On the removal of anti and output dependences

Pierre-Yves Calland Alain Darte Yves Robert
Frédéric Vivien*

Laboratoire LIP, URA CNRS 1398
Ecole Normale Supérieure de Lyon, F - 69364 LYON Cedex 07
e-mail: Firstname.Lastname@lip.ens-lyon.fr

Abstract

In this paper we build upon results of Padua and Wolfe [9], who introduce two graph transformations to eliminate anti and output dependences. We first give a unified framework for such transformations. Then, given a loop nest, we aim at determining which statements should be transformed so as to break artificial cycles involving anti or output dependences. The problem of finding the mininum number of statements to be transformed is shown to be NP-complete in the strong sense, and we propose two efficient heuristics.

Key-words: node splitting, anti dependences, output dependences, dependence graph, NP-completeness, heuristics.

1: Introduction

Flow dependences are the only "true" dependences of a program. Anti dependences and output dependences are due to storage re-use and can be eliminated at the price of more memory usage. Removing anti and output dependences may prove very useful to break data dependence cycles and thereby enabling vectorization and/or improving parallelization.

Many papers have been devoted to the problem of eliminating anti and output dependences. Proposed methods include "array data flow analysis" [5, 8], "array privatization" [7], "variable expansion" [3], "variable renaming" [9] and "node splitting" [9]. See the survey papers of Banerjee, Eigenmann, Nicolau and Padua [2] and Bacon, Graham and Sharp [1], as well as the books of Wolfe [15] and Zima [16], for further references.

In this paper we build upon results of Padua and Wolfe [9], who introduce two graph transformations to eliminate anti and output dependences. We first give a unified framework for such transformations in Section 2. Then, given a loop nest, we aim at determining which statements should be transformed so as to break artificial cycles involving anti or output dependences. The problem of finding the mininum number of statements to be transformed is shown to be difficult: in Section 3, we prove it NP-complete in the strong sense. This justifies the introduction of heuristics in Section 4. Finally, we give some conclusions in Section 5.

*Supported by the CNRS-INRIA Project *ReMaP* and by the Eureka project *EuroTOPS*. Pierre-Yves Calland is supported by a grant of Région Rhône-Alpes.

1063-6862/96 $5.00 © 1996 IEEE

2: Graph transformations

2.1: Two well-known elementary transformations

Padua and Wolfe [9] propose two transformations to break data dependence cycles in the presence of anti or output dependences. These transformations are best illustrated with the original examples of their paper.

Anti dependences

Consider the following loop, denoted as L_1:

$$\begin{array}{l} \textsf{for } i := 1 \textsf{ to } N \textsf{ do} \\ \quad S_1\text{: } a(i) := b(i) + c(i) \\ \quad S_2\text{: } d(i) := (a(i) + a(i+1))/2 \end{array}$$

There is a flow dependence from S_1 to S_2 because S_1 writes $a(i)$ and S_2 uses it immediately after. There is also an anti dependence from S_2 to S_1 because $a(i+1)$ must be read in S_2 before being written in S_1 at the next iteration. As a consequence, there is a data dependence cycle, as illustrated[1] in Figure 1(a).

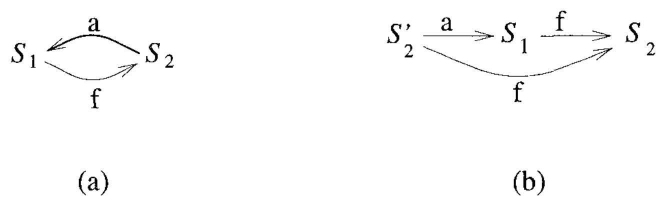


Figure 1. Dependence graph of loop nest L_1: (a) before transformation; (b) after transformation.

