Parallel Processing Letters

Combining retiming and scheduling techniques for loop parallelization and loop tiling*

ALAIN DARTE, GEORGES-ANDRÉ SILBER and FRÉDÉRIC VIVIEN
CNRS URA 1398, Ecole Normale Supérieure de Lyon, 69364 LYON Cedex 07
e-mail: [darte,gsilber,fvivien]@lip.ens-lyon.fr

Received
Revised
Communicated by (Name of Editor)

ABSTRACT

Tiling is a technique used for exploiting medium-grain parallelism in nested loops. It relies on a first step that detects sets of permutable nested loops. All algorithms developed so far consider the statements of the loop body as a single block, in other words, they are not able to take advantage of the structure of dependences between different statements. In this paper, we overcome this limitation by showing how the structure of the reduced dependence graph can be taken into account for detecting more permutable loops. Our method combines graph retiming techniques and graph scheduling techniques. It can be viewed as an extension of Wolf and Lam's algorithm to the case of loops with multiple statements. Loop independent dependences play a particular role in our study, and we show how the way we handle them can be useful for fine-grain loop parallelization as well.

Keywords: Automatic parallelization, nested loops, permutable loops, tiling, medium grain.

1 Introduction

Affine scheduling techniques - from the simplest and earliest one, Lamport's hyperplane method [11], to the most sophisticated one, Feautrier's multi-dimensional affine scheduling [6] - are used to transform a set of nested loops into a semantically equivalent code, consisting in parallel loops surrounded by sequential loops. Lamport's method, and its extension, the linear scheduling, transform n perfectly nested loops into $n-1$ nested parallel loops surrounded by a single sequential loop. When this is not feasible, multi-dimensional scheduling can be used to transform the original loops into $n-r$ sequential loops surrounding r innermost parallel loops, with $n-r>1$. The goal is to make r (roughly speaking the *degree of parallelism*) as large as possible.

*Supported by the CNRS-INRIA project *ReMaP*. G.-A. Silber is supported by a DRET fellowship.

The underlying computational model in which these techniques are developed is nothing but a PRAM. Additional constraints such as the cost of communications, the cost of synchronizations, the number of processors, the ratio communications/computations, are not taken into account. The claim (the hope) is that they can be optimized a posteriori, for example by merging virtual PRAM processors into fewer physical processors. However, especially when r is small, the granularity of computations can be too fine, leading to poor performances especially on distributed memory systems. To circumvent this problem, the granularity of computations has to be increased. This can be achieved by a technique called *tiling*, introduced by Irigoin and Triolet [7] as *supernode partitioning*.

Tiling consists in aggregating several loop iterations that will be considered as an elemental computation. The size and shape of a tile are chosen following various criteria, for achieving better vectorization of communications and/or computations, for improving cache-reuse, reducing communications, etc. All these criteria are very machine-dependent, and despite the large amount of different optimization strategies [7,10,13,14,16,2], choosing a "good" tiling remains an open problem.

However, before even defining the size and shape of the tiles, one has to make sure that they will be atomic, i.e. that they can be computed with no intervening synchronization or communication. This atomicity property is fulfilled if the dependence graph between tiles is acyclic which is guaranteed if the tiles partition the iteration domain into identical rectangles, and if the iteration domain is described by *permutable loops*.

Until now, all algorithms proposed for detecting permutable loops have the following restrictions:

- The original loops are perfectly nested.
- The dependences are uniform, except for Wolf and Lam's algorithm [17] where dependences can be approximated by direction vectors.
- The statements of the loop body are considered as a single block. This may enforce complicated skews, even if a simple shift between statements is sufficient to make the loops permutable.

Taking into account the structure of the reduced dependence graph has been proved very useful for the detection of parallel loops: see for example the algorithms of Allen and Kennedy [1], Darte and Vivien [5], or Feautrier [6]. In this paper, we show that it can also be useful for the detection of permutable loops. Our method combines graph retiming and graph scheduling techniques.

