N$. Figure 2 describes the data-flow relations between instances of S: An arrow from (i', j') to (i, j) means that instance (i', j') defines a value that *may* reach (i, j).

```
 program example
 real A[1..4*N-1]
 do i = 1 to 2*N
   do j = 1 to 2*N
     if P(i,j) then
S      A[i-j+2*N] = ... A[i-j+2*N] ...
     end if
T    if j = N then A[i+N] = ... end if
   end do
 end do
```

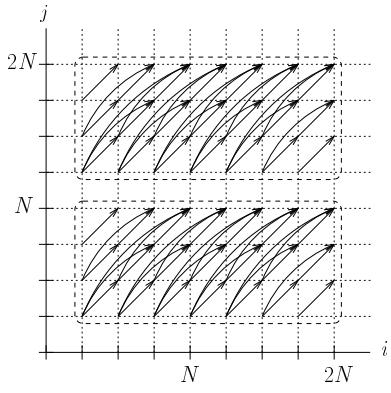

Figure 2: Program example and data-flow graph.

Formally, the source of an instance $\langle S, i, j\rangle$ of statement S can be given by a closed-form

formula parameterized by symbolic constants such as N.

$$\sigma(\langle S,i,j\rangle) = \left| \begin{array}{l} \textbf{if } j \leq N \\ \textbf{then } \{\langle S,i',j'\rangle : 1 \leq i' \leq 2N \wedge 1 \leq j' < j \wedge i'-j' = i-j\} \cup \{\bot\} \\ \textbf{else} \left| \begin{array}{l} \textbf{if } i-j+N < 1 \\ \textbf{then } \{\langle S,i',j'\rangle : 1 \leq i' \leq 2N \wedge N < j' < j \wedge i'-j' = i-j\} \cup \{\bot\} \\ \textbf{else } \left(\begin{array}{l} \{\langle S,i',j'\rangle : 1 \leq i' \leq 2N \wedge N < j' < j \wedge i'-j' = i-j\} \\ \cup \{\langle T, i-j+N, N\rangle\} \end{array} \right) \end{array} \right. \end{array} \right. \quad (2)$$

This result is found automatically by the fuzzy array data-flow analysis.

3.7 Related Work

Work on non-linear constraints in dependence analysis can be divided into two classes. In the first one, the dependence analyzer uses a limited amount of mathematical knowledge to decide whether dependences exist [48, 47]. In the other class of methods, to which this paper belongs, one uses syntactical information only: This may include the structure of the original program, the shape of subscript expressions and the list of variables which occur in them.

The work nearest to our own in that direction is the one by Pugh and Wonnacott [52, 51]. The engine behind Pugh's ADA is the Omega Calculator, a logical Presburger formula simplifier. The main difference between their work and ours is the use of *uninterpreted function symbols* in Presburger formulae to express non-linear constraints and the inability of their analysis to cope with some relations on these constraints. See [7, 5] for a detailed comparison.

From the results of ADA or FADA, one may deduce many useful abstractions, like reaching definitions, upward and downward exposed regions, and so on. In the case of scalars, this information can be obtained more directly by iterative data-flow analysis. These methods can be extended to arrays: An example is the work of Tu [53, 55]. Regions are approximated by coarser objects than polyhedra: For instance, regular sections [11]. When solving data-flow equations, one has to compute unions and complements of regular sections, which are not regular sections in general. Hence, one introduces approximate operations. The information obtained in this way is less precise than the one given by ADA or FADA, but the analysis is faster and is precise enough for solving some problems like array privatization. Another case in point is the work of Duesterwald et al. [21]. In our opinion, the main interest of FADA is that it gives an exhaustive analysis of the source program, and hence is more versatile than other, less precise techniques.

3.8 In Brief

We have given a method to build a conservative approximation of the flow of values in programs whose control flow and array accesses cannot be known at compile-time. Such programs include control flow constructs such as `while`s and `if ... then ... else` constructs, making both control and data flow unpredictable at compile-time . More importantly, the net effect of our handling of `while` loops and tests is to add *equations* to the definition of the candidate set, thus improving the probability of success of fast analysis schemes like [49, 39].

As a concluding remark, note that a $\bot$ in a source set points to a possible programming error. Beyond automatic parallelization, a fuzzy array data-flow analysis may therefore be

a general tool for translators, compilers and program checkers, as array data-flow analysis was.

4 Conversion to Single Assignment Form

We present in this section a general algorithm to perform conversion into single-assignment form. It is based on the results of our fuzzy array data-flow analysis.

4.1 Problem Statement

The reaching definitions given by array data-flow analysis capture the flow of data in programs. However, memory-based dependences remain. To eliminate them, we can use a special intermediate representation, called *single-assignment form*, of the program. This representation can be used internally by the compiler (e.g., to schedule the operations, as in Section 5), or serve to actually generate the final code. The latter case is assumed in this section, since automatic generation of single-assignment code is a general and useful technique.

The single-assignment property states that each memory cell in the program is written at most once. Obviously, this property seldom holds in imperative programming. The benefits of the single-assignment property, however, are well known: Programs having this property are easier to reason about, they have more parallelism, etc. This is the reason why the SSA framework [19] has been used so successfully in the compiler community.

Extension to arrays is therefore an important issue. We first present our algorithm for single-assignment form transformation, then compare with recent work published on array SSA.

4.2 An Algorithm Sketch

Our algorithm to convert programs into array single-assignment form, presented in [36], proceeds as follows. This algorithm follows directly from the general scheme given in Section 2.

Let us first define $\mathsf{Stmt}(\langle S, \vec{x}\rangle) = S$ and $\mathsf{Index}(\langle S, \vec{x}\rangle) = \vec{x}$.

1. For each assignment S whose iteration vector is $\vec{x}$:
   (a) Create a data structure $\mathtt{D}_S$ in one-to-one mapping with $\mathrm{Dom}(S)$. The new function mapping operations to memory cells is $f' = \mathsf{Index}$.
   (b) Let $\vec{x}$ be an index in $\mathrm{Dom}(S)$. The control structures surrounding S, sweeping over $\vec{x}$, are left unchanged.
   (c) Replace the left-hand side with $\mathtt{D}_S\,[\,\vec{x}\,]$.
   (d) Replace each reference r in the right-hand side with $\mathsf{Convert}(r)$, where:
   - If $\sigma(\langle S, \vec{x}\rangle) = \{u\}$, then $\mathsf{Convert}(r) = \mathtt{D}_{\mathsf{Stmt}(u)}\,[\,u\,]$.
   - If $\sigma(\langle S, \vec{x}\rangle) = \{\bot\}$, then $\mathsf{Convert}(r) = r$ (the initial reference expression).
   - If $\sigma(\langle S, \vec{x}\rangle)$ is a non-singleton set, then $\mathsf{Convert}(r) = \phi_r(\sigma(\langle S, \vec{x}\rangle))$.
   - If $\sigma(\langle S, \vec{x}\rangle) =$ **if** p **then** r_1 **else** r_2, then $\mathsf{Convert}(r) = \mathtt{if}\, p\, \mathtt{then}\, \mathsf{Convert}(r_1)\, \mathtt{else}\, \mathsf{Convert}(r_2)$.

ϕ functions are needed to preserve the original data flow when the control flow cannot be predicted at compile-time. They were introduced, for scalars only, in the *static single-assignment* framework: ϕ-functions may be needed to "merge" multiple reaching definitions, i.e. possible data definitions due to several incoming control paths [19].

