Loop Shifting for Loop Parallelization

Alain Darte, Guillaume Huard

Loop Shifting for Loop Parallelization

Alain Darte and Guillaume Huard

May 2000

Research Report N° 2000-22

Loop Shifting for Loop Parallelization

Alain Darte and Guillaume Huard

May 2000

Abstract

The automatic detection of parallel loops is a well-known problem. Sophisticated polynomial algorithms have been proposed to produce code transformations that reveal parallel loops, in an optimal way as long as the only objective is to reveal as much parallelism as possible. However, a weakness of these techniques is that there is very few control on the solutions they built. For example, it may happen that the produced solution needs a very complex unimodular transformation plus a shift of the different statements even though a much simpler solution exists. It would be more useful to first select its own unimodular transformation and to be able to check that it can be completed by a shift while revealing the same parallelism. Unfortunately, the main result of this paper is that this is a hard problem: finding a (multi-dimensional) shift that makes an innermost loop parallel is strongly NP-complete. Nevertheless, we show that several subcases of this problem are easily solvable and that the general problem can be solved thanks to integer linear programming.

Keywords: loop parallelization, code transformation, retiming, loop shifting

Résumé

La détection de boucles parallèles est un problème bien connu. Plusieurs algorithmes polynômiaux sophistiqués ont été proposés pour déterminer des transformations de code qui révèlent des boucles parallèles et ce, de façon optimale, tant que l'objectif est de révéler le plus de parallélisme possible. Cependant, une faiblesse de ces techniques est qu'elles offrent peu de contrôle sur les solutions qu'elles produisent. Par exemple, il peut arriver qu'une solution produite conduise à une transformation unimodulaire très complexe, associée à un décalage d'instructions, alors qu'une solution bien plus simple existe. Il serait bien plus utile de pouvoir sélectionner une transformation unimodulaire de son choix et d'être capable de vérifier qu'elle peut être complétée par un décalage et révéler autant de parallélisme. Malheureusement, le résultat principal de cet article est que ceci est un problème difficile : trouver un décalage (multi-dimensionnel) qui rend une boucle interne parallèle est un problème NP-complet au sens fort. Néanmoins, nous montrons que plusieurs sous-cas de ce problème peuvent être résolus simplement et que le problème général peut être résolu par programmation linéaire en nombres entiers.

Mots-clés: parallélisation de boucles, transformations de code, décalage d'instructions

Contents

1 Introduction **2**

2 Illustrating Examples and Related Problems **4**
2.1 Loop Alignment Versus Loop Shifting 5
2.2 Loop Shifting for Fusion of Parallel Loops 6
2.3 Multidimensional Shifting 7

3 General Properties **9**
3.1 Dependence Graphs and Correct Codes 9
3.2 Shifting and Shortest Path Properties 11

4 Parallelization by Internal Shifting **13**
4.1 Onedimensional Case 13
4.2 Extension to Nested Loops 15

5 Parallelization by External Retiming **17**
5.1 Characterization of the Problem 17
5.2 NP-Completeness 19
5.3 Formulation by Linear Constraints 24

6 Polynomially-Solvable Cases and Heuristics **25**
6.1 Inner Parallel Loop by Internal Retiming Alone 25
6.2 Inner Parallel Loop by External Retiming Alone 25
6.3 A Multidimensional Heuristic 26
6.4 Non-Existence of a Retiming 27

7 Conclusions and Extensions **27**
7.1 Summary of Results 27
7.2 Improving Parallelism Through Code Replication 28
7.3 Combining Shifting with Linear Transformations 28
7.4 Extension to Nonuniform Dependences 29

Loop Shifting for Loop Parallelization

Alain Darte and Guillaume Huard
LIP, ENS-Lyon, 46, Allée d'Italie, 69007 Lyon, France.
E-mail: `Firstname.Lastname@ens-lyon.fr`

May 2000

Abstract

The automatic detection of parallel loops is a well-known problem. Sophisticated polynomial algorithms have been proposed to produce code transformations that reveal parallel loops, in an optimal way as long as the only objective is to reveal as much parallelism as possible. However, a weakness of these techniques is that there is very few control on the solutions they built. For example, it may happen that the produced solution needs a very complex unimodular transformation plus a shift of the different statements even though a much simpler solution exists. It would be more useful to first select its own unimodular transformation and to be able to check that it can be completed by a shift while revealing the same parallelism. Unfortunately, the main result of this paper is that this is a hard problem: finding a (multi-dimensional) shift that makes an innermost loop parallel is strongly NP-complete. Nevertheless, we show that several subcases of this problem are easily solvable and that the general problem can be solved thanks to integer linear programming.

1 Introduction

Now that the architectures of parallel machines are getting more stable, programmers can hope to take advantage of these new resources directly with their old "sequentially" written programs. But developing a parallel program from a sequential one still requires a strong knowledge of parallel programming, therefore an automatic procedure is the only way for a nonspecialist to achieve the transformation. The problem of this automatization is the complexity of the task, and today it is not yet reasonable to think of a stand-alone procedure that would be able to provide an "optimal" parallel version of a whole sequential program. A more realistic approach would be to help nonspecialists with semi-automatic parallelization tools based on languages extended with compilation directives. This is, for example, the design idea of Tera's compiler [32]: try to do as much as possible with the compiler, but propose to the user some ways to guide the compiler. This is also the idea of languages such as HPF [18] and OpenMP [25]. Such an approach has to provide procedures both to quickly parallelize "easy" parts of the program (thereby saving development time for the user) and to implement complex techniques that cannot be easily handled by the user. For his part, the user may have to guide the tool by choosing which transformation has to be applied and where, thereby ensuring a better interaction between successive parts of the program. For instance, in the context of compilation for distributed-memory systems, this human intervention can help to choose either a good data alignment for a long and complex program or a well-suited transformation in presence of constraints for data layout.

In this paper we study some of these automatic transformations, especially those devoted to the detection of parallelism in simple loops or nested loops. Loops are interesting for several reasons:

they involve both regular and predictable computations and they are often parts of the program in which most of the computation time is spent (especially for scientific applications). Furthermore, this regularity and predictability make them good candidates for an automatic transformation. Since we want to develop techniques that can be integrated in semi-automatic parallelization tools (like our own tool Nestor, see [30], or SUIF [31]), we would like to use, when possible, transformations as simple as possible, i.e., easily understandable by the user at high-level, but powerful and complex enough to help the compiler work.

Numerous algorithms have been already proposed to detect parallelism in a loop nest. A wide family of transformations is based on the notion of *schedule*. The idea is to consider the operations described by the different iterations of the loop, in a global way, as an *iteration domain*, and to restructure this domain so as to execute the different iterations following a new order defined by the schedule: this schedule specifies for each operation an execution date such that all the operations executed at the same time are independent (in other words parallel). Such algorithms typically lead to codes with outer sequential loops (the time) surrounding a fully parallel loop nest. The first algorithm in this category is the hyperplane method, proposed by Leslie Lamport [19]: it transforms a set of perfectly nested loops with uniform dependences (i.e., with a finite number of non parameterized constant dependence distances) into one outer sequential loop surrounding parallel loops. This method was generalized by Wolf and Lam [33], using unimodular transformations of the iteration domain (this includes loop reversal, loop interchange, and loop skewing). The drawback of both methods in terms of parallelism is that the body of the loop is viewed as a single block: potential parallelism due to the structure of the dependence graph (i.e., dependences between different statements) cannot be exploited. More complex algorithms (especially Feautrier's algorithms [12, 13]) have then been proposed to capture more general transformations, through multidimensional affine schedules. For example, the intermediate algorithm proposed in [11] combines unimodular transformations, loop distribution and shifting.

These algorithms are, in theory, perfect for parallelism detection. But their power makes also their weakness: the more general the form of the schedule, the more complicated the generation of the code and the less control we have on the simplicity of the solution. For example, when the linear part of the transformation is found automatically by a linear programming approach, we cannot guarantee that the "simplest" solution or the "most obvious" solution for the programmer would be chosen. A complex unimodular transformation, even a skew for example, can cause several problems when generating the new code. One problem is that the conversion between new and old loop indices can produce extra costly [1] operations such as modulos, multiplications, minima and maxima, and this complexity is not due to the code generation algorithms [16, 4, 6] but to the transformation itself: the simplest to avoid it is sometimes just to consider another transformation! Another problem (especially in a distributed-memory environment) is that the data that could be properly aligned for the initial execution order can now be badly located and can generate a heavy communication overhead (unless remapping is performed). This problem is hard to fix automatically and fixing it manually is not simple because even a skewed code is difficult to read. Also, even if parallelism is increased, a complex transformation may not be good for other objectives. For example, it can change the number of sequential iterations (that correspond to synchronizations). On shared-memory environments, it can also have an impact on locality. Indeed, all these algorithms follow the same greedy strategy: a schedule is built so that dependences after code transformation occur between operations executed at different iterations of the sequential loops (they are now loop carried dependences). Therefore, codes with data reuse in the body of the loop such as parallel

[1]These operations are not only time-consuming. They are also costly in terms of hardware when such transformed loops are used in circuit synthesis. Avoiding operations such as modulos and multiplications saves gates and power.

loops with internal (i.e., loop independent) dependences cannot be obtained, surprisingly.

Among all these code transformations that reveal parallelism, only two of them can be considered as "simple" in the sense that they do not change the structure of the iteration domain: these are loop distribution and loop shifting. Loop distribution (which replicates the loop structure into several loops, without changing the iteration domain) is the basic tool for Allen, Callahan, and Kennedy's algorithm [1]. This transformation is sufficient for parallelism detection in many practical cases, and its simplicity makes possible its study with other objectives: for example, partial loop distribution [7] can be used to derive parallel loops including loop independent dependences, fusion techniques can derive codes with better locality [23], loop distribution can be extended for complex flow programs [17], etc. Our goal here is to make a similar study when shifting statements, the main question being the following: can we decide when loop shifting is sufficient for finding parallel loops? More generally, given the linear part of a transformation, in other words a particular axis choice for the iteration domain, can we decide if it can be completed by an adequate loop shifting?

The idea to use shifting alone (i.e., with no axis change) for loop parallelism was studied by Okuda [24] who suggested to shift statements to transform some dependences into loop independent dependences. A similar technique, called loop alignment, was previously developed by Peir [27] (see also [34]). None of these techniques however answers completely the question we stated above. The idea to decompose the construction of an affine transformation in two steps – a step for the linear part, a step for the shifting part – was developed in [10]. But, again, they were not able to completely characterize the cases when a loop shifting exists that completes the unimodular part of the transformation. In [20], our main problem is addressed: a polynomial-time algorithm is given, but the technique misses one point and, as we will show, the problem "solved" by this polynomial algorithm is actually strongly NP-complete. We should also mention the problem of fusion with loop shifting [2], which is a particular onedimensional case of our problem, and the shift-and-peel technique [22], which combines loop shifting and loop peeling. This last technique allows to detect more parallelism in loops, but the number of required parallel chunks must be known (i.e., the number of targeted processors). Here, we restrict to loop shifting alone and we look for loops where all iterations can be performed in parallel (not blocks of iterations). The shift-and-peel transformation is then an interesting transformation (as is loop skewing) that can be applied when loop shifting alone fails.

The paper is organized as follows. First, in Section 2, we illustrate the problem through several examples and state it more formally. In Section 3, we first give some general properties of multidimensional loop shifting. In Section 4, we address the problem of loop parallelization by shifting only along the loops we want to parallelize (internal shifting). The problem is polynomially solvable but the conditions for a solution to exist are quite strong, in other words, loops that can be parallelized by internal shifting are quite rare. We thus explore, in Section 5, how shifting along surrounding loops (what we call external shifting) can make this internal shifting feasible. We show that this problem is strongly NP-complete. Nevertheless, it can be formulated as a polynomial number of integer linear constraints. In Section 6, we give some simple heuristics that identify particular solvable cases and that can be used for integrating loop shifting as a practical tool for loop parallelization. Finally, we conclude in Section 7 and we suggest some possible extensions.

2 Illustrating Examples and Related Problems

The goal of this section is to illustrate through examples what are the "shifting" problems we want to solve, and how they are linked to similar approaches and problems.

2.1 Loop Alignment Versus Loop Shifting

Example 1

Consider the well-known example from Peir and Cytron [28] recalled with its associated dependence graph in Figure 1. To find parallelism is this code, Peir and Cytron use multidimensional *loop alignment* as a preliminary step. The idea of the technique is to group into a single edge all the dependences formed by a recurrence circuit in the reduced dependence graph, by shifting statements with respect to each other. By grouping dependences this way, it may happen that each circuit has now only loop independent dependences, except one which may be carried by the outermost loop.

```
DO i=-1,13
  DO j=-4,25
    S1  a[i,j]=b[i,j-6]+d[i-1,j+3]
    S2  b[i+1,j-1]=c[i+2,j+5]
    S3  c[i+3,j-1]=a[i,j-2]
    S4  d[i,j-1]=a[i,j-1]
  ENDDO
ENDDO
```

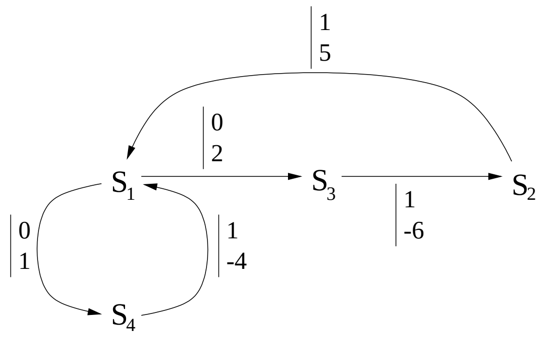


Figure 1: An example borrowed from Peir and Cytron.

Here loop alignment leads to the code of Figure 2, where the outermost loop is sequential and the innermost loop is now parallel. The statement $S4$ is shifted by $(1, -4)$ (i.e., 1 iteration for the i-loop, -4 iterations for the j-loop), $S3$ by $(2, -1)$, and $S2$ by $(1, 5)$, and statements are reordered in the loop body, following the loop independent dependences obtained after the shift. Thus, loop alignment solves our problem on this example (but this is not true in general).

```
prologue (with a DO j...)
DO i=1,13
  prologue
  DOALL j=1,21
    S4  d[i-1,j+3]=a[i-1,j+3]
    S3  c[i+1,j]=a[i-2,j-1]
    S2  b[i,j-6]=c[i+1,j]
    S1  a[i,j]=b[i,j-6]+d[i-1,j+3]
  ENDDO
  epilogue
ENDDO
epilogue (with a DO j...)
```

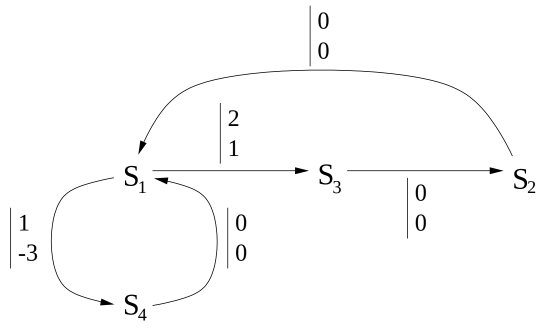


Figure 2: The transformation proposed by Peir and Cytron.