We do not overcome all restrictions listed above as we still consider only perfectly nested loops. However, our algorithm can be applied even if the dependences are described by a polyhedral approximation of distance vectors (which is more general than direction vectors), and we do exploit the fact that the loop body may have more than one statement, i.e. that the reduced dependence graph may have more than one vertex.

The paper is organized as follows. In Section 2, we explain why some particular structures of codes cannot be obtained by standard linear scheduling techniques, although they correspond to useful optimizations. These are codes containing loop independent dependences, i.e. codes that express sequentiality in parallel loops. In Section 3, we show how such codes can be generated for exploiting fine-grain parallelism. The technique is to modify standard scheduling techniques while introducing graph retiming techniques. In Section 4, we use a similar combination for extending Darte and Vivien's algorithm [5] (first designed for detecting innermost parallel loops, i.e. fine-grain parallelism) to the detection of maximal sets of permutable loops (i.e. medium-grain parallelism). Finally, in Section 5, we summary our main results, and we point out some open problems.

2 Sequentiality in parallel loops

Loops parallelized by scheduling techniques have a particular structure: each statement in the parallelized code is surrounded by a set of nested *parallel* [a] loops, surrounded by a set of *sequential* loops. The term "scheduling" comes from the fact that the outermost sequential loops can be interpreted as a description of the time steps, or synchronization steps, needed for computing the loops in a PRAM manner. The innermost parallel loops describe the set of computations carried at a given time step. By construction, these computations are completely independent: each dependence is carried by one of the sequential loops. Indeed, the general principle is to transform all dependences into dependences carried by the outermost loop (level 1 dependences). If this is not possible, as many dependences as possible are transformed into level 1 dependences, then as many as possible into level 2 dependences, and so on, until all dependences are carried by one of the constructed loops (which are therefore sequential). The remaining dimensions are completely independent. With such a principle, the final code never contains a loop independent dependence (null dependence distance). A consequence of this restriction is that some code structures that also describe fine-grain parallelism cannot be generated. We illustrate this fact on the following code structure:

```
for i=1 to n
    for j=1 to n
        S1
        S2
    endfor
endfor
```

Suppose that we succeeded to parallelize the above code with the scheduling technique called shifted-linear scheduling. This means that we have found an integral [b] vector $\vec{X} = (a, b)$ and two constants ρ_1 and ρ_2 such that iteration $\vec{I} = (i, j)$ of statement S_1 (resp. S_2) is carried (in the PRAM model) at logical time $\vec{X}.\vec{I} + \rho_1 = ai + bj + \rho_1$ (resp. $\vec{X}.\vec{I} + \rho_2$). Forgetting the time interpretation, this simply means that we apply a loop transformation for which $i' = ai + bj + \rho_1$ (resp.

[a] A loop is said parallel if it carries no dependences, i.e. if there is no dependences between different iterations of the loop.

[b] $\vec{X}$ may be chosen with rational components, in this case the logical time is $\lfloor \vec{X}.\vec{I} + \rho \rfloor$, and code generation may involve loop unrolling. We will not discuss this here. We assume all along the paper that timing vectors such as $\vec{X}$ are integral vectors.

$i' = ai + bj + \rho_2$) is the new loop counter for the first loop surrounding S_1 (resp. S_2). Now, two main cases can occur:

- The components of $\vec{X}$ are relatively prime: for each iteration of the outermost loop (corresponding to $\vec{X}$), a hyperplane of computations can be carried out in parallel, for $S1$ *and* for S_2. The resulting parallel code looks like the code of type (a) on Figure 1, possibly with some guards. This is typically the case if $\vec{X} = (1, 0)$, and for any ρ_1 and ρ_2. All dependences are carried by the first loop, and potential parallelism between S_1 and S_2 is exploited.

- The components of $\vec{X}$ are not relatively prime. A typical example is $\vec{X} = (2, 0)$, ρ_1 odd, and ρ_2 even. In this case, the even iterations of the outermost loop correspond to iterations of S_2, and the odd iterations to iterations of S_1. This can be written into a parallel code with the structure of type (b) on Figure 1, possibly with some guards. The dependences are either carried by the first loop, or occur between the first and the second parallel loop.