In our method, a ϕ-function implements the result of fuzzy array data-flow analysis since the returned value is the one produced by the last possible *executed* source, or by the initial element of the initial array if no possible source is executed. The reader is referred to [36] for the algorithms to generate ϕ-functions.

Notice that our technique does not include any "pseudo-assignments" of ϕ-functions to intermediary arrays. The benefit is that placing ϕ-nodes is trivial for us. The drawback is that, when the same ϕ-function is used several times, our scheme generates several instances of the same function. Notice also that, when iteration domains are not bounded at compile-time, the data structures $\mathtt{D}_S$ we allocate are not bounded either; we thus have to allocate them dynamically, or to tile the iteration space.

```
 program example
 real  D_S [1..2*N,1..N]
 real  D_T [1..N,N..N]
 do i = 1 to 2*N
   do j = 1 to 2*N
     if  P(i,j)  then
                          | if j ≤ N
                          | then φ1({⟨S,i',j'⟩ : 1 ≤ i' ≤ 2N ∧ 1 ≤ j' < j ∧ i'-j' = i-j} ∪ {⊥})
S       D_S [i,j] = ...   |      | if i-j+N < 1
                          | else | then φ2({⟨S,i',j'⟩ : 1 ≤ i' ≤ 2N ∧ N < j' < j ∧ i'-j' = i-j} ∪ {⊥})
                          |      | else φ3( {⟨S,i',j'⟩ : 1 ≤ i' ≤ 2N ∧ N < j' < j ∧ i'-j' = i-j},
                          |      |          {⟨T,i-j+N,N⟩} )
     end if
T    if j = N then  D_T [i,N] = ... end if
   end do
 end do
```

Figure 3: Conversion to single-assignment on the running example.

The algorithm above, applied to the running example, yields the program in Figure 3.

4.3 Related Work

Our method to convert programs with arrays to single-assignment form [27, 36], uses the result of an array data-flow analysis, and then transforms the code. Recent work by Knobe and Sarkar [44], on the other hand, does not make this separation. So, is this an important issue?

We believe the answer is yes. Cutting the conversion to SA into two phases has several benefits:

- A ϕ-function is needed only when the (compile-time) analysis fails to find the unique (run-time) source. In [44], ϕ-functions are inserted even for affine programs, whereas

well-known analyses such as [27] can easily prove that ϕ-functions are not necessary. This has practical applications for compiler writers: When a useless ϕ-function has been inserted in a program, we know what to blame: The data-flow analysis.

- Element-wise analysis allows optimizations that are not related to single-assignment. For instance, the element-wise dead code elimination cited in [44] is simply given by eliminating the set of operations $\{u : \nexists v, u \in \sigma(v)\}$. Several other applications (program checking, etc.) do not need the single-assignment property either.

- The last reason is that the program may not be converted to single-assignment, perhaps because ϕ-functions are considered too expensive. A maximal static extension [6] may be preferred instead, on top of the data-flow analysis (cf Figure 1).

To sum things up, array data-flow analysis is not opposed to array SSA. Actually, they *cannot* be opposed: The former is a *compile-time analysis*, whereas the latter is a framework for single-assignment form transformation. We believe that the first phase of an array SSA should be an array data-flow analysis.

Notice finally that the work by Knobe and Sarkar is also, in a sense, more advanced, because they give a better description of how the actual code generation is performed in the general case. In our mind, they prefer a robust "safety net" for general programs to more precise but also more restricted methods. How to use an array data-flow analysis to improve array SSA is definitely an interesting future work.

5 Scheduling

Dependence or data-flow analyses derive a graph where nodes are operations and edges are constraints on the execution order. The problem is now to traverse the graph in a partial order; This order is the execution order for the parallel program. The more partial the order, the higher the parallelism. Obviously, this partial order cannot be expressed as the list of relation pairs: One needs an expression of the partial order that does not grow with problem size, i.e., a closed form. Additional constraints on the expression of partial orders are: Have a high expressive power; Be easily found and manipulated; Allow optimized code generation. A suitable solution is to use scheduling functions.

5.1 Problem Statement

When constructing parallel programs, one may use a *schedule*, i.e. a function θ from the set Ω of all operations to the set $\mathbb{N}$ of positive integers. the schedule must satisfy the following constraint:

$$u \,\delta\, v \Rightarrow \theta(u) < \theta(v).$$

where δ is some dependence relation on operations. If no expansion on data structures is applied, then δ is the classical order given by true, anti- and output dependences. If the program is converted to single-assignment form, then δ is the data flow (i.e. the relation given by function σ). If a more general—intermediate—expansion scheme is preferred (cf. Section 6), then δ is given by the memory-based dependences that are not removed by the expansion.

On the other hand, since θ is integer valued, the constraint above is equivalent to:

$$u \, \delta \, v \Rightarrow \theta(u) + 1 \leq \theta(v). \tag{3}$$

This system of functional inequalities, called *causality constraints*, must be solved for the unknown function θ. As it is often true for system of inequalities, it may have many different solutions. One can minimize various objective functions, as e.g., the number of synchronization points or the latency.

5.2 A Scheduling Algorithm

Let us introduce $\vec{x}$, the vector of all variables in the problem: $\vec{x}$ is obtained by concatenating u, v, and the vector of size parameters in the problem. It so happens that, in the context of this work, $u \, \delta \, v$ is the disjunction of conjunctions of linear inequalities. In other words, the set $\{u, v \mid u \, \delta \, v\}$ is a union of convex polyhedra. This is true for ordinary dependences where $u \, \delta \, v$ is the usual dependence relation of [50]. It is also true when the dependence relation is approximate in various ways (dependence cones, direction vectors, dependence level, etc.). Lastly, it is true in the case of our fuzzy data-flow analysis, since the source sets are linearly described.

Therefore, the constraints in the antecedent of (3) are affine; Let us denote them by $C_i(\vec{x}) \geq 0, \ 1 \leq i \leq M$. Similarly, let $\psi(\vec{x}) \geq 0$ be the consequent $\theta(v) - \theta(u) - 1 \geq 0$ in (3). Then, we can apply the following lemma:

Lemma 1 (Affine Form of Farkas' Lemma) *An affine function $\psi(\vec{x})$ is non-negative on a polyhedron $\{\vec{x} | \, C_i(\vec{x}) \geq 0, 1 \leq i \leq M\}$ if there exists a set of non-negative integers $\lambda_0, \ldots, \lambda_M$ (the Farkas coefficients) such that:*

$$\psi(\vec{x}) = \lambda_0 + \sum_{i=1}^{M} \lambda_i C_i(\vec{x}) \tag{4}$$

This relation is valid for all values of $\vec{x}$. Hence, one can equate the constant term and the coefficient of each variable in each side of the identity, to get a set of linear equations where the unknowns are the coefficients of the schedules and the Farkas multipliers, λ_i. Since the latter are constrained to be positive, the system must be solved by linear programming.

Unfortunately, some loop nests do not have "simple" affine schedules. The solution in this case is to use a *multidimensional affine schedule* [29], whose domain is $\mathbb{N}^d, d > 1$ ordered according to the lexicographic order. Such a schedule can have as low a degree of parallelism as necessary, and can even represent sequential programs. The selection of a multidimensional schedule can be automated by using algorithms from [28, 29]. It can be proved that any loop nest in an imperative program has a multidimensional schedule. Notice that multidimensional schedules are particularly useful in the case of dynamic control programs, since we have in that case to overestimate the dependences and hence to underestimate the degree of parallelism.

