Optimizing Storage Size for Static Control Programs in Automatic Parallelizers

Vincent Lefebvre and Paul Feautrier

Laboratoire PRiSM, Université de Versailles-St. Quentin,
45, Avenue des États-Unis, 78 035 Versailles cédex, FRANCE
e-mail: {Vincent.Lefebvre,Paul.Feautrier}@prism.uvsq.fr
Workshop: 03 Automatic Parallelization and High-Performance Compilers

Abstract. This article deals with automatic parallelization of static control programs. During the parallelization process the removal of artificial dependences is usually realized by translating the original program into a single assignment form. This total data expansion has a very high memory cost. We present a technique of partial data expansion which leaves untouched the performances of the parallelization process, with the help of algebra techniques given by the polytope model.

1 Introduction

This article deals with the automatic parallelization technique based on the polytope model. This method can be applied provided that source programs are static control programs, i.e. are limited to `do` loops and assignment statements to array with affine subscripts. The first step is an array data flow analysis in order to extract exact dependences on memory cells. All artificial dependences, which are due to reuse of data, are deleted by a total data expansion. The transformed program has the single assignment property and residual dependences constitute the data flow. The program is then parallelized by scheduling a method which automatically satisfies the sequential constraints inherent in the data flow.
The single assignment form translation has a very high memory cost. The aim of this paper is to present a new technique for partial data expansion. We show that starting with a schedule function given by the data flow, it is possible to build a parallel program in which memory is reused.

2 The polyedric method

All techniques and algorithmes described in this section are directly taken from the PAF compiler developped at the university of Versailles by P. Feautrier and his team. All Parametric Integer Programming problems, are solved with the PIP algorithm as described in [5].

2.1 Static Control Programs

Static control programs are built from assignment statements and `DO` loops. The only data structures are arrays of arbitrary dimensions. Loop bounds and

array subscripts are affine functions in the loop counters and integral structure parameters.
An operation may be named $\langle R, \mathbf{x} \rangle$ where R is a statement and $\mathbf{x}$ the iteration vector built from the values of the surrounding loop counters. The iteration domain $\mathcal{D}(R)$ of a statement R, is the set of instances of R and can be described by the conjunction of all inequalities associated to the surrounding loops. One will take as running example in this article, the following program:

```
      program matrix-vector
      real s, a(n,n), b(n), c(n)
      integer i,j,n
      do i=1,n
S1      s = 0.
        do j=1,n
S2        s = s + a(i,j)*b(j)
        end do
S3      c(i) = s
      end do
      end
```

In the program $\langle S2, i, j \rangle$ is an instance of $S2$ and $\mathcal{D}(S2) = \{i, j \,|\, 1 \leq i \leq n \wedge 1 \leq j \leq n\}$ is the iteration domain of $S2$.

2.2 Sequential Execution Order

Let us introduce some notations. The lexicographic order is noted $\ll$. The expression $R \lhd S$ indicates that statement R is before statement S in the program text. N_{RS} is the number of loops surrounding both R and S. One has $\mathbf{x} \ll_p \mathbf{y} \equiv \mathbf{x}[1..p] = \mathbf{y}[1..p] \wedge \mathbf{x}[p+1] < \mathbf{y}[p+1]$ and $\ll$ is given by

$$\mathbf{x} \ll \mathbf{y} \equiv \bigvee_{p=0}^{|\mathbf{x}|-1} \mathbf{x} \ll_p \mathbf{y} \tag{1}$$

The fact that operation $\langle R, \mathbf{x} \rangle$ is executed before the operation $\langle S, \mathbf{y} \rangle$ is written: $\langle R, \mathbf{x} \rangle \prec \langle S, \mathbf{y} \rangle$. It is shown in [5] that:

$$\langle R, \mathbf{x} \rangle \prec \langle S, \mathbf{y} \rangle \equiv \mathbf{x} \ll \mathbf{y} \vee (\mathbf{x}[1..N_{RS}] = \mathbf{y}[1..N_{RS}] \wedge R \lhd S) \equiv \bigvee_{p=0}^{N_{RS}} \langle R, \mathbf{x} \rangle \prec_p \langle S, \mathbf{y} \rangle \tag{2}$$

where

$$\langle R, \mathbf{x} \rangle \prec_p \langle S, \mathbf{y} \rangle \Leftrightarrow \begin{cases} 0 \leq p < N_{RS} : \mathbf{x} \ll_p \mathbf{y} \\ p = N_{RS} : \mathbf{x}[1..N_{RS}] = \mathbf{y}[1..N_{RS}] \wedge R \lhd S \end{cases} \tag{3}$$