Code of type (a):
```
forseq
    forpar
        S1//S2
    endforpar
endforseq
```

Code of type (b):
```
forseq
    forpar
        S2
    endforpar
    forpar
        S1
    endforpar
endforseq
```

Code of type (c):
```
forseq
    forpar
        S2
        S1
    endforpar
endforseq
```

Fig. 1: Codes of type (a), (b) and (c).

On the other hand, with standard scheduling techniques, it is not possible to obtain a code such as the code of type (c) on Figure 1, which may contain a loop independent dependence (from S_2 to S_1 here). Yet, it can be interesting to generate such a code, for several reasons:

- If parallelism between S_1 and S_2 cannot be exploited anyway because of the machine programming model, a code of type (a) reveals too much parallelism. This is the case for example for a parallelizer that generates parallel code in an intermediate language such as HPF, and expresses parallel loops as `!hpf$ independent` directives: the potential parallelism $S_1//S_2$ cannot be exploited. Instances of S_1 and S_2 will be sequentialized, even if they can be carried out in parallel. In this case, a code of type (c) is sufficient. Of course, any code of type (a) can be sequentialized into a code of type (c). However, all codes of type (c) cannot be obtained this way (see our example in Section 3).

- A code of type (c) can lead to better performance than a code of type (b) when the minimization of communications and/or synchronizations is important. Indeed, for a code of type (b), a synchronization (or a phase of communications) is needed between the two parallel loops. In a code of type (c), all iterations can be carried in parallel, and possible communications from S_2 to S_1 take place *inside* a given iteration of the parallel loop. This principle is similar to the one used in Allen and Kennedy's algorithm where loop fusion (more precisely partial loop distribution) is shown useful to minimize synchronizations.

- Defining loop transformations that lead to codes of type (c) can also be useful for enlarging the set of valid schedules, and having more flexibility. This freedom gives us a better control on the code shape. We can use it to avoid loop skewing when it is not necessary, to keep loops perfectly nested if possible (which can be useful for tiling), to impose loop transformations to be unimodular if loop strides are not desirable, etc.

To conclude this short study, let us point out that codes of type (c) can be obtained simply by allowing loop independent dependences in the transformed codes. We now show that this can be done by combining standard scheduling techniques with graph retiming techniques, linked to Bellman-Ford's algorithm, for fine-grain parallelism detection (Section 3) as well as for medium-grain parallelism detection (and when possible, parallel outer loops detection) (Section 4).

Notations and hypotheses In the rest of the paper, we consider n perfectly nested loops whose dependences are represented by a polyhedral reduced dependence graph (PRDG), i.e. where set of distance vectors are approximated by non parameterized polyhedra. As shown in [5], we can capture such dependences through a modified reduced dependence graph with edges labeled by n-dimensional integral vectors, i.e. a uniform dependence graph. This graph has particular vertices (called virtual vertices) which are handled in a special way. However, to make the discussion simpler, we will forget about these vertices. Taking them into account is indeed mainly technical, but does not bring any fundamental difficulty. Going back, from the modified dependence graph, to the original dependence graph, is also conceptually not difficult. See [5] for a complete explanation of this "uniformization" process. The only important point is that the vectors that label the edges are not necessarily lexicographically positive. Therefore, in the rest of our study, we will make the following assumptions:

- The reduced dependence graph is uniform.
- There is no cycle of null weight.
- Dependence vectors are not necessarily lexicographically positive.

We will use the following notations: G is the reduced dependence graph (RDG), V the set of vertices and E the set of edges. $\#V$ (resp. $\#E$) is the number of vertices (resp. edges). $e = (x, y)$ is an edge of G from vertex x to vertex y. $\vec{w}_e$ is the weight of e. $\vec{w}_C$ denotes the weight of a cycle C (sum of the weights of its edges) and l_C denotes its length (number of edges).