Using Lemma 1, we can compute an affine parallel schedule for our running example:

$$\begin{aligned} \theta(S, i, j) &= j - 1, \text{ if } j \leq N \\ \theta(S, i, j) &= j - N, \text{ if } j > N \\ \theta(T, i, N) &= 0 \end{aligned}$$

The resulting latency is N.

We have shown in Section 4 how to eliminate data dependences. Constraints on the schedule have been alleviated in this way. When the control flow is not precisely known at compile-time, these constraints may still be too stringent to extract a satisfying degree of parallelism. Hence, ignoring some *control dependences* is a complementary issue; This is the purpose of speculative execution.

5.3 Speculative Execution

Before exploring speculation, let us define *control dependences*: There is a control dependence between u and v if the *very execution* of v depends on the result of u. u is called the *governing operation*. Such a dependence is denoted by δ^c. Notice that such control dependences are part of the program data flow. Therefore, when the result of the data-flow analysis is approximate, the control flow may not be precisely known at compile-time.

First, why does speculation have a chance to give more parallelism? This is simply because every dependence, including control dependences, is an obstacle for parallel execution, and that removing one of them may give a better schedule. Suppose that operation u guards the execution of v, i.e. u is in control dependence with v. If the control dependence is satisfied when computing schedule θ, we have $\theta(u) < \theta(v)$. If this control dependence is discarded, then the above constraint may not hold. In this case, v may be executed before being sure that it was executed in the original program. The execution of v is speculative.

A speculative operation must not modify the program memory. Its result must be held in temporary storage until its governing operations are executed. If the outcome is **true**, then the result is *committed* by moving it to permanent storage. In the opposite case, the result is discarded from temporary storage.

Our objective now is to give rules for the correctness of a speculative program.

Intuitively, not taking a control dependence into account may yield, first, incorrect results and, second, a non-terminating behavior. The first problem arises when one of the arguments is the result of a speculative operation. This result cannot be used until it has been *committed*, i.e. until its governing operations have terminated. This can be enforced by the introduction of *compensating dependences*. For more details, the reader is referred to [31].

Let us focus on the second problem. When there is a single and outermost `while` loop, a necessary and sufficient condition for correctness is that fronts must be finite [15]. Our aim here is to give a termination criterion for more general speculative programs. Together with the preceding theorem, it will entail the total correctness of the object program.

Firstly, we must have some idea about the structure of the target code. As it is the case for synchronous parallel programs, we will suppose that the outer loop is sequential, its function being the enumeration of the successive values of time. Contrary to the static control case, this is a `while` loop[2]. The operations which are scheduled at time t constitute the *front* at time t. The definition of a front is more complicated than in the static control case. An operation which is scheduled at time t is to be executed provided that all its governing operations which have been executed before t have been evaluated to **true**.

[2] In the case of multidimensional schedules, one may have several loops of this kind, the number of loops being the dimension of the schedule.

The result of operation u will be written $\rho(u)$. With this notation, the speculative front $\mathcal{F}(t)$ of operation scheduled at time t is:

$$\mathcal{F}(t) = \bigcup_S \mathcal{F}_S(t), \tag{5}$$

$$\text{where } \mathcal{F}_S(t) = \{u \in \widehat{\text{Dom}}(S) | \, \theta(u) = t, \forall v : (\theta(v) < t \wedge v \, \delta^c \, u \Rightarrow \rho(v) = \textbf{true})\}. \tag{6}$$

To write the speculative program, we need the following auxiliary functions:

$$\text{first} = \min\{t | \, \mathcal{F}(t) \neq \emptyset\} \tag{7}$$

$$\text{next}(\tau) = \min\{t | \, \mathcal{F}(t) \neq \emptyset \wedge t > \tau\} \tag{8}$$

with the provision that these functions take the undefined value $\perp$ if the set over which the minimum is computed is empty. "first" gives the first clock tick at which there is work to be done, and "next(τ)" is the first clock tick after τ at which there is work to be done.

With these notations, the abstract speculative program is:

```
do t = first while t ≠ ⊥
  doall F(t)
  t = next(t)
end do
```

Let us introduce:

$$\mathcal{B}(t) = \bigcup_{\tau < t} \mathcal{F}(\tau).$$

To be correct, the abstract program above has to satisfy several conditions:

1. An uncommitted value has to be held in temporary storage until the results of all governing predicates are known. The size of the temporary storage has to be finite.

2. Each operation has to be executed in finite *real* time. Since the execution time of a parallel program is bounded from below by the number of its operations divided by the number of processors, this means that the total number of operations before any logical instant t has to be finite:

$$\text{Card } \mathcal{B}(t) < \infty \tag{9}$$

3. Lastly, the speculative program must terminate whenever the original program does.

The set of uncommitted results is a subset of the set of all results. Hence, condition 2 implies point 1 above. In [31], we prove that:

Proposition 1 *A program with speculative schedule terminates provided that the original program terminates and the schedule is such that all sets $\mathcal{B}(t)$ are finite.*

The complete sufficient condition for the total correctness of the target speculative program is given described in [31].

Obviously, the more speculative the schedule, the larger the memory needed to store intermediate uncommitted result. For simple cases, this issue is addressed in [15]. The interplay of expansion and speculative scheduling, in the general case, is left for future work.

In the rest of this paper, schedules are supposed to be non-speculative.

5.4 Related Work

The scheduling problem has been widely studied since the first Kennedy and Allen algorithm. It is not the purpose of this paper to compare these algorithms, the interested reader may refer to [20, 29, 25] for details.

6 The Interplay of Array Expansion and Scheduling

The previous section presented a method to express the parallelism in a dependence graph. It computes a schedule satisfying the partial order given by the dependence graph. Moreover, memory expansion aims at deleting dependences due to memory reuse. Applying memory expansion and then scheduling seems a natural way to extract additional parallelism.

However, several problems arise from a straightforward application of our compilation process:

1. ϕ-functions are a significant overhead when data structures hold several scalar elements (like arrays) and when their elements are spread across processors. Therefore, we may want to expand the initial data structures, but not convert into SA, so as to avoid ϕ-functions. More formally, when the source $\sigma(u)$ of some operation u is a not a singleton, then we may want to build an expansion of the program such that the new function f' mapping operations to memory cells verifies:

$$\forall v, w \in \sigma(u), f'(v) = f'(w).$$

   The benefit is that we always know which memory cell stores the value needed by u: This cell always is $f'(v) = f'(w) = f'(\sigma(u))$. The drawback is that some parallelism may be lost, i.e. we may have to choose a schedule with higher latency.

2. It may happen that a single memory cell can store two successive, distinct values (without consequences on the program schedule). Let us consider a program with exactly four operations w_1, r_1, w_2 and r_2, such that $\sigma(r_1) = \{w_1\}$ and $\sigma(r_2) = \{w_2\}$, scheduled at time steps 0, 1, 2 and 3, respectively. Then, assigning two memory cells (one for w_1, r_1 and one for w_2, r_2) is a waste, since a single cell can store the first value (defined by w_1 and last read by r_1), and *then* store the second value used by w_2 and r_2. In other words, when looking at the schedule, the two memory cells given by SA can be *folded* into one.