The cycle can be broken by inserting a new assignment to a compiler temporary array as follows:

$$\begin{array}{l} \textsf{for } i := 1 \textsf{ to } N \textsf{ do} \\ \quad S'_2\text{: } temp(i) := a(i+1) \\ \quad S_1\text{: } a(i) := b(i) + c(i) \\ \quad S_2\text{: } d(i) := (a(i) + temp(i))/2 \end{array}$$

There is now an extra dependence (the flow of the temporary from S'_2 to S_2) but the new dependence graph has no cycle (see Figure 1(b)). Therefore the new loop can be directly vectorized:

$$\begin{array}{l} S'_2\text{: } temp(1:N) := a(2:N+1) \\ S_1\text{: } a(1:N) := b(1:N) + c(1:N) \\ S_2\text{: } d(1:N) := (a(1:N) + temp(1:N))/2 \end{array}$$

Output dependences

In the presence of a data dependence cycle due to an output dependence, a similar transformation can be performed. Consider the following loop, denoted as L_2:

[1] In all figures, flow, anti and output dependence edges are labeled with a *"f"*, a *"a"* and a *"o"* respectively.

for $i := 1$ to N do
S_1: $a(i) := b(i) + c(i)$
S_2: $a(i+1) := a(i) + 2 \times d(i)$

There is an output dependence from S_2 to S_1 because $a(i+1)$ is written in S_2 before being re-written in S_1 at the next iteration. We still have a flow dependence from S_1 to S_2 because of $a(i)$, hence the dependence graph of Figure 2(a). Now, adding a temporary array leads to the following loop:

for $i := 1$ to N do
S_1': $temp(i) := b(i) + c(i)$
S_2: $a(i+1) := temp(i) + 2 \times d(i)$
S_1: $a(i) := temp(i)$

The new loop has no cycles (see Figure 2(b)) and therefore can be vectorized.

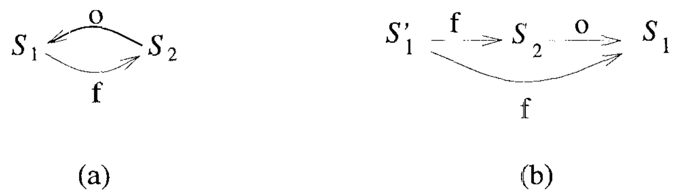


Figure 2. Dependence graph of loop nest L_2: (a) before transformation; (b) after transformation.

To summarize this section, we see that both transformations have broken a cycle in the dependence graph, thereby enabling vectorization and/or improving parallelization. Of course the price to pay is an increase in the memory requirements. In both cases, we have used an extra temporary array. In Section 2.2, we give a unified framework for generalizing Padua and Wolfe's transformations.

2.2: A unified transformation

We unify the two transformations of the previous section in a general setting. Then we identify the transformations induced on the dependence graph, and we formally state the problem of minimizing memory overhead when removing anti and output dependences.

2.2.1 Defining the transformation

Consider the following loop L_3:

for $i := 1$ to N do
...
S_k: $lhs(f(i)) = rhs(\ldots)$
...

and assume we want to remove some anti and output dependences due to the access to the array lhs[2] in statement S_k (say because there are cycles due to such dependences in the dependence graph). What would be the effect on the dependence graph of a transformation like:

[2] lhs and rhs stand for left-hand side and right-hand side respectively.

for $i := 1$ to N do

$\ldots$

S'_k: $temp(f(i)) = rhs(\ldots)$

S_k: $lhs(f(i)) = temp(f(i))$

$\ldots$

Note that we simply evaluate the right hand side into a new temporary array *temp* which we copy back to *lhs*. Of course, any access to an array element $lhs(g(i))$ that depends upon the value calculated in statement S'_k should be replaced by $temp(g(i))$. Thus we need to know what are the statement instances which depend upon the value calculated in statement S'_k, and we may have to rely on a powerful dependence analyzer such as Tiny [14], Petit [10], Partita [11], PAF [13] or PIPS [12] (to quote but a few).