Nevertheless, in this example, we could also use a simpler shifting technique: actually, what is really needed is to align only the edges that are not carried by the outermost loop to make them loop independent. No need here to shift in both dimensions, a shift in the innermost dimension is sufficient. As we can see in Figure 3, this results in a simpler code because the amount of shifting is smaller (the more an instruction is shifted, the bigger is the size of the prologue and epilogue) and because we apply a shift only in one dimension (again multidimensional shifting produces bigger prologue and epilogue). □

```
DO i=-1,13
  prologue
  DO j=-3,24
    S1  a[i,j-1]=b[i,j-7]+d[i-1,j+2]
    S2  b[i+1,j-1]=c[i+2,j+5]
    S3  c[i+3,j]=a[i,j-1]
    S4  d[i,j-1]=a[i,j-1]
  ENDDO
  epilogue
ENDDO
```

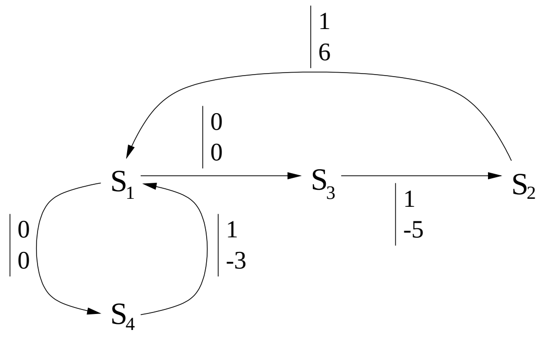


Figure 3: Our solution for Example 1.

As shown by Example 1, it is possible to derive a parallel loop just by applying a shift of instructions in the same dimension. The question are: When is it possible? And how can we find the shift? Note that, in this code, a standard unimodular approach would apply a loop skewing to transform the code into one parallel loop surrounded by one sequential loop, but with a completely different result: the resulting code would have no prologue nor epilogue, but loop bounds and access functions to arrays would be much more complicated (see Figure 4).

```
DO j=-11,45
  DOALL i=max(-1,ceil((j-25)/7)),
          min(13,floor((j+4)/7))
    S1  a[i,j-7i]=b[i,j-7i-6]+d[i-1,j-7i+3]
    S2  b[i+1,j-7i-1]=c[i+2,j-7i+5]
    S3  c[i+3,j-7i-1]=a[i,j-7i-2]
    S4  d[i,j-7i-1]=a[i,j-7i-1]
  ENDDO
ENDDO
```

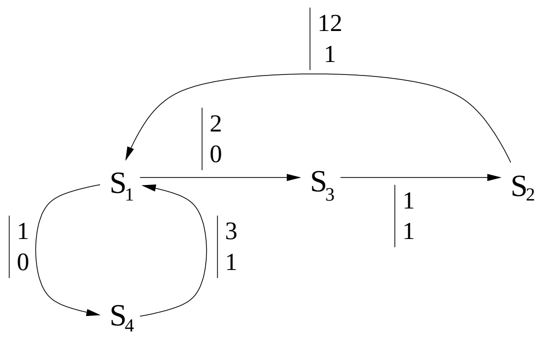


Figure 4: The simplest solution with a unimodular transformation for Example 1.

2.2 Loop Shifting for Fusion of Parallel Loops

In the previous section, we shifted statements that belong to the same loop body. The general situation can be more complicated, for example when we try to shift statements that belong to different loops so as to group the loops into as few sequential and parallel loops as possible, in other words when we try to combine loop shifting with loop fusion while keeping parallelism. This problem was addressed in [2] and illustrated by the following example.

Example 2

The code on the left of Figure 5 can be “compacted” into the code on the right (assuming $N \geq 1$), which minimizes the number of parallel loops that can be obtained by shift and fusion.

Fusing two loops, thanks to loop shifting, is theoretically always possible when dependences are uniform: one just has to shift enough so that dependence distances become nonnegative. However,

```
loop1: DOALL i=1,N
           a[i]=1
       ENDDO
loop2: DOALL i=1,N
           b[i]=1
       ENDDO
loop3: DOALL i=1,N
           c[i]=a[i-1]+b[i]
       ENDDO
loop4: DOALL i=1,N
           d[i]=a[i]+b[i-1]
       ENDDO
loop5: DOALL i=1,N
           e[i]=d[i]+d[i-1]
       ENDDO
```

```
            a[1]=1
            d[1]=a[1]+b[0]
loop1-2-4:  DOALL i=2,N
                a[i]=1
                b[i-1]=1
                d[i]=a[i]+b[i-1]
            ENDDO
            b[n]=1
loop3-5:    DOALL i=1,N
                c[i]=a[i-1]+b[i]
                e[i]=d[i]+d[i-1]
            ENDDO
```

Figure 5: Loop shifting for fusion.

for fusing parallel loops, a shift must be found so that all dependences between statements in the same loop are loop independent, i.e., with dependence distance equal to 0 (otherwise, the loop would be either incorrect if a distance is negative, or sequential if a distance is positive). In this example, a solution that leads to the minimal number of parallel loops is obtained by shifting the statement in `loop2` by 1. Now, `loop1`, `loop2`, and `loop4` can be fused. Fusing `loop3` and `loop5` does not require loop shifting however. □

Determining if there exists a shift for a set of parallel loops so that they can be fused in at most K parallel loops has been proved to be NP-complete [2], if K is arbitrary. In other words, shifting and fusing parallel loops into as few parallel loops as possible is hard.

2.3 Multidimensional Shifting

The NP-completeness result we mentioned above does not hold for the fixed value $K = 1$: Determining if a shift exists that results in a single parallel loop can be quickly checked as we will show in Section 4.1. One just have to verify that, in the graph formed by the dependences to be considered, all (undirected) cycles [2] have a zero weight. We call this transformation internal shifting. But, when internal shifting is not sufficient for parallelizing a loop, another question arises: Can we take advantage of a shift along a surrounding loop so that the innermost loop becomes parallelizable by shifting? This can indeed happen because shifting along a surrounding loop can change the edges of the dependence graph that are not carried, i.e., the edges to be considered for parallelizing the innermost loop. In other words, if we are able to carry the "good" dependences with the surrounding loop, the innermost loop may become parallelizable. This is what we call external shifting.

Example 3

Example 3 is a case for which a well-chosen shift of the surrounding loop can transform the dependences in a way that allow a subsequent internal shift and parallelization. In the code of Figure 6, the computation of `b[i,j]` (statement S2) depends on two values of the array `a` that are computed

[2]We call cycle a path from a vertex to itself where edges can be followed backwards or forwards. If all edges are followed forwards, the cycle is a circuit.

by statement S1 in the same iteration of the surrounding loop but in different iterations of the innermost loops (if they were fused). This situation prevents the parallel fusion of the two loops because the dependence graph to consider for the fusion of the two loops have two edges from S1 to S2 with weights 0 and 1, thus an undirected cycle of nonzero weight.

```
DO i=1,N
  DOALL j=1,M
    S1: a[i,j]=b[i-1,j-1]
  ENDDO
  DOALL j=1,M
    S2: b[i,j]=a[i,j]+a[i,j-1]
  ENDDO
ENDDO
```

Figure 6: Initial code for Example 3.

```
DOALL j=1,M
  a[1,j]=b[0,j-1]
ENDDO
DO i=2,N
  DOALL j=1,M
    S1: a[i,j]=b[i-1,j-1]
  ENDDO
  DOALL j=1,M
    S2: b[i-1,j]=a[i-1,j]+a[i-1,j-1]
  ENDDO
ENDDO
DOALL j=1,M
  b[N,j]=a[N,j]+a[N,j-1]
ENDDO
```

Figure 7: Code after external shifting.

Now, with a shift of the surrounding loop, these two dependences become carried by the surrounding loop. This results in two nested loops that are still sequentialized (Figure 7, assuming $N \geq 1$) but now, internal shifting enables the parallel fusion. Indeed, the only dependence not carried by the outer loop is now from S2 to S1, with weight 1. The dependence graph has no undirected cycle of nonzero weight since it has no cycle. In the final code, both statements are in the same loop (Figure 8, assuming $N \geq 1$ and $M \geq 1$). We have been able to eliminate the synchronization between the two parallel loops. Here, we can even push S1' and S2' out of the i loop, since all dependences are nonnegative (Figure 9, assuming $N \geq 1$ and $M \geq 1$). $\Box$

To summarize, the "external shifting problem" is to find a shift of the surrounding loop such that the dependence graph to consider for the innermost loop has only zero-weight undirected cycles (so that internal shifting can make the innermost loop parallel). We will prove in Section 5.2 that this problem is strongly NP-Complete.

Nevertheless, for practical implementations, we could try to identify particular cases that are simpler to solve. The first subproblem that comes in mind is to find a shift of the surrounding loop such that the dependence graph of the innermost loop is a forest because, in this case, there is no cycles at all and internal shifting will lead to a parallel loop. Unfortunately, we will see that the graph built in our NP-completeness proof is such that the only way to eliminate all the nonzero-weight cycles is also the only way to get a forest or even a set of chains. Therefore, finding an external shift such that the dependence graph of the innermost loop is a forest (resp. a set of strings) is also NP-complete!

The simplest subproblem we could think of is then the problem of finding an external shift such that the dependence graph of the innermost loop has only zero-weight edges. In this case, there will be no need of a further internal retiming because the innermost loop will be already parallel. We will see that this last problem can be solved efficiently.

```
DOALL j=1,M
  a[1,j]=b[0,j-1]
ENDDO
DO i=2,N
  S1': a[i,1]=b[i-1,0]
  DOALL j=2,M
    S2: b[i-1,j-1]=a[i-1,j-1]+a[i-1,j-2]
    S1: a[i,j]=b[i-1,j-1]
  ENDDO
  S2': b[i-1,M]=a[i-1,M]+a[i-1,M-1]
ENDDO
DOALL j=1,M
  b[N,j]=a[N,j]+a[N,j-1]
ENDDO
```

Figure 8: Code after external and internal shiftings.

```
DOALL j=1,M
  a[1,j]=b[0,j-1]
ENDDO
DOALL i=2,N
  S1': a[i,1]=b[i-1,0]
ENDDO
DO i=2,N
  DOALL j=2,M
    S2: b[i-1,j-1]=a[i-1,j-1]+a[i-1,j-2]
    S1: a[i,j]=b[i-1,j-1]
  ENDDO
ENDDO
DOALL i=2,N
  S2': b[i-1,M]=a[i-1,M]+a[i-1,M-1]
ENDDO
DOALL j=1,M
  b[N,j]=a[N,j]+a[N,j-1]
ENDDO
```

Figure 9: Final code after distribution (not always possible though).

3 General Properties

Loop shifting means to delay the execution of each statement of the loop by a fixed number of iterations (the shift), possibly different for each statement. In Section 4, we will first restrict to what we call internal shifting, which means shifting along the loop we want to parallelize (and possibly along the loops it contains). In Section 5, we will address the problem of external shifting, which means shifting also along loops surrounding the loop we are interested in.

3.1 Dependence Graphs and Correct Codes

For the sake of simplicity, we first assume that we are given a sequence of simple [3] loops with the same iteration domain, the same loop step equal to 1, the same loop counter (after loop peeling, loop reversal, and renaming if necessary). We model the sequence of loops by a dependence graph, i.e., a directed graph $G = (V, E, w)$ where the set of vertices V is the set of generic instructions (an instruction that appear textually inside one of the loops), the set of edges E is the set of dependences between instructions and w is an integer weight over the edges (a mapping from E into $\mathbb{Z}$). In other words, we assume uniform dependences as in all previous examples (we will mention, each time this is required, what happens for direction vectors). We define the weight of a path p in the graph as $w(p) = \sum_{e \in p} w(e)$.

If G comes from a correct code, an edge e of weight $w(e)$ from a vertex u to a vertex v means that the computation of u for a given iteration i has to be executed before the computation of v for the iteration $i + w(e)$ (even if, in the original code, u and v do not belong to the same loop). Furthermore, if u and v belong to two different loops then the loop containing u has to be textually

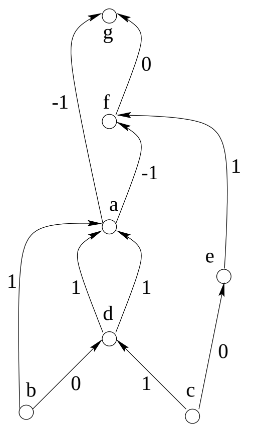
[3]Internal shifting for nested loops will be addressed in Section 4.2 and external shifting in Section 5.

before the loop containing v. If they belong to the same loop then $w(e) \geq 0$ and furthermore, if $w(e) = 0$, then u has to come textually before v. In other words, all dependences with negative weight belong to different strongly connected components and there is no circuit of zero weight. A consequence is that all circuits have a positive weight. Example 4 illustrates these facts. We say that a graph that has only positive-weight circuits is a legal graph.

Example 4

The following code is a toy program with its reduced dependence graph.

```
DO i=1,N
   a[i]=d[i-1]*d[i-1]+b[i-1]
   b[i]=sin(h[i])
   c[i]=cos(h[i])
   d[i]=b[i]+c[i-1]
   e[i]=sin(c[i])
ENDDO
DO i=1,N
   f[i]=a[i+1]+e[i-1]
   g[i]=f[i]*a[i+1]
ENDDO
```



We will illustrate in Section 4.1 how internal shifting can be used to parallelize this code into a single parallel loop. □

When dealing with nested loops, we consider a dependence graph $G = (V, E, w)$ where the mapping $w : E \longrightarrow \mathbb{Z}^n$ assigns to each edge an integral vector of dimension n. We say that G is a graph of dimension n: n is the number of loops surrounding each statement and the ith component $w(e)_i$ of the vector $w(e)$ is the dependence distance relative to loops at depth i in the nests. We denote by $\tilde{G}$ the graph obtained from G by keeping only the edges e that are loop independent with respect to the outermost dimension, i.e., such that the first component of $w(e)$ is zero, and we define the weight of an edge e in $\tilde{G}$ as the vector of dimension $(n-1)$ whose components are the $(n-1)$ last components of $w(e)$. We denote by G_i the graph deduced from G by restricting the weights to their ith component: G_i is a graph of dimension 1.

Proposition 1 *If G is the dependence graph of a correct code, then all circuits of G have a lexico-positive weight where the lexicographic order is defined as follows:*

$$a <_{lex} b \iff \exists k \in \{1, \ldots, n\},\ a_i = b_i \text{ for } i \in \{1, \ldots, k-1\}, \text{ and } a_k < b_k.$$

Proof: We prove this property by induction, as an extension of the case $n = 1$ studied above.

Let us view the code as a sequence of simple loops by considering that a statement is any instruction (even a loop) surrounded by only one loop: the associated dependence graph of dimension 1 is nothing but G_1 where all vertices that correspond to statements with several common surrounding loops are merged into a single vertex (and self-loops with zero weight are ignored). Therefore (case $n = 1$), G_1 has only nonnegative-weight circuits, and the weights between two vertices in the same merged group are nonnegative. A zero-weight circuit in G_1 is possible, but it is a circuit with zero-weight edges, and involving vertices of the same merged group, i.e., vertices surrounded by at least two common loops. By induction hypothesis (case $n-1$), the graph $\tilde{G}$ (restricted to this group, i.e., to statements in these two loops) is the graph of a correct code, thus it has only lexico-positive weight circuits. Thus, either the weight of a circuit in G has a first positive component, or its first component is zero and its last $(n-1)$ components form a lexico-positive vector of dimension $n-1$. In other words, G has only circuits of lexico-positive weight. $\square$

Note that Proposition 1 also holds if dependence weights are direction vectors, i.e., vectors whose components belong to $\mathbb{Z} \cup \{*, +, -\} \cup (\mathbb{Z} \times \{+, -\})$ with the natural addition defined for such vectors. We will come back to direction vectors in Proposition 3 and in Section 7.4.

We call legal a graph such that all circuits have a lexico-positive weight. As shown by Proposition 1, this is a necessary condition for a graph to correspond to a correct code, but it is not sufficient. However, we will see in Proposition 2 that this is a necessary and sufficient condition for a graph to correspond to a correct code transformed by loop shifting.