One simple solution would be to first expand, then to schedule, and finally to fold, but this is not very elegant. How to solve both problems simultaneously is left for future work. In Section 6.1, we solve the first problem and assume scheduling is applied later. In Section 6.2, we restrict ourselves to affine loop nests and show how to take benefit from scheduling information to decrease memory usage.

6.1 Maximal Static Expansion

In this section, we show how to automatically find the static expansion which expands all data structures as much as possible without introducing ϕ-functions. Maximal static expansion may be considered as a trade-off between parallelism and memory usage. We present an algebraic framework to derive the maximal static expansion; The input of this framework is

the (perhaps approximate) output of a data-flow analysis. Our framework is valid for any imperative program, without restriction—the only restrictions being those of your favorite data-flow analysis. In the sequel, we use the analysis presented in Section 3.

6.1.1 Problem Definition

Let us consider two operations u and v belonging to the same set of possible sources of some read r. If they both write in the same memory cell $f(u) = f(v)$ and if we assign two distinct memory cells to u and v ($f'(u) \neq f'(v)$), then a ϕ-function is needed to restore the data flow since we do not know which of the two cells has the value needed by r. Static expansion enforces $f'(u) = f'(v)$.

Definition 1 (Static expansion) *A static expansion is a mapping f' from operations to memory cells such that*

$$\forall u, v \in \mathcal{W} \ : \ \left(\exists r, \ u \in \sigma(r) \wedge v \in \sigma(r) \wedge f(u) = f(v)\right) \implies f'(u) = f'(v).$$

Notice that the condition $f(u) = f(v)$ is necessary in presence of non-affine array subscripts: A set of sources $\sigma(r)$ may hold operations writing in different memory cells.

Notice also that, according to this definition, even a constant function on $\mathcal{W}$ is a static expansion. Because we are interested in maximizing the memory expansion, the range of a "good" static expansion should be as large as possible. Such an expansion would be constant on sets as small as possible:

Definition 2 (Maximal static expansion) *A static expansion f' is maximal on the set of operations $\mathcal{W}$ iff for any static expansion f''*

$$\forall u, v \in \mathcal{W} \ : \ f'(u) = f'(v) \implies f''(u) = f''(v).$$

Intuitively, if f' is maximal, then any f'' cannot do better and maps two writes to the same memory cell when f' does.

We need to characterize the sets of operations on which a maximal static expansion f' is constant. By definition, these sets are exactly the equivalence classes of relation $\{u, v \in \mathcal{W} \ : \ f'(u) = f'(v)\}$. The set of theses classes is denoted by $\mathcal{W}/_{f'}$. The number of memory cells after maximal static expansion is thus equal to the cardinal of $\mathcal{W}/_{f'}$.

However, this hardly gives us an expansion scheme, because this result does not tell us how much each individual memory cell should be expanded. The purpose of Section 6.1.2 is to give a similar result for each memory cell c used in the original program. This result appears in Theorem 2. This theorem is then used to give a practical expansion scheme.

6.1.2 Expansion Scheme

Let us define the relation:

$$u \mathcal{R} v \iff \exists r, \ u \in \sigma(r) \wedge v \in \sigma(r). \tag{10}$$

Relation $\mathcal{R}$ is obviously symmetric. Definition 1 requires that a static expansion f' verifies $f'(u) = f'(v)$ when $f(u) = f(v)$ and $u \mathcal{R} v$. Given u, v and w in $\mathcal{W}$, if $f(u) = f(v) = f(w)$, $u \mathcal{R} v$ and $v \mathcal{R} w$ then $f'(u) = f'(v) = f'(w)$. Therefore, given $u \in \mathcal{W}$, f' is constant on the set of all $v \in \mathcal{W}$ such that $f(u) = f(v)$ and $u \mathcal{R}^* v$, $\mathcal{R}^*$ being the transitive closure of $\mathcal{R}$. We may give an equivalent definition of a static expansion:

Definition 3 *A static expansion is a mapping f' from operations to memory cells such that*

$$\forall u, v \in \mathcal{W} \ : \ u\mathcal{R}^* v \wedge f(u) = f(v) \implies f'(u) = f'(v).$$

From this characterization, we proved in [6] that f' is a maximal static expansion iff

$$\forall u, v \in \mathcal{W} \ : \ u\mathcal{R}^* v \wedge f(u) = f(v) \iff f'(u) = f'(v). \tag{11}$$

For a memory cell c, the set of operations writing into c is $\mathcal{W}(c) = \{u \in \mathcal{W} \ : \ f(u) = c\}$. Given a memory cell c, a static expansion f' is maximal iff

$$\forall u, v \in \mathcal{W}(c) \ : \ f'(u) = f'(v) \iff u\mathcal{R}^* v.$$

Therefore, classes of $\mathcal{R}^*$ in $\mathcal{W}(c)$ are exactly the sets we are looking for:

Proposition 2 *The sets on which a maximal static expansion f' is constant are given by*

$$\mathcal{W}/_{f'} = \bigcup_{c \in M} \mathcal{W}(c)/_{\mathcal{R}^*} \tag{12}$$

Therefore, the expansion factor of each individual memory cell c is $\mathrm{Card}(\mathcal{W}(c)/_{\mathcal{R}^*})$.

To generate the transformed code, one first has to remember which equivalence class an operation belongs to. This can be done by picking a representative in each class: Let φ be the function mapping an operation u to the (unique) representative of its equivalence class. The second stage consists in labeling every element (i.e. equivalence class) of $\mathcal{W}(c)/_{\mathcal{R}^*}$. In terms of representatives, each element of $\varphi(\mathcal{W}(c))$ should be labeled. Such a labeling scheme is obviously arbitrary, but all programs transformed using our method are equivalent up to a permutation of these labels. We denote by $\nu(u)$ the label we choose for the elements of $\varphi(\mathcal{W}(f(u)))$. Then, $f' = (f, \nu)$.

Our expansion scheme depends on the transitive closure calculator and on the part calculating $\mathcal{W}(c)$. We would like to stress the fact that the expansion produced is static and maximal with respect to the results yielded by these parts, whatever their accuracy:

- The transitive closure may be too complicated to give an exact result. Therefore, it may be over-approximated. The expansion factor of a memory cell c is then lower than $\mathrm{Card}(\mathcal{W}(c)/_{\mathcal{R}^*})$. However, the expansion remains static and is maximal with respect to the transitive closure given to the algorithm.
- The sets $\mathcal{W}(c)$ may not be known precisely at compile-time. (E.g. when dealing with non-affine array subscripts.) One may use some approximation $\widetilde{\mathcal{W}(c)}$ such that $\mathcal{W}(c) \subseteq \widetilde{\mathcal{W}(c)}$, then label all classes of $\mathcal{W}/_{\mathcal{R}^*}$, which in turn gives labels to the classes of all $\widetilde{\mathcal{W}(c)}/_{\mathcal{R}^*}$. The drawback of this method is that some memory cells not used during program execution may be allocated.

Notice that the definitions given in Section 6.1.1 and the expansion scheme are valid for any imperative program. The only restrictions and limitations are those of the data-flow analysis and of the algorithm to compute transitive closures.

6.1.3 Example

Let us apply maximal static expansion to a practical example: The expansion of array `A` in the program of Figure 2.

Because sources in (2) are non-singleton sets, a straightforward conversion of this program into single-assignment form would require run-time computation of the memory location read by S. However, we notice that the iteration domain of S may be split into *separate subsets* by grouping together operations involved in the same data flow. These subsets build a partition of the iteration domain. Each subset may have its own memory cell, a cell that will not be written nor read by operations outside the subset. The partition is given in Figure 4.a.

