International Journal of Parallel Programming, Vol. 26, No. 3, 1998

On the Removal of Anti- and Output-Dependences[1]

Pierre-Yves Calland,[2] Alain Darte,[2] Yves Robert,[2] and Frédéric Vivien[2]

In this paper we build upon results of Padua and Wolfe,[1] who introduced two graph transformations to break dependence paths including anti- and output-dependences. We first formalize these two transformations. Then, given a loop nest, we aim at determining which statements should be transformed so as to break artificial dependence paths involving anti- or output-dependences. The problem of finding the minimum number of statements to be transformed is shown to be NP-complete, 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,"[2, 3] "array privatization,"[4] "variable expansion,"[5] "variable renaming" and "node splitting."[1] See the survey papers of Banerjee

[1] Supported by the CNRS-ENS Lyon-INRIA Project *ReMaP*. Pierre-Yves Calland is supported by a grant of Région Rhône-Alpes.

[2] Laboratoire LIP, URA CNRS 1398, École Normale Supérieure de Lyon, F-69364 Lyon Cedex 07. E-mail: {Firstname.Lastname}@lip.ens-lyon.fr.

et al.,[6] and Bacon *et al.*,[7] as well as the books of Wolfe,[8] and Zima,[9] for further references.

Removing *all* memory-based or "false" (i.e., anti- and output-) dependences may have a prohibitive cost.[10] A complete removal of false dependences is usually achieved, if feasible, via conversion of the original program into single assignment form. This turns out to be unnecessarily costly. Indeed, there are some memory-based dependences whose removal will not improve the parallelization.

In this paper we build upon results of Padua and Wolfe,[1] who introduce two graph transformations to break dependence paths including anti- and output-dependences. Our motivation is threefold:

- formalize Padua and Wolfe's transformations (which were only stated through examples in their original paper);
- analyze their potential to break dependence paths or cycles;
- identify the minimum number of transformations required to break all "spurious" dependence cycles (see Sections 3 and 4 for a more precise statement).

The rest of the paper is organized as follows. In Section 2 we formally define and study these two transformations. Then, given a loop nest, we aim at determining which statements should be transformed so as to break artificial paths involving anti- or output-dependences. In Section 3 we prove that Padua and Wolfe's transformations can be repeatedly applied to break all "spurious" dependence cycles. The price to pay is some extra memory overhead (a new temporary array) each time a transformation is applied to a statement. Unfortunately, the problem of finding the *minimum* number of statements to be transformed is shown to be difficult: in Section 4, we prove it is NP-complete in the strong sense. This justifies the introduction of heuristics in Section 5. Finally, we give some conclusions in Section 6.

2. GRAPH TRANSFORMATIONS

2.1. Two Well-Known Elementary Transformations

Padua and Wolfe[1] 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.

2.1.1. Anti Dependences

Example 1. Consider Fig. 1. 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

```
For i := 1 to N do
  $S_1$: a(i) := b(i) + c(i)
  $S_2$: d(i) := (a(i) + a(i+1)) / 2
```

$$\begin{tikzcd}S_1 \arrow[r, bend right, "f"'] & S_2 \arrow[l, bend right, "a"']\end{tikzcd}$$

Fig. 1. Example 1 and its dependence graph *before* transformation.

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 in Fig. 1. [Note: In all figures, flow-, anti- and output-dependences edges are labeled with a "*f*," a "*a*" and a "*o*" respectively.] This cycle can be broken by inserting a new assignment to a compiler temporary array as shown in Fig. 2.

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 Fig. 2). Therefore the new loop can be directly vectorized:

```
$S'_2$: temp(1:N) := a(2:N+1)
$S_1$: a(1:N) := b(1:N) + c(1:N)
$S_2$: d(1:N) := (a(1:N) + temp(1:N)) / 2
```

2.1.2. Output Dependences

In the presence of a data dependence cycle due to an output-dependence, a similar transformation can be performed.

Example 2. Consider Fig. 3. There is an output-dependence from S_2 to S_1 because a(i+ 1) is written in S_2 before being rewritten 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 Fig. 3. Now, adding a temporary array leads to the program in Fig. 4. The new loop has no cycles (see Fig. 4) and therefore can be vectorized.

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

```
For i := 1 to N do
  $S'_2$: temp(i) := a(i+1)
  $S_1$: a(i) := b(i) + c(i)
  $S_2$: d(i) := (a(i) + temp(i)) / 2
```

$$\begin{tikzcd}S'_2 \arrow[r, "a"] \arrow[rr, bend right, "f"'] & S_1 \arrow[r, "f"] & S_2\end{tikzcd}$$

Fig. 2. Example 1 and its dependence graph *after* transformation.

```
For i := 1 to N do
  $S_1$: a(i) := b(i) + c(i)
  $S_2$: a(i+1) := a(i) + 2 × d(i)
```

$$\begin{tikzcd} S_1 \arrow[r, bend right, "f"'] & S_2 \arrow[l, bend right, "o"'] \end{tikzcd}$$

Fig. 3. Example 2 and its dependence graph *before* transformation.

in the memory requirements. In both cases, we have used an extra temporary array.

Padua and Wolfe[1] have described their transformations only through an example. In Sections 2.2 and 2.3, we formally define both transformations, and we discuss their respective usefulness.

2.2. The First Transformation

2.2.1. Defining the Transformation

Padua and Wolfe have transformed Example 1 by introducing a temporary array to store the value contained in the memory location that generates the anti-dependence that prevents vectorization.

The access to $a(i+1)$ in statement S_2 of Example 1 is an access to a precise memory location and the reading of a value which is necessary to the computation. There is absolutely no need to access the memory location $a(i+1)$ in S_2: one only needs to access a memory location where has already been stored a copy of (the value stored in) $a(i+1)$. Therefore, Padua and Wolfe have split the access to $a(i+1)$ into two parts: in the new statement S'_2 they access the memory location $a(i+1)$, and copy the value stored there in another memory location, using a temporary array; then in the original statement S_2 they replace the access to the memory location $a(i+1)$ by the access to the temporary array. So to speak, they have separated the access to the memory location and the subsequent use of the read value.