3.2 Shifting and Shortest Path Properties

The shifting technique is well known in the context of VLSI technology (and called retiming, see [21]) that has been intensively studied to solve many circuit optimization problems (see [29]). The use of retiming as a loop shifting technique for program transformations is not new and has been mainly used for software pipelining (see [5, 3, 8]), taking advantage of results from the VLSI community. We can also notice some attempts to study the possibilities of retiming as a tool for loop parallelization (see [26, 20, 10]) or for other code optimizations (see [14]). In the following, we will use indifferently a shift or a retiming of a dependence graph as a function assigning an integer value $r(u)$ (scalar or vector) to each vertex u of the graph.

The principle of a scalar retiming is to move an operation u from iteration i to iteration $i + r(u)$. Therefore applying a retiming $r : V \longrightarrow \mathbb{Z}$ to a dependence graph G of dimension 1 will change the weights of its edges, because for an edge $e = (u, v)$, the dependence is now from iteration $i + r(u)$ to iteration $i + w(e) + r(v)$, therefore a dependence distance equal to $w(e) + r(v) - r(u)$. For a dependence graph of dimension n, a (multidimensional) retiming is the composition of n onedimensional retimings, one for each graph G_i. We let $G_r = (V, E, w_r)$ the graph obtained from G after the application of a retiming r: the weight w_r after retiming is such that, for any edge $e = (u, v)$, $w_r(e) = w(e) + r(v) - r(u)$.

Note that a shift does not destroy our assumption for the legality of a graph (all circuits with lexico-positive weight) since a retiming does not change the weight of a circuit (see [21], this is a simple summation of the retimed weights on a circuit). Furthermore, using the terminology of Leiserson and Saxe, we say that a retiming r is a legal retiming for G if, in G_r, all edge weights are nonnegative for a onedimensional retiming, and lexico-nonnegative for a multidimensional retiming. We now recall how to find a legal retiming for a graph G.

Proposition 2 *Given a graph $G = (V, E, w)$, there exists a legal retiming r for G if and only if G has no circuit of lexico-negative weight.*

Proof: Assume first that G has no circuit of negative weight. Then, for each vertex u, the shortest (i.e., with smallest weight) path $\pi(u)$ leading to u can be defined. Because π is a shortest path, then for each edge $e = (u, v) \in E$, $\pi(v) \geq \pi(u) + w(e)$ since it is always longer (or equal) to go to v through u. Therefore, for each edge, we have $-\pi(v) + \pi(u) + w(e) \geq 0$, and $-\pi$ is a legal retiming for G.

Conversely, assume that there exists a legal retiming r for G. Let c be a circuit of G. Because the retiming keeps the weight of a circuit unchanged, $w_r(c) < 0$, which contradicts the fact that r is legal (all edges have a nonnegative weight).

Note: in the previous proof, $\geq$ is the classical order in $\mathbb{Z}$ if G is a graph of dimension 1. Otherwise, the proof is the same for a graph of dimension n, but where $\geq$ denotes the lexicographic order. □

This legal retiming can be found by computing the shortest path leading to any vertex using a variant of the Bellman-Ford algorithm in $O(|V||E|)$ steps, where each step involves the lexicographic order of dimension n. Note also that, by this construction, the retiming $-\pi$ we obtain is nonnegative. A legal retiming can also be found with n calls to a onedimensional Bellman-Ford algorithm: consider first G_1 to find a legal (onedimensional) retiming for the first dimension (possible since G_1 has no circuit with negative weight), then start again recursively with $\tilde{G}$ for the $(n-1)$ last dimensions.

Proposition 2 shows that, starting from a dependence graph with lexico-positive weight circuits, there always exists a way to shift the corresponding statements so that all dependence vectors become lexico-nonnegative. Furthermore, since there is no zero-weight circuits, the statements can be reordered so that all zero-weight dependences follow the textual order. In other words, a sequence of arbitrarily nested loops, with uniform dependences and such that all statements are surrounded by n loops, can always been transformed (by shifting) into a single set of n nested loops (possibly with a prelude and postlude for each dimension).

Note however that this situation is slighly different if we assume that the dependence weights are direction vectors. Indeed, in the proof of Proposition 2, the shortest paths π are not necessarily integral vectors but may belong also to $\{+, -, *\}$, which is not desirable for defining a shift. Thus, we will not be able to transform the code into a single set of n nested loops. But, as the following proposition shows, we can still get a correct code by combining loop shifting and loop distribution.

Proposition 3 *If G is a dependence graph with n-dimensional direction vectors such that all circuits have a lexico-positive weights, then there exists a shift and a way to generate the loops so as to get a correct code.*

Proof: Since G has only circuits with lexico-positive weights, then G_1 has only circuits with nonnegative weights. We can first compute the strongly connected components of G and apply a corresponding loop distribution. Now, in each strongly connected component, we can use the Bellman-Ford algorithm and compute the shortest paths as in Proposition 2. This time, since all edges belong to a cycle, there is no edge with weight involving $*$ or $-$. Therefore, the shortest-path values are integers. A shift can be defined so that all dependence weights in G_1 are nonnegative. The construction goes on for the graphs $\tilde{G}$ corresponding to each strongly connected component.

Note: the previous proof uses maximal loop distribution. This technique can be refined so as to get better loop fusion, by computing the shortest paths in the whole graph (while handling the weights $*$ and $-$ smartly) so as to generate separate loops only if there is a weight $*$ or $-$ between the two loops. □

To conclude this section, starting from a n-dimensional dependence graph that has only circuits with lexico-positive weights, it is easy to find a shift that leads to a correct code with all statements (in the case of uniform dependences) inside the same sequential loops (except for preludes and postludes). The real difficulty will arise when we introduce loop parallelism, i.e., when we require some loops to be parallel. This question is the heart of the paper and is addressed in the next two sections.

4 Parallelization by Internal Shifting

In this section, we deal with the case of a sequence of loops that has to be grouped so as to get an outermost parallel loop (or a block of outermost parallel loops), i.e., a loop whose iterations are independent. For that, we want to find a retiming of G such that all dependences become loop independent, in other words have a zero weight after retiming. This problem is already known as loop alignment (see [34]) but the solution below has two advantages: it unifies this problem with other retiming problems and it solves it with a lower complexity than the algorithm suggested in [34, p. 404] (which looks for zero-weight circuits in a directed weighted graph).

We start with the case of a sequence of simple loops, i.e., of a dependence graph of dimension 1.

4.1 Onedimensional Case

We first state the conditions for a retiming r to exists such that the retimed graph G_r has only edges of zero weight. These results can be found under a slightly different form in [9].

We recall that a cycle is a undirected path from a vertex to itself where edges can be followed backwards or forwards. If all edges are followed forwards, the cycle is a circuit. An elementary cycle (resp. circuit) is a cycle (resp. circuit) that does not contain twice the same vertex. An elementary cycle c can be represented by a function $\mu_c : E \longrightarrow \{-1, 0, +1\}$ defined as follows:

$$\forall e \in E,\, \mu(e) = \begin{cases} +1 & \text{if } e \text{ is a forward edge of } c \\ -1 & \text{if } e \text{ is a backward edge of } c \\ 0 & \text{otherwise, i.e., if } e \text{ does not belong to } c \end{cases}$$

We define the weight of a cycle c in the following way:

$$w(c) = \sum_{e \in c} \mu_c(e) w(e)$$

and we prove that a retiming does not change the weight of a cycle.

Proposition 4 *Given a cycle c of a graph G, $w(c) = w_r(c)$ for any retiming r of G.*

Proof: Without loss of generality, we can assume that c is elementary (otherwise it can be decomposed into a union of elementary cycles).

$$\begin{aligned} w_r(c) &= \sum_{e \in c} \mu_c(e) w_r(e) = \sum_{e=(u,v) \in c} \mu_c(e)(w(e) + r(v) - r(u)) \\ &= \sum_{e \in c} \mu_c(e) w(e) + \sum_{e=(u,v) \in c} \mu_c(e)(r(v) - r(u)) \end{aligned}$$

Since c is elementary, each vertex w in the cycle belongs to exactly two edges e_1 and e_2 of the cycle. Two cases can occur:

- e_1 and e_2 are followed in the same direction; in this case $\mu_c(e_1) = \mu_c(e_2)$ and w appears exactly twice in the sum $\sum_{e=(u,v) \in c} \mu_c(e)(r(v) - r(u))$, once as $r(w)$ and once as $-r(w)$.
- e_1 and e_2 are followed in opposite directions (one forwards, one backwards); in this case $\mu_c(e_1) = -\mu_c(e_2)$ and w appears exactly twice in the sum $\sum_{e=(u,v) \in c} \mu_c(e)(r(v) - r(u))$, either both as $r(w)$ or both as $-r(w)$.

It follows that $\sum_{e=(u,v)\in c} \mu_c(e)(r(v) - r(u)) = 0$, and therefore $w_r(c) = w(c)$. □

Now we state the conditions for a graph to be retimed so that all its edges have a zero weight. We start with an intermediate result, which is a subcase of our problem:

Lemma 1 *For any graph G with no (undirected) cycles, there exists a retiming r such that G_r has only zero-weight edges.*

Proof: The proof is a straightforward constructive proof. We pick an arbitrary vertex v_n for each connected component n of G and we set $r(v_n) = 0$. Then, we apply a marking procedure of each component from v_n, following edges in any direction, and we choose, for each new edge $e = (u, v)$, the correct value $r(u)$ or $r(v)$, depending on the direction, such that $r(v) - r(u) + w(e) = 0$. Because the value is fixed for the vertex which we come from, the choice is unique, and because there is no cycle, each vertex is marked only once: thus, there is no ambiguity for defining r. Finally, since we apply the procedure for each connected component, all edges are visited and they all have a zero weight after retiming. □

We now give a necessary and sufficient condition for the existence of the desired retiming.

Theorem 1 *Let $G = (V, E, w)$ be a graph. Then, there exists a retiming r such that G_r has only zero-weight edges if and only if G has only zero-weight cycles.*

Proof: The first part is obvious. If, for a given retiming r, G_r has only zero-weight edges, then the weight $w_r(c)$ of any cycle c in G is zero. Since a retiming does not change the weight of cycles (Proposition 4), then G has only zero-weight cycles too.

Now, assume that G has only cycles of zero weight. Let T be a spanning tree of G (considered as an undirected graph). T has no cycle at all (because this is a tree) thus, by Lemma 1, there is a retiming r such that T_r has only edges of zero weight. Now assume that G_r has an edge e of nonzero weight, then adding e to T forms a cycle c (because T is a spanning tree). This cycle has a nonzero weight (equal to $\pm w_r(e)$ depending on the direction of e in c) since all edges in c, except e, have a zero weight. Therefore, $w_r(c) \neq 0$ and, by Proposition 4, $w(c) = w_r(c) \neq 0$: G has a cycle with nonzero weight, a contradiction. Thus, all edges of G_r have a zero weight and r is the desired retiming. □

Theorem 1 shows that is not sufficient that the graph has only zero-weight circuits. It must have only zero-weight **cycles**, and this property will make everything more complicated when considering external retiming. Note also that this characterization contradicts the result of [20], which states that a graph without circuits can always be retimed to have only zero-weight edges (this property is wrong, a graph with no *circuit* may have a nonzero-weight *cycle*). A similar mistake seems to have been made in [10], even if this property is not formally stated. However, the result of Theorem 1 is not new, it was stated by Peir [27, Lemma 4.1].

The algorithm suggested in [34], to transform by retiming a sequence of simple loops with constant dependence distances into a single parallel loop, is to add, for each edge (u, v) in G, a new edge (v, u) with opposite weight and to check that this graph has only zero-weight circuits. This can be done with the Bellman-Ford algorithm with complexity $O(|V||E|)$. Actually, there is no need to work on this directed graph, which is very redundant. The proof of Theorem 1 suggests a much simpler algorithm based on a simple graph traversal of G considered as an undirected graph. Here is the algorithm that corresponds to the constructive proof of Theorem 1. Below, we assume that G is connected, otherwise we apply the procedure on each connected component.

Algorithm: Perfect Alignment (G, r)

```
• for each vertex $v$ of $G$, set $m(v) = 0$.
• set $S = \{v\}$ where $v$ is an arbitrary vertex of $G$, set $m(v) = 1$ and $r(v) = 0$.
• while $S$ is not empty, choose $v \in S$, and set $S = S - \{v\}$:
    – for each edge $e = (u, v)$ with $m(u) = 0$, set $r(u) = r(v) + w(e)$, $m(u) = 1$, and $S = S \cup \{u\}$.
    – for each edge $e = (v, t)$ with $m(t) = 0$, set $r(t) = r(v) - w(e)$, $m(t) = 1$, and $S = S \cup \{t\}$.
• if $G_r$ has only zero-weight edges, then return TRUE, else return FALSE.
```

Initializations are $O(|V|)$ and the marking procedure is $O(|V|+|E|)$. Checking that all edges have a zero weight after retiming is $O(|E|)$ giving a total complexity $O(|V|+|E|)$. This obvious algorithm is nothing but Algorithm 4.1 in [27], but it seems to have been forgotten.

Back to Example 4

Figure 10 gives the solution found for Example 4. The dependence graph has only zero-weight cycles, therefore there is a retiming that leads to a unique parallel loop (here the retiming is naturally nonnegative, but this is a particular case, it depends which starting vertex we use).

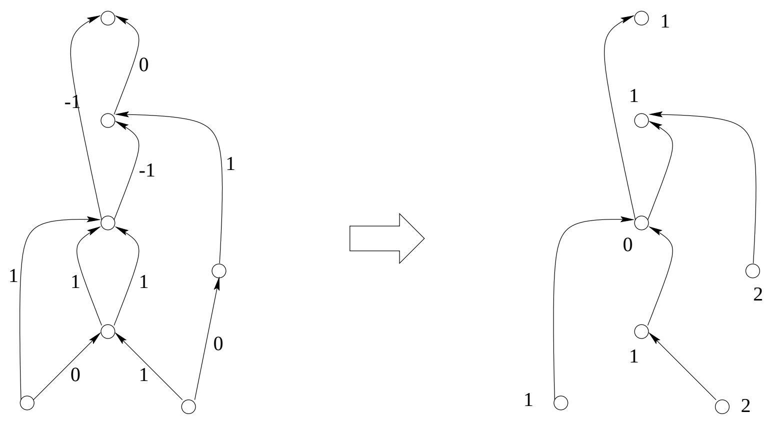
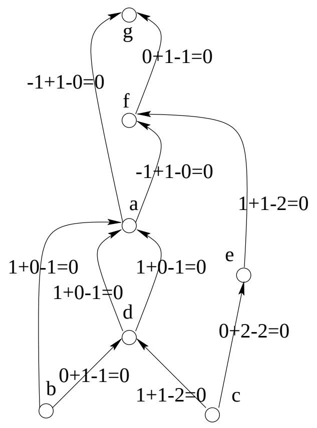


Figure 10: Parallelizable graph with a spanning tree and its according retiming.

The resulting dependence graph and the corresponding code are given in Figure 11. All dependence distances are equal to zero, and since we assumed that the initial dependence graph has only positive weight circuits, there is no circuit after retiming with zero weight (actually, there is no circuit at all, since a positive weight circuit will imply a nonzero weight cycle). We can thus reorder the statements in the code so that all directed edges follow the textual order. To make the code simpler, we omitted the details of the required prelude and postlude. □

4.2 Extension to Nested Loops

In this section, we see how the algorithm for obtaining a single parallel loop from a succession of simple loops can be extended to (fuse and) parallelize several loops of a succession of loop nests. There are two possible extensions of the perfect alignment problem defined in Section 4.1: the



```
prelude
DO i=3,N
   b[i-1]=sin(h[i-1])
   c[i-2]=cos(h[i-2])
   d[i-1]=b[i-1]+c[i-2]
   a[i]=d[i-1]*d[i-1]+b[i-1]
   e[i-2]=sin(c[i-2])
   f[i-1]=a[i]+e[i-2]
   g[i-1]=f[i-1]*a[i]
ENDDO
postlude
```

Figure 11: Retimed dependence graph and parallelized code for Example 4.

problem of transforming a sequence of loop nests of same depth n into n nested parallel loops, and the same problem when only the d outermost loops are parallel.