In our running example, we have: $\langle S1, 2 \rangle \prec_0 \langle S2, 3, 1 \rangle$ and $\langle S2, 2, 3 \rangle \prec_0 \langle S2, 3, 1 \rangle$

2.3 Array Data Flow Analysis

To each operation v we associate two sets: $\mathcal{R}(v)$ is the set of memory cells which are read by v; $\mathcal{M}(v)$ is the set of memory cells which are modified by v. Berstein's conditions distinguish three kinds of dependences between v and u, where $v \prec u$. If $\mathcal{M}(v) \cap \mathcal{R}(u) \neq \emptyset$, there is a **flow dependence**, written $v \, \delta \, u$. If $\mathcal{R}(v) \cap \mathcal{M}(u) \neq \emptyset$, there is an **anti-dependence**, written $v \, \overline{\delta} \, u$. If

$\mathcal{M}(v) \cap \mathcal{M}(u) \neq \emptyset$, there is an **output dependence**, written $v\,\delta^{\circ}\,u$. One may be more precise and associate a dependence to a depth p. For instance, if two operations v and u are in flow dependence at depth p, written $v\,\delta_p\,u$, it means that: $v \prec_p u \wedge \mathcal{M}(v) \cap \mathcal{R}(u) \neq \emptyset$.
The real dependences which define the inherent semantic of a program, are a subset of flow dependences: the **direct flow dependences**. All others dependences are due to memory reuse and are artificial. A direct flow dependence is a data flow from a definition by an operation v to a use by an operation w of a same memory cell c and provided there is no write on c between the executions of v and w. It means that the value read by w in c is the one produced by v. **Direct flow dependences** are computed by data flow analysis [5]. It must determine for each memory cell c read by an operation w, the last operation in $\prec$ which gives a value to c before the execution of w. This operation is called the *source* function of the read:

$$source(c, w) = \max_{\prec}\{v \mid v\,\delta\,w\} \tag{4}$$

The computation of the *source* function can be done by PIP (Parametric Integer Programming) algorithm (cf [5] for more details). The result of the analysis is a quasi-affine tree or quast, i.e. a many-level conditionnal in which predicates are tests for the positiveness of affine forms in the loop counters and structure parameters. The Leaves are either operation names, or $\perp$. $\perp$ indicates that the array cell under study is not modified.
Sources functions are gathered in the Data Flow Graph (DFG). The DFG of our running example is:

memory cell referenced	read operation	source operation
s	$\langle S2, i, j\rangle$	**if** $j-2 \geq 0$ **then** $\langle S2, i, j-1\rangle$ **else** $\langle S1, i\rangle$
$a(i,j)$	$\langle S2, i, j\rangle$	$-$
$b(j)$	$\langle S2, i, j\rangle$	$-$
s	$\langle S3, i\rangle$	$\langle S2, i, n\rangle$

2.4 Total Data Expansion

The aim is to delete all artificial dependences. Total data expansion gives to the program the **single assignment property**: each memory cell allocated to data will only receive one value produced by one operation during all the execution of the program. In this way, one associates a memory cell to an operation. One can find the algorithm of translation of a static control program into a single assignment form in [5]. The first step is a **complete renaming**: for each statement R one associates a specific data structure `InsR`, used to store all values produced by the operations instances of R. Then one **totally expands** all data structures: `InsR` is indexed by the iteration vector of R.

$$R:\ \texttt{A[f(x)]} = \ldots \text{ becomes } R:\ \texttt{InsR[x]} = \ldots$$

Finally one **reconstitutes the data flow** by replacing each rhs reference by its corresponding *source*

2.5 Parallelization by Scheduling

One computes a time function θ which gives the partial execution order of the parallel program by taking into account the sequential constraints of the data flow. For any operation u, if $\theta(u)$ is its execution time, one must have:

$$\forall c \in \mathcal{R}(u), \theta(source(c,u)) \ll \theta(u) \tag{5}$$

It defines a set of linear constraints. For complexity reasons finding the exact solution of (5) is not practicable. One limits oneself to affine one-dimensionnal and multi-dimensionnal [6] schedules. In the case of our running example, one can have the following schedule function θ:

$$\begin{cases} \theta(S1,i) = 0 \\ \theta(S2,i,j) = j \\ \theta(S3,i) = n+1 \end{cases} \tag{6}$$