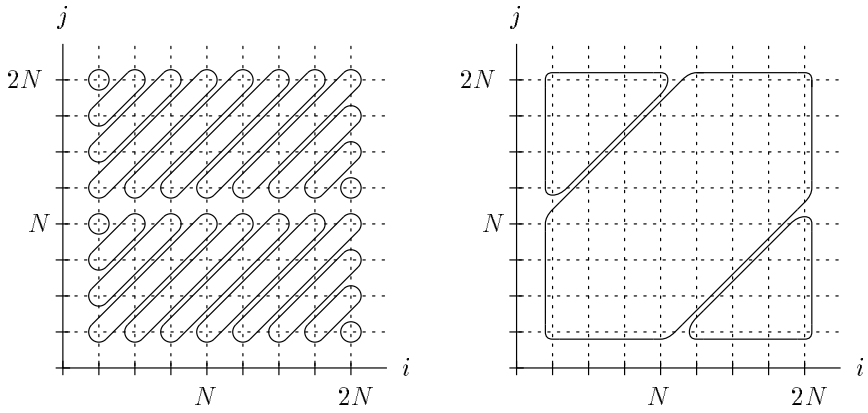

Figure 4.a. *Figure 4.b.*

Figure 4: Partition of the iteration domain ($N = 4$).

Using this property, we can duplicate only those elements of `A` that are used twice. These are all the array elements `A[`c`]`, $1 + N \leq c \leq 3N - 1$. They are accessed by operations in the large central set in Figure 4.b. Let us label with 1 the subsets in the lower half of this area, and with 2 the subsets in the top half. We add one dimension to array `A`, subscripted with 1 and 2 in statements S_2 and S_3 in Figure 5, respectively. Elements `A[`c`]`, $1 \leq c \leq N$ are accessed by operations in the upper left triangle in Figure 4.b and have only one subset each (one subset in the corresponding diagonal in Figure 4.a), which we label with 1. The same labeling holds for sets corresponding to operations in the lower right triangle.

The maximal static expansion is shown in Figure 5. Notice that this program has the same degree of parallelism as the corresponding single-assignment program, without the run-time overhead of ϕ-functions.

6.2 Optimizing Memory Usage

It is clear that converting a program to single-assignment has a very high memory cost. In the particular case of *affine loop nests*, this section presents a technique to reduce memory expansion without hampering the performances of the parallelization process.

To each operation u is associated two sets of memory cells: $\mathcal{R}(u)$, the set of read cells, and $\mathcal{M}(u)$ the set of modified cells. Bernstein's conditions distinguish three kinds of dependences between u and v, where $u \prec v$. If $\mathcal{M}(u) \cap \mathcal{R}(v) \neq \emptyset$, there is a *true dependence*. If $\mathcal{R}(u) \cap \mathcal{M}(v) \neq \emptyset$, there is an *anti-dependence*. If $\mathcal{M}(u) \cap \mathcal{M}(v) \neq \emptyset$, there is an *output dependence*.

```
 real A[1..4*N-1,1..2]
 do i = 1 to 2*N
   do j = 1 to 2*N
     {expansion of statement S}
     if -2*N+1 <= i-j <= -N then
       if P(i,j) then
S1       A[i-j+2*N,1] = ... A[i-j+2*N,1] ...
       end if
     elsif -N+1 <= i-j <= N-1 then
       if j <= N then
         if P(i,j) then
S2         A[i-j+2*N,1] = ... A[i-j+2*N,1] ...
         end if
       else
         if P(i,j) then
S3         A[i-j+2*N,2] = ... A[i-j+2*N,2] ...
         end if
       end if
     else
       if P(i,j) then
S4       A[i-j+2*N,1] = ... A[i-j+2*N,1] ...
       end if
     end if
     {expansion of statement T}
T    if j = N then A[i+N,2] = ... end if
   end do
 end do
```

Figure 5: Maximal static expansion for the example.

In terms of memory dependences, array data-flow analysis (see Section 3) computes the data-flow graph which is a subset of true dependences. All other dependences are due to memory reuse: They are artificial in the sense that conversion to single-assignment form eliminates them. Our aim is now to define a method for partial data expansion which *builds a parallel program with memory reuse.*

We suppose that a schedule function θ has been deduced from the data-flow graph. The constraint for our partial expansion being that this schedule should remain valid in presence of output and anti-dependences.

6.2.1 Memory Reuse in Parallelized Programs

Let $\mathcal{V}(v)$ be the value produced by an operation v. Let $\mathcal{C}(v)$ be the memory cell in which $\mathcal{V}(v)$ is stored. Let $\mathcal{U}(v)$ be the set which gathers all operations u such that there is a data-flow from v to u. $\mathcal{U}(v)$ is usually called the *utilization set* of v. Let $L_\theta(v)$ be the execution time of the last read of $\mathcal{V}(v)$ in the parallel program scheduled by θ.

Consider a memory cell $\mathcal{C}(v)$ during execution of a parallel program in single-assignment form. The memory cell *stays empty* until the execution of v at $\theta(v)$. The operations of $\mathcal{U}(v)$ read $\mathcal{V}(v)$ until $L_\theta(v)$. The memory cell is not read anymore after $L_\theta(v)$, nevertheless $\mathcal{V}(v)$

is still in $\mathcal{C}(v)$ until the end of the execution of the parallel program.

It is clear that the *utility span of* $\mathcal{V}(v)$ is between $\theta(v)$ and $L_\theta(v)$. Before and after this utility span, $\mathcal{C}(v)$ can store other values without changing the data-flow from v to operations in $\mathcal{U}(v)$: Output dependences between v and some other operations can be reintroduced in the parallel code. Such output dependences are called *neutral dependences.*

6.2.2 Neutral Dependences

Definition 4 *An output dependence is neutral for a schedule function θ iff it does not change the data-flow in the parallel program built with the help of θ.*

One can precisely give the characteristics of a *neutral output dependence* between two operations v and w in the parallel program:

- *v must be executed before w* $(\theta(v) < \theta(w))$.
- *There is an access conflict* $(\mathcal{C}(v) = \mathcal{C}(w))$.
- *The utility spans are separate* $(L_\theta(v) < \theta(w))$.

To decide if an output dependence is neutral in a parallel program, one must have a precise estimation of the utility span of each value $\mathcal{V}(v)$. Then this estimation can help us to reconstruct the data space of the program by adjusting data size to utility spans. The final purpose is to build a parallel program with neutral output dependences. Refer to [45] for details.

6.2.3 Utility Span

Consider the utility span of a value $\mathcal{V}(v)$: $[\theta(v), L_\theta(v)]$. The lower bound of this time subsegment is directly given by θ. The problem is to compute the upper bound $L_\theta(v)$. Determining this time uses techniques from data-flow analysis. The main difference is that the lexicographic maximum computation is not on the sequential execution order $\prec$, but on the execution order given by the schedule function θ.

$L_\theta(v)$ is the execution time of the last operations of $\mathcal{U}(v)$ according to θ:

$$L_\theta(v) = \max\{\theta(u) \,|\, u \in \mathcal{U}(v)\} \tag{13}$$

6.2.4 Data Reconstruction

The first step is a *partial array and scalar expansion process* that decides the shape of each statement left-hand side (LHS). The second step consists in a *partial renaming process* and decides which statements can share the same data structure.