The first problem is very similar to the perfect alignment of simple loops studied before. Indeed, all previous results can be used either dimension by dimension (because a cycle has a zero weight if and only if, in each dimension, its weight is zero) or by directly considering vectors instead of scalar values. Therefore, a multidimensional retiming exists such that all dependence vectors are zero if and only if the dependence graph has only zero-weight cycles, and the same perfect alignment algorithm can be used with vectors instead of scalars.

However, a (small) complication arises for the second problem, i.e., when we want to group a sequence of loop nests of same depth n (represented by a dependence graph G of dimension n) into n nested loops so that only the d outermost loops are parallel. The problem is that if we apply the previous perfect alignment algorithm just for the first d dimensions, then an edge with negative weight may appear that was carried before by one of the d first loops. But, the graph obtained by considering only the remaining $(n-d)$ dimensions is still legal since all circuits in G have a lexico-positive weight and their first d components are zero. Thus, all circuit weights considering the last $(n-d)$ dimensions are also lexico-positive. Therefore, using Proposition 2 (or Proposition 3 in the case of direction vectors), we can get a correct code thanks to an extra retiming in the remaining dimensions.

This remark allows us to find a retiming that makes the first loop parallel while ensuring that the graph still satisfies the legality constraints. Consequently, applying the perfect loop alignment algorithm, loop after loop, starting from the outermost loop one until a loop that cannot be parallelized is found gives a simple and efficient way to find a maximal group of surrounding parallel loops. Once an impossibility is found, an extra retiming can fix the problem of lexico-negative weights in the remaining dimensions. Figure 12 illustrates the technique.

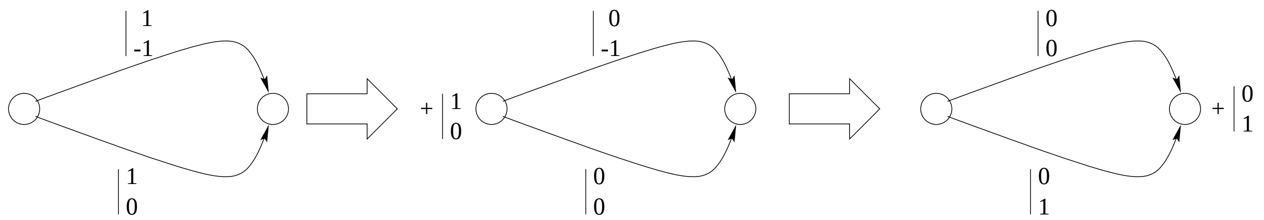


Figure 12: Parallelization of the outermost loop only.

To conclude this section, parallelization by internal shifting, i.e., parallelization of outermost loops by shifting along the same dimensions (and possibly along inner dimensions) is easy. One has just to check that the dependence graph has only zero-weight cycles and to shift statements so that all dependences become loop independent in the required dimensions (and lexico-positive in the remaining dimensions). In the next section, we address the problem of external shifting, which is how to shift statements along a dimension so that parallelization of an inner loop is feasible by internal shifting. As we will see, it turns out that this problem is much more complex.

5 Parallelization by External Retiming

In this section, we address the case where internal shifting is not sufficient for parallelizing loops, in other words, when whatever the dimension we consider, the dependence graph of the corresponding portion of code has a cycle of nonzero weight. The idea is then to try to exploit the sequentiality of the outermost loops and the dependences they carry. Because a dependence carried by an outer loop does not prevent the parallelization of inner loops, what we want is shift along the outer loops (this is what we call external shifting) so as to carry by outer loops the "good" dependences and allow the rest of the nest to be parallelizable by internal shifting. The resulting code in 2 dimensions will be typically a sequential loop surrounding a parallel loop.

We first study the problem where the x innermost loops of a nest of dimension n have to be made parallel. Then, we look at the subproblem of parallelizing the inner loop of a nest of two loops, which we prove to be strongly NP-complete. It follows that more general parallelization problems, such as how to shift so that a given loop is parallel, or how to shift so that as many loops as possible are parallel, are also NP-complete. Nevertheless, we will show that formulations by systems of integer linear constraints are possible.

5.1 Characterization of the Problem

In this section, we first characterize a subproblem, the problem of finding a shift so as to obtain all statements (except for preludes and postludes) surrounded by the same sequential outer loops and the same nest of parallel loops. We consider a graph $G = (V, E, w)$ of dimension n for which we want to parallelize the x innermost loops, i.e., we want to find a multidimensional retiming r such that there is no dependence carried by one of these x loops (of course $1 \leq x < n$). This happens when each dependence is either carried by one of the $d = n - x$ first loops, or not carried by any loop, i.e., is loop independent. More formally, the x innermost loops are parallel when:

$$\forall e = (u, v) \in E,\ w(e) \geq_{lex} (\underbrace{0, \ldots, 0}_{d-1}, 1, \underbrace{-\infty, \ldots, -\infty}_{x}) \text{ or } w(e) = 0$$

Note that, with this definition, we are looking for retimings that produce only lexico-nonnegative weights, in other words legal retimings. This is because we want a perfect nest as a result (except for preludes and postludes), which corresponds to a dependence graph with lexico-nonnegative weights (lexico-negative weights are allowed only between successive loops, but not for nested loops). Thanks to Proposition 2, we can assume that we start from a graph whose dependence weights are already lexico-nonnegative. If a retiming is needed to obtain this graph, it can be added to the final result producing a single retiming for the whole problem.

We first show an intermediate result that states that, to carry dependences by the d first loops, a retiming in the dth dimension only is as efficient as a retiming in the d first dimensions. In other words, if the d first loops have to be sequential, we can replace a combination of d retimings in the d first dimensions by a single retiming in the dth dimension [4].

Theorem 2 *Let G be a graph of dimension d with only lexico-nonnegative weights. For any legal multidimensional retiming r of G, there exists a legal retiming r' of the dth dimension only, such that all the edges that are loop carried in G_r are also loop carried in $G_{r'}$.*

Proof: Consider $G' = (V, E', w')$ the subgraph of G obtained by setting

$$E' = \{e \in E \mid w(e) <_{lex} (0, \dots, 0, 1, -\infty)\}$$

in other words all edges not carried by one of the $d-1$ first loops in G, and

$$\forall e \in E',\, w'(e) = \begin{cases} w(e)_d & \text{if } w_r(e) = (0, \dots, 0) \\ w(e)_d - 1 & \text{otherwise} \end{cases}$$

in other words, $w(e)_d$ if e is loop independent in G_r and $w(e)_d - 1$ if e is loop carried in G_r. We want to find a retiming r' of G' such that, for each $e \in E'$, $w'(e) + r'(v) - r'(u) \geq 0$ because if such a retiming can be found, all edges that are loop carried in G_r will be loop carried in $G_{r'}$, either by one of the first $d-1$ loop (as in the original graph G) or, by definition of w', in the dth dimension.

We know that we can find such a retiming if and only if G' has no circuit of negative weight. By definition of E', a circuit c of G' corresponds to a circuit of G such that all edges have a component equal to zero in their $(d-1)$ first dimensions. Since a retiming does not change the weight of a circuit, the edges of c have also a zero weight in their $(d-1)$ first components in G_r (otherwise one of them would have a lexico-negative weight). In other words, a dependence that is not loop carried in G for the $(d-1)$ first dimensions cannot be loop carried in G_r for the $(d-1)$ first dimensions. Furthermore, since r is legal (edges have lexico-nonnegative weights), all edges of c have a nonnegative dth component in G_r and $w_r(c)_d \geq p$ if p of them are loop carried in G_r. So, again because the weight of a circuit does not change by retiming, $w(c)_d \geq_{lex} p$ and by definition of w', $0 \leq w'(c) \leq +\infty$. Therefore, G' has no circuit of negative weight and we can find the desired retiming.

To say it differently (and more intuitively), retiming in the first $(d-1)$ dimensions cannot help for the circuits whose edges have a zero weight in the first $(d-1)$ dimensions. Furthermore, these circuits are the only "hard" constraints for retiming in the dth dimension since edges that are not part of a circuit can always been carried thanks to a sufficiently large retiming. Therefore, the multidimensional retiming r and the retiming r' in dimension d have the same "real" constraints. □

Theorem 2 shows that, without loss of generality, we can assume that we are looking for a single sequential loop surrounding a set of loops that have to be made parallel. Furthermore, as we saw in Section 4.2, parallelizing a single loop or a block of nested loops (possibly containing sequential loops) is actually the same problem. In both cases, we require the dependence graph to have only cycles of zero weight and the algorithm of Section 4.1 can be applied the same way just by looking for vectors as retiming values, or by reasoning dimension by dimension. So, we can also assume that we are looking for one single parallel inner loop. To summarize, we assume that we are given a nest of dimension 2 and that we are looking for a 2-dimensional retiming such that the outer loop is sequential and the inner loop is parallel.

[4]In practice however, if a onedimensional retiming in an outer dimension is sufficient, it may be more desirable to use it since this will push the prelude and postlude out of loops as much as possible.

The following proposition separates the problem in two subproblems, allowing us to search first for a retiming in the first dimension before looking at the second dimension. We recall that $\tilde{G} = (V, \tilde{E}, \tilde{w})$ denotes the graph deduced from $G = (V, E, w)$ by removing all the dependences carried by the first loop, that is $\tilde{E} = \{e \in E \mid w(e) <_{lex} (1, -\infty)\}$ and $\tilde{w}$ is the second dimension of the weight: $\tilde{w}(e) = w(e)_2$ if $e \in \tilde{E}$. The following proposition is a direct consequence of Theorem 1.

Proposition 5 *Given a graph G of dimension 2, there exists a legal multidimensional retiming r such that the inner loop of G is parallel if and only if there exists a legal retiming r_1 in the first dimension of G such that $\tilde{G}_{r_1}$ has only cycles of zero weight.*

We have now formally separated the problem in two subproblems, the problem of retiming along the outer loop that should carry the "good" dependences in order to allow the remaining dependences to become loop independent (this is what we called external shifting), and the problem of the remaining loop that has to be made parallel (this is what we called internal shifting). Figure 13 gives an example of external shifting. Dotted edges are those carried by the first loop. Consider the graph on the left: the outer loop cannot be parallelized by retiming since there is a cycle with nonzero weight (in the first dimension), for example one of the circuit. Now consider $\tilde{G}$ (the solid edges): it has no circuit, but a nonzero-weight cycle therefore the second loop cannot be parallelized by internal shifting. However, if we first shift in the first dimension, we can change the structure of $\tilde{G}$ so that it now has no cycle at all (thus, no cycle of nonzero weight): a shift in the second dimension can then be found that makes all remaining dependences loop independent.

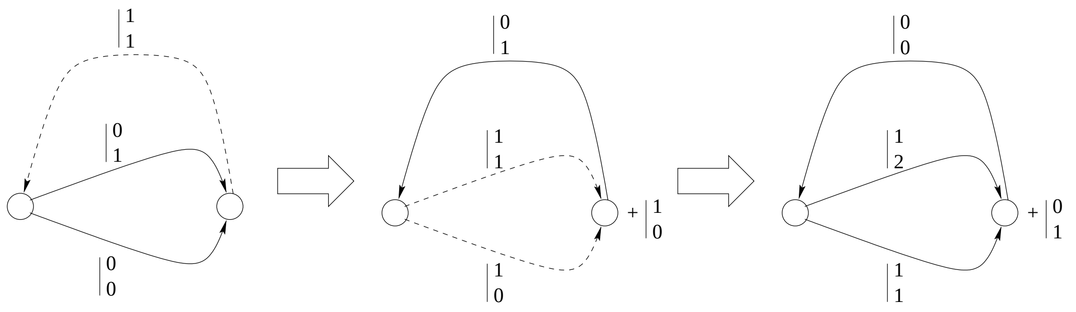


Figure 13: A two-step parallelization of the inner loop.

Even if, for the previous example, the suitable external shifting was obvious, we now show that the problem is, in general, NP-complete in the strong sense.

5.2 NP-Completeness

The goal of this section is to prove that the problem of parallelizing the innermost loop by retiming is NP-complete for a graph of dimension 2. It follows that the problem is also NP-complete for nested loops of larger depth. Note that, except if P=NP, this goes against the results of [20] where a polynomial algorithm is given to solve the problem (actually it seems that their necessary and sufficient condition of feasibility is neither necessary nor sufficient).

The proof is by a polynomial reduction from 3SAT (problem LO2 in [15, p. 259]). As we will see, the graph G built by the reduction is a positive instance of our problem if and only if a solution r_1 (a retiming) is such that $\tilde{G}_{r_1}$ has no cycles at all. This remark proves that, as stated in Section 2, the other problem of finding a retiming r_1 in the first dimension of G such that $\tilde{G}_{r_1}$ has no cycles at all is also NP-complete. We first recall the formulation of 3SAT.

Problem: 3SAT

Instance A set U of n boolean variables and C of m clauses over U (for each variable u of U, we denote by u and $\bar{u}$ literals over U, a clause over U is a set of literals over U) such that for each clause $c \in C$, $|c| = 3$.

Question Does there exist a truth assignment for U (i.e., a function $t : U \longrightarrow \{T, F\}$, with $t(\bar{u}) = T$ if $t(u) = F$ and $t(\bar{u}) = F$ if $t(u) = T$) such that every clause in C has at least one true literal (i.e., at least one $u \in C$ such that $t(u) = T$)?

Because of Proposition 5, our problem in the case of dimension 2 is stated as follows.

Problem: 2D EXTERNAL SHIFTING

Instance A directed graph $G = (V, E, w)$ of dimension 2, where V is the set of vertices of G, $E \subset V^2$ the set of edges, $w : E \longrightarrow \mathbb{Z}^2$ a 2-dimensional weight function over the edges.

Question Does there exists a legal retiming $r : V \longrightarrow \mathbb{Z}$, in the first dimension, such that $\tilde{G}_r$ has only zero-weight cycles?

We first need an intermediate result before the complete proof. The following lemma states that we can look for retiming values that are bounded when trying to carry some dependences.

Lemma 2 *For any legal retiming r of a graph $G = (V, E, w)$ of dimension 1, there exists a legal retiming r' of G such that $\forall v \in V$, $|r'(v)| \leq \sum_{e \in E} |w(e)| + |V|$ and such that $G_{r'}$ has exactly the same loop-carried edges as G_r.*

Proof: We build, from G and r, a graph $G' = (V, E', w')$ such that $E \subset E'$. If $e = (u, v) \in E$ is such that $w_r(e) = 0$, we let $w'(e) = w(e)$ and we add in E' an edge $e' = (v, u)$ with weight $w'(e') = -w(e)$ (e and e' form a zero-weight circuit). If $e = (u, v)$ is such that $w_r(e) \geq 1$, we let $w'(e) = w(e) - 1$.

By construction, G'_r has only nonnegative edges. Therefore, r is legal for G' and, by Proposition 2, G' has only nonnegative circuits. We can thus find a retiming r', as in Section 3.2, defined as a shortest path, such that for any edge $e = (u, v) \in E', w'(e) + r'(v) - r'(u) \geq 0$. Defined this way, r' is nonnegative. By construction of G', after this new retiming r', exactly the same edges as in G_r have a zero weight in $G_{r'}$, and all edges that are loop carried in G_r are loop carried in $G_{r'}$. Finally, because the shortest path can be defined as an elementary path, the new retiming r' satisfies $\forall v \in V$, $0 \leq r'(v) \leq \sum_{e \in E} |w'(e)| \leq \sum_{e \in E} |w(e)| + |V|$. □

We are now ready to prove that the problem of 2d external shifting is strongly NP-complete.