Before going further, we point out that the loop nest can be multidimensional. Our discussion is presented for a single loop, but all results hold for several nested loops. The loop nest need not be perfect or uniform or whatever, what really matters is the availability of a good dependence analyzer capable of providing sources and sinks of all dependences. We do have a restriction, however: to perform our transformation we must assume that the access function f to the left-hand side array *lhs* is one-to-one. The reason for this is explained below in Section 2.2.2, when discussing self output dependences. Note that the two transformations of Padua and Wolfe have the same requirement.

2.2.2 Applying the transformation

Consider statement S_k in loop L_3. There can be flow, anti and output dependences going to or coming from S_k, hence six kinds of arrows[3] in the dependence graph (see Figure 3). We discuss hereafter the impact of our transformation on each of these arrows. Of course there is a new flow dependence f_{new} from S'_k to S_k. Note also that there is no self output loop on S'_k because we have supposed that the access functions to the left-hand side arrays are one-to-one.

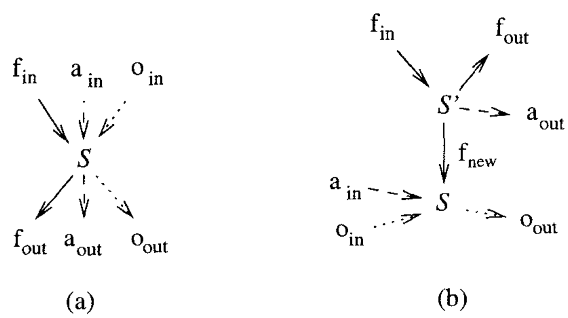


Figure 3. A statement S with in-coming and out-going dependences a) before and b) after transformation.

- **In-coming flow dependence** (Figure 4). One of the data read in the right-hand side of statement S_k was previously produced in the left-hand side of a statement

[3] In the figure f_{in} stands for an in-coming flow dependence, f_{out} stands for an out-going flow dependence, and so on.

S_l. After the transformation, the data is read in the right-hand side of statement S'_k. Thus there is a flow dependence from S_l to S'_k.

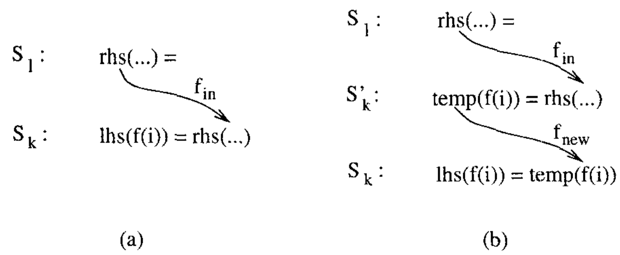


Figure 4. In-coming flow dependence a) before and b) after transformation.

In-coming anti dependence (Figure 5). A statement S_l reads $lhs(f(i))$ before S_k writes it. After the transformation, $lhs(f(i))$ is still read by S_l and is still written by S_k. Thus, the anti-dependence from S_l to S_k is left unchanged.

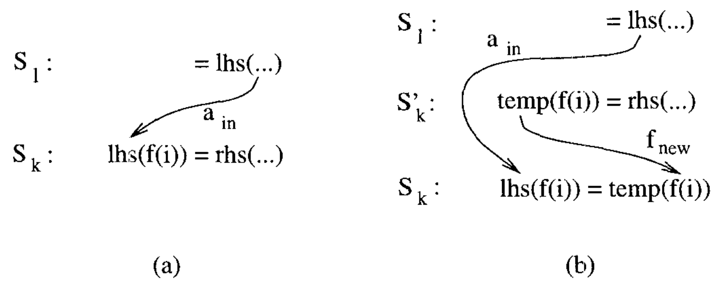
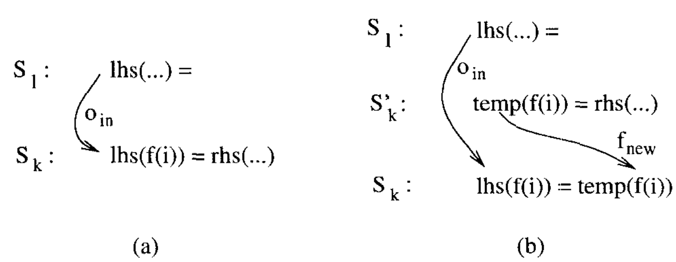


Figure 5. In-coming anti dependence a) before and b) after transformation.