We formally define Padua and Wolfe's first transformation in Fig. 5. The transformation [*lhs* and *rhs* stands for "left-hand side" and "right-hand side" respectively] is applied on any read access in the statement, namely

```
For i := 1 to N do
  $S'_1$: temp(i) := b(i) + c(i)
  $S_2$: a(i+1) := temp(i) + 2 × d(i)
  $S_1$: a(i) := temp(i)
```

$$\begin{tikzcd} S'_1 \arrow[r, "f"] \arrow[rr, bend right, "f"'] & S_2 \arrow[r, "o"] & S_1 \end{tikzcd}$$

Fig. 4. Example 2 and its dependence graph *after* transformation.

```
for i := 1 to N do
   ...
   S_k: lhs(f(i)) = ... rhs(g(i)) ...
   ...
```

```
for i := 1 to N do
   ...
   S'_k: temp(i) = rhs(g(i))
   S_k: lhs(f(i)) = ... temp(i) ...
   ...
```

Fig. 5. Formal definition of Padua and Wolfe's first transformation.

here the access to $rhs(g(i))$. A new statement and a temporary array are introduced. The temporary array is a new array which is used nowhere else in the program. [Note that we can also apply the transformation to a read access in the left-hand side of statement S_k, i.e., to a read access inside the function $f(i)$ itself.]

Before going further, we point out that this transformation can be applied to a multidimensional loop nest. Our discussion is presented for a single loop, but all results extend to several (possibly non perfectly) nested loops.

2.2.2. Applying the Transformation

The goal of this transformation is to break some dependence paths which go through statement S_k. Indeed, consider S_k before the transformation (Fig. 5). There can be flow-, anti- and output-dependences coming into or going out of S_k, hence six kinds of dependence edges in the dependence graph [In Fig. 8 f_{in} stands for an incoming flow-dependence, f_{out} stands for an outgoing flow-dependence, and so on.] Clearly, since we have manipulated the right-hand side of S_k, the transformation may only modify incoming flow- and outgoing anti-dependences. We have to discuss the impact of the transformation on such dependence edges. Of course there is a new flow-dependence f_{new} from S'_k to S_k.

- *Incoming flow-dependence* (Fig. 6). Data item $rhs(g(i))$ read in statement S_k was previously produced in the left-hand side of another statement S_l. After the transformation, this data item is read in the right-hand side of statement S'_k. Thus there is a flow-dependence from S_l to S'_k.

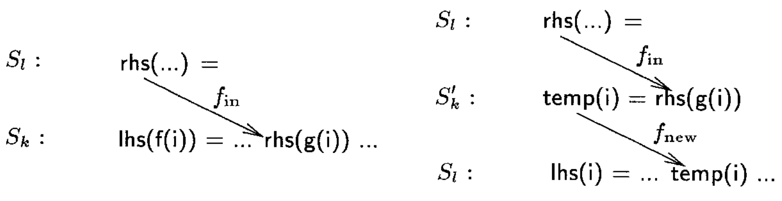


Fig. 6. Incoming flow-dependence before and after transformation.

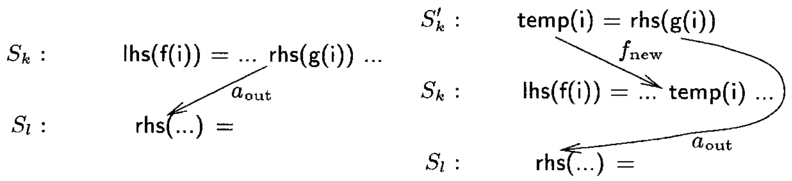


Fig. 7. Outgoing anti-dependence before and after transformation.

- *Outgoing anti-dependence* (Fig. 7). Data item *rhs*($g(i)$) read in the right-hand side of statement S_k is later written in another statement S_l. After the transformation, this data item is read in the right-hand side of statement S'_k. Thus there is an anti-dependence from S'_k to S_l.

Note that incoming flow- or outgoing anti-dependences that are *not* related to *rhs*($g(i)$) (but to another data access in the right-hand side of S_k) are not modified by the transformation. For instance, there is an incoming flow-dependence to S_2 in Example 1. But this dependence is not due to $a(i+1)$, which is the target of the transformation; rather it is due to $a(i)$, which is not concerned by the transformation: hence this incoming flow-dependence from S_1 to S_2 is not modified by the transformation.

The impact of the transformation is summarized in Fig. 8. Self loops are processed as the other edges. Consider for instance 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.

From the point of view of breaking paths, the transformation of a given statement S may be useful if it has an incoming anti- or output-dependence edge, and an outgoing anti-dependence edge (see Fig. 8 again):

$$? \xrightarrow{a,\,o} S \xrightarrow{a} ?$$

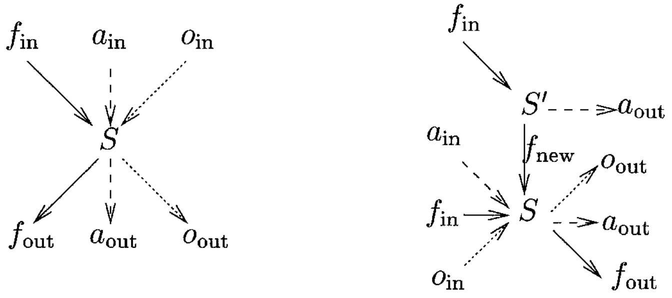


Fig. 8. A statement S with incoming and outgoing dependences, before and after Padua and Wolfe's first transformation.

The transformation is also useful if S has an incoming flow-dependence and an outgoing anti-dependence, *provided that* the two dependences are due to two different read accesses in S. Again, this is illustrated by Example 1: the incoming flow-dependence in S_2 is due to accessing $a(i)$, while the outgoing anti-dependence is due to $a(i+1)$.

2.3. The second transformation

We formalize Padua and Wolfe's second transformation, and we study its consequences on dependence graphs. Then we discuss in detail the scope and applicability of the transformation.

2.3.1. Defining the Transformation

Consider the following loop:

```
For i := 1 to N do
   ...
   $S_k$: lhs(f(i)) = rhs( ... )
   ...
```

and assume we want to remove some anti- and output-dependences due to the access to the array *lhs* 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:

```
For i := 1 to N do
   ...
   $S'_k$: temp(f(i)) = rhs( ... )
   $S_k$: lhs(f(i)) = temp(f(i))
   ...
```

Note that we simply evaluate the right-hand side into a new temporary array *temp* which we copy back to *lhs*. We replace any access to an array element $lhs(g(i))$ that depends upon the value calculated in statement S'_k by an access $temp(g(i))$. To be able to do so, we need to know what are the statement instances which depend upon the value calculated in statement S'_k, hence we have to rely on a powerful dependence analyzer such as Tiny,(11) Petit,(12) Partita,(13) PAF,(14) or PIPS(15)(to quote but a few).

2.3.2. Applying the Transformation

As before, we have to describe the impact of the transformation on each type of incoming and outgoing dependence edges. Consider statement S_k:

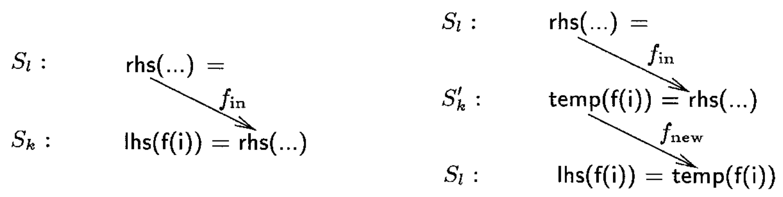
there are six kinds of dependence arrows (see Fig. 15), which we discuss successively later. Of course there is a new flow-dependence f_{new} from S'_k to S_k.