Theorem 3 *The problem* 2D EXTERNAL SHIFTING *is NP-complete in the strong sense.*

Proof: We first prove that the problem belongs to NP, then we describe how we transform an instance of 3SAT into an instance of 2D EXTERNAL SHIFT, and finally we show the equivalence between both.

The problem is in NP Consider a positive instance of our problem: a graph $G = (V, E, w)$ of dimension 2 and a legal retiming r in the first dimension such that $\tilde{G}_r$ has only zero-weight cycles. By Lemma 2, there is another retiming r' whose values are bounded as a polynomial function of the absolute values of the weights in the first dimension, and the number of vertices. This retiming, applied in the first dimension of G, leads to the same loop-carried edges as r, thus $\tilde{G}_r = \tilde{G}_{r'}$. Therefore, there is a polynomial certificate for our instance (the size of r' is polynomial in the instance size and checking that this retiming leads to a graph $\tilde{G}_{r'}$ that has only zero-weight cycles can be done in polynomial time). Therefore, the problem belongs to NP.

Transformation Given an instance (U,C) of 3SAT, we define $f(U, C) = G = (V, E, w)$ a transformation to an instance of 2D External Shifting.

First, we describe a construction that we will use as a basic component of the transformation, we call it a copy vertex. We build a copy vertex y of some vertex x of G as follows: we add the new vertex y to V, two edges from x to y of respective weights $(1, 0)$ and $(1, 1)$, and two edges from y to x of respective weights $(1, 0)$ and $(1, 1)$ (see Figure 14).

Now we construct G in the following manner:

- We start with a unique vertex r_G: $V = \{r_G\}$, $E = \emptyset$.
- For each variable $u \in U$, we add a copy vertex r_u of r_G, a new vertex v_u, and two edges $e_u = (v_u, r_u)$, $e_{\bar{u}} = (r_u, v_u)$ with $w(e_u) = (1, 0)$ and $w(e_{\bar{u}}) = (0, 0)$ (see Figure 15).

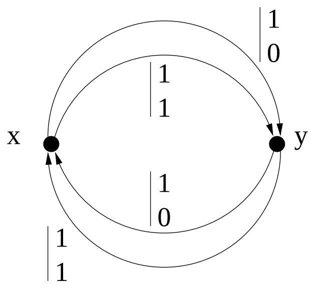


Figure 14: The copy structure.

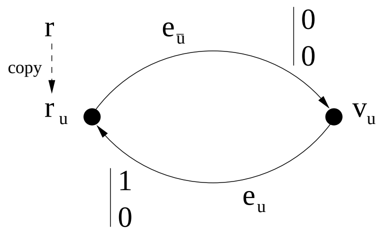


Figure 15: Transformation of a variable u.

- For each clause $c = \{x, y, z\} \in C$ (see Figure 16), we add three copy vertices $r^c_{x,y}$, $r^c_{y,z}$, $r^c_{z,x}$ of r_G, and for each literal $a \in c$ involving a variable u (i.e., either as $a = u$ or $a = \bar{u}$), we add:
  - a copy v^c_a of v_u.
  - if $a = u$, two new edges $e^c_a = (v^c_a, r^c_{a,.})$ and $e'^c_a = (v^c_a, r^c_{.,a})$, of respective weights $(1, 0)$ and $(1, 1)$ (i.e., the same weight, in the first dimension, as e_u).
  - if $a = \bar{u}$, two new edges $e^c_a = (r^c_{a,.}, v^c_a)$ and $e'^c_a = (r^c_{.,a}, v^c_a)$, of respective weights $(0, 0)$ and $(0, 1)$ (i.e., the same weight, in the first dimension, as $e_{\bar{u}}$).

Obviously the above transformation is polynomial, since we add a copy, a vertex and two edges for each variable, and six copies and six edges for each clause. Each copy representing a vertex and four edges, we have a total of $2|U| + 6|C| + 1$ vertices and $6|U| + 30|C|$ edges, so the size of the graph built by the transformation is $O(|U| + |C|)$.

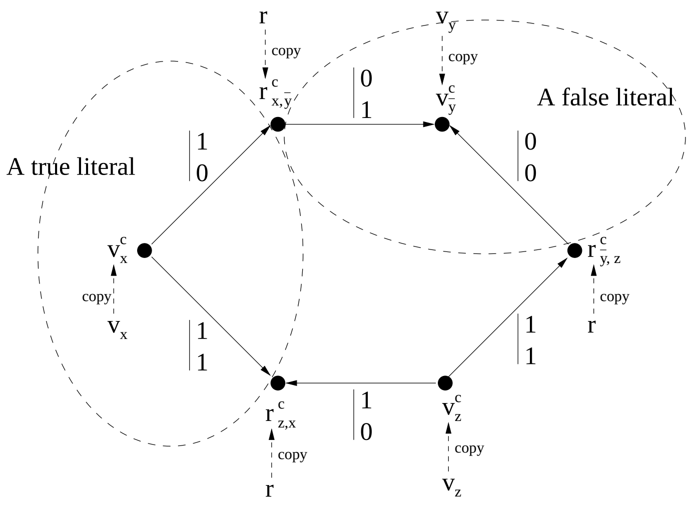


Figure 16: Transformation of a clause $c = \{x, \bar{y}, z\}$.

Reduction Let (U, C) be an instance of 3SAT and $G = (V, E, w) = f(U, C)$. We prove that:

- if (U, C) is a positive instance of 3SAT, then G is a positive instance of 2D External Shifting. First let t be a truth assignment over U satisfying all the clauses of C. We set:

$$r(r_G) = 0$$
$$\forall u \in U,\ r(r_u) = 0 \text{ and } r(v_u) = \begin{cases} 1 & \text{if } t(u) = F \\ 0 & \text{if } t(u) = T \end{cases}$$
$$\forall c \in C,\ r(r^c_{.,.}) = 0 \text{ and } \forall a \in c \text{ involving } u \in U,\ r(v^c_a) = r(v_u)$$

  Now we show that r, considered as a onedimensional retiming in the first dimension, is legal, and that $\tilde{G}_r$ has no cycle of nonzero weight (actually, it has no cycle at all):

  - by definition of r and by construction of G, all the copies of a given vertex have the same retiming value as their original, so the copy edges still have a (strictly) positive weight in the first dimension after retiming, and are not in $\tilde{G}_r$. Therefore, no copy vertex is connected to its original in $\tilde{G}_r$.
  - for each literal $u \in U$, either $r(r_u) = r(v_u) = 0$, $w_r(e_u) = (1, 0)$ and $w_r(e_{\bar{u}}) = (0, 0)$, or $r(r_u) = 0$, $r(v_u) = 1$, $w_r(e_u) = (0, 0)$ and $w_r(e_{\bar{u}}) = (1, 0)$. In any case, there is only one of the two edges, generated by a literal, that belongs to $\tilde{G}_r$. Moreover, these vertices r_u and v_u are connected to the rest of graph only through copy structures. Thus, with the previous remark, they are not connected to anything else in $\tilde{G}_r$ and the remaining edge (e_u or $e_{\bar{u}}$) is isolated in $\tilde{G}_r$: there is no cycle generated in $\tilde{G}_r$ by a literal. Furthermore, by definition of $r(v_u)$, we deduce that for any literal a, $w_r(e_a) = (1, 0)$ if $t(a) = T$ and $w_r(e_a) = (0, 0)$ if $t(a) = F$.
  - by construction, in each clause $c \in C$ and for any literal $a \in c$ (either $a = u$ or $a = \bar{u}$, for a variable $u \in U$), the two edges generated by the literal a, e^c_a and e'^c_a, have the same weight as e_a in their first dimension. Furthermore, by definition of r ($r(v^c_a) = r(v_u)$ and $r(r^c_{.,a}) = r(r^c_{a,.}) = r(r_u) = 0$) and by choice of the direction of these edges in the construction, e^c_a and e'^c_a have the same retiming values for their heads and tails as e_a. Thus, e^c_a and e'^c_a have the same retimed weight as e_a in the first dimension, that is 1

if $t(a) = T$ and 0 otherwise. Since at least one literal is true in each clause, at least one edge (actually a pair of "twin" edges) of the cycle generated by the clause c has a (strictly) positive weight in the first dimension, so the cycle is "cut" in $\tilde{G}_r$. Furthermore, because the vertices of c are all copies, the clause is not connected to anything else in $\tilde{G}_r$.

To summarize, $\tilde{G}_r$ has no cycle at all: it is a set of (undirected) chains of length 1 (for the variable structures), and 0, 2, or 4 (for the clause structures). Furthermore, all the retimed weights are nonnegative in the first dimension (and nonnegative in the second dimension) thus r is legal, and G is a positive instance of 2D External Shifting.

- if G is a positive instance of 2D external shifting, then (U, C) is a positive instance of 3SAT. First, let r be a legal retiming of the first dimension of G such that there is no nonzero-weight cycle in $\tilde{G}_r$. We set:

$$\forall u \in U,\ t(u) = \begin{cases} T & \text{if } w_r(e_u)_1 > 0 \\ F & \text{otherwise} \end{cases} \quad \text{and } t(\bar{u}) = \begin{cases} T & \text{if } w_r(e_{\bar{u}})_1 > 0 \\ F & \text{otherwise} \end{cases}$$

We first give some properties of the retiming values in G:

  – for each vertex y generated as a copy of some vertex x, by construction of the edges between them and since r is a legal retiming of the first dimension of G, the retiming values of x and y differ by at most one (otherwise a negative value would appear). Furthermore, since $\tilde{G}_r$ has no cycle of nonzero weight, $r(x)$ and $r(y)$ are equal, otherwise two of the copy edges in one of the directions would have a zero retimed weight in the first dimension, thus creating a nonzero cycle in $\tilde{G}_r$ (since their weights in the second dimension are not equal). In other words, any copy vertex has the same retiming value as its original.

  – for any clause $c \in C$ and any literal $a \in c$ involving a variable $u \in U$ (i.e., either $a = u$ or $a = \bar{u}$), since the vertices $r^c_{.,.}$ are copies of r_G and the v^c_a is a copy of v_u, we have $r(r^c_{.,.}) = r(r_G) = r(r_u)$ and $r(v^c_a) = r(v_u)$. The edges e^c_a and e'^c_a, generated by a, have by construction the same weight, in the first dimension, as e_a and as shown above, the same retiming value for their head and tail as copies of either those of e_a or r_G. Thus, they have the same retimed weight as e_a (in the first dimension).

We can now prove that t is a truth assignment (i.e., $t(u) \neq t(\bar{u})$) and that it satisfies all the clauses of C.

  – for each variable $u \in U$, $w(e_u) = (1, 0)$ and $w(e_{\bar{u}}) = (0, 0)$. Since r is legal and since the weight of a circuit does not change by retiming, we have either $w_r(e_u) = (1, 0)$ and $w_r(e_{\bar{u}}) = (0, 0)$, or $w_r(e_u) = (0, 0)$ and $w_r(e_{\bar{u}}) = (1, 0)$. This proves that $t(u) \neq t(\bar{u})$.

  – $\tilde{G}_r$ has no cycle of nonzero weight, and, by construction, the cycle generated by a clause has a nonzero weight in the second dimension. Indeed, there is three 1 in the cycle, so whatever the direction of the edges, the total weight of the cycle in the second dimension is -3, -1, 1, or 3. Thus, this cycle has to be "cut" in $\tilde{G}_r$: there is at least one edge with a positive weight in its first dimension. Let e be one of these edges: e corresponds to some literal a, and as shown above, it has the same weight in its first dimension as e_a. Thus, e_a has a positive weight in its first dimension, and by definition $t(a)$ is true. In other words, each clause has at least one true literal and therefore it is satisfied.

To summarize, t is a truth assignment that satisfies all the clauses of C: (C, U) is a positive instance of 3SAT.

This proves that (C, U) is a positive instance of 3SAT if and only if $f(C, U)$ is a positive instance of 2D EXTERNAL SHIFTING. Thus, 2D EXTERNAL SHIFTING is strongly NP-complete. □

5.3 Formulation by Linear Constraints

In this section, we propose a formulation of the external shifting problem as a system of integer linear constraints. For the sake of simplicity, we limit our study to the case where we look for one sequential loop surrounding a parallel subnest. Because of Theorem 2 and the remarks in Section 4, all other situations are similar and can be solved as variations of this problem. We have the following proposition.

Proposition 6 *Given a graph $G = (V, E, w)$ of dimension n, let $M_d = \sum_{e \in E} |w(e)_d|$ for $2 \le d \le n$. Then r is a legal n-dimensional retiming such that the $(n-1)$ innermost dimensions are parallel if and only if r is a solution of the following integer linear system:*

$$\begin{aligned}
\forall e = (u, v) \in E, \quad & M_2(w(e)_1 + r_1(v) - r_1(u)) + (w(e)_2 + r_2(v) - r_2(u)) \ge 0 \\
& M_2(w(e)_1 + r_1(v) - r_1(u)) - (w(e)_2 + r_2(v) - r_2(u)) \ge 0 \\
& \qquad \vdots \\
& M_n(w(e)_1 + r_1(v) - r_1(u)) + (w(e)_n + r_n(v) - r_n(u)) \ge 0 \\
& M_n(w(e)_1 + r_1(v) - r_1(u)) - (w(e)_n + r_n(v) - r_n(u)) \ge 0
\end{aligned}$$

Proof: Assume first that the system has a solution r. For any edge $e = (u, v)$, the first two constraints imply $2M_2(w(e)_1 + r_1(v) - r_1(u)) \ge 0$ and then, $w(e)_1 + r_1(v) - r_1(u) \ge 0$ since $M_2 \ge 0$: thus r is a legal retiming in the first dimension. Now, there are two cases: either $w(e)_1 + r_1(v) - r_1(u) = 0$ and, in this case, the constraints lead to $w(e)_d + r_d(v) - r_d(u) = 0$ for any dimension $d \ge 2$ and the dependence is loop independent after the retiming r, or $w(e)_1 + r_1(v) - r_1(u) > 0$ and the dependence is carried by the first loop while $w(e)_d + r_d(v) - r_d(u)$ can take (more or less) any value in the remaining dimensions d. Therefore, r is a legal retiming that leads to a code with a first (in general) sequential loop and all other parallel.

Conversely, assume that there exists a retiming r such that the first loop is sequential and the other are parallel after retiming. The retiming r is legal, it produces only lexico-positive weights: in particular, the first component of the retimed weight of any edge $e = (u, v)$ is nonnegative: $w(e)_1 + r_1(v) - r_1(u) \ge 0$. Furthermore, $\tilde{G}_r$ has only edges of zero weight, but if r is any solution, it may be possible that the retimed weights are too large for the linear constraints to be satisfied. We are going to change r.