An operation front $\mathcal{F}(\mathbf{t})$ gathers all operations which have a same execution time. The operations of a same front can be executed in parallel. Let τ be the set of lexicographical enumeration of each possible execution time ($\mathbf{t} \in \tau \Rightarrow \mathcal{F}(\mathbf{t}) \neq \emptyset$). The parallel program must enumerate each possible date $\mathbf{t} \in \tau$. If one translates in Fortran 90 the parallel program built with (6) as new operations execution order, one gets the following code:

```
        program matrix-vector
        real InsS1(n), InsS2(n,n), InsS3(n), a(n,n), b(n)
        do t=0,n+1
          if (t .EQ. 0) then
S1          InsS1(1:n:1)=0.
          end if
          if (t .EQ. 1) then
S2          InsS2(1:n:1,t) = InsS1(1:n:1) + a(1:n:1,t)*b(t)
          end if
          if (t .GE. 2 .AND. t .LE. n) then
S2          InsS2(1:n:1,t) = InsS2(1:n:1,t-1) + a(1:n:1,t)*b(t)
          end if
          if (t .EQ. n+1) then
S3          InsS3(1:n:1) = InsS2(1:n:1,n)
          end if
        end do
        end
```

Notice that the total data expansion has created one one-dimensionnal array `InsS1` with n elements and a two-dimensionnal array with n^2 elements. Moreover it has induced the split of $S2$ in two different statements in the parallel code.

3 Reduced Data Expansion in Parallelized Programs

Translating the sequential program in single assignment form has a very high memory cost. It is clear in the case of our running example: from a scalar `s` and an array `c(n)`, one gets three arrays with a data space of $\mathcal{O}(n^2)$.
Our aim is now to define a method of partial data expansion which **reduces the memory expansion** induced by parallelization and **replaces the single assignment form translation** during the parallelization process. The constraint is that the schedule which has been deduced from the DFG should remain valid in the presence of output and anti dependences. An intuitive presentation of the method is given below.

3.1 An intuitive Approach

One must precise some conventions and notations. One writes $\mathcal{V}(v)$ for the value produced by an operation v. $\mathcal{C}(v)$ is the memory cell in which $\mathcal{V}(v)$ is stored. The set $\mathcal{U}(v)$ gathers all operations u such that there is a direct data flow from v to u. $\mathcal{U}(v)$ is the set of all operations which will be executed after v and will read $\mathcal{V}(v)$:

$$\mathcal{U}(v) = \{u \mid source(\mathcal{C}(v), u) = v\} \tag{7}$$

$\mathcal{U}(v)$ is usually called the **utilization set** of v.
$\mathcal{L}(v)$ is the execution time of the last read of $\mathcal{V}(v)$ in the parallel program. $L(v)$ is the operation which executes this last read:

$$\mathcal{L}(v) = \theta(L(v)) = \max \theta(u), u \in \mathcal{U}(v) \tag{8}$$

Consider a memory cell $\mathcal{C}(v)$ during the execution of a parallel program in single assignment form. One can distinguish three periods:

1. **Period (I)**: the memory cell **stays empty** until the execution of v with which it is associated.
   In our running example, `InsS2[i,j]` stays **"empty"** until the execution of $\langle S2, i, j\rangle$ (`InsS2[i,j]` $= \mathcal{C}(S2, i, j)$) at $\theta(S2, i, j) = j$, if $1 \leq j \leq n \perp 1$.
2. **Period (II)**: the execution of v stores $\mathcal{V}(v)$ in $\mathcal{C}(v)$. The operations of $\mathcal{U}(v)$ read $\mathcal{V}(v)$ until $\mathcal{L}(v)$. During this time, $\mathcal{V}(v)$ is **useful**.
   One has $\mathcal{U}(S2, i, j) = \{\langle S2, i, j+1\rangle\}$. $\mathcal{V}(S2, i, j)$ is read by $\langle S2, i, j+1\rangle$ at $\theta(S2, i, j+1) = j+1$. This time is the last read of $\mathcal{V}(S2, i, j)$: $\mathcal{L}(S2, i, j) = j+1$
3. **Period (III)**: the memory cell is not read anymore after $\mathcal{L}(v)$, nevertheless $\mathcal{V}(v)$ is still in $\mathcal{C}(v)$ until the end of the execution of the parallel program. $\mathcal{V}(v)$ becomes **useless**.
   $\mathcal{V}(S2, i, j)$ becomes useless after $\theta(S2, i, j+1) = j+1$ and stays in `InsS2[i,j]` until the end of the program at $\theta(S3, i) = n + 1$

It is clear that during the period (I) and (III), $\mathcal{C}(v)$ can store others values. If one stores others values in $\mathcal{C}(v)$, output dependences appear in the parallel program. The problem is to define an automatic method for partial data expansion which ensures that the parallel program obtained is valid.