- *Incoming flow-dependence* (Fig. 9). 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 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.
- *Incoming anti-dependence* (Fig. 10). 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.
- *Incoming output-dependence* (Fig. 11). 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.
- *Outgoing flow-dependence* (Fig. 12). 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.
- *Outgoing anti-dependence* (Fig. 13). 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.
- *Outgoing output-dependence* (Fig. 14). 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.

The impact of the transformation is summarized in Fig. 15. From the point of view of breaking paths, the transformation of a given statement S may be useful if it has an incoming anti- or output-dependence edge, and an outgoing flow- or anti-dependence edge (see Fig. 15 again):

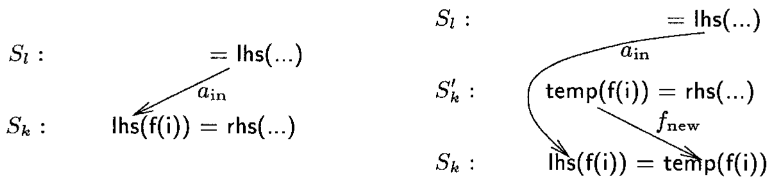
$$? \xrightarrow{a,\,o} S \xrightarrow{f,\,a} ?$$

We come back to the usefulness of the transformation in Section 3. Beforehand, we refine our analysis of its applicability.



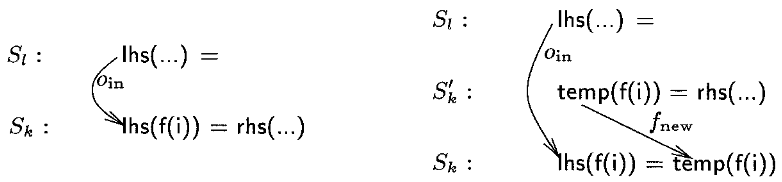
S_l : rhs(...) =
f_{in}
S_k : lhs(f(i)) = rhs(...)
S_l : rhs(...) =
f_{in}
S'_k : temp(f(i)) = rhs(...)
f_{new}
S_l : lhs(f(i)) = temp(f(i))

Fig. 9. Incoming flow-dependence before and after transformation.



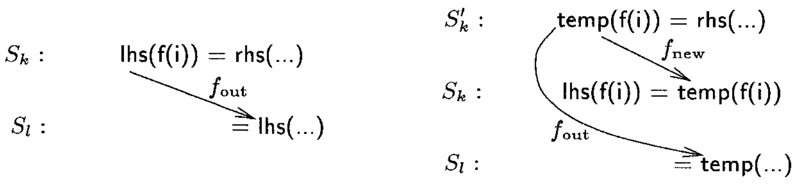
S_l : = lhs(...)
a_{in}
S_k : lhs(f(i)) = rhs(...)
S_l : = lhs(...)
a_{in}
S'_k : temp(f(i)) = rhs(...)
f_{new}
S_k : lhs(f(i)) = temp(f(i))

Fig. 10. Incoming anti-dependence before and after transformation.



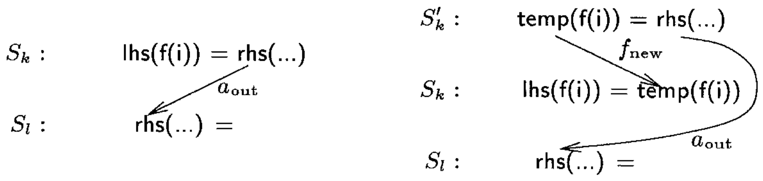
S_l : lhs(...) =
o_{in}
S_k : lhs(f(i)) = rhs(...)
S_l : lhs(...) =
o_{in}
S'_k : temp(f(i)) = rhs(...)
f_{new}
S_k : lhs(f(i)) = temp(f(i))

Fig. 11. Incoming output-dependence before and after transformation.



S_k : lhs(f(i)) = rhs(...)
f_{out}
S_l : = lhs(...)
S'_k : temp(f(i)) = rhs(...)
f_{new}
S_k : lhs(f(i)) = temp(f(i))
f_{out}
S_l : = temp(...)

Fig. 12. Outgoing flow-dependence before and after transformation.



S_k : lhs(f(i)) = rhs(...)
a_{out}
S_l : rhs(...) =
S'_k : temp(f(i)) = rhs(...)
f_{new}
S_k : lhs(f(i)) = temp(f(i))
a_{out}
S_l : rhs(...) =

Fig. 13. Outgoing anti-dependence before and after transformation.

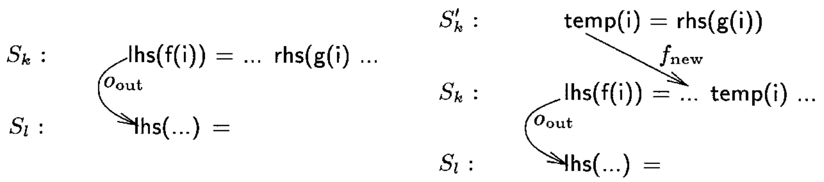

S_k : lhs(f(i)) = ... rhs(g(i) ...
o_{out}
S_l : lhs(...) =
S'_k : temp(i) = rhs(g(i))
f_{new}
S_k : lhs(f(i)) = ... temp(i) ...
o_{out}
S_l : lhs(...) =

Fig. 14. Outgoing output-dependence before and after transformation.

2.3.3. *Scope and Applicability of the Transformation*

2.3.3.1. On the necessity of statement S_k. After the transformation, why do we need to keep statement S_k (which is a copy into *lhs* now)? If we delete S_k, we have done nothing but a renaming. This is valid if and only if none of the values stored in array *lhs* by S_k is used outside the loop nest, e.g., written in an output; if this is the case, executing S_k is a pure time loss. In other words, if at least one of the values written by S_k is re-used later in the program then we must keep this statement, else we should delete it. Now, if statement S_k is kept, where to place it in the new code? Figure 16 shows three valid possibilities for Example 2. The first solution is the one proposed in our formal definition, the second is the original solution found by Padua and Wolfe, and the third involves two loops instead of one (the copy statement is moved outside the original loop). In the example, the first two solutions are equivalent, but this is not always the case: statement S_k cannot always be pushed at the end of the loop body. One can update *lhs* at the end of the loop body only if there is no output-dependence outgoing of statement S_k (a dependence outgoing of statement S_k is always an output-dependence, see Fig. 15). If there is a dependence