For each connected component s of $\tilde{G}_r$ and for each dimension $d \ge 2$, pick a vertex $u_{s,d}$ that has the minimal value r_d in s. Then, define for each $v \in s$, $R_d(v) = r_d(v) - r_d(u_{s,d})$ and let $R_1(v) = r_1(v)$. Then, R is also a solution: we did not change the weight of edges in the first dimension, and we did not change the weight of edges in $\tilde{G}_r$ for the other dimensions. Furthermore, for all vertices $v \in V$ and all dimensions $d \ge 2$, we have $R_d(v) \ge 0$ and $R_d(v)$ is the weight (in dimension d) of an undirected elementary path P_v from $u_{s,d}$ to v if $v \in s$. Thus, $0 \le R_d(v) \le \sum_{e \in P_v} |w(e)_d|$. Now, if $w(e)_1 + R_1(v) - R_1(u) = 0$ (i.e., if $e \in \tilde{G}_R$), then for any dimension $d \ge 2$, $w(e)_d + R_d(v) - R_d(u) = 0$ since R is a solution: the constraints are satisfied. Otherwise, $w(e)_1 + R_1(v) - R_1(u) > 0$ (i.e.,

$e \notin \tilde{G}_R$), we have $w(e)_d + R_d(v) - R_d(u) \leq w(e)_d + \sum_{e \in P_v} |w(e)_d| \leq M_d$ (since $e \notin \tilde{G}_R$ implies that $e \notin P_v$): the constraints are also satisfied. □

We are thus able to look for a solution leading to one sequential loop surrounding $(n-1)$ parallel loops, with a relatively small problem, $n|V|$ variables (the retiming) and $2n|E|$ constraints. All other similar problems we mentioned before can be solved using the same approach, possibly with additional linear constraints.

6 Polynomially-Solvable Cases and Heuristics

Because of the exponential complexity of the exact approach proposed in Section 5.3, we now mention some particular cases that can be solved in polynomial time (they can be viewed as heuristics for the general problem). To make the discussion simpler, we restrict to the case of two loops, for which we want to find a shift that leads to a sequential loop surrounding a parallel loop. The first cases are when retiming in only one dimension is sufficient, either along the outer loop, or along the inner loop. These two cases correspond to situations in which we can find a parallel loop by looking only to one of the two parts of the problem (see Proposition 5). Then, we give a multidimensional heuristic that tries to combine both ideas. Finally, we also mention a sufficient condition that guarantees that the problem has no solution.

6.1 Inner Parallel Loop by Internal Retiming Alone

This is the simplest case, when all the dependences in $\tilde{G}$ can be transformed into zero-weight edges, in other words when what we called internal shifting is sufficient. This case has been already studied in Section 4: it can be solved by the perfect alignment algorithm. Figure 17 illustrates this situation ($\tilde{G}$ is the graph with solid edges). Note that this example cannot be solved by external shifting alone (see next section). Another example where internal shifting alone is sufficient is the example of Peir and Cytron (see Figures 1 and 3).

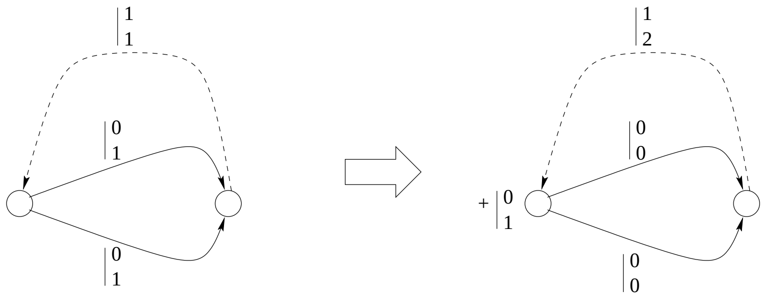


Figure 17: Parallelization by internal retiming.

6.2 Inner Parallel Loop by External Retiming Alone

If all the dependences that have a nonzero weight in the inner dimension can be carried by the outer loop, then the inner loop will be parallel. This case can be identified as follows. We want to find a retiming r_1 in the first dimension such that, for each edge $e = (u, v)$ that has a nonzero component in the second dimension, $w(e)_1 + r_1(v) - r_1(u) \geq 1$, and for all other edges $w(e)_1 + r_1(v) - r_1(u) \geq 0$. In other words, we want to find a legal retiming of $G' = (V, E, w')$ where $w'(e) = w(e)_1 - 1$ or

$w'(e) = w(e)_1$ depending on the edge we consider. By Proposition 2, such a retiming exists if and only if G' has no circuit of negative weight, and, as explained in Section 3.2, such a retiming can be found in polynomial time, when it exists. Figure 18 illustrates this situation. Note that this example cannot be solved by internal shifting alone (see previous section).

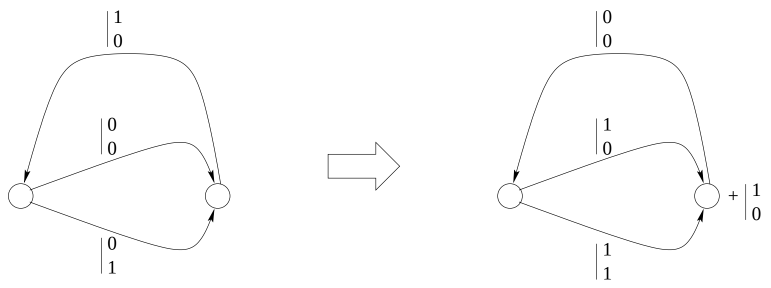


Figure 18: Parallelization by external retiming.

External shifting alone fails when there is too few edges that can be carried by the first dimension to "eliminate" all the dependences that have a nonzero weight in the inner dimension. This is the case for the example of Peir and Cytron (see Figure 1). Formulated differently, we answer here the following question: Is there a retiming r in the first dimension of G such that $\tilde{G}_r$ has only zero-weight edges? We have thus classified the following list of problems: deciding if a retiming r in the first dimension exists such that 1) $\tilde{G}_r$ has no edge, 2) $\tilde{G}_r$ has only zero-weight edges, 3) $\tilde{G}_r$ is a set of chains, 4) $\tilde{G}_r$ has no cycle, 5) $\tilde{G}_r$ has only zero-weight cycles. Problems 1 and 2 are polynomially solvable, and Problems 3, 4, and 5 are strongly NP-complete.

6.3 A Multidimensional Heuristic

It is possible to combine external shifting and internal shifting into a simple heuristic as follows. We know what graphs can be transformed into graphs with zero-weight edges, these are the graphs with only zero-weight cycles. So, the idea is to build two sets of edges $E = E_1 \cup E_2$, with $E_1 \cap E_2 = \emptyset$, such that $G_1 = (V, E_1)$ has no cycle of nonzero weight. To build E_1, we start from a spanning tree T of the dependence graph (thus with no cycle) and we add to T all edges, considered in any order, that do not create a nonzero-weight cycle. It remains to find, if it exists, a legal retiming r in the first dimension such that all edges in E_2 are carried by the first loop. This can be done by a similar procedure as in the previous section. Figure 19 illustrates this technique. For the first spanning tree, there is a solution but not for the second one. Of course, picking the right spanning tree is the hard problem. Note however that, for this example, retiming in both dimensions is necessary: external shifting alone and internal shifting alone are not sufficient.

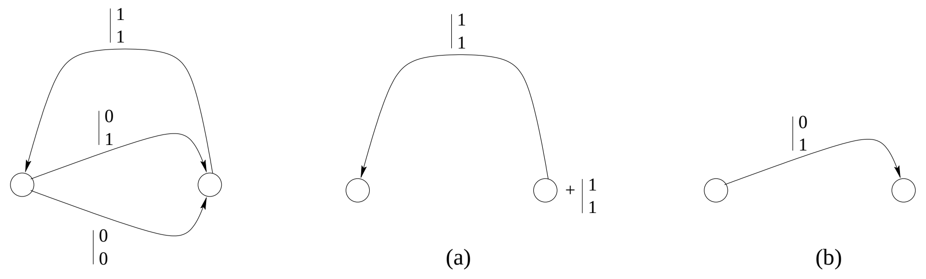


Figure 19: Two different spanning trees: a) leads to a solution b) cannot lead to a solution.

6.4 Non-Existence of a Retiming

We end this section by a particular case where we can be sure that no 2-dimensional retiming exists such that the inner loop can be parallelized by shifting. This is the case where G has a circuit c such that $0 <_{lex} w(c) \leq_{lex} (0, +\infty)$. Indeed, whatever the legal retiming r in the first dimension, all edges of c belong to $\tilde{G}_r$. Thus, $\tilde{G}_r$ has a circuit. Furthermore, since $w(c) \neq 0$, $\tilde{G}_r$ has a nonzero-weight circuit: there is no way to shift in the second dimension so that all edges have a zero-weight. Note that this condition is sufficient but, of course, not necessary.

Such a case can be detected by first choosing a legal retiming r in the first dimension, i.e., such that all edges have a nonnegative weight in the first dimension. Then, if there is a circuit with zero-weight in the first dimension, all its edges have a zero weight and they belong to $\tilde{G}_r$. Therefore, we just have to check if $\tilde{G}_r$ has a circuit of zero-weight. Figure 20 gives an example that cannot be parallelized by a 2-dimensional retiming.

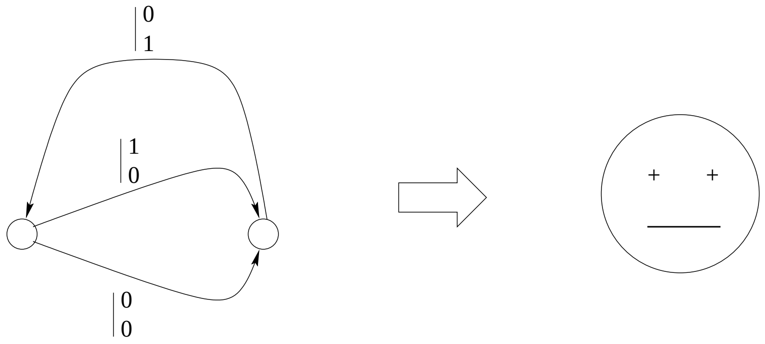


Figure 20: A nonparallelizable example.

7 Conclusions and Extensions

7.1 Summary of Results

The initial goal of this work was to be able to apply simple parallelization techniques such as loop fusion/distribution and loop shifting, when they are sufficient to reveal parallel loops. Loop shifting has indeed a similar property as loop fusion/distribution: it does not change the iteration domains of loops, it just shifts the domain of each statement with respect to each other.

Shifting in one dimension so that all loops can be fused into a single parallel loop is well-known: this is more or less the loop alignment problem. We recalled it here because this is a particular case of our study and because we wanted to make the reader more familiar with the retiming techniques we use. The main result of this paper concerns shifting in several dimensions. We showed that, unfortunately, the simplicity of the problem in one dimension disappears in higher dimensions: determining if a shift exists for two nested loops such that the innermost loop is parallel is a strongly NP-complete problem. (We also show in appendix the NP-completeness of similar shifting problems related to optimization of locality).

This result is, at first sight, very surprising: indeed, all algorithms to detect parallel loops ([19, 1, 33, 12, 13, 11] to quote but a few) are polynomial (based on rational linear programming) and most of them do capture loop shifting. But, first they are not able to enforce a given linear part (i.e., control the complexity of the solution), second they are not able to detect codes whose parallel loops contain loop independent dependences (they try to carry all dependences). Even if a simple shift can be sufficient, they may give a solution where a loop skewing is needed (this is for example the case for the code of Peir and Cytron, see Section 2). Thus, there is no contradiction, the fact that they look for any solution in a larger set of transformations makes that the problem is no longer NP-complete. Finding some affine transformation is less complex than checking if a solution with a given linear part exists. However, our NP-complete result is in contradiction with the polynomial algorithm proposed in [20], which in incorrect.

Despite the complexity of the problem, we showed that, for not too large problems, an exact solution can be found as an integral solution of a linear (in the number of dependences) number of integral linear constraints. We also showed that many subcases can be solved with similar variants of the Bellman-Ford algorithm.

In most results we presented here, we assumed a very simple case, a set of loops (not necessarily perfectly nested) that can be described by a graph with uniform dependences, and we only considered shifting transformations (combined with loop fusion). We now give some possible extensions that we plan to add to this simplified model. Some of these extensions are inspired by other works on retiming and loop parallelization. We first present the interest of code replication for improving loop parallelization. Then, we illustrate how our study can be used for more general transformations, in particular to check if a linear transformation can be completed by an adequate shifting. Finally, we explain how our results can be adapted to dependences represented by direction vectors. All these extensions are just briefly mentioned here and are left for future work.

7.2 Improving Parallelism Through Code Replication

As presented in [24] by Okuda, a well-chosen replication of statements can change the structure of the dependence graph so that parallelization by retiming becomes possible. The idea is that some statements compute values that are used by several other statements in the loop. These statements may prevent the parallelization if they belong to a cycle of nonzero weight. Replicating them allow to break the cycle because each dependent statement now depends on its own copy.

This idea is applied on the dependence graph by duplicating vertices that correspond to replicated statements. Although original, the technique presented by Okuda makes a use of replication that is neither necessary nor sufficient. Actually, Okuda's method replicates all the statements that are "extreme", i.e., vertices of the graph that are connected to a unique other vertex. What is really necessary is to break all cycles of nonzero weight. If the graph contains a circuit, it is in general impossible to make such a transformation, but if the graph is acyclic, the procedure becomes feasible. In this case, a simple algorithm consists in starting from a spanning tree and adding the remaining edges one after the other. When adding the edges, two cases can occur:

- the edge belongs to a zero-weight cycle: it can be left unchanged.
- the edge belongs to a nonzero-weight cycle: replicating all its ancestors breaks the cycle.

The problem is that we cannot predict the amount of replication produced by this technique (although it is obviously bounded by $|V||E|$), and the choice of the spanning tree is crucial to make as few replication as possible.

7.3 Combining Shifting with Linear Transformations

As we mentioned before, our shifting technique can be easily combined with a unimodular transformation of the loop. Such a combination was proposed in [10] but with an "unpredictable" linear part found by linear programming. Our idea is to apply a fixed transformation before parallelizing the loops by shifting. Indeed, in practice, very few linear parts have to be checked, mainly the identity matrix, the matrices corresponding to interchange, and skew with small factors. This technique has the advantage to give a control on the chosen transformation, consequently we are free to limit our linear transformations to simple transformations.

As Figure 21 illustrates, the first graph cannot be retimed to parallelize the inner loop [5]. But, if we use a loop interchange before searching for a retiming, the graph becomes parallelizable both by internal or external retiming alone.

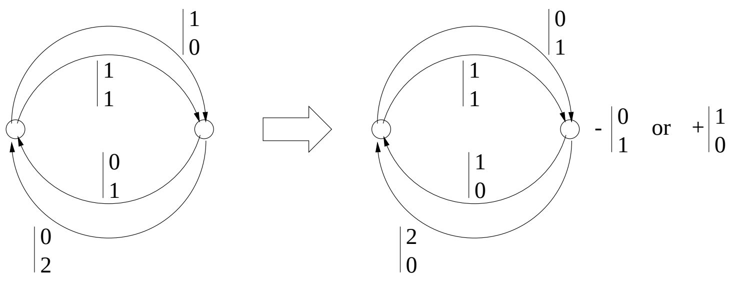


Figure 21: A nonparallelizable example that is parallelizable after loop interchange.

7.4 Extension to Nonuniform Dependences

In the case of non uniform dependences, we may be able to handle some situations. We assume here that dependences are approximated by direction vectors (see [34] for a detailed presentation of direction vectors). Direction vectors are vectors whose components may have the single value $*$, or any integral value possibly associated with the symbol $+$ or $-$. The meaning of these values is the following:

- n means the integral value n.
- $n+$ means n or more (and $+$ alone means $1+$).
- $n-$ means n or less (and $-$ alone means $(-1)-$).
- $*$ means any integral value.

Our retiming technique can only change the integral parts of these vectors. It results that the values that can become zero after retiming are those with an integral part only (no $+$, $-$, or $*$), and the values that can become positive after retiming are the integral values possibly associated with a $+$ (no $-$ or $*$). We will assume as before that all circuits have a lexico-positive weight (defined using the standard sum for direction vectors, for example, $+$ plus $-$ is $*$, $2+$ minus 3 is $(-1)+$, etc.).