In-coming output dependence (Figure 6). A statement S_l writes $lhs(f(i))$ before S_k writes it. After the transformation $lhs(f(i))$ is still written by S_l and by S_k. So, there is an output dependence from S_l to S_k.



gure 6. In-coming output dependence a) before and b) after transformation.

Out-going flow dependence (Figure 7). A statement S_l reads the value of $lhs(f(i))$ produced by S_k. Thus the access to lhs in S_l, denoted $lhs(g(i))$, was replaced by an access to *temp*, denoted $temp(g(i))$. Now, as $temp(f(i))$ is written by S'_k, there is a flow dependence from S'_k to S_l.

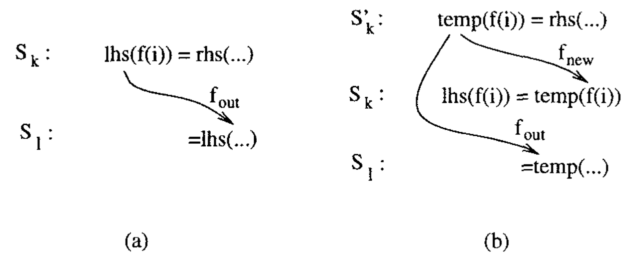


Figure 7. Out-going flow dependence a) before and b) after transformation.

- **Out-going anti dependence** (Figure 8). One of the data read in the right-hand side of statement S_k is written afterwards in a statement S_l. After the transformation this data is read in the right-hand side of statement S'_k. Thus there is an anti dependence from S'_k to S_l.

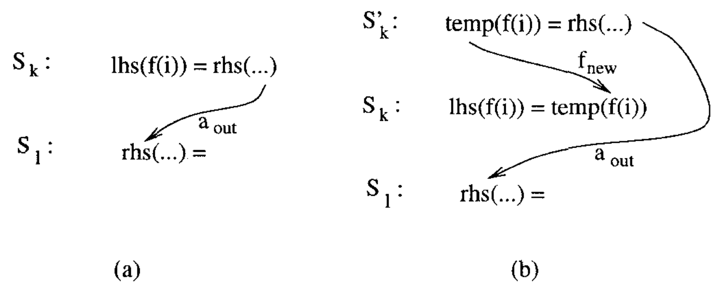


Figure 8. Out-going anti dependence a) before and b) after transformation.

- **Out-going output dependence** (Figure 9). A statement S_l writes $lhs(f(i))$ after S_k writes it. After the transformation, $lhs(f(i))$ is still written by S_l and by S_k. Thus, there is an output dependence from S_k to S_l.

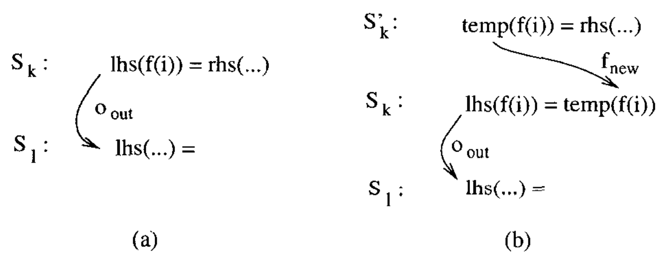


Figure 9. Out-going output dependence a) before and b) after transformation.