3 Application to the detection of fine-grain parallelism

In this section, we explain how a technique based on shifted-linear scheduling can be modified so as to allow the generation of codes of type (c), i.e. codes with parallel loops and sequential bodies. As recalled in Section 2, shifted-linear scheduling consists in defining a logical time for computing the iterations of each statement S: iteration $\vec{I}$ of S is scheduled at time $\vec{X}.\vec{I} + \rho_S$. $\vec{X}$ will be used for building the outermost loop of the parallelized code. All dependences are carried by this outer loop, if possible. (We do not discuss here parallel outer loop detection.)

The two following lemmas show the differences between the "pure" shifted-linear approach and our modified approach which does not require all dependences to be loop carried in the transformed code. With this technique, we can find as much parallelism, and with more freedom on the choice of $\vec{X}$. The constraints on $\vec{X}$ given by Lemma 1 (resp. Lemma 2) are the constraints imposed by the pure shifted-linear approach (resp. the modified one).

Lemma 1 *Let G be a RDG. Let $\vec{X}$ be a vector which induces on each cycle C of G a delay greater than the length of C:*

$$\forall C \text{ cycle of } G,\ \vec{X}.\vec{w}_C \geq l_C$$

Then, for each vertex $v \in V$, there exists a constant ρ_v such that the shifted linear schedule built from $\vec{X}$ and the constants ρ_v is valid. In other words:

$$\forall e = (x, y) \in E,\ \vec{X}.\vec{w}_e + \rho_y - \rho_x \geq 1$$

Lemma 2 *Let G be a RDG. Let $\vec{X}$ be an integral vector which induces on each cycle C of G a delay greater than one:*

$$\forall C \text{ cycle of } G,\ \vec{X}.\vec{w}_C \geq 1$$

Then, for each vertex $v \in V$, there exists a constant ρ_v such that:

$$\forall e = (x, y) \in E,\ \vec{X}.\vec{w}_e + \rho_y - \rho_x \geq 0$$

Furthermore, the subgraph generated by the edges with null delay is acyclic.

The proofs use Bellman-Ford's algorithm. All details are provided in the extended version of this paper [3].

Note that the number of elementary cycles in a graph can be exponential in the number of vertices and edges of the graph. Therefore, checking directly that $\vec{X}.\vec{w}_C \geq 1$ or $\vec{X}.\vec{w}_C \geq l_C$ for all elementary cycles can be exponential, even if in practice it can be fast when the number of cycles is small. However, Lemma 1 shows that finding a $\vec{X}$ such that $\vec{X}.\vec{w}_C \geq l_C$ for all cycles is equivalent to solving $\#E$ inequalities with $\#V$ additional variables (the constants ρ_v), thus a polynomial number of inequalities. Expressing the constraints $\vec{X}.\vec{w}_C \geq 1$ with a polynomial number of inequalities and variables is more tricky, but also feasible.

With the example on Figure 2, we illustrate the differences between the three following techniques: linear schedule, shifted-linear schedule, and shifted-linear schedule allowing loop independent dependences.

```
for i=1 to N
    for j=1 to N
        S1: a(i, j) = b(i-1, j) + a(i-1,j)
        S2: b(i, j) = a(i, j-1)
    endfor
endfor
```

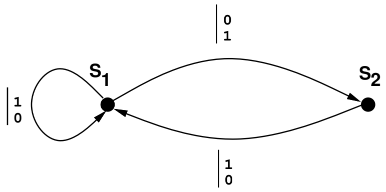

Fig. 2: First example and its RDG.

Linear schedule We look for a vector $\vec{X}$ such that $\vec{X}.\vec{w} \geq 1$ for each dependence vector $\vec{w}$ in the RDG. Because of the values of the two inter-dependences, both components of $\vec{X}$ must be greater than one. Hence we have to do at least one loop skewing. We choose $\vec{X} = (1,1)$ and we complete it into a unimodular matrix with the vector $(0,1)$. After transformation, we obtain the code of Figure 3.