Partial Array Expansion We want to build a structure $\texttt{lhs}_S$ which is specifically associated to the statement S. It will give the shape (number of dimensions and size of each dimension) and the index function which constitute the data in the left-hand side of S in the restructured program. The aim is that $\texttt{lhs}_S$ provides memory reuse, i.e. *neutral* output dependences between some operations instances of S. Moreover the elaboration of $\texttt{lhs}_S$ must be independent from the original data structure in the LHS of S.

A neutral output dependence cannot kill a value $\mathcal{V}(S,\vec{x})$ during its utility span. To respect this rule for any instance of S, one must take into account the maximum duration that the utility span of $\mathcal{V}(S,\vec{x})$ can have in the parallel program. For an operation $\langle S,\vec{x}\rangle$ this duration is obtained by subtracting the lower bound of its utility span from the upper bound. Let $d(S) = \max_{\vec{x}\in \mathrm{Dom}(S)}(L_\theta(S,\vec{x}) - \theta(S,\vec{x}))$ be the maximum utility span duration of all instances of S.

To protect every instance of S during its utility span, $\mathtt{lhs}_S$ must be built in such a way that no value $\mathcal{V}(S,\vec{x})$ can be killed between $\theta(S,\vec{x})$ and $\theta(S,\vec{x}) + d(S)$.
The algorithm that builds the data structure $\mathtt{lhs}_S$ can be summarized like this:

1. One starts with a scalar $\mathtt{lhs}_S$. The elaboration of $\mathtt{lhs}_S$ is iterative, the number of iterations is equal to N_S (number of loops surrounding S, cf. Section 3). Each iteration is called *partial expansion of S at depth p* where p is the depth of the loop considered.

2. A *partial expansion of S at depth p* consists in computing the *expansion degree E_S^p of S at depth p* (it gives the number of elements of a new dimension that one adds to $\mathtt{lhs}_S$).

The problem is now to compute E_S^p. The partial expansion of S at depth p avoids non-neutral output dependences between two operations $\langle S,\vec{x}\rangle$ and $\langle S,\vec{x'}\rangle$ if $\langle S,\vec{x}\rangle \prec_p \langle S,\vec{x'}\rangle$. For an operation $\langle S,\vec{x}\rangle$, we build the set of candidates gathering all the operations $\langle S,\vec{x'}\rangle$ which cannot share the same memory cell as $\langle S,\vec{x}\rangle$ because their utility spans are *not separate*. Let $\langle S,\vec{x}_c\rangle$ be the last operation in this set. No output dependence can appear between operations $\langle S,\vec{x}\rangle$ and $\langle S,\vec{x'}\rangle$ with $\langle S,\vec{x}\rangle \prec_p \langle S,\vec{x'}\rangle \preceq_p \langle S,\vec{x}_c\rangle$. From this follows the inequalities on the iteration vectors: $\vec{x}[p+1] < \vec{x'}[p+1] \leq \vec{x}_c[p+1]$.

If $\mathtt{lhs}_S$ is expanded at depth p with $\vec{x}_c[p+1] - \vec{x}[p+1] + 1$ elements, we are sure that no non-neutral output dependence at depth p can appear concerning $\langle S,\vec{x}\rangle$. But it must be verified for every instance of S, hence the expansion degree E_S^p is the maximum of $\vec{x}_c[p+1] - \vec{x}[p+1] + 1$ for all $\vec{x} \in \mathrm{Dom}(S)$.

Partial Renaming For two statements S and T, partial expansion builds two structures $\mathtt{lhs}_S$ and $\mathtt{lhs}_T$ which can have different shapes. If at the end of the renaming process S and T are authorized to share the same array, this one would have to be the rectangular hull of $\mathtt{lhs}_S$ and $\mathtt{lhs}_T$: $\mathtt{lhs}_{ST}$. It is clear that these two statements can share the same data iff this sharing does not generate non-neutral dependence between S and T with $\mathtt{lhs}_{ST}$ in the LHS of the two statements. Let f_{ST} be the index function of $\mathtt{lhs}_{ST}$. One must verify for each operation $\langle S,\vec{x}\rangle$ and $\langle T,\vec{y}\rangle$ that would be in output dependence (i.e. $f_{ST}(\vec{x}) = f_{ST}(\vec{y})$) that $\mathcal{V}(S,\vec{x})$ cannot be killed by $\langle T,\vec{y}\rangle$ before the end of its utility span and that $\mathcal{V}(T,\vec{y})$ cannot be killed by $\langle S,\vec{x}\rangle$ before the end of its utility span.

Finding the minimal number of renaming is an NP-complete problem (see [1]). Our method consists in building a graph similar to an interference graph as used in code generation process of a classical compiler to optimize registers allocation. In this graph, each vertex represents a statement of the program. There is an edge between two vertices S and T iff it has been shown that they cannot share the same data structure in their LHS. Then one applies on this graph a greedy coloring algorithm. Finally it is clear that vertices that have the same color can have the same data structure.

6.2.5 An Example

Consider the `matrix-vector` program. It is given in Figure 6 with its data-flow graph.

```
 program matrix-vector
 real s, A[N,N], B[N], C[N]
 do i = 1 to N
S1  s = 0
    do j = 1 to N
S2    s = s + A[i,j]*B[j]
    end do
S3  C[i] = s
 end do
```

$$\sigma(s,\langle S_2,i,j\rangle) = \left| \begin{array}{l} \textbf{if } j \geq 2 \\ \textbf{then } \langle S_2,i,j-1\rangle \\ \textbf{else } \langle S_1,i\rangle \end{array} \right.$$

$$\sigma(a(i,j),\langle S_2,i,j\rangle) = \bot$$

$$\sigma(b(j),\langle S_2,i,j\rangle) = \bot$$

$$\sigma(s,\langle S_3,i\rangle) = \langle S_2,i,N\rangle$$

Figure 6: Program `matrix-vector` and its data-flow graph

Converting the `matrix-vector` program into single-assignment yields a program whose dependence graph equals the data-flow graph in Figure 6. The following schedule θ is thus a valid one for the expanded program:

$$\theta(S_1,i) = 0 \qquad \theta(S_2,i,j) = j \qquad \theta(S_3,i) = N+1 \tag{14}$$

The utility spans are:

Operation v	$L_\theta(v)$	Utility span of $\mathcal{V}(v)$
$\langle S_1,i\rangle$	1	$[0,1]$
$\langle S_2,i,j\rangle$	$j+1$	$[j,j+1]$
$\langle S_3,i\rangle$	$N+1$	$[N+1,N+1]$

The partial expansion step gives the following results:

Statement	Maximum utility span duration	Expansion degree	Final data structure	Final LHS
S_1	$d(S_1)=1$	$E^0_{S_1}=N$	$\texttt{lhs}_{S_1}$`[N]`	$\texttt{lhs}_{S_1}$`[i] = ...`
S_2	$d(S_2)=1$	$E^0_{S_2}=N$		
		$E^1_{S_2}=0$	$\texttt{lhs}_{S_2}$`[N]`	$\texttt{lhs}_{S_2}$`[i] = ...`
S_3	$d(S_3)=0$	$E^0_{S_3}=N$	$\texttt{lhs}_{S_3}$`[N]`	$\texttt{lhs}_{S_3}$`[i] = ...`

Notice that the array in the LHS of S_3 is left untouched even if its values are never read, because it stores output values.