All these results are summarized in Figure 3. Note that self loops are processed as the other edges: the less obvious case is a self anti-dependence loop on statement S_k; since it comes from S_k and goes to S_k, it will be replaced by an anti dependence edge coming from S'_k and going to S_k.

Next we show the usefulness of our transformation. If we transform all vertices of a dependence graph, then the only cycles that may remain are pure flow dependence cycles (only made up with edges labeled f) or pure output dependence cycles (only made up with edges labeled o). See [4] for a proof of the following result:

Theorem 1 *Let G be the dependence graph of a loop nest L, and let G' be the graph obtained from G by transforming all its nodes. Then a cycle C of G' is only composed of flow dependences or is only composed of output dependences. Furthermore, C corresponds to a cycle that was already a cycle of G.*

In other words, pure flow dependence cycles and pure output dependence cycles are not broken when transforming all vertices. But if the original dependence graph contains no such cycles, then the transformed graph is *acyclic*. In fact, from the point of view of breaking cycles, the transformation of a given vertex v may be useful only if it has an incoming anti or output dependence edge, and an outgoing flow or anti dependence edge (see Figure 3 again). We can summarize this discussion by the following schema:

$$? \xrightarrow{a,o} v \xrightarrow{f,a} ?$$

These are the only paths that are broken by applying our transformation to vertex v.

Determining the minimum number of vertices to transform (i.e. the minimum number of temporary arrays to use) so that the new dependence graph has only pure flow dependence cycles and pure output dependence cycles turns out to be a difficult problem. Before stating this formally, we work out an example, so as to illustrate our transformation and the heuristics introduced later.

2.3: Target example

Consider the following loop nest L_4:

for $i := 4$ to N do
S_1: $a(i+5) := c(i-3) + b(2i+2)$
S_2: $b(2i) := a(i-1) + 1$
S_3: $a(i) := c(i+5) - 1$
S_4: $c(i) := b(2i-4)$

The dependence graph is represented in Figure 10. There are six dependences in the loop:

3 **flow dependences** from S_3 to S_2 (because of array a), from S_2 to S_4 (because of array b), and from S_4 to S_1 (because of array c),

2 **anti dependences** from S_1 to S_2 (because of array b) and from S_3 to S_4 (because of array c),

1 **output dependence** from S_1 to S_3 (because of array a).

Note that Tiny [14] does find the six dependences listed above (see Table 1). In fact Tiny finds a seventh dependence (the second one in Table 1), but recognizes that this dependence is killed. Indeed, we might have found a flow dependence from S_1 to S_2 because $a(i+5)$ is written in $S_1(i)$ (the i-th instance of S_1) and used in $S_2(i+6)$. But meanwhile, $a(i+5)$

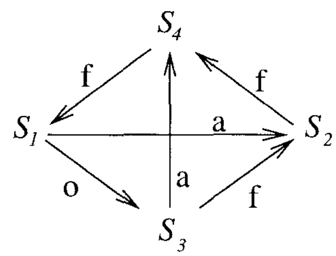


Figure 10. The target dependence graph before transformation.

anti	S_1 $b(2i+2)$	$\longrightarrow$	S_2	$b(2i))$	
flow	S_1 $a(i+5)$	$\longrightarrow$	S_2	$a(i-1))$	[killed]
output	S_1 $a(i+5)$	$\longrightarrow$	S_3	$a(i))$	
flow	S_2 $b(2i))$	$\longrightarrow$	S_4	$b(2i-4)$	
anti	S_3 $c(i+5)$	$\longrightarrow$	S_4	$c(i)$	
flow	S_3 $a(i)$	$\longrightarrow$	S_2	$a(i-1)$	
flow	S_4 $c(i)$	$\longrightarrow$	S_1	$c(i-3)$	

Table 1. The dependences found by Tiny.

is re-written in $S_3(i+5)$, and this new value is used in $S_2(i+6)$, hence the source of the dependence for using $a(i+5)$ in $S_2(i)$ is $S_3(i+5)$ rather than $S_1(i)$. In other words, this flow dependence is overlapped by the succession of the output dependence from S_1 to S_3 and of the flow dependence from S_3 to S_2. It turns out, in our example, that it is of tremendous importance to have an accurate dependence analyzer capable of detecting that this seventh dependence is a false one. Otherwise we would have considered that there is a pure flow dependence cycle in the dependence graph !