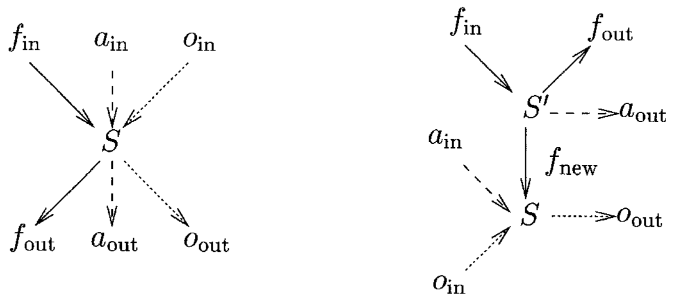


Fig. 15. A statement S with incoming and outgoing dependences before and after transformation.

```
For i := 1 to N do
  $S'_1$: temp(i) := b(i)+c(i)
  $S_1$: a(i) := temp(i)
  $S_2$: a(i+1) := temp(i)+2×d(i)
```

```
For i := 1 to N do
  $S'_1$: temp(i) := b(i)+c(i)
  $S_2$: a(i+1) := temp(i)+2×d(i)
  $S_1$: a(i) := temp(i)
```

```
For i := 1 to N do
  $S'_1$: temp(i) := b(i)+c(i)
  $S_2$: a(i+1) := temp(i)+2×d(i)
For i := 1 to N do
  $S_1$: a(i) := temp(i)
```

Fig. 16. Three different versions of Example 2 after transformation.

from S_k to another statement S_l then lhs must be updated before the execution of S_l.

The third transformation (with two loops) has three drawbacks: (i) more loops may imply more control; (ii) if loops can be made parallel, a barrier synchronization must be added after the first loop, which can be expensive; (iii) if we keep S_k near S'_k, we ensure that the value *temp*$(f(i))$ still resides in the cache (but we lose this property when using loop distribution). On the other hand, the third transformation may prove useful if the code contains a cycle of output-dependences which prevents parallelization. In this case, we might end up with a parallel computational loop, and a sequential loop involving all the S_k (copy) statements.

We illustrate the problems of moving statement S_k with Fig. 17. In this example, S_1 (after the transformation) cannot be executed at the end of the loop because the value of a is modified in the last statement. After the transformation, the dependence graph does not contain dependence cycles any more. The generated code may be rewritten as one HPF FORALL loop (inside the loop body, the ordering of statements being S'_1, S_2, S_1, and S_3) or as two parallel loops (the first one containing S'_1 and S_2, and the second one S_1 and S_3). On the other hand, if array a is to be updated independently (outside the loop), then we must split the initial loop into three different parallel loops.

To conclude, it is always possible to update *lhs* using an external loop, because after the transformation there is no flow-dependence going out of S_k (see Fig. 15). This may speed up the computational loop, but the execution of the loop updating the *lhs* arrays may well outweigh the benefits!

```
For i := 1 to N do
  $S_1$: a(i) := ...
  $S_2$: a(i+1) := a(i) + ...
  $S_3$: if b(i) > 1 then
           a(i) := ...
```

```
For i := 1 to N do
  $S'_1$: temp(i) := ...
  $S_1$: a(i) := temp(i)
  $S_2$: a(i+1) := temp(i) + ...
  $S_3$: if b(i) > 1 then
           a(i) := ...
```

```
For i := 1 to N do
  $S'_1$: temp(i) := ...
  $S_2$: a(i+1) := temp(i) + ...
For i := 1 to N do
  $S_1$: a(i) := temp (i)
For i := 1 to N do
  $S_3$: if b(i) > 1 then
           a(i) := temp(i)
```

Fig. 17. The leftmost loop is the initial code; the middle loop is the code obtained after our transformation; in the rightmost loop nest, array a is updated outside the loop.

Anyway, both solutions are possible, and loop distribution, which we chose not to implement, can still be used for further optimizations.

Hypotheses. As already said, the transformation extends to multi-dimensional loops. What really matters is the availability of a good dependence analyzer capable of providing sources and sinks of all dependences under consideration. 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* of statement S_k is injective. This is to prevent the occurrence of self output-dependence loops on S'_k. Note that Padua and Wolfe's second transformation has the same requirement.

Another hypothesis is that there are no dependences concerning the access function $f(i)$ of the left-hand side $lhs(f(i))$ of statement S_k: indeed, the function f may itself access variables that are accessed elsewhere in the code. This is not an actual restriction of the transformation, however. We explain how to handle such dependences in the Appendix.

2.3.3.2. Comparing both transformations. The first transformation can be applied for each read access in the original statement, but a new temporary array must be introduced at each time. On the contrary, the second transformation requires the use of a single temporary array per statement: instead of storing in a temporary array the result of a single read access, all read accesses are performed and the result of the computation is stored in the temporary array. Later the value is copied from the temporary back to the original array. Therefore, from the point of view of memory requirements, the second transformation seems more interesting than the first one.

We reported in our summary that the second transformation breaks more dependence paths (all paths $? \xrightarrow{a,\, o} S \xrightarrow{f,\, a} ?$, against paths $? \xrightarrow{a,\, o} S \xrightarrow{a} ?$). But we mentioned that the first transformation can also break paths like $? \xrightarrow{f} S \xrightarrow{a} ?$ provided that both dependences f and a apply to different data accesses. Fortunately, it turns out that the use of the second transformation is enough to break all "false dependence cycles" in the dependence graph, as will be stated more formally in Section 3.1. Here is a small example (see Fig. 18) to illustrate this point. There is a cycle of dependences:

$$S_1 \xrightarrow{f} S_2 \xrightarrow{f} S_3 \xrightarrow{a} S_1.$$

This cycle cannot be broken by the first transformation (even if we apply it to each statement). However, applying the second transformation to S_1 does break the cycle.

```
For i := 1 to N do
  S1: a(i-1) := b(i)
  S2: a(i) := a(i-1)
  S3: c(i) := a(i)
```

```
For i := 1 to N do
  S'1: tmp1(i) := b(i)
  S1: a(i-1) := tmp1(i)
  S'2: tmp2(i) := a(i-1)
  S2: a(i) := tmp2(i)
  S'3: tmp3(i) := a(i)
  S3: c(i) := tmp3(i)
```

```
For i := 1 to N do
  S'1: tmp(i-1) := b(i)
  S1: a(i-1) := tmp(i-1)
  S2: a(i) := tmp(i-1)
  S3: c(i) := a(i)
```

Fig. 18. The leftmost code is the initial code, the middle code is obtained after applying the first transformation to each statement, and the rightmost code is the outcome of the second transformation applied to S_1.

However, the second transformation does not subsume the first one. Indeed, the second transformation can break the dependence cycle in Example 1, just as the first transformation does: but it should be applied to statement S_1 instead of statement S_2.