As noticed in Section 3.2 (see in particular Proposition 3), we may not be able to obtain fully fused loops due to negative values (such as $-$) that cannot be transformed into nonnegative values by (bounded) retiming: for those, loop distribution will be needed. Another possibility is that these dependences are carried by an outer loop. This can be enforce, when possible, in a way similar to the case of external shifting alone: we just have to add stronger constraints for the retiming in the outer dimension (for example in the 2d case, we can impose that the retimed weight in the outer dimension is greater than 1 if the second dimension is $-$ or $*$). Figure 22 gives two examples, the first one can be parallelized by retiming, the second one cannot.

[5]Note that this example shows that the condition given in Section 6.4 is sufficient but not necessary, as we mentioned.

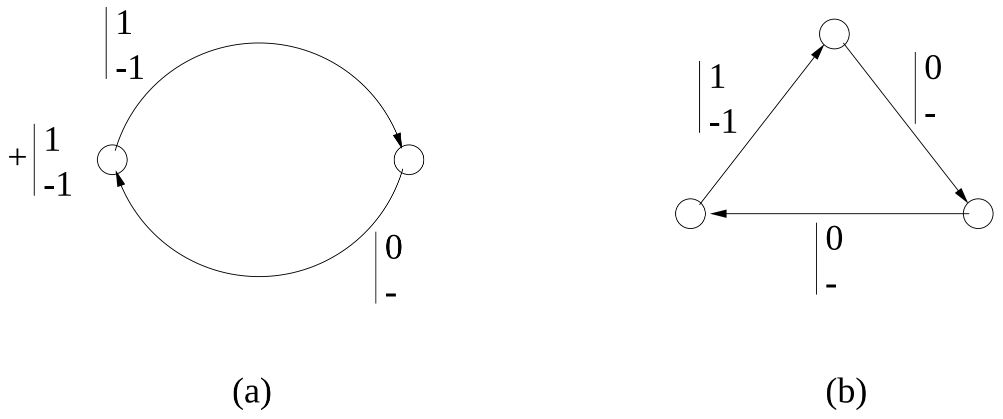


Figure 22: a) Total fusion with inner parallel loop is a) possible b) not possible.

References

[1] John Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, Munich, Germany, January 1987.

[2] Pierre Boulet, Alain Darte, Georges-André Silber, and Frédéric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Journal of Parallel Computing*, 24(3), 1998. Special issue on Languages and Compilers for Parallel Computers.

[3] P.-Y. Calland, A. Darte, and Y. Robert. Circuit retiming applied to decomposed software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, January 1998.

[4] Zbigniew Chamski. *Environnement logiciel de programmation d'un accélérateur de calcul parallèle.* PhD thesis, Université de Rennes, Rennes, France, 1993. numéro 957.

[5] J.C. Chung. Optimal loop parallelization based on a retiming technique. In *International Conference on Parallel Processing*, 1992.

[6] Jean-François Collard, Paul Feautrier, and Tanguy Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3):421–436, September 1995.

[7] Alain Darte. On the complexity of loop fusion. In *International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, Newport Beach, CA, October 1999.

[8] Alain Darte and Guillaume Huard. Loop shifting for loop compaction. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *The Twelfth International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science. Springer-Verlag, San Diego, CA, August 1999.

[9] Alain Darte and Yves Robert. A graph-theoretic approach to the alignment problem. Technical Report 93-20, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, July 1993.

[10] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[11] Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism detection in polyhedral reduced dependence graphs. *International Journal of Parallel Programming*, 25(6):447–497, December 1997.

[12] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part I: One-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

[13] Paul Feautrier. Some efficient solutions to the affine scheduling problem, Part II: Multi-dimensional time. *International Journal of Parallel Programming*, 21(6):389–420, December 1992.

[14] A. Fraboulet, G. Huard, and A. Mignotte. Loop alignment for memory access optimization. In *Twelfth International Symposium on System Synthesis*, 1999.

[15] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

[16] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *The 5th Symposium on Frontiers of Massively Parallel Computation*, pages 332–341, McLean, Virginia, February 1995.

[17] Ken Kennedy and Kathryn S. McKinley. Loop distribution with arbitrary control flow. In *Supercomputing'90*, August 1990.

[18] Charles H. Koelbel, David B. Loveman, Robert S. Schreiber, Guy L. Steele Jr., and Mary E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[19] Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[20] C. Lang, N. Passos, and E. Sha. Polynomial-time nested loop fusion with full parallelism. In *International Conference on Parallel Processing*, volume 3, pages 9–16, Bloomingdale, IL, 1996.

[21] C. E. Leiserson and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6(1):5–35, 1991.

[22] Naraig Manjikian and Tarek S. Abdelrahman. Loop fusion for parallelism and locality. *IEEE Trans. Parallel Distributed Systems*, 8(2):193–209, February 1997.

[23] Kathryn S. McKinley and Ken Kennedy. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *The Sixth Annual Languages and Compiler for Parallelism Workshop*, volume 768 of *Lecture Notes in Computer Science*, pages 301–320. Springer-Verlag, 1993.

[24] Kunio Okuda. Cycle shrinking by dependence reduction. In *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 398–401. Springer-Verlag, 1996.

[25] OpenMP Standard for Shared-memory parallel directives. World Wide Web document, `http://www.openmp.org`.

[26] N. Passos and E. Sha. Achieving full parallelism using multi-dimensional retiming. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1150–1163, November 1996.

[27] J. K. Peir. *Program Partitioning and Synchronization on Multiprocessor Systems.* PhD thesis, University of Illinois at Urbana-Champaign, March 1986. Report UIUC-DCS-R-86-1259.

[28] J. K. Peir and R. Cytron. Minimum distance: A method for partitioning recurrences for multiprocessors. *IEEE Transactions on Computers*, 38(8):1203–1211, August 1989.

[29] N. Shenoy. Retiming: Theory and practice. *INTEGRATION, the VLSI journal*, 22:1–21, 1997.

[30] Georges-André Silber and Alain Darte. The Nestor library: A tool for implementing Fortran source to source transformations. In *High Performance Computing and Networking (HPCN'99)*, volume 1593 of *Lecture Notes in Computer Science*, pages 653–662. Springer-Verlag, April 1999.

[31] Stanford Compiler Group. The SUIF Compiler System. World Wide Web document, `http://suif.stanford.edu/suif/suif.html`.

[32] Tera Computer Company. TERA's multithreaded architecture. Electronic document `http://www.tera.com`.

[33] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.

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

Appendix: Maximizing the Number of Local Accesses

In this section, we present another retiming NP-completeness result that concerns the maximization of locality by retiming. This problem does not concern the main topic of the paper – loop parallelization – this is why we address it here only in the appendix, but it uses the same retiming techniques. We thus found interesting to mention it to make a summary of known results on the complexity of retiming.

The problem is, assuming that it is not possible to find a retiming so that all edges have a zero weight (which is our "perfect alignment problem" addressed in Section 4.1), to find a legal retiming such that as many edges as possible have a zero weight, in other words to "align" the statements as much as possible. It has some similarities with the data alignment problem considered in [9] and with the locality optimization problems studied in [23]. The problem in [9] is the same, except that the retiming does not have to be legal, and it has been shown (weakly) NP-complete for a directed graph with circuits. The problem in [23] concerns loop fusion to maximize aligned accesses: it is mainly an optimization on a directed graph with no circuits (DAG), obtained from a sequence of loops.

We deal here with the problem of loop shifting to maximize aligned accesses. We show that the problem is strongly NP-complete [6], even for graphs with no circuits (which is often the case when we try to optimize a sequence of loops). Our proof for graphs with no circuits is more complicated than for general graphs. We therefore give both proofs, first a more tricky proof for graphs with no circuits, then an easy proof for general graphs.

NP-Completeness for a Dependence Graph With No Circuit

In this section, we address the problem of maximizing, by retiming, the number of edges of zero weight of an acyclic graph (acyclic here means with no circuit, but cycles may exist). We call this problem ACYCLIC MAXIMAL LOCALITY because an edge with a zero weight is a dependence local to the loop body (the producer and the consumer belong to the same loop, at the same iteration). More formally, the associated decision problem is stated as follows.

Problem: ACYCLIC MAXIMAL LOCALITY

Instance An acyclic directed graph $G = (V, E, w)$, where V is the set of vertices of G, $E \in V^2$ its set of edges, $w : E \longrightarrow \mathbb{Z}$ a weight function over the edges and a positive integer value K.

Question Does there exist a retiming $r : V \longrightarrow \mathbb{Z}$ such that G_r has at least K edges of zero weight?

We prove that this problem is strongly NP-complete. The proof is by a polynomial reduction from the problem ONE-IN-THREE 3SAT. We recall the definition of ONE-IN-THREE 3SAT (problem LO4 in [15, p. 259]).

Problem: ONE-IN-THREE 3SAT

Instance A set U of n boolean variables and C of m clauses over U (for each variable u of U we denote by u and $\bar{u}$ literals over U, a clause over U is a set of literals over U), such that for each clause $c \in C$, $|c| = 3$.

[6]Note: the dual problem of minimizing the number of zero-weight edges is simpler, it can be solved in polynomial time as a particular cost-flow optimization, see [8].

Question Does there exist a truth assignment for U (i.e., a function $t : U \longrightarrow \{T, F\}$, with $t(\bar{u}) = T$ if $t(u) = F$ and $t(\bar{u}) = F$ if $t(u) = T$) such that every clause in C has exactly one true literal (i.e., exactly one $u \in c$ such that $t(u) = T$)?

Before the proof, we need to slightly modify Lemma 2, because we do not want to require, for the general proof, the retiming to be legal (but, as we will see, if the retiming is required to be legal, the NP-complete result still holds).

Lemma 3 *For any retiming r of a graph $G = (V, E, w)$ of dimension 1, there exists a retiming r' of G such that $\forall v \in V$, $0 \leq r'(v) \leq \sum_{e \in E} |w(e)|$, and $G_{r'}$ has at least the same edges of zero weight as G_r.*

Proof: We build a graph $G' = (V, E', w)$ from G and r, by removing from G all edges that have a nonzero weight in G_r. By construction, all the edges of G' have a zero weight after the retiming r, so G' can obviously have only edges of zero weight by retiming. Therefore, applying the algorithm of Section 4.1, we can find a retiming r' such that all edges of G' have zero weight. By construction of G', if we apply this retiming r' to G, all edges with zero weight in G_r have a zero weight in $G_{r'}$. Furthermore, as in the proof of Proposition 6, we can add a suitable constant to r' so that it is nonnegative and defined as a shortest path in G', and we have $\forall v \in V$, $0 \leq r'(v) \leq \sum_{e \in E} |w(e)|$. $\square$

Theorem 4 Acyclic Maximal Locality *is strongly NP-complete.*

Proof:

The problem is in NP Consider a positive instance of our problem given by a graph $G = (V, E, w)$ and a positive integral value K: there exists some retiming r such that G_r has at least K edges of zero weight. Because of Lemma 3, we know that there exists another retiming r' such that $G_{r'}$ has at least K edges of zero weight, and whose values are bounded by the sum of the absolute values of the weights of G. We can check in polynomial time that $G_{r'}$ has more than K zero-weight edges. Thus, Acyclic Maximal Locality belongs to NP.

Transformation Given an instance (U, C) of One-in-Three 3SAT, that is a set U of n variables and a set C of m clauses over literals of U, we transform it to $f(U, C) = (G, K)$, an instance of Acyclic Maximal Locality, in the following way.

- We start from a graph $G = (V, E, w)$ with two vertices a and b ($V = \{a, b\}$), and E is a set of $48mn + 8m$ edges from a to b, all with a zero weight. We call this part of the transformation the base structure of G (see Figure 23).

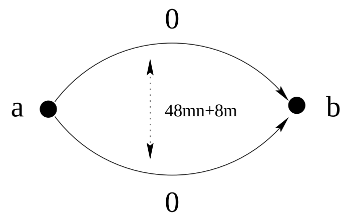


Figure 23: Starting point of the transformation (base structure).

- For each variable $u \in U$, we add two vertices to V, u and $\bar{u}$, and we add to E, $24m$ edges of weight 1 from $\bar{u}$ to b, $16m$ edges of weight 1 from u to $\bar{u}$, $24m$ edges of weight 0 from a to u, $16m$ edges of weight 0 from a to $\bar{u}$, and $16m$ edges of weight 1 from u to b (see Figure 24).

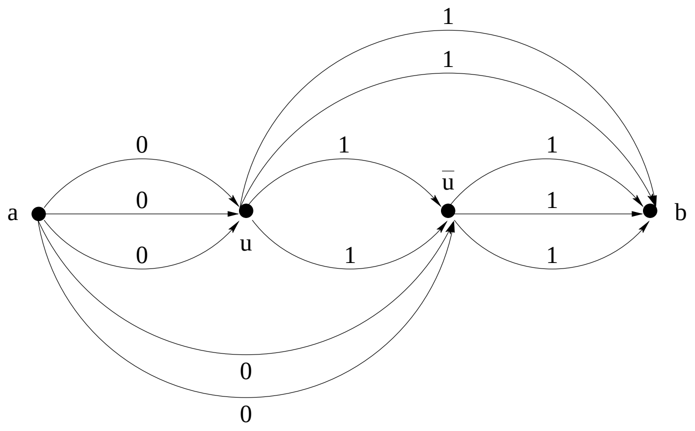


Figure 24: Transformation of a variable (each depicted edge is repeated $8m$ times).

- For each clause $c = \{x, y, z\} \in C$, we consider what we call the derivated clauses $c_0 = \{x, \bar{y}, \bar{z}\}$, $c_1 = \{\bar{x}, y, \bar{z}\}$, $c_2 = \{\bar{x}, \bar{y}, z\}$, and we add three vertices c_0, c_1, and c_2 to V. Then, for each $c_i = \{x', y', z'\}$, $i \in \{0, 1, 2\}$, we add one edge of weight 0 from a to c_i, one edge of weight 0 from c_i to each of x', y', and z', and one edge of weight 1 from each of x', y', and z' to b (see Figure 25).

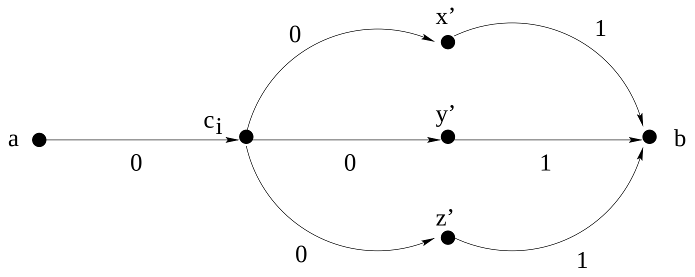


Figure 25: Elementary part of the clause transformation (for one of the derivated clauses).

- We set $K = 96mn + 22m$.

The graph G generated this way has obviously no circuit, and the transformation is polynomial, since G has $2n + 3m + 2$ vertices and $144mn + 29m$ edges, and K is polynomial in n and m.

Reduction Let (U, C) be a positive instance of One-in-Three 3SAT, and $f(U, C) = (G, K)$. We prove that:

- if (U, C) is a positive instance of One-in-Three 3SAT, then (G, K) is a positive instance of Acyclic Maximal Locality. Let t be a truth assignment for U satisfying all the clauses of C with exactly one true literal. We define r, a retiming of G, by:

$$r(a) = r(b) = 0$$

$$\forall u \in U,$$
$$r(u) = \begin{cases} 1 & \text{if } t(u) = T \\ 0 & \text{otherwise} \end{cases} \quad r(\bar{u}) = \begin{cases} 1 & \text{if } t(u) = F \\ 0 & \text{otherwise} \end{cases}$$

$$\forall c \in C, \forall c_i = \{x', y', z'\}, \, i \in \{0, 1, 2\},$$
$$r(c_i) = \begin{cases} 1 & \text{if } t(x') = t(y') = t(z') = T \\ 0 & \text{otherwise} \end{cases}$$

Since $r(a) = r(b)$, the retiming is legal for the base construction, and the $48mn+8m$ associated edges remain of zero weight.