Consider the effect of transforming vertices S_2 and S_3 in the dependence graph. The new graph G' is represented in Figure 11.

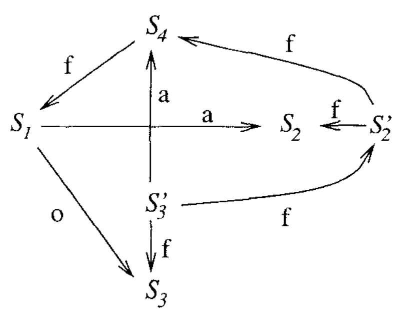


Figure 11. The target dependence graph after transforming S_2 and S_3.

To illustrate the impact of transforming vertices S_2 and S_3, we can rewrite the loop using the two temporary arrays `a-temp` (introduced to transform S_3) and `b-temp` (introduced to transform S_2):

for $i := 4$ to N do

S_1: $a(i+5) := c(i-3) + b(2i+2)$

S_2': b-temp$(2i) := \begin{cases} \text{if } i \geq 5 \text{ then a-temp}(i-1)+1 \\ \text{else } a(i-1)+1 \end{cases}$

S_2: $b(2i) :=$ b-temp$(2i)$

S_3': a-temp$(i) := c(i+5) - 1$

S_3: $a(i) :=$ a-temp(i)

S_4: $c(i) := \begin{cases} \text{if } i \geq 6 \text{ then b-temp}(2i-4) \\ \text{else } b(2i-4) \end{cases}$

Note that conditional statements are required to process dependences coming from several sources.

3: NP-completeness

In this section we prove that the problem of determining the minimal number of statements to split with our transformation is NP-hard. First, we formally state the problem and then we prove that the associated decision problem is NP-complete by reduction from the 3-SAT satisfiability problem. This theoretical result states the complexity of the problem and motivates the search for efficient heuristics (see Section 4). Due to lack of space, all proofs are omitted (see [4]). However, we point out that in the proof we use loop nests with anti dependences only. Even with this simple assumption, the problem still exhibits hard complexity.

3.1: Problem statement

Let $G = (V, E, \ell)$ be the dependence graph of a loop nest L. Vertices represent statements. Edges represent dependences between statements. The label of an edge is given by the function $\ell : E \longrightarrow \{f, a, o\}$ (flow, anti or output dependence). Our problem is to determine the minimum number of statements which we should transform using the transformation of Figure 3 so that there remains only pure flow dependence cycles and pure output dependence cycles. The associated decision problem can be stated as follows:

Definition 1 *Given a loop nest L (and its dependence graph $G = (V, E, \ell)$) and a non-negative integer bound K, can we find $k \leq K$ vertices of G such that transforming these k vertices leads to a graph G' where there remains only pure flow dependence cycles and pure output dependence cycles? (if the answer is yes, we say that $L \in$ PURE-CYCL(K)).*

Theorem 2 *PURE-CYCL is NP-complete (in the strong sense).*

4: Heuristics

In this section we briefly sketch some heuristics to find out which vertices of the dependence graph $G = (V, E)$ of a loop nest should be transformed so that there remains only pure cycles in G'. We give two heuristics, both quite natural. The first one might be very expensive in the worst case, but could be of interest for small dependence graphs. The second one always requires a polynomial time. It runs in time $O(t^2(|V| + |E|))$, where t is the number of transformed vertices, hence a worst case bound $O(|V|^2(|V| + |E|))$.