Shifted linear schedule We look here for a vector $\vec{X}$ such that $\vec{X}.\vec{w}_C \geq l_C$ for each cycle C in the RDG. The two cycle weights are $(1,1)$ (cycle of length 2), and $(1,0)$ (cycle of length 1), thus we can choose $\vec{X} = (2,0)$. Using Lemma 1, we find two constants, $\rho_1 = 0$ and $\rho_2 = 1$, to complete the schedule. Once again, we complete $\vec{X}$ into a unimodular matrix using the vector $(0,1)$. After transformation, we obtain the code of Figure 4.

```
forseq i=2 to 2N
    forall j=max(1, i-N) to min(i-1, N)
        S1: a(i-j, j) = b(i-j-1, j) + a(i-j-1,j)
        S2: b(i-j, j) = a(i-j, j-1)
    endforall
endforseq
```

Fig. 3: Resulting code, type (a).

```
forseq i=1 to N
    forall j=1 to N
        S1: a(i, j) = b(i-1, j) + a(i-1,j)
    endforall
    forall j=1 to N
        S2: b(i, j) = a(i, j-1)
    endforall
endforseq
```

Fig. 4: Resulting code, type (b).

Shifted linear schedule allowing loop independent dependences We look here for a vector $\vec{X}$ such that $\vec{X}.\vec{w}_C \geq 1$ for each cycle C in the RDG. Using Lemma 2, we will find some constants to complete the schedule. This schedule is said to be *allowing loop independent dependences* because it does not induce a delay greater than one on all the edges as usually required, but only a non negative delay. Since all values are integers, a delay is either greater than one or null. If a delay is equal to zero, the dependence will only be satisfied by the ordering of the statements in the loop body. This can be achieved through a topological ordering of the subgraph generated by the edges with null delay, since this subgraph is acyclic (cf Lemma 2). Such a dependence will finally be transformed into a loop independent dependence.

In our example we can choose $\vec{X}$ equal to $(1,0)$, and both constants to be null.

But then S_1 must precede S_2 in the final loop body as the delay on the edge from S_1 to S_2 is null. Furthermore, constants in the remaining dimension must be carefully chosen so that the dependences from S_1 to S_2 is not carried (otherwise the remaining loop will be sequential). Once again, we complete $\vec{X}$ into a unimodular matrix using the vector $(0, 1)$. For the second dimension, we choose $\rho_1 = 1$, and $\rho_2 = 0$. After transformation, we obtain the following code, of type (c):

```
forseq i=1 to N
    forall j=1 to N+1
        S1: if (j > 1) then a(i, j-1) = b(i-1, j-1) + a(i-1,j)
        S2: if (j < N+1) then b(i, j) = a(i, j-1)
    endforall
endforseq
```

This technique completely solves the problem stated by Okuda in [12]: in a uniform nested loops, shift statements *before* searching a schedule, so that the latency of the best linear schedule is minimized. This can be done simply by minimizing the latency induced by a vector $\vec{X}$ subject to the constraints of Lemma 2.

4 Detecting fully permutable loops

Consider the piece of code described in Figure 5. It is a uniform program with four dependence vectors. Fine-grain parallelism detection will lead to a code with one sequential and one parallel loop, which may not be sufficient for achieving good performance on distributed memory machines. To increase the granularity of computations, we can use the tiling technique, introduced by Irigoin and Triolet [7], by first transforming the original loops into permutable loops. The condition of permutability is easy to check: two consecutive loops are permutable if and only if all dependence vectors, not carried by outermost loops, have non negative components in these dimensions.

```
for i=1 to N
    for j=1 to N
        S1: a(i, j) = b(i, j-1) + a(i, j-1)
        S2: b(i, j) = a(i-1, j+1) + b(i-1, j)
    endfor
endfor
```

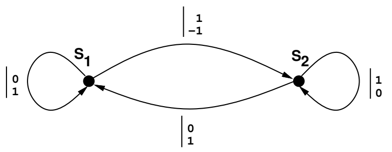

Fig. 5: Second example and its RDG.