Depending on t, for each variable $u \in U$, we have either $r(u) = 1$ and $r(\bar{u}) = 0$, or $r(u) = 0$ and $r(\bar{u}) = 1$. In both cases, this generates exactly six groups of $8m$ edges of zero weight in each construction associated with a variable (i.e., six edges on Figure 24) so a total of $48mn$ edges of zero weight (and all edges have a nonnegative weight, the retiming is legal for them too).

For each clause $c \in C$, depending on t, there is exactly one true literal, thus exactly one clause $c_i = \{x', y', z'\}$ of the derivated clauses $\{c_0, c_1, c_2\}$ has all its literals true under t. For this derivated clause, we have $r(x') = r(y') = r(z') = 1$ since they are all true, and $r(c_i) = 1$, thus the retiming is legal for the construction associated with c_i, and the construction has six edges of zero weight. Any other clause c_j in $\{c_0, c_1, c_2\} - \{c_i\}$ has, by definition, $r(c_j) = 0$ since at least one of its literals is false for t. Thus, whatever the retiming value of its literals, the retiming is in all cases legal, and the construction associated with c_j has four edges of zero weight. Finally, there are $6 + 4 + 4 = 14$ edges of zero weight for each clause, so a total of $14m$ edges for the clauses.

To summarize, we get a total of $96mn + 22m$ edges of zero weight for the entire graph: (G, K) is a positive instance of Acyclic Maximal Locality. Note also that (G, K) is also a positive instance even if we require the retiming to be legal.

- if (G, K) is a positive instance of Acyclic Maximal Locality, then (U, C) is a positive instance of One-in-Three 3SAT. Let r be a retiming of G such that G_r has at least K edges of zero weight. Without loss of generality we can assume that $r(a) = 0$ (otherwise, we can add a constant c to all retiming values so that $r(a) + c = 0$). We define t a truth assignment for U by:

$$\forall u \in U, \, t(u) = \begin{cases} F & \text{if } r(u) = 0 \\ T & \text{otherwise} \end{cases} \quad \text{and } t(\bar{u}) = \begin{cases} F & \text{if } r(\bar{u}) = 0 \\ T & \text{otherwise} \end{cases}$$

Before going further, let us prove some properties of any retiming satisfying our particular instance. First, we prove that $r(a) = r(b) = 0$, then for any variable $u \in U$, we discuss about the number of edges of zero weight that can be found in the construction they generate, and we prove that $r(u)$ and $r(\bar{u})$ are either 0 or 1, and that $r(u) \neq r(\bar{u})$.

We can notice that each structure constructed from a variable generates $12 * 8m = 96m$ edges and each structure constructed from a clause generates $3 * 7 = 21$ edges, so a total of $96mn + 21m$ edges. Consequently, whatever the retiming r we choose, if G_r has $K =$

$96mn + 22m$ or more edges of zero weight, then some edges of its base structure have a zero weight. Since all edges of the base structure have the same weight (0), and the same head (b) and tail (a), all of them are of zero weight as soon as one has a zero weight: thus, $r(a) = r(b) = 0$, and there are $48mn + 8m$ edges of zero weight in the base structure.

Consider the structure built from a variable $u \in U$. Assume first that $r(u) < 0$ or $r(u) > 1$, then in both cases, the edges from a to u and the edges from u to b have a nonzero weight after retiming. Furthermore, the edges from a to $\bar{u}$ and from u to $\bar{u}$ cannot have a zero weight simultaneously (otherwise, we would have $r(\bar{u}) = 0$ and $r(u) = 1$). Thus, at most $40m$ edges have a zero weight after retiming, the edges from a to $\bar{u}$, or the edges from u to $\bar{u}$, plus possibly the edges from $\bar{u}$ to b. Now, assume that $r(\bar{u}) < 0$ or $r(\bar{u}) > 1$, then in both cases, the edges from a to $\bar{u}$ and the edges from $\bar{u}$ to b have a nonzero weight after retiming. Furthermore, whatever the value we choose for $r(u)$, either the edges from u to $\bar{u}$, or the edges from u to b have a nonzero weight, because they have the same initial weight, the same tail (u), and different retiming values for their heads. Thus, again, at most $40m$ edges have a zero weight after retiming. Moreover, we can easily check that if $r(u) = r(\bar{u}) = 0$ or $r(u) = r(\bar{u}) = 1$, then exactly $40m$ edges are of zero weight after retiming. But, if $r(u) = 0$ and $r(\bar{u}) = 1$, or $r(u) = 1$ and $r(\bar{u}) = 0$ then exactly $48m$ edges have of zero weight after retiming.

Finally, assume that at least one of the structures generated by the variables has $40m$ zero-weight edges or less after retiming, then the total number of zero-weight edges generated by the base structure and the structures built from the variables is at most $48mn + 8m + 48m(n - 1) + 40m = 96mn$: by the choice of K, we still need $22m$ zero-weight edges in the structures of the clauses but, the total number of edges in these structures is only $21m$. Thus, because r is a solution of our instance, all the structures generated by the variables have exactly $48m$ zero-weight edges after retiming and for each variable $u \in U$, either $r(u) = 0$ and $r(\bar{u}) = 1$, or $r(u) = 1$ and $r(\bar{u}) = 0$.

It follows that for any variable $u \in U$, $t(u) \neq t(\bar{u})$ (by definition of t) and so t is a truth assignment. Finally, it remains to prove that t satisfies all the clauses of C, with a single true literal.

Consider a derivated clause $c_i = \{x', y', z'\}$ from some clause $c \in C$: it generates a structure in which $r(x')$, $r(y')$, and $r(z')$ are either 0 or 1 (because these are vertices from a structure generated by a variable). Assume that $r(c_i) < 0$ or $r(c_i) > 1$, then the edge from a to c_i and the edges from c_i to any of x', y', or z' have a nonzero weight after retiming. So, in this case, at most 3 edges have a zero weight after retiming. Furthermore, if $r(c_i) = 0$, then the structure has exactly 4 zero-weight edges after retiming and, if $r(c_i) = 1$, then either $r(x') = r(y') = r(z') = 1$ and the structure has exactly 6 zero-weight edges after retiming, or it is easy to check that the structure has at most 4 zero-weight edges after retiming. In general, it may be possible that r is not a legal retiming and that only 3 edges have a zero weight, but we can slightly change $r(c_i)$ so that r is legal and the number of zero-weight edges does not decrease. For that, we let $r(c_i) = 1$ if $r(x') = r(y') = r(z') = 1$, otherwise $r(c_i) = 0$. Thus, if there is a retiming for our graph with at least K edges, there is a legal retiming with at least as many edges and such that all derivated clauses have either 4 or 6 zero-weight edges.

Finally, because $\forall u \in U$, $r(u) \neq r(\bar{u})$, and by definition of the derivated clauses c_0, c_1, c_2 of a given clause c, there is at most one generated structure with 6 zero-weight edges after retiming, and the two other have 4 zero-weight edges after retiming. Now, assume that for some clause $c \in C$, the three generated structures have 4 zero-weight edges after retiming. There are thus at most $(6+4+4)*(m-1)+(4+4+4) = 14m-2$ zero-weight edges generated by the clauses. Added to the $48mn + 8m + 48mn$ zero-weight edges generated by the base structure and the structures of the variables, we get a total of $96mn + 22m - 2$ edges and the retiming is not a solution of our instance. This proves that, for each clause $c \in C$, at least one of the generated structures has 6 zero-weight edges after retiming, which means that all the literals of the associated derivated clause are true, and by definition of the derivated clauses, that exactly one literal of c is true. Thus t is a truth assignment for U that satisfies all clauses of C with exactly one true literal: (U, C) is a positive instance of One-in-Three 3SAT.

This completes the proof: Acyclic Maximal Locality is strongly NP-complete. □

Note that if we require, in the definition of the problem Acyclic Maximal Locality, that the retiming is legal, then the same proof is valid because we proved that the retiming solution of our instance is legal. Note also that if we forbid multi-edges (i.e. several edges between two vertices), the problem is still strongly NP-complete since we can add dummy vertices to break each edge of the proof in two.

NP-Completeness for a Dependence Graph With Circuits

In this section, we deal with the problem of maximizing the number of zero-weight edges of a cyclic (i.e., with circuits) dependence graph by retiming. Although the previous proof for a graph with no circuit is still valid simply by adding a back edge of weight 1 from b to a, we give here a simpler and more elegant proof. We can state the problem of zero-weight edge maximization by retiming as follows:

Problem: Cyclic Maximal Locality

Instance A directed graph $G = (V, E, w)$ and a positive integer K, where V is the set of vertices of G, $E \in V^2$ its set of edges, $w : E \longrightarrow \mathbb{Z}$ a weight function over the edges.

Question Does there exist a retiming $r : V \longrightarrow \mathbb{Z}$ such that G_r has at least K edges of zero weight?

We prove the NP-completeness of Cyclic Maximal Locality by a polynomial reduction from the problem Not-All-Equal 3SAT (problem LO3 in [15, p. 259]).

Problem: Not-All-Equal 3SAT

Instance A set $U = \{u_1, \dots, u_n\}$ of boolean variables and $C = \{c_1, \dots, c_m\}$ of clauses over U (for each variable u of U we denote by u and $\bar{u}$ literals over U, a clause over U is a set of literals over U), such that for each clause $c \in C, |c| = 3$.

Question Does there exist a truth assignment for U (i.e., a function $t : U \longrightarrow \{T, F\}$, with $t(\bar{u}) = T$ if $t(u) = F$ and $t(\bar{u}) = F$ if $t(u) = T$) such that every clause in C has at least a true literal (i.e., $u \in c$ such that $t(u) = T$) and a false literal (i.e., $u \in c$ such that $t(u) = F$)?

Theorem 5 Cyclic Maximal Locality *is strongly NP-Complete.*

Proof:

The problem is in NP for the same reason as in the acyclic case. There is a certificate of polynomial size that can be checked in polynomial time.

Transformation Given an instance of Not-All-Equal 3SAT, i.e., given U and C, we define $f(U,C) = (G,K)$ a transformation into an instance of Cyclic Maximal Locality in the following way:

- We start from $G = (V,E)$ with $V = \emptyset$ and $E = \emptyset$.
- For each variable $u \in U$, we add two new vertices u and $\bar{u}$, and two edges with weight equal to 1 from u to $\bar{u}$, and from $\bar{u}$ to u (see Figure 26).

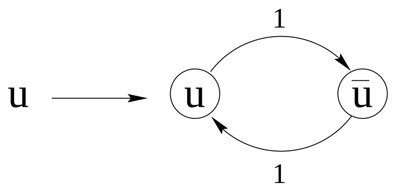


Figure 26: Transformation of a variable.

- For each clause $c = \{x,y,z\} \in C$, we add 6 edges with weight equal to 1 from x to y, from y to x, from x to z, from z to x, from y to z, and from z to y (see Figure 27).

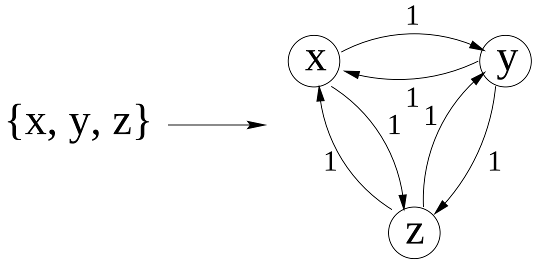


Figure 27: Transformation of a clause.

- We let $K = 2m + n$.

The transformation is obviously polynomial (two vertices and two edges generated by each literal, six edges generated by each clause, then a transformation in $O(m+n)$).

Reduction Given an instance (U,C) of Not-All-Equal 3SAT, with $(G,K) = f(U,C)$, we prove that:

- if (U,C) is a positive instance of Not-All-Equal 3SAT then $(G,K) = f(U,C)$ is a positive instance of Cyclic Maximal Locality. Let t be a truth assignment satisfying all the clauses of C with at least one false literal, we define for all variables $u \in U$,

$$r(u) = \begin{cases} 1 & \text{if } t(u) = T \\ 0 & \text{if } t(u) = F \end{cases} \qquad r(\bar{u}) = 1 - r(u)$$

  First, note that for each edge $e = (u,v) \in E$, $-1 \leq r(v) - r(u)$, then $w_r(e) = r(v) - r(u) + w(e) \geq 0$ since all edges of G have a weight equal to 1. So r is a legal retiming of G.

  For each variable $u \in U$, $r(u) - r(\bar{u}) = \pm 1$. So, either $w_r((u,\bar{u})) = 0$ and $w_r((\bar{u},u)) = 2$, or $w_r((u,\bar{u})) = 2$ and $w_r((\bar{u},u)) = 0$, then there are n zero-weight edges generated by the set of variables.

For each clause $c = \{x, y, z\} \in C$, at least one literal is true and at least one is false. Thus there exist two literals, for instance x and y, such that $r(x) = r(y)$ and $r(x) - r(z) = r(y) - r(z) = \pm 1$. Thus there is no zero-weight edge between x and y, and exactly one between both x and z, and y and z, in other words, two zero-weight edges for each clause, for a total of $2m$ for the set of clauses.

Finally, there are $2m + n = K$ zero-weight edges in G_r, and (G, K) is a positive instance of Cyclic Maximal Locality.

- if $f(U, C) = (G, K)$ is a positive instance of Cyclic Maximal Locality then (U, C) is a positive instance of Not-All-Equal 3SAT. Let r be a retiming such that G_r has at least K zero-weight edges, and define for all the literals $u \in U$,

$$t(u) = \begin{cases} T & \text{if } r(u) \bmod 2 = 1 \\ F & \text{if } r(u) \bmod 2 = 0 \end{cases}$$

  Note first that at most one of the two edges generated by a variable can have a zero weight after retiming since a retiming does not change the weight of a circuit.

  We now show that at most two edges generated by a clause can have a zero weight after retiming. First, the same observation as above limits this number to three, i.e., at most one zero-weight edge between two distinct literals. Now, assume that, for a clause $c = \{x, y, z\}$, there are at least two zero-weight edges after retiming, for instance, between x and y (thus $r(y) = r(x) \pm 1$) and between x and z (thus $r(z) = r(x) \pm 1$). Then either $r(y) = r(z)$ or $r(y) = r(z) \pm 2$, and there is no zero-weight edge between y and z. Thus, there are at most 2 zero-weight edges.

  To summarize, whatever the retiming, there are at most $2m + n$ zero-weight edges. Furthermore, if there are $K = 2m + n$ zero-weight edges, then there is exactly one zero-weight edge between a literal and its opposite, and two in each clause.

  It remains to show that t is a truth assignment and that it satisfies all the clauses with at most two true literals in each.

  - because there exists a zero-weight edge between u and $\bar{u}$, we have $r(u) = r(\bar{u}) \pm 1$. Thus $r(u) \bmod 2 \neq r(\bar{u}) \bmod 2$ and $t(u) \neq t(\bar{u})$.
  - each clause $c = \{x, y, z\}$ contains exactly two zero-weight edges, then at least one zero-weight edge, for example one between x and y, so $r(x) = r(y) \pm 1$ and $t(x) \neq t(y)$.

  Finally, (U, C) is a positive instance of Not-All-Equal 3SAT.

This completes the proof: Cyclic Maximal Locality is strongly NP-complete. □

Note again that the same proof shows that the same problem, with the additional constraint that the retiming is legal, is strongly NP-complete. Indeed, in the first part of the proof, if (U, C) is a positive instance of Not-All-Equal 3SAT, then the retiming we built is legal. In the second part of the proof, if there is a legal retiming for $f(U, C)$ with at least K zero-weight edges, then the same conclusion holds since we did not exploit the fact that the retiming is or is not legal.