4.1: A heuristic based on the hypergraph of the cycles of G

The most natural heuristic is to build the hypergraph $H = (V, F)$ of the cycles of G. F is defined as a collection of subsets $\{ \subset V$, where each $\{$ is the set of the vertices of an elementary cycle C of G. See Figure 12 for the hypergraph H of our target example. Furthermore each vertex v in $\{$ is marked *breakable* if C is broken when v is transformed, i.e. v is marked *breakable* if the in-coming edge of v in C is an anti or output dependence, and the out-coming edge a flow or anti dependence.

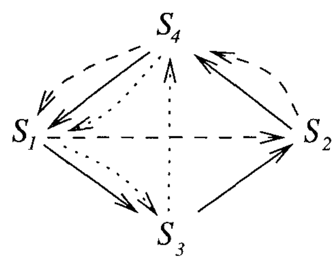


Figure 12. Hypergraph of the target example. The three elementary cycles are shown with different arrow formats.

Once H is built, we have to transform one breakable vertex per cycle, which is related to the NP-complete hitting set problem [6, problem SP8]. Therefore we apply a greedy strategy and transform the vertex v_0 which belongs to, and is *breakable* for, the maximal number of subsets $\{ \in F$. We delete all cycles that are going through v_0 and for which v_0 is breakable. We redo the operation until there remains no cycle in the graph with *breakable* vertices[4].

The drawback of this heuristic is its high cost in the worst case. The number of cycles can be exponential in the size $O(|V| + |E|)$ of the graph, and the construction of H might therefore have a very high cost.

The heuristic applied to the target example

We show here the transformation of the target example of Section 2.3 using this heuristic. Figure 13 shows the hypergraph corresponding to the dependence graph of Figure 10. The table below (Figure 14) shows for each vertex how many elementary cycles include it as a *breakable* vertex.

According to this table, the heuristic first transforms node S_3. The hypergraph of the new graph is shown in Figure 13. As the hypergraph still contains *breakable* nodes, and as S_2 is the only *breakable* node, the heuristic transform S_2 and stops. We obtain the same result as in Section 2.3 (see Figure 11).

4.2: A polynomial-time heuristic

Transforming a vertex may be useful only if the corresponding statement has an incoming anti or output dependence, and an outgoing flow or anti dependence. For each vertex v of

[4] Here is a small improvement: search whether there exists a subset $\{ \in F$ which contains a single *breakable* vertex v; if such a vertex exists then transform it (because we have to break it later on anyway to delete the cycle); else search a vertex which belongs to and is *breakable* for the maximal number of subsets $\{ \in F$.

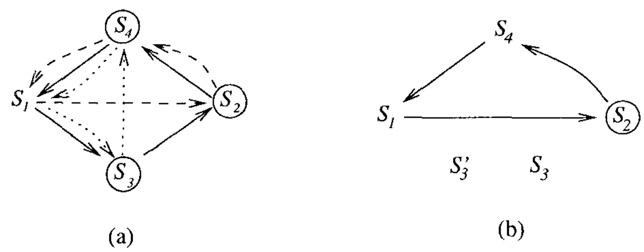


Figure 13. Hypergraph of the target example a) before transformation and b) after transformation of node S_3.

S_1	S_2	S_3	S_4
0	1	2	1

Figure 14. Number of circuits which include a vertex as *breakable*.

the dependence graph we can count its "utility", i.e. the number *Util*(v) of pairs (e_{in}, e_{out}) such that:

1. $e_{in} \in E$, $e_{in} : ? \longrightarrow v$ and $\ell(e_{in}) \in \{a, o\}$
2. $e_{out} \in E$, $e_{out} : v \longrightarrow ?$ and $\ell(e_{out}) \in \{f, a\}$

We transform one of the vertices v such that *Util*(v) is maximal. We obtain a graph G'. We remove from G' all the edges which are not in a strongly connected component. If there is at least one anti dependence edge in G' or if G' has an elementary circuit which contains both an output dependence edge and a flow dependence edge, we apply recursively the heuristic on G'.