3.2 Previous Techniques to Automatically Reduce Storage Size

Most of papers from the automatic parallelization community deal with array privatization. Privatization is a technique that allows each thread on a processor to allocate a distinct instance of a variable. It may require less space than total expansion because it creates one copy per processor and the number of processors cooperating in the execution of the parallel loop is less than the number of iterations ([9],[7]). Lam [1] proposes to optimize array privatization with the help of the Data Flow Graph. Another solution has been proposed by the systolic community ([3],[10]). Programs in this case are directly given in single assignment form. They try to create output dependences which don't invalidate the data flow by estimating the lifetime of each variable. It is interesting to notice that these techniques are similar to data-localization methods ([4],[11]).

3.3 Utility Span of a Value

Our method of partial data expansion is based on the notion of **utility span of a value**. The main advantage over the notion of variable lifetime is that it can be applied to programs which are not necessarily in single assignment form. The atomic entity in our study is not the memory cell $\mathcal{C}(v)$ like in most previous methods, but the value $\mathcal{V}(v)$.
The utility span of a value is a subsegment of $[\mathbf{0}, \mathbf{L}]$ where L is the latency i.e. the execution time of the last front executed in the parallel program. It is clear that it corresponds to the period (II): $\mathcal{V}(v)$ must reside in memory during $\mathbf{t} \in [\theta(v), \mathcal{L}(v)]$.

Definition 1 *The utility span of $\mathcal{V}(v)$ is the span between the time of production of $\mathcal{V}(v)$ and the time of its last read in the parallel program, where $\mathcal{V}(v)$ must reside in memory.*

$$\mathbf{t} \in [\theta(v), \mathcal{L}(v)] \Rightarrow \mathcal{V}(v) \in \mathcal{C}(v) \tag{9}$$

One can estimate the utility span of $\mathcal{V}(S2, i, j)$ in our running example. If $1 \leq i \leq n \wedge 1 \leq j \leq n \perp 1$, then $\mathcal{V}(S2, i, j)$ must reside in $\mathcal{C}(v)$ for $\mathbf{t} \in [\theta(S2, i, j), \theta(S2, i, j+1)] = [j, j+1]$.
Before and after this utility span, $\mathcal{C}(v)$ can store others values without changing the data flow from v to operations in $\mathcal{U}(v)$: one can reintroduce output dependences between v and some others operations. The next subsection show which are the conditions that an output dependence must verify to be tolerable in the parallel program. Such output dependences are called **neutral dependences**.

3.4 Neutral Dependences

Consider two operations v and w. The rule (9) imposes that:

1. $\mathcal{V}(v) \in \mathcal{C}(v)$ for $\mathbf{t} \in [\theta(v), \mathcal{L}(v)]$
2. $\mathcal{V}(w) \in \mathcal{C}(w)$ for $\mathbf{t} \in [\theta(w), \mathcal{L}(w)]$

In the case of a program in single assignment form, one has systematically $\mathcal{C}(v) \neq \mathcal{C}(w)$ because there is no output dependence. Optimizing the storage, means that one introduces memory reuse in the parallel program, i.e. we want to have some operations v and w such as $\mathcal{C}(v) = \mathcal{C}(w)$. It is clear that is possible iff the basic rule (9) is still verified for v and w in spite of this output dependence. Hence an output dependence is valid in the parallel program if the subsegments which are the utily spans of v and w are separate. A such output dependence is called **neutral output dependence**.

Definition 2 *An output dependence is neutral for a schedule function θ iff it doesn't change the data flow in the parallel program built with the help of θ.*

One can precisely gives the characteristics of a **neutral output dependence** $v\, \delta^{\circ}\, w$ in the parallel program:

- v **must be executed before** w: $\theta(v) \ll \theta(w)$.

– **there is an access conflict**: $\mathcal{C}(v) = \mathcal{C}(w)$
– **the utility spans are separate**: $\mathcal{L}(v) \ll \theta(w)$