3. MANIPULATING DEPENDENCE GRAPHS

In the rest of the paper we concentrate on Padua and Wolfe's second transformation, as formally defined (and modified) in Section 2.3.1. In this section we show its usefulness to break "false" cycles in dependence graphs.

3.1. Removing cycles

If we transform all the 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):

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 vertices (statements). 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.

Proof. Figure 15 may help follow the proof. Assume that G' has a cycle C, and consider an arbitrary edge e of C. Then e corresponds either to a flow-, an output- or an anti-dependence:

- e is an **output-dependence edge**. Then, according to Fig. 15, e is an edge from a node S_k to a node S_l. As the only edges going out S_l are output-dependences, the edge following e in C is an output-dependence edge. Thus C is only composed of output-dependence edges. Furthermore, all edges of C are also edges of G. Thus C is also a cycle of G.

- e is an **anti-dependence edge**. Then e goes from a node S'_k to a node S_l. As the only edges coming from S_l are output-dependences, the edge following e in C is an output-dependence edge. From the previous case (e is an output-dependence), we conclude that C is only composed of output-dependence edges. This contradicts the hypothesis that e is an anti-dependence edge. Thus C contains no anti-dependence edges.
- e is a **flow-dependence edge**. Then either e is a new flow edge from a node S'_k to the node S_k, or e goes from a node S'_k to a node S'_l:
  - — $e : S'_k \xrightarrow{f_{\text{new}}} S_k$. As the only edges coming from a node S_k are output-dependences, the edge following e in C is an output-dependence edge. From the first case (e is an output-dependence), we conclude that C is only composed of output-dependence edges. This contradicts the hypothesis that e is a flow-dependence edge.
  - — $e : S'_k \xrightarrow{f} S'_l$. There can be flow- and anti-dependence edges coming from a node S'_l. However, the edge which follows e in C cannot be an anti-dependence edge (because of the conclusion of the case "e is an anti-dependence"). Thus the edge which follows e in C is a flow-dependence edge and C is only composed of flow-dependence edges. Furthermore, a flow-dependence edge e in C goes from a node S'_k and to a node S'_l. Thus there is in G a flow-dependence edge from S_k to S_l, and 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*.

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. In Section 4, we state this problem formally and prove that it is NP-hard. This justifies the introduction of heuristics in Section 5. Beforehand, we work out an example, so as to illustrate the second transformation and heuristics.

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

$$\begin{tikzcd}
 & S_4 \arrow[dl, "f"'] & \\
S_1 \arrow[rr, "a"] \arrow[dr, "o"'] & & S_2 \arrow[ul, "f"'] \\
 & S_3 \arrow[uu, "a"] \arrow[ur, "f"'] &
\end{tikzcd}$$

Fig. 19. The target dependence graph before transformation.

3.2. Target example

Consider the following loop nest:

The dependence graph is represented in Fig. 19. 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[11] does find the six dependences listed above (see Table I). In fact Tiny finds a seventh dependence (the second one in Table I), 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 ith instance of S_1) and used in $S_2(i+6)$. But meanwhile, $a(i+5)$ is re-written in $S_3(i+5)$, and it is this new value which 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 spurious one. Otherwise we

Table I. The Dependencies Found by Tiny

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

$$\begin{tikzcd}
& S_4 \arrow[dl, "f"'] & & \\
S_1 \arrow[rr, "a"] \arrow[ddr, "o"'] & & S_2 & S_2' \arrow[l, "f"'] \arrow[ull, bend right, "f"'] \\
& S_3' \arrow[uu, "a"] \arrow[urr, bend right, "f"'] \arrow[d, "f"] & & \\
& S_3 & &
\end{tikzcd}$$

Fig. 20. The target dependence graph after transforming S_2 and S_3.

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 Fig. 20.

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 (another possibility would be to use loop peeling).

Finally, it is important to point out that we have broken all dependence cycles using only two temporary arrays. Other techniques such as converting the code into single assignment form would have achieved the same result at twice the price (introducing four temporaries).

4. 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 5). 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.

4.1. 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 $\ell : E \to \{f, a, o\}$ (flow, anti- or output-dependence). Our problem is to determine the minimum number of statements which we should transform using the transformation of Fig. 15 so that there remains only pure flow-dependence cycles and pure output-dependence cycles. We need some definition to formulate the associated decision problem:

Definition 1.

- Given a loop nest L (and its dependence graph $G = (V, E, \ell)$) and a nonnegative integer bound K, can we find no more than K vertices of G such that transforming these 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 $L \in PURE\text{-}CYCL(K)$).
- A loop nest L is admissible iff its dependence graph $G = (V, E, \ell)$ only contains anti-dependence edges:

$$\forall e \in E, \qquad \ell(e) = a$$

- Given an admissible loop nest L (and its dependence graph $G = (V, E, \ell)$ where $\forall e \in E$, $\ell(e) = a$) and a nonnegative integer bound K, can we find no more than K vertices of G such that transforming these vertices leads to an acyclic graph G' (if the answer is yes, we say that $L \in NO\text{-}CYCL(K)$).

We will prove that *NO-CYCL* is NP-complete in the strong sense, and therefore that *PURE-CYCL* is NP-complete in the strong sense. We use a reduction from a graph-theoretic problem that formalizes our transformation:

Definition 2. Let $G = (V, E)$ be a directed graph. Transforming $s \in V$ amounts to create a new graph $G' = (V', E')$ such that

1. G' has a new vertex s' : $V' = V \cup \{s'\}$
2. E' has a new edge $e = (s', s)$
3. let $e = (u, v) \in E$:

   (a) if $u \neq s$ then $e \in E'$

   (b) if $u = s$, e is replaced by an edge $e' = (s', v) \in E'$

Note that the transformation of several vertices of a graph can be performed in any order. We state the following decision graph-theoretic problem:

Definition 3. Given a graph $G = (V, E)$ and a nonnegative integer bound K, can we find no more than K vertices of G such that transforming these vertices leads to an acyclic graph G' (if the answer is yes, we say that $L \in GRAPH\text{-}CYCL\,(K)$).

We first prove that *GRAPH-CYCL* is NP-complete, by using a reduction from the satisfiability problem $3SAT$. Then we show that *NO-CYCL* is NP-complete.

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

Proof. First, *GRAPH-CYCL* belongs to NP: given a graph $G = (V, E)$, a bound K, and the list of the vertices to be transformed, we can check in polynomial time whether the new graph G' is acyclic (this can be done even in linear time $O(|V| + |E|)$ by traversing it).