The strongly connected components of G' can be built in $O(|V| + |E|)$. To check the presence of an elementary circuit which contains an output dependence edge and a flow dependence edge, we consider a vertex v with an incoming output dependence and an outgoing flow dependence. If there is a path from a vertex reached by an outgoing flow dependence of v to a vertex from which starts an incoming output dependence of v, and if this path does not include v, then G' contains at least one non pure circuit. One can check the existence of such a path in one "smart" graph traversal, and thus in time $O(|V| + |E|)$. As there are $|V|$ nodes, the total complexity of this circuit checking is $O(|V|(|V| + |E|))$.

In the worst case, all nodes will be transformed and the heuristic complexity is $O(|V|^2(|V| + |E|))$.

The polynomial-time heuristic on the target example

We show here the processing of the target example of Section 2.3 by the polynomial heuristic. The table below shows the value of *Util* for each of the graph vertices.

	S_1	S_2	S_3	S_4
Util(v)	0	1	2	1

Once again, S_3 is transformed first. The new graph has one strongly connected component with an anti dependence (from S_1 to S_2): the heuristic is applied once again. The new value of *Util* is then:

	S_1	S_2	S_4
Util(v)	0	1	0

Thus the polynomial-time heuristic transforms S_2. We retrieve the same result as before.

5: Conclusion

In this paper we have formalized Padua and Wolfe's transformation [9], to eliminate anti and output dependences. We have stated a complexity result that shows the difficulty of the problem, even in the restricted framework that we have considered.

Note that we have dealt with transformations which increase memory requirements only by a factor proportional to the number of statements. In the general case we also aim at suppressing output dependence cycles, which may require array expansions, thus changing the order of magnitude for the memory requirements: e.g. for a single loop with k statements, we might go from $O(k \times N)$ memory units to $O(k \times N^2)$. Further work will be devoted to the systematic study of such transformations.

References

[1] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4), 1994.

[2] U. Banerjee, R. Eigenmann, A. Nicolau, and D.A. Padua. Automatic program parallelization. *Proceedings of the IEEE*, 81(2):211–243, 1993.

[3] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *International Conference of Supercomputing*, pages 407–417, 1988.

[4] Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien. On the removal of anti and output dependences. Technical Report 2800, INRIA, France, February 1996.

[5] Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–51, 1991.

[6] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1991.

[7] Junjie Gu, Zhiyan Li, and Gyungho Lee. Symbolic array dataflow analysis for array privatization and program parallelization. In *Supercomputing 95*, 1995.

[8] Dror E. Maydan, Saman P. Amarasinghe, and Monica Lam. Array data-flow analysis and its use in array privatization. In *Principles of Programming Languages*, 1993.

[9] David A. Padua and Michael J. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[10] William Pugh. Release 0.96 of petit. World Wide Web document, URL: `http://www.cs.umd.edu/projects/omega/petit.html`.

[11] SIMULOG S.A. *FORESYS, Manuel de Référence*, April 1994.

[12] PIPS Team. Pips (interprocedural parallelizer for scientific programs). World Wide Web document, URL: `http://www.cri.ensmp.fr/~pips/index.html`.

[13] PRiSM SCPDP Team. Systematic construction of parallel and distributed programs. World Wide Web document, URL: `http://www.prism.uvsq.fr/english/parallel/paf/autom_us.html`.

[14] Michael Wolfe. The Tiny loop restructuring research tool. In H.D. Schwetman, editor, *International Conference on Parallel Processing*, volume II, pages 46–53. CRC Press, 1991.

[15] Michael Wolfe. *High Performance Compilers For Parallel Computing.* Addison-Wesley Publishing Company, 1996.

[16] Hans Zima and Barbara Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1990.