By extension an output dependence between v and w can be considered as neutral if w is $L(v)$, i.e. the operation which executes the last read of $\mathcal{V}(v)$. Here the utility spans of $\mathcal{V}(v)$ and $\mathcal{V}(w)$ are not separate because $\mathcal{L}(v) = \theta(w)$. Nevertheless these two operations can share the same memory cell because w must read $\mathcal{V}(v)$ before computing $\mathcal{V}(w)$. It means that the write of $\mathcal{V}(w)$ occurs after the read of $\mathcal{V}(v)$ by w.
An output dependence between $\langle S2, i, j\rangle$ and $\langle S2, i, j+2\rangle$ would be neutral because $\langle S2, i, j+2\rangle$ is executed after the utility span of $\mathcal{V}(S2, i, j)$ in the parallel program.
Notice that if **two operations** v and w belong to the **same operations front**, an output dependence $v\,\delta^{\circ}\,w$ would be **non neutral** in the parallel program. Hence one must use data expansion to ensure that they are stored in two different memory cells. In fact, the memory requirement of a parallel program is strongly linked to the parallelism degree (size of operations fronts) given by the schedule function. As we have seen, the utility span of $\mathcal{V}(S2, i, j)$ for $j < n$ is between $\mathbf{t} = j$ and $\mathbf{t} = j + 1$ in our running example. An output dependence between $\langle S2, i, j\rangle$ and $\langle S2, i+1, j\rangle$ would not be neutral because the two operations belong to the same front $\mathcal{F}(\mathbf{t}) = j$.
To decide if an output dependence is neutral in a parallel program, one must have a precise estimation of a 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 program with direct flow dependences and output dependences that will be neutral. Our first approach has consisted to maintain neutral output dependences from the original program to its parallel version [8]. But this method is directly dependent from the original data space and can't be used to reduce data size of programs provided in single assignment form. We have decided to improve our technique to become independent from the original data: with the new method presented in this article, the output dependences existing in the program after partial expansion are not necessarily present in the original version.

3.5 Determinating Utility Span

Consider an operation $\langle R, \mathbf{x}\rangle$. One wants to determine the subsegment of $[\mathbf{0}, \mathbf{L}]$ which corresponds to the utility span of this operation: $[\theta(R, \mathbf{x}), \mathcal{L}(R, \mathbf{x})]$. The lower bound of this subsegment is directly given by θ. The problem is to compute the upper bound $\mathcal{L}(R, \mathbf{x})$. We recall that it is the last execution time in the parallel program of an operation of the utilization set $\mathcal{U}(R, \mathbf{x})$.
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. Consider two statements R and S:

$$R : a[\mathbf{f}(\mathbf{x})] = \ldots$$
$$S : \ldots = \ldots a[\mathbf{h}(\mathbf{y})] \ldots$$

The operation $L_S(R, \mathbf{x})$ is the last read of $\mathcal{V}(R, \mathbf{x})$ in the parallel program among the operations instances of S which belong to $\mathcal{U}(R, \mathbf{x})$. The set of candidates is $\langle S, B_{RS}(\mathbf{x}) \rangle$ with

$$\begin{aligned} B_{RS}(\mathbf{x}) = \{\, \mathbf{y} \mid\ & \mathbf{x} \in \mathcal{D}(R) \wedge \\ & \mathbf{y} \in \mathcal{D}(S) \wedge \\ & source(a[\mathbf{h}(\mathbf{y})], \langle S, \mathbf{y} \rangle) = \langle R, \mathbf{x} \rangle \} \end{aligned} \tag{10}$$

This set is built by scanning the Data Flow Graph. It is clear that the last operation which reads $\mathcal{V}(R, \mathbf{x})$ between instances of S is the last one executed according to θ:

$$L_S(R, \mathbf{x}) = \langle S, \max_{\ll_\theta} B_{RS}(\mathbf{x}) \rangle \tag{11}$$

The set $B_{RS}(\mathbf{x})$ is a disjunction of $\mathbb{Z}$-polyhedra. All statements which may read the data a must be taken into account. The real last read is their maximum according to θ:

$$L(R, \mathbf{x}) = \max_{\ll_\theta} L_S(R, \mathbf{x}) \tag{12}$$

Like the source function, $L(R, \mathbf{x})$ is a quast. To determine $\mathcal{L}(R, \mathbf{x})$ one just applies the function θ to each leaf of $L(R, \mathbf{x})$ except for leaves which are the symbol $\perp$ which are left untouched. The different utility spans are gathered in the Utility Span Graph (USG) which gives to each operations v the utility span of $\mathcal{V}(v)$ and the operation executing the last read of $\mathcal{V}(v)$. The symbol $\perp$ indicates that $\mathcal{V}(v)$ is either useless or an output value. For our running example one obtains:

Operation v	$L(v)$	$\mathcal{L}(v)$	Utility span of $\mathcal{V}(v) = [\theta(v), \mathcal{L}(v)]$
$\langle S1, i \rangle$	$\langle S2, i, 1 \rangle$	1	$[0, 1]$
$\langle S2, i, j \rangle$	**if** $j \leq n-1$ **then** $\langle S2, i, j+1 \rangle$ **else** $\langle S3, i \rangle$	**if** $j \leq n-1$ **then** $j+1$ **else** $n+1$	**if** $j \leq n-1$ **then** $[j, j+1]$ **else** $[j, n+1]$
$\langle S3, i \rangle$	–	–	$[n+1, -]$

3.6 Partial Data Expansion

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

Partial Array Expansion The aims of partial array expansion for each statement R are the following:

- We want to build a structure `lhsR` which is specifically associated to the statement R. 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 R in the restructured program.
- The specifications used to build `lhsR` is that if `lhsR` provides memory reuse, i.e. output dependences between some operations instances of R, these output dependences have to be **neutral** in the parallel program.