We use a reduction from the satisfiability problem $3SAT$. An instance of the 3SAT problem[16] consists of a Boolean expression B in conjunctive normal form,

$$B = \bigwedge_{i=1}^{t} C_i$$

- where $C_j = \ell_j^1 \vee \ell_j^2 \vee \ell_j^3$, $1 \leqslant j \leqslant t$, is a clause
- where each literal ℓ_j^k is a variable or its negation in the set of variables $X = \{x_1, \ldots, x_r\}$

The associated decision problem is represented as follows: does there exist a value assignment $w : X \to \{true, false\}$ such that B evaluates to *true* under w? (we say $B \in 3SAT$).

Here is an example that we shall use throughout the proof: $t = 3$, $r = 4$, and

$$B = (x_1 \vee \neg x_2 \vee x_4) \wedge (\neg x_1 \vee x_3 \vee \neg x_4) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$$

Given B, we have to construct an instance $g(B)$ of our problem (i.e., a graph $G = (V, E)$ and a bound K) such that $g(B) \in GRAPH\text{-}CYCL\ (K) \Leftrightarrow B \in 3SAT$. Furthermore, the construction function g must be polynomial in the size of B, i.e., in the number of clauses t and of variables r.

Construction. To help the reader follow the construction, we give intuitive names to the nodes of the graph.

There are $2 \times t \times r$ vertices in G. For each variable x_i we introduce a widget W_i made of $2 \times t$ vertices. These vertices are labeled $T(i, j)$ and $F(i, j)$, $1 \leqslant j \leqslant t$ (see Fig. 21). Intuitively, vertex $T(i, j)$ or $F(i, j)$ will be used if variable x_i appears in clause C_j: we use vertex $T(i, j)$ if variable x_i is un-negated in clause C_j, otherwise we use vertex $F(i, j)$. As illustrated in Fig. 21, the widget is a complete bipartite graph: there is an edge from any vertex $T(i, j)$ to every vertex $F(i, k)$ and vice-versa, with $1 \leqslant j, k \leqslant t$, which leads to $2 \times t^2$ edges per widget.

Widgets are connected according to clauses. Consider clause C_1 in the example: $C_1 = x_1 \vee \neg x_2 \vee x_4$. We link vertices $T(1, 1)$, $F(2, 1)$ and $T(4, 1)$ so as to make a cycle $T(1, 1) \to F(2, 1) \to T(4, 1) \to T(1, 1)$ in the dependence graph G (see Fig. 22). Similarly, since $C_2 = \neg x_1 \vee x_3 \vee \neg x_4$,

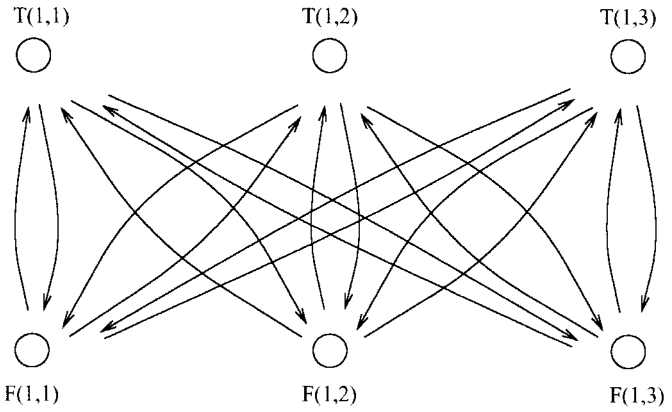


Fig. 21. The widget W_1.

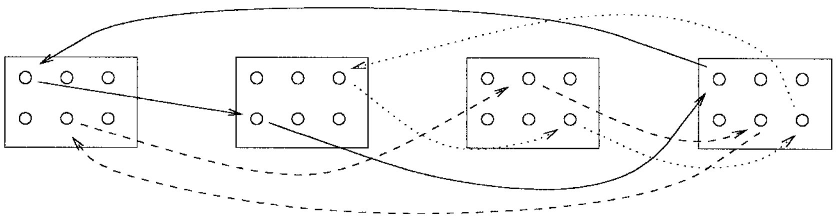


Fig. 22. Connecting widgets from the clauses.

we link vertices $F(1, 2)$, $T(3, 2)$ and $F(4, 2)$ to make another cycle in G. Therefore, for each clause we add 3 edges to E, leading to a total of $2\times r\times t^2+ 3\times t$ edges in the graph. Clearly, the dependence graph G does have a size polynomial in r and t.

Finally, we let $K= r\times t$, hence we ask the question whether it is possible to transform half the vertices so that the resulting graph G' has no cycle.

Equivalence. Now we prove that $g(B) \in GRAPH\text{-}CYCL\ (K) \Leftrightarrow B \in 3SAT$. Assume first that $B \in 3SAT$, and let $w : X \rightarrow \{true, false\}$ be a value assignment such that B evaluates to *true* under w. If variable x_i is assigned to *true* (i.e., $w(x_i)= true$), then in the widget W_i we transform the t vertices $T(i, j)$, $1\leqslant j\leqslant t$, otherwise we transform the t vertices $F(i, j)$, $1\leqslant j\leqslant t$. Therefore we do transform $K= r\times t$ vertices in total.

Transforming all the $T(i, *)$ vertices or all the $F(i, *)$ vertices ensures that there does not remain any cycle internal to the widget W_i. Indeed, since the widget is bipartite, all internal cycles go at least through a T and through a F node, and one of them is transformed, thereby breaking the cycle.

There remains to prove that the cycles due to the connection of the widgets are broken too. Since $B \in 3SAT$, each clause C_j evaluates to *true* under the assignment w. Hence there is at least one variable x_i in C_j whose assignment $w(x_i)$ raises C_j to *true*. If x_i appears un-negated in C_j then $w(x_i)= true$ and we transform vertex $T(i, j)$. By construction, the cycle due to clause C_j goes through vertex $T(i, j)$ and therefore is broken. The reasoning is similar with $F(i, j)$ if x_i appears negated in C_j. Hence there remains no cycle in G', and $g(B) \in GRAPH\text{-}CYCL\ (K)$.

Conversely, let $g(B) \in GRAPH\text{-}CYCL\ (K)$, i.e., assume that it is possible to transform at most $K= r\times t$ vertices so that the resulting graph G' has no cycle. We have to build a value assignment w such that B evaluates to *true* under w.

Consider a widget W_i. Since there does not remain any cycle in G', at least all the vertices $T(i, j)$, $1\leqslant j\leqslant t$, or all the vertices $F(i, j)$, $1\leqslant j\leqslant t$, must have been transformed. Otherwise, there would remain a vertex $T(i, j_0)$ and a vertex $F(i, j_1)$ that have not been transformed, and the cycle of length 2: $T(i, j_0) \rightarrow F(i, j_1) \rightarrow T(i, j_0)$ would not have been broken. Since at most $K = r \times t$ vertices are transformed in total, and since at least t vertices per widget are transformed, then exactly t vertices are transformed per widget, either all the $T(i, *)$ or all the $F(i, *)$.