The technique is as follows: a 2-by-2 non singular integral transformation matrix H is generated, such that $H\vec{w}_e \geq \vec{0}$ for each dependence vector $\vec{w}_e$. In particular, each row $\vec{X} = (a, b)$ of H satisfies $\vec{X}.\vec{w}_e \geq 0$. This leads to the following constraints:

$$a \geq 0 \quad b \geq 0 \quad a - b \geq 0 \quad b \geq 0$$

Here, the simplest linear independent solutions are: $\vec{X}_1 = (1, 0)$ and $\vec{X}_2 = (1, 1)$. H is a matrix for performing a *loop skewing*. The corresponding permutable code, in which tiling can be achieved, is the following:

```
for i=1 to N
      for j=1+i to N+i
            S1: a(i, j+i) = b(i, j+i-1) + a(i, j+i-1)
            S2: b(i, j+i) = a(i-1, j+i+1) + b(i-1, j+i)
      endfor
endfor
```

Now, let us use the same technique as in Section 3 so as to exploit the structure of the dependence graph. Instead of transforming each iteration vector $\vec{I} = (i, j)$ into $\vec{I'} = H\vec{I}$ for all statements, we allow statements to be shifted between each other. In other words, we transform iteration $\vec{I}$ of statement S into $H\vec{I} + \vec{\rho}_S$ where $\vec{\rho}_S$ is a *shift* vector, possibly different for each statement S. For the new loops to be permutable, the constraints are now that for each edge $e = (x, y)$ of the graph, $H(\vec{I} + \vec{w}_e) + \vec{\rho}_y \geq H\vec{I} + \vec{\rho}_x$, i.e. $H\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x \geq \vec{0}$. Reasoning row by row, it means that, for each row $\vec{X} = (a, b)$ of H, there are constants ρ_1 and ρ_2 such that:

$$a \geq 0 \quad b \geq 0 \quad a - b + \rho_2 - \rho_1 \geq 0 \quad b + \rho_1 - \rho_2 \geq 0$$

Here, the simplest linear independent solutions are: $\vec{X}_1 = (1, 0)$ (with $\rho_1 = \rho_2 = 0$) and $\vec{X}_2 = (0, 1)$ (with $\rho_1 = 0$ and $\rho_2 = 1$). H is simply the identity matrix and S_2 is moved forward one iteration along the j loop. The corresponding permutable code, in which tiling can be achieved, is the following:

```
for i=1 to N
   for j=1 to N+1
      S2: if (j > 1) then b(i, j-1) = a(i-1, j) + b(i-1, j-1)
      S1: if (j < N+1) then a(i, j) = b(i, j-1) + a(i, j-1)
   endfor
endfor
```

Remark that we interchanged S_1 and S_2 in the loop body. This is because, after transformation, all dependence vectors are now non negative, and some of them can even be null (loop independent dependences). To keep the semantic of the code, we have to order the statements inside the loop body so that loop independent dependences follow the textual order. For this to be possible, we have to make sure that the subgraph of G generated by loop independent dependences is acyclic. Once again, the technique is related to Lemma 2. The main difference with Section 3 is that, for tiling, we are looking for a family of independent vectors $\vec{X}$ (and not only for one vector $\vec{X}$) that form a matrix H of full rank. The condition on the weights of the cycles $\vec{w}_C$ given in Lemma 2 is now too strong. What we need is $H\vec{w}_C \geq \vec{0}$ and $H\vec{w}_C \neq \vec{0}$. It is now possible that one of the rows $\vec{X}$ of H satisfies $\vec{X}.\vec{w}_C = 0$, as long as at least one of the other rows satisfies $\vec{X}.\vec{w}_C > 0$.

We are now ready to generalize this technique to arbitrary reduced dependence graphs with uniform (but not necessarily lexicographically positive) dependences, as long as the graph has no cycle of null weight. We combine two ideas:

- Wolf and Lam's idea [17] that a set of perfectly nested loops can be transformed by unimodular transformations into a canonical form consisting of *nested blocks of fully permutable loops*. The technique is greedy and recursive. First, as many outermost permutable loops as possible are generated. All dependences have now non negative components in these dimensions. Some of them have at least one positive component: they are carried by at least one loop and are not considered any longer. The other ones are taken into account for building a new block of permutable loops. This recursive procedure ends when all dependences (except loop independent dependences of the original loops that remain unchanged) are finally carried by at least one of the generated loops.