– The elaboration of `lhsR` must be independent from the original data structure in the lhs of R.

The problem is now to build `lhsR`. One recalls that a neutral output dependence can't kill a value $\mathcal{V}(R,\mathbf{x})$ during its utility span. To respect this rule for any instance of R, one must take into account the maximum duration that the utility span of $\mathcal{V}(R,\mathbf{x})$ can have in the parallel program. For an operation $\langle R,\mathbf{x}\rangle$ this duration is obtained by subtracting the lower bound of its utility span from the upper bound. One writes $d(R,\mathbf{x})$ this parameter:

$$d(R,\mathbf{x}) = \mathcal{L}(R,\mathbf{x}) - \theta(R,\mathbf{x}) \tag{13}$$

One considers that $\bot\bot\theta(R,\mathbf{x}) = \bot$. Each leaf of $d(R,\mathbf{x})$ is a multi-dimensionnal linear expression in term of loop counters and structure parameters. The maximum duration $D(R)$ that the utility span of instances of R can have, is the maximum value of $d(R,\mathbf{x})$ on the iteration domain of R:

$$\forall \mathbf{x} \in \mathcal{D}(R), d(R,\mathbf{x}) \underline{\ll} D(R) \tag{14}$$

$D(R)$ is a multidimensionnal linear expression in term of structure parameters or the symbol $\bot$. Notice that one considers that if $d(R,\mathbf{x}) \neq \bot$, then $\bot \ll d(R,\mathbf{x})$. For our running example, one finds:

Statement R	Utility span duration of an instance of R	Maximum utility span duration on R
$S1$	$d(S1,i) = 1$	$D(S1) = 1$
$S2$	$d(S2,i,j) = \begin{cases} \textbf{if } j \leq n-1 \\ \textbf{then } 1 \\ \textbf{else } 1 \end{cases}$	$D(S2) = 1$
$S3$	$d(S3,i) = -$	$D(S3) = -$

(9) implies that $\mathcal{V}(R,\mathbf{x})$ must be in $\mathcal{C}(R,\mathbf{x})$ between $\theta(R,\mathbf{x})$ and $\mathcal{L}(R,\mathbf{x}) = \theta(R,\mathbf{x}) + d(R,\mathbf{x})$. If one wants to protect each instance of R during its utility span, one must build `lhsR` in such a way that (9) is verified for the greatest utility span that an instance of R can have. Hence one has chosen to impose that no value $\mathcal{V}(R,\mathbf{x})$ can be killed between $\theta(R,\mathbf{x})$ and $\theta(R,\mathbf{x}) + D(R)$:

$\mathcal{V}(R,\mathbf{x}) \in$ `lshR` for $\mathbf{t}$ in $[\theta(R,\mathbf{x}), \theta(R,\mathbf{x})+D(R)]$ where $\theta(R,\mathbf{x})+d(R,\mathbf{x}) \underline{\ll} \theta(R,\mathbf{x})+D(R)$

The algorithm that builds the data structure `lhsR` can be summarized like this:

– One starts with a scalar `lhsR`.
– The elaboration of `lhsR` is iterative, the number of iterations is equal to N_{RR} (number of loops surrounding R). Each iteration is called **partial expansion of R at depth** p where p is the depth of the loop considered ($p \in [0, N_{RR} \bot 1]$).
– A **partial expansion of R according to** $(p+1)$ consists in
  1. Computing the **expansion degree of R at depth** p: E_R^p. It gives the number of elements of a new dimension that one adds to `lhsR`.
  2. Indexing this new dimension of `lhsR`:

$$\texttt{lhsR}[\mathbf{F}'(\mathbf{x})] \text{ becomes } \texttt{lshR}[\mathbf{F}'(\mathbf{x}),\ i_{p+1} \bmod E_R^p]$$

  where $\mathbf{F}'(\mathbf{x})$ is the index function built by previous iterations on p; i_{p+1} is the counter of the loop $(p+1)$ (from the outer one surrounding R); "mod" is the modulo operator and E_R^p is the expansion degree computed in the previous step.