According to this discussion, there are exactly t vertices transformed in widget W_i, namely either the t vertices $T(i, j)$, $1 \leqslant j \leqslant t$ or the t vertices $F(i, j)$, $1 \leqslant j \leqslant t$. We derive a truth assignment function w by letting $w(x_i) =$ *true* if the transformed vertices are the $T(i, *)$ and $w(x_i) =$ *false* if the transformed vertices are the $F(i, *)$. We have to show that w is a value assignment such that B evaluates to *true* under w. Consider a clause C_j, $1 \leqslant j \leqslant t$. The cycle of G linking the three jth nodes of the widgets W_i such that variable x_i appears in C_j has been broken. Hence at least one of these nodes has been transformed, say the one corresponding to variable x_{i_0}. This transformed node can be either $T(i_0, j)$ or $F(i_0, j)$, depending upon whether x_{i_0} appears un-negated or negated in C_j. But if we have transformed $T(i_0, j)$ then $w(x_i) =$ *true*, and if we have transformed $F(i_0, j)$ then $w(x_i) =$ *false*. Therefore C_j evaluates to *true* under w, and so does B. Consequently, $B \in 3SAT$, and the proof is complete. □

Theorem 3. *NO-CYCL* is NP-complete (in the strong sense).

Proof. First, *NO-CYCL* belongs to NP: consider an admissible loop nest L and its dependence graph $G = (V, E)$. If the vertices to be transformed are given, we can check in polynomial time whether the new graph G' is acyclic (this can be done even in linear time $O(|V| + |E|)$ by traversing it).

We use a reduction from *GRAPH-CYCL*. Given a graph $G = (V, E)$, we construct an admissible loop nest L whose dependence graph is G.

So let $G = (V, E)$ be given. All edges in G must correspond to anti-dependences in the loop nest L. To each vertex $v \in V$ we associate a linear array *tab.v* of size 100 (say). We build the loop nest L as a single loop surrounding $|V|$ statements. There is one statement S_v per vertex v, whose left-hand side is simply $tab.v(i) =$ .... For each edge $e = (u, v) \in E$ we obtain an anti-dependence from S_u to S_v by inserting a reference to $tab.v(i+1)$ in the right-hand side of S_u as follows:

```
for $i := 1$ to 99 do
        Statement $S_u$: $tab.u(i) = \ldots + tab.v(i+1) + \ldots$
        ...
        Statement $S_v$: $tab.v(i) = \ldots + \ldots$
```

The loop nest L is clearly admissible, as there are neither flow- nor output-dependences in its dependence graph G. This construction is clearly polynomial in the size of G, and the result follows immediately. □

Corollary 1. *PURE-CYCL* is NP-complete (in the strong sense).

5. 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 exponentially 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 $O(|V|^2 (|V| + |E|))$.

5.1. A heuristic based on the hypergraph of the cycles of G

Maybe 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 $f \subset V$, where each f is the set of the vertices of an elementary cycle C of G. See Fig. 23 for the hypergraph H of our target example. We mark each vertex v in f as *breakable* if C is broken when v is transformed, i.e., v is marked *breakable* if the incoming edge of v in C is an anti- or output-dependence, and the outgoing edge a flow- or anti-dependence.

Once H is built, we apply a greedy strategy and transform the vertex v_0 which belongs to, and is *breakable* for, the maximal number of subsets $f \in F$. We delete all cycles that were going through v_0 and for which v_0 was

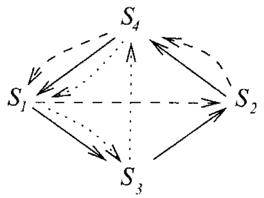


Fig. 23. Hypergraph of the target example. The three elementary cycles are shown with different arrow formats.

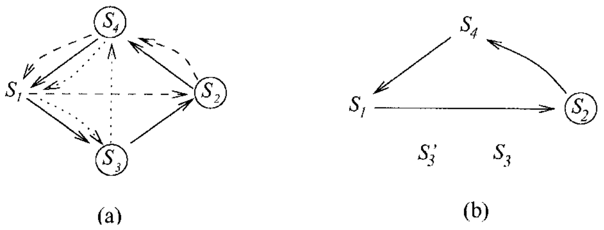


Fig. 24. Hypergraph of the target example (a) before transformation; and (b) after transformation of node S_3.

breakable. We redo the operation until there remains no cycle in the graph with *breakable* vertices. [Note: Here is a small improvement: search whether there exists a subset $f \in F$ 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 $f \in F$.]

The drawback of this heuristic is its high cost in the worst case. Although this is unlikely to happen, 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 prohibitive cost.

5.1.1. The Heuristic Applied to the Target Example

We show here the transformation of the target example of Section 3.2 using this heuristic. Figure 24 shows the hypergraph corresponding to the dependence graph of Fig. 19. The table in Fig. 25 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 Fig. 24. 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 3.2 (see Fig. 20).

5.2. 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

S_1	S_2	S_3	S_4
0	1	2	1

Fig. 25. Number of circuits which include a vertex as *breakable*.

anti-dependence. For each vertex v of the dependence graph we can count its "utility," i.e., the number $Util(v)$ of pairs $(e_{\text{in}}, e_{\text{out}})$ such that

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

We transform one of the vertices v such that $Util(v)$ is maximal. We obtain a graph G'. Remove from G' all the edges which are not in a strongly connected component. If there is at least an anti-dependence edge in G' or if G' includes 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 nonpure 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 $O(|V|^2 (|V| + |E|))$.

5.2.1. The Polynomial-Time Heuristic on the Target Example

We show in Table II the processing of the target example of Section 3.2 by the polynomial heuristic. Table II shows the value of *Util* for each of the graph vertices.

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 depicted in Table III.

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

Table II. Values of *Util* for Each of the Graph Vertices

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

Table III. The New Values of *Util*

	S_1	S_2	S_4
$Util(v)$	0	1	0

6. CONCLUSIONS

In this paper we have formalized Padua and Wolfe's transformations,[1] to eliminate dependence paths including anti- and output-edges. 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.