Applying our coloring algorithm shows that S_1, S_2 and S_3 have the same color. The memory requirement is finally a one-dimensional array with N elements which can be the array `C`. Hence the statement S_3 can be deleted. After partial expansion and code generation the parallel version of the program is:

```
 real A[N,N], B[N], C[N]
 do t = 0 to N
   if t = 0 then
     doall i = 1 to N
S1     C[i] = 0
```

```
      end do
    else
      doall i = 1 to N
S2      C[i] = C[i] + A[i,t]*B[t]
      end do
    end if
  end do
```

Our method can effectively reduce the memory cost of data expansion process for affine loop nests. The conversion of the source program to single-assignment would give a memory space of $O(N^2)$ instead of $O(N)$ with partial expansion. Notice that in this example the data size of the parallel program is less than the original data size. Moreover the simplification of memory accesses can in some cases simplify the complexity of the parallel code (removal of S_3).

6.3 Related Work

Many studies are related to array privatization [53, 49]. Maydan et al. [49] proposed an algorithm to privatize arrays. Their method only applies to affine loop nests. Tu and Padua [54] proposed a privatization technique for a very large class of programs. But it resorts to dynamic restoration of the data flow. Another accurate approach using array regions has been described by Creusillet [18], her method avoids the cost of a dynamic restoration and copies back the privatized elements into the original arrays.

Array privatization may require less space than total expansion, but only detects parallelism along the enclosing loops; It is thus less powerful than general array expansion techniques—in terms of parallelism extraction.

De Greef and Catthoor have addressed the memory reuse problem for affine loop nests. They stop at the formulation of the constraints to be satisfied [22]. Another solution has been proposed by the systolic community [56]. Programs in this case are directly given in single-assignment form. They try to create output dependences which do not invalidate the data flow by estimating the lifetime of each variable.

6.4 Perspectives

We have presented in this section two techniques for array expansion, addressing two different but complementary problems: The first one kills as much memory-based dependences as possible without introducing any run-time overhead, and the second one reduces the memory cost of the generated code according to the parallel schedule.

As discussed earlier (see Section 6), our aim is now to make these two techniques work together. Interestingly enough, the static expansion algorithm does not require any precision level of the data-flow analysis, nor does it require the closure computation to be exact. Conservative approximate results are fine as well, the only drawback being a probable loss in static expansion. When the data-flow analysis and/or the transitive closure tool give poor results, our expansion scheme does not fail but degrades gracefully. A fundamental issue in designing a unified technique is thus to extend the memory optimization framework to a more general class of programs and analyses.

7 Related Work on Compilation Frameworks

Initial but restricted parallelizing frameworks aimed at reordering techniques for perfectly nested loops [4]. These approaches cannot handle imperfectly nested loops or represent transformations such as loop distribution or fusion.

The SUIF [23] and Polaris [24] compilers are among the major compilers in the field. The first obvious difference between these compilers and our approach is that they are much more robust than PAF. However, Polaris is based on array privatization only. SUIF is, in that respect, more advanced, and features an array data-flow analysis and affine mappings.

Several automatic parallelizing compilers rely on the polytope model. The Loopo parallelizer [37], developed at the University of Passau, implements the scheduling algorithms proposed in [28] and [20]. Moreover, Loopo is an excellent platform to compare scheduling techniques. The Omega project [43], is very similar in spirit to PAF. The findings of Pugh and Wonnacott on data-flow analysis of general programs [57] are very close to ours, even though both results were achieved independently. Their analysis and scheduling techniques are based on the Omega Calculator [50, 43]. But their mappings allow additional transformations such as loop fusion, loop distribution and statement reordering.

We may roughly say that the compiler efforts cited above, with the exception of Polaris, are based on affine transformations. Another paradigm in the field is supernode partitioning, also known as *tiling* [42, 9, 12]. In particular, tiling may be useful to solve some code generation issues addressed in Section 4. However, even state-of-the-art tiling techniques [46] restrict themselves to loop nests with affine bounds and affine array subscripts. Notice that the partition given in Figure 4.a can be seen as a special case of tiling.

Finally, let us recall that several compiler projects have focused on pointer analysis instead of arrays. For instance, the McCAT compiler includes sophisticated pointer analyses [34, 35]. How to extend our techniques to pointers is left for future work.

8 Conclusion

The research issues that have been reported here are part of a long term project whose aim is to improve automatic parallelization techniques for arbitrary programs. The problem is so complex that one has to divide in the remote hope of conquering. A possible division line is to separate problems in which the size of data structures is known beforehand (let us say, at program loading-time), and those where data structures are built dynamically. In this paper, we deal only with the first type of programs. The second type will be the subject of future work [13, 32].

It is clear that, for programs with dynamic control structures and/or complex array subscripts, the basic objects of automatic parallelization (iteration domains, dependences, data flow) cannot be computed exactly and have to be approximated. This in turn generates two subproblems:

- How to reduce uncertainty in approximations.
- How to synthesize a parallel program in the presence of approximate information.

8.1 Analysis

Automatic parallelization in the presence of approximate dependences is not a new idea. Many parallelizers ignore non linear constraints in dependence tests, with the result that dependences are overestimated. One can apply classical algorithms to such overestimates (as, e.g. the Allen and Kennedy algorithm), but the end result often is simply a copy of the source program. In fact, one needs much more information for the use of some of the most powerful parallelization algorithms, like scheduling and placement. On the other hand, classical parallelizers cannot handle dynamic control structures. `while` loops, for instance, are considered as inherently sequential.

Our answer has been to devise FADA (see Section 3), a fuzzy array data-flow analysis which is able to handle any array program. FADA is more a framework than a completely specified method. It can use information from preliminary analyses (including a preliminary application of itself) to increase its precision. When this information is deduced from the structure of the program, we have been able to prove that our analysis has maximum precision. This cannot be generalized, since it would contradict the undecidability of predicate calculus. There are many sources of information about a program that one would like to integrate in the data-flow analysis process: Structural analysis, property analysis like [33], [40] or [17], more general results from abstract interpretation [16] and so on. How to organize this wealth of information is still a mystery to us.

8.2 Synthesis

Whether the array data-flow analysis is exact (i.e. its result is a singleton) or approximate (i.e. its result is expressed as an affine relation), one can construct affine schedules (see [28, 29]). One must observe that the causality condition takes into account all the memory-based dependences that are not eliminated by array expansion. The program can be brought into single-assignment form, or, more cleverly, one can determine the minimum amount of memory which is able to support the parallelism of the schedule. There is in fact a tradeoff between parallelism and memory: Let us suppose that useless operations have been removed. Values which are generated in a front cannot be used before the next front. For this, one needs at least as many memory cells as there are operations in the front. Hence, the size of the largest front is a lower bound for the size of memory. Suppose that the total number of operations is s and that the number of fronts is ℓ. s is a rough measure of the duration of the sequential program, and ℓ is a rough measure of the duration of the parallel program. The mean size of a front is s/ℓ, and if we suppose that the size of fronts is more or less constant, then this is also the size of the needed memory. But s/ℓ is the maximum speed-up for a large number of processors. Hence, we conclude that the maximum speed-up of a program is of the order of the size of its working memory. Using this result as a practical estimator is complicated by the fact that one must consider the size of the input and of the output, and that these subsets of the data space are not necessarily separate.

In the case of fuzzy analysis, the situation becomes more complicated, because, firstly, some constructions cannot be parallelized without recourse to speculation, and, secondly, because too much expansion requires the use of ϕ-functions. Both of these devices incur overhead, which may or may not be compensated by more parallelism.