– At the end of the process, `lhsR` only provides neutral output dependences on $R, \forall p \in N_{RR}$.

The problem is now to compute E^p_R. The partial expansion of R at depth p avoids non neutral output dependences between two operations $\langle R, \mathbf{x}\rangle$ and $\langle R, \mathbf{x}'\rangle$ if $\mathbf{x} \ll_p \mathbf{x}'$. For an operation $\langle R, \mathbf{x}\rangle$, we build the set of candidates gathering all the operations $\langle R, \mathbf{x}'\rangle$ which can't share the same memory cell than $\langle R, \mathbf{x}\rangle$:

– **the operations exist**: $\mathbf{x} \in \mathcal{D}(R)$ and $\mathbf{x}' \in \mathcal{D}(R)$
– **the sequential execution order is**: $\langle R, \mathbf{x}\rangle \prec_p \langle R, \mathbf{x}'\rangle$
– **the utility spans are not separate**:

$$[\theta(R, \mathbf{x}), \theta(R, \mathbf{x}) + D(R)] \cap [\theta(R, \mathbf{x}'), \theta(R, \mathbf{x}') + D(R)] \neq \emptyset$$

Let be $C^p_{RR}(\mathbf{x})$ the set of candidates, it can be decomposed in disjunctions of $\mathbb{Z}$-polyhedra. Let $e_R^{C,p}$ be its lexicographic maximum:

$$e_R^{C,p} = \max_{\prec_p} C^p_{RR}(\mathbf{x})$$

One can't have output dependences between operations $\langle R, \mathbf{x}\rangle$ and $\langle R, \mathbf{x}'\rangle$ with:

$$\langle R, \mathbf{x}\rangle \prec_p \langle R, \mathbf{x}'\rangle \preceq_p \langle R, \mathbf{x}_e\rangle = e_R^{C,p}$$

From this follows the inequalities on the iteration vectors:

$$\mathbf{x}[p+1] < \mathbf{x}'[p+1] \leq \mathbf{x}_e[p+1]$$

If one expands `lhsR` at depth p with $E^p_{\langle R,\mathbf{x}\rangle} = \mathbf{x}_e[p+1] \perp \mathbf{x}[p+1] + 1$, we are sure that no non neutral output dependence at depth p can appear concerning $\langle R, \mathbf{x}\rangle$. But it must be verified for each instance of R, hence the expension degree E^p_R is the maximum value that $E^p_{\langle R,\mathbf{x}\rangle}$ can have for $\mathbf{x} \in \mathcal{D}(R)$:

$$E^p_R = \max_{\mathbf{x} \in \mathcal{D}(R)} E^p_{\langle R,\mathbf{x}\rangle} \tag{15}$$

For our running example, one obtains the following results:

Statements	Expansion degrees	Final data structure	Final lhs
S1	$E^0_{S1} = n$	lhsS1[n]	lhsS1[i] = ...
S2	$E^0_{S2} = n$	lhsS2[n]	lhsS2[i] = ...
	$E^1_{S2} = 0$		
S3	$E^0_{S3} = n$	lhsS3[n]	lhsS3[i] = ...

There can't be output dependences on S1 and S2 at depth 0, hence `lhsS1` is fully expanded and `lhsS2` becomes a one-dimensionnal array with n elements. But all output dependences on $S2$ at depth 1 will be neutral in the parallel program, hence there is no expansion at depth 1 for $S2$. Notice that for the last statement one leaves untouched the shape of the array in the lhs of $S3$ even if its values are never read. It is due to the fact that it stores the final results of the program.

Partial Renaming The partial renaming process must decide if two different statements can share the same data structure. Consider two statements R and T. Partial expansion builds two structures `lhsR` and `lhsT` which can have different shapes. If at the end of the renaming process R and T are authorized to share the same array, this one would have to be the rectangular hull of `lhsR` and `lhsT`: `lhsR-T`. It is clear that these two statements can share the same data iff this sharing does not generate non neutral dependence between R and T with `lhsR-T` in left hand side of the two statements. Let $\mathbf{F}_{R-T}$ be the index function of `lhsR-T`. One must verify for each operation $\langle R, \mathbf{x}\rangle$ and $\langle T, \mathbf{z}\rangle$ that would be in output dependence (i.e. $\mathbf{F}_{R-T}(\mathbf{x}) = \mathbf{F}_{R-T}(\mathbf{z})$) that:

1. $\mathcal{V}(R, \mathbf{x})$ can't be killed by $\langle T, \mathbf{z}\rangle$ before the end of its utility span:
$$\theta(R, \mathbf{x}) \underline{\ll} \theta(T, \mathbf{z}) \underline{\ll} \theta(R, \mathbf{x}) + D(R)$$
2. $\mathcal{V}(T, \mathbf{z})$ can't be killed before by $\langle R, \mathbf{x}\rangle$ before the end of its utility span:
$$\theta(T, \mathbf{z}) \underline{\ll} \theta(R, \mathbf{x}) \underline{\ll} \theta(T, \mathbf{z}) + D(T)$$

As in the case of partial expanding, one can decompose candidates sets in disjunctions of $\mathbb{Z}$-polyhedra. All these $\mathbb{Z}$-polyhedra must be empty for this transformation to be legal. If there are no integral solutions, R and T can share the same data structure else they can't.
Finding the minimal number of renaming is a NP-complete problem (see [2]). Our method consists in building a graph similar to an interference graph as used in code genaration 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 R and T iff it has been shown that they can't share the same data structure in their left hand side: there is at least one non neutral output dependence $R\,\delta_p^\circ\,T$. Then one applies on this graph a greedy coloring algorithm. Finally it is clear that vertices that have the same colour can have the same data structure in their lhs. In our running example, one finds that $S1$ and $S2$ have the same colour in the interference graph. It means that $S1$ and $S2$ can share the same data structure. $S3$ must have a specific data structure. One just has to reconstruct the data flow. Then the program can be parallelized. Its translation in Fortran 90 after partial expansion is:

```
      program matrix-vector
      real s(n), a(n,n), b(n), c(n)
      integer i,j,n
      do t=0,n+1
        if (t .EQ. 0) then
S1        s(1:n:1) = 0.
        end if
        if (t .GE. 1 .AND. t .LE. n)
S2        s(1:n:1) = s(1:n:1) + a(1:n:1,t)*b(t)
        end if
        if (t .EQ. n+1)
S3        c(1:n:1) = s(1:n:1)
        end if
      end do
      end
```

4 Conclusion

Our aim has been reached, our method can effectively reduce the memory cost in the data expansion process of static control programs. In our running example

the expansion is limited to an expansion of the scalar s in an one-dimensionnal array with n elements. Notice that if one builds a schedule function equivalent to the sequential execution order, one finds as final structure the scalar s and the array c. It means that if the source program is provided in single assignment form for instance, then our method reduces the two arrays in the lhs of $S1$ and $S2$ to a single scalar. We have then obtained an important result: our method can reduce the original data size of the program if the memory requirement necessary for the schedule function is less than the original data size. Our method can be used now to reduce data space of program directly provided in single assignment form.

References

1. D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. *Array data-flow analysis and its use in array privatization.* In Principles of Programming Languages, 1993.
2. P.Y Calland, A. Darte, Y. Robert, F. Vivien. *On the removal of anti and output dependences.* Technical report RR96-04, laboratoire LIP - école normale supérieure de Lyon - Feb 1996.
3. Z. Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* Thèse de l'université de Rennes I - chapitre IV - 1993, numéro d'ordre 957.
4. C. Eisenbeis, W. Jalby, D. Windheiser, and F. Bodin. *A strategy for array management in local memory.* In Proc. of the 4th Worlshop on Languages and Compilers for Parallel Computing, Aug. 1991.
5. P. Feautrier. *Dataflow Analysis of Array and Scalar References.* Int. J. of Parallel Programming, 20(1):23-53, February 1991.
6. P. Feautrier. *Some efficient solutions to the affine scheduling problem part II : multidimensional time.* Int J. of Parallel Programming, 21(6):389-420, December 92.
7. Z. Li, G. and G. Lee. *Symbolic array dataflow analysis for array privatization and program parallelization.* In Supercomputing 95, 1995.
8. V. Lefebvre and P. Feautrier. *Storage Management in Parallel Programs.* In Proc. of the Fith Euromicro Workshop on Parallel and Distributed Processing Conf, Pages 181-188. London. Jan 1997.
9. P. Tu and D. Padua. *Array privatization for shared and distributed memory machines.* In Proc. Third Workshop on Languages and Compilers for Distributed Memory Machines, Boulder, Colorado 1992.
10. S. Rajopadhye and D. Wilde. *Memory Reuse Analysis in the Polyhedral Model.* In Bougé, Fraignaud, Mignotte and Robert, editors, Euro-Par'96 Parallel Processing, Vol I, pages 389-397. Springer-Verlag, LNCS 1123, Aug 1996.
11. M. Wolf and M. Lam. *A data locality optimizing algorithm.* In Proc. ACM SIGPLAN 91 Conf. on Programming Language Design and Implementation, June 1991.

This article was processed using the LaTeX macro package with LLNCS style