APPENDIX: READ ACCESSES IN LEFT-HAND SIDES

When analyzing Padua and Wolfe's second transformation in Section 2.3.2, we have assumed that there was no read access in the left-hand side of statements. This hypothesis prevents the existence of incoming flow-dependences and outgoing anti-dependences on the left-hand side of statements. We consider here such dependences:

- *Incoming flow-dependence in the left-hand side* (Fig. 26). One of the data read in the left-hand side of statement S_k was previously produced in the left-hand side of another statement S_l. This data is used to compute the access function to array lhs in statement S_k. After the transformation, the data is used to compute the access functions to array temp in S'_k and to arrays lhs and temp in S_k. Thus there is one flow-dependence from S_l to S'_k, and there are two dependences from S_l to S_k.
- *Outgoing anti-dependence in the left-hand side* (Fig. 27). One of the data read in the left-hand side of statement S_k is written afterwards in another statement S_l. This data is used to compute the access

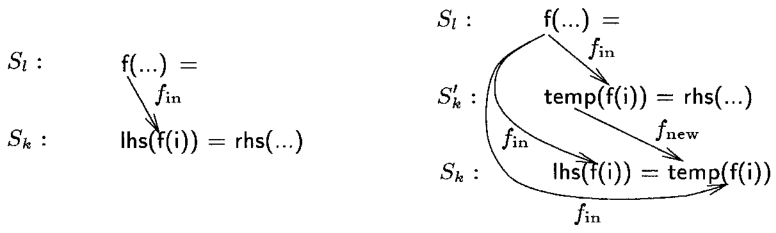


Fig. 26. Incoming flow-dependence before and after transformation.

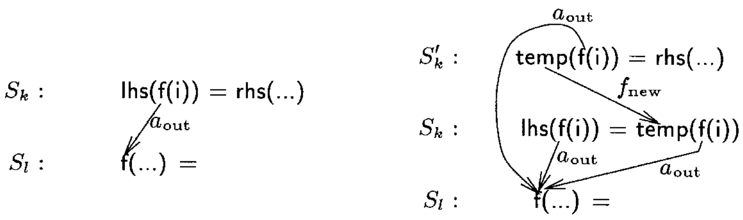


Fig. 27. Outgoing anti-dependence before and after transformation.

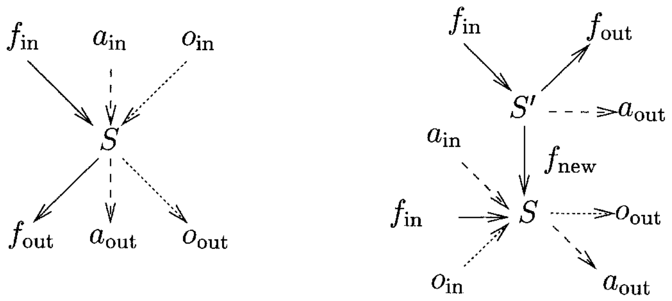


Fig. 28. A statement with incoming and outgoing dependences before and after transformation, with read in access functions.

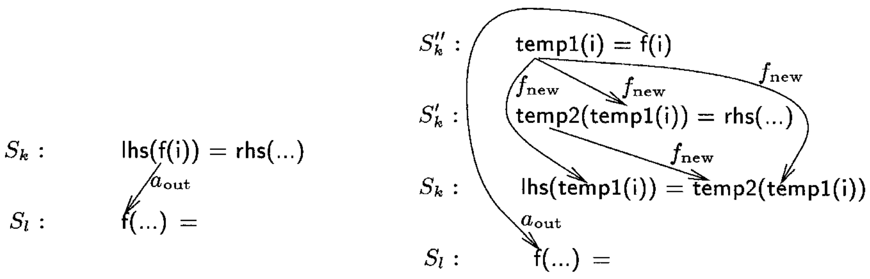


Fig. 29. Outgoing anti-dependence before and after double transformation.

function to array lhs in statement S_k. After the transformation, the data is used to compute the access functions to array temp in S'_k and to arrays lhs and temp in S_k. Thus there is one anti-dependence from S'_k to S_l, and there are two dependences from S_k to S_l.

We summarize these results and those previously established (Section 2.3.2) in Fig. 28. Looking at Fig. 28, we might think that the transformation of a statement may be useful only if it has an incoming anti- or output-dependence edge, and an outgoing flow-dependence edge (see Fig. 15 again), because we now have anti-dependences going out of S. However, these anti-dependences going out of S come from read access in array access functions. To solve the problem, we should have previously applied the first transformation to these faulty accesses before applying the second transformation to the statement. The result of this double transformation is shown on Fig. 29. The reader can check that the double transformation breaks the same paths as before: $? \xrightarrow{a,\,o} S \xrightarrow{f,\,a} ?$. Theorem 1 still holds.

ACKNOWLEDGMENTS

We thanks the reviewers whose comments and suggestions have greatly improved both the presentation and the contents of the paper.

REFERENCES

1. David A. Padua and Michael J. Wolfe, Advanced Compiler Optimizations for Supercomputers, *Comm. ACM* **29**(12):1184–1201 (December 1986).
2. Paul Feautrier, Dataflow Analysis of Array and Scalar References, *IJPP* **20**(1):23–51 (1991).
3. Dror E. Maydan, Saman P. Amarasinghe, and Monica Lam, Array Data-Flow Analysis and Its Use in Array Privatization, *Principles of Progr. Lang*. (1993).
4. Junjie Gu, Zhiyan Li, and Gyungho Lee, Symbolic Array Dataflow Analysis for Array Privatization and Program Parallelization, *Supercomputing* (1995).
5. Thomas Brandes, The Importance of Direct Dependences for Automatic Parallelization, *Int'l Conf. Supercomputing*, pp. 407–417 (1988).
6. Uptal Banerjee, Rudolph Eigenmann, Alexandru Nicolau, and D. A. Padua, Automatic program parallelization, *Proc. IEEE* **81**(2):211–243 (1993).
7. David F. Bacon, Susan L. Graham, and Oliver J. Sharp, Compiler Transformations for High-Performance Computing, *ACM Computing Surveys* **26**(4):345–420 (1994).
8. Michael Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley Publishing Company (1996).
9. Hans Zima and Barbara Chapman, *Supercompilers for Parallel and Vector Computers*, ACM Press (1990).
10. Pierre-Yves Calland, Alain Darte, Yves Robert, and Frédéric Vivien, Plugging Anti- and Output-Dependence Removal Techniques into Loop Parallelization Algorithms, *Parallel Computing* **23**(1,2):251–266 (1997).

11. Michael Wolfe, The Tiny Loop Restructuring Research Tool. In H. D. Schwetman, (ed.), *Int'l. Conf. Parallel Processing*, Volume II, CRC Press, pp. 46–53 (1991).
12. William Pugh, Release 0.96 of petit. World Wide Web document, URL: http://www.cs.umd.edu/projects/omega/petit.html.
13. SIMULOG S.A. *FORESYS*, *Manuel de Référence* (April 1994).
14. 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.
15. PIPS Team, Pips (interprocedural parallelizer for scientific programs). World Wide Web document, URL: http://www.cri.ensmp.fr/˜pips/index.html.
16. Michael R. Garey and Davis S. Johnson, *Computers and Intractability, a Guide to the Theory of NP-Completeness*, W. H. Freeman and Company (1991).