Hence, we see that the designer of a parallel program has to tradeoff memory for speed in a complicated way. If the running time is a constraint and memory size is to be minimized, one can compute the best schedule as in Section 5, then slow it down artificially if needed.

Conversely, if one has to find the fastest program under memory-size constraints, a tentative solution is to state that front cardinals should never exceed a given memory size and to solve the corresponding scheduling problem. These issues are left for future work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass, 1986.

[2] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. on Software Engineering*, 18(3):237–251, March 1992.

[3] B. S. Baker. An algorithm for structuring programs. *J. of the ACM*, 24:98–120, 1977.

[4] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1992.

[5] D. Barthou. *Array Dataflow Analysis in Presence of Non-affine Constraints.* PhD thesis, PR*i*SM, Université de Versailles, February 1998.

[6] D. Barthou, A. Cohen, and J.-F. Collard. Maximal static expansion. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 98–106, San Diego, CA, January 1998.

[7] D. Barthou, J.-F. Collard, and P. Feautrier. Fuzzy array dataflow analysis. *Journal of Parallel and Distributed Computing*, 40:210–226, 1997.

[8] R. Bodík and S. Anik. Path-sensitive value-flow analysis. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 237–251, San Diego (CA), January 1998.

[9] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? In *Scalable High-Performance Computing Conf.*, pages 568–576, Knoxville, Tenn., May 1994. IEEE Computer Society Press.

[10] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. *ACM SIGPLAN Notices*, 21(7):162–175, July 1986.

[11] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN PLDI*, pages 47–56, Atlanta, Ga, June 1988. ACM.

[12] L. Carter, J. Ferrante, and S. Flynn Hummel. Efficient multiprocessor parallelism via hierarchical tiling. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1995.

[13] A. Cohen and J.-F. Collard. Applicability of algebraic transductions to data-flow analysis. Technical Report 98/002, PR*i*SM, U. of Versailles, January 1998.

[14] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. of the Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G 10.3*, pages 185–194, Caracas, Venezuela, April 1994. North Holland.

[15] J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 429–436, Knoxville, TN, May 1994. IEEE.

[16] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 238–252, Los Angeles, CA, 1977.

[17] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *ACM Symp. on Principles of Programming Languages (PoPL)*, San Diego, CA, 1978.

[18] B. Creusillet. *Array Region Analyses and Applications.* PhD thesis, École des Mines de Paris (ENSMP), December 1996.

[19] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[20] A. Darte and F. Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *Int. Journal of Parallel Programming*, 25(6):447–496, December 1997.

[21] E. Duesterwald, R. Gupta, and M.-L. Soffa. A practical data flow framework for array reference analysis and its use in optimization. In *ACM SIGPLAN'93 Conf. on Prog. Lang. Design and Implementation*, pages 68–77, June 1993.

[22] F. Catthoor E. de Greef and H. de Man. Reducing storage size for static control programs mapped to parallel architectures. In *Dagstuhl Seminar on Loop Parallelization*, April 1996.

[23] M. Hall et al. Maximizing multiprocessor performance with the SUIF compiler. *IEEE Computer*, 29(12):84–89, December 1996.

[24] W. Blume et al. Parallel programming with Polaris. *IEEE Computer*, 29(12):78–82, December 1996.

[25] Alain Darte et Yves Robert. Affine-by-statement scheduling of uniform loop nests over parametric domains. Technical Report 92-16, LIP, ENS Lyon, France, 1992.

[26] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22:243–268, September 1988.

[27] P. Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[28] P. Feautrier. Some efficient solution to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[29] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), December 1992.

[30] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103. Springer-Verlag, June 1996.

[31] P. Feautrier. Basis of parallel speculative execution. In *Europar'97*, pages 3–14. Springer-Verlag, LNCS 1300, 1997.

[32] P. Feautrier. A parallelization framework for recursive tree programs. Technical report, PR*i*SM, University ov Versailles, 1998. To appear.

[33] R. J. Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science*. AMS, 1967.

[34] R. Ghiya and L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 1–15, St. Petersburg, Florida, January 1996.

[35] R. Ghiya and L. J. Hendren. Putting pointer analysis to work. In *ACM Symp. on Principles of Programming Languages (PoPL)*, San Diego, California, January 1998.

[36] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In S. Haridi, K. Ali, and P. Magnusson, editors, *EURO-PAR '95*, Lecture Notes in Computer Science 966, pages 315–326. Springer-Verlag, 1995.

[37] M. Griebl and C. Lengauer. The loop parallelizer LooPo — announcement. *LNCS*, 1239:603–607, 1997.

[38] M.W. Hall, J.M. Mellor-Crummey, A. Carle, and R. Rodriguez. Fiat: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Annual Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, January 1994. updated version for book.

[39] C. Heckler and L. Thiele. Computing linear data dependencies in nested loop programs. *Parallel Processing Letters*, 4(3):193–204, September 1994.

[40] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21, 1978.

[41] F. Irigoin. Interprocedural analyses for programming environments. In J. J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, pages 333–350. Elsevier, 1993.

[42] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.

[43] W. Kelly and W. Pugh. A Framework for Unifying Reordering Transformations. Technical Report CS-TR-2995.1, U. of Maryland, November 1992.

[44] K. Knobe and V. Sarkar. Array SSA form and its use in parallelization. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 107–120, San Diego (CA), January 1998.

[45] V. Lefebvre and P. Feautrier. Optimizing storage size for static control programs in automatic parallelizers. In S. Gorlatch C. Lengauer, M. Griebl, editor, *Euro-Par'97 Parallel Processing*, pages 356–363. Springer-Verlag, August 1997.

[46] A. W. Lim and M. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *ACM Symp. on Principles of Programming Languages (PoPL)*, pages 201–214, Paris, January 1997.

[47] F. Masdupuy. Semantic analysis of interval congruences. In D. Børner, M. Broy, and I.V. Pottosin, editors, *Int. Conf. on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, pages 142–155, Academgorodok, Novosibirsk, Russia, June 1993. Springer-Verlag.

[48] V. Maslov and W. Pugh. Simplifying polynomial constraints over integers to make dependence analysis more precise. Technical Report CS-TR-3109.1, U. of Maryland, February 1994.

[49] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993.

[50] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.

[51] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):27–47, August 1992.

[52] W. Pugh and D. Wonnacott. An exact method for analysis of value-based data dependences. Technical Report CS-TR-3196, U. of Maryland, December 1993.

[53] P. Tu and D. Padua. Array privatization for shared and distributed memory machines. In *Third Workshop on Languages and Compilers for Distributed Memory Machines*, 1992.

[54] P. Tu and D. Padua. Automatic array privatization. In *Proc. Sixth Workshop on Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 500–521, August 1993. Portland, Oregon.

[55] P. Tu and D. Padua. Gated SSA-Based demand-driven symbolic analysis for parallelizing compilers. In *ACM Int. Conf. on Supercomputing*, pages 414–423, Barcelona, Spain, July 1995.

[56] D. Wilde and S. Rajopadhye. Memory reuse analysis in the polyhedral model. In Mignotte Bougé, Fraignaud and Robert, editors, *Europar'96 Parallel Processing*, pages 389–397. Springer Verlag, LNCS 1123, 1996.

[57] D. G. Wonnacott. *Constraint-Based Array Dependence Analysis*. PhD thesis, University of Maryland, 1995.