- Darte and Vivien's idea [5] that fine-grain parallelism can be detected by "uniformizing" the polyhedral reduced dependence graph into the dependence graph of a system of uniform recurrence equations, which can be scheduled. The technique is also greedy and recursive. First, an outermost loop is generated that carries as many dependences as possible, possibly after shifting the different statements between each other. Then, all carried dependences are removed from the graph. The procedure keeps going on each strongly connected component of the remaining graph (called G') and the recursive procedure ends when all dependences are finally carried by at least one of the generated loops.

We mixed these two approaches: we aim at finding a nested structure of blocks of permutable loops as in Wolf and Lam's algorithm, but we exploit the structure of the reduced dependence graph, as in Darte and Vivien's algorithm, by allowing shifts between statements.

Each statement S is transformed by a multi-dimensional affine function: iteration $\vec{I}$ of S is represented by the new iteration vector $\vec{I'} = H_S\vec{I} + \vec{\rho}_S$ where H_S is a non singular n-by-n matrix. Following the technique used in [5] (called shifted-linear multi-dimensional schedules), we look for transformation matrices H_S whose first rows (as many as possible) are the same [c] for all statements within a given strongly connected component of G. After transformation, the first common r rows of the matrices H_S correspond to r permutable loops if

$$\forall e = (x, y) \in G,\ \ M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x \geq \vec{0} \tag{1}$$

where M is the r-by-n matrix of full rank formed by these row vectors. Our goal is to build such a matrix M while maximizing r.

Of course, M defines only one part of the final transformation. To be valid, the final transformation has to respect all dependences. Some of them are already carried by the loops defined by M. The other ones, corresponding to edges $e = (x, y)$ such that $M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x = \vec{0}$, will be satisfied either by a topological sort as in Section 3, or recursively in the subsequent dimensions.

For the sake of clarity, we only focus on the construction of the outermost block of permutable loops. We will explain briefly at the end of the section how to adapt this study to the whole recursive construction. Our problem is therefore the following: build a *full, maximal rank* matrix M (and its corresponding vectors $\vec{\rho}_v$) that can be extended to a n-dimensional valid transformation. Due to a lack of

[c] Such a restriction keeps optimality for maximal parallel loops detection in polyhedral reduced dependence graphs (PRDG), we conjecture it is also true for maximal permutable loops detection.

space, we only state the main results that are needed to understand the technique. All detailed proofs are available in the extended version of this paper [3].

Condition 1 is a necessary condition, expressed in terms of edges. It can be reformulated as a necessary condition on cycles:

Lemma 3 (Condition on cycles) *Let M be a matrix. M satisfies Condition 1 for some vectors $\vec{\rho}_v$, $v \in V$, if and only if $M\vec{w}_C \geq \vec{0}$ for each cycle C of G.*

We now show the fundamental role of G', the subgraph of G generated by the multi-cycles (union of cycles) of null weight, i.e. the subgraph generated by the edges of G that belong to a multi-cycle of null weight. We point out that G' is also the base of Karp, Miller and Winograd's decomposition [8] for the computability of systems of uniform recurrence equations, and of Darte and Vivien's algorithm [5] for the detection of parallelism in PRDGs. G' can be built by rational linear programming, with a polynomial number of constraints and variables (see [4]).

Lemma 4 (Condition on edges) *Condition 1 is equivalent to:*

$$\begin{array}{rcl} (i)\ \forall e \notin G',\ M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x & \geq & \vec{0} \\ (ii)\ \forall e \in G',\ M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x & = & \vec{0} \end{array}$$

The matrix M is composed by r row vectors $\vec{X}_i$, with $1 \leq i \leq r$. Our goal is to maximize r. Let U be the vector space generated by the weights of the cycles of G'. Let k be the dimension of U. We have the following result:

Lemma 5 ($\mathbf{Vect}(\vec{X}_i) = U^{\perp}$) *The rows of M, i.e. the r vectors $\vec{X}_i$, form a basis of $U^{\perp}$. Thus, $r = n - k$.*

There is an infinite number of matrices M, of rank $n-k$, satisfying $M\vec{w}_C \geq \vec{0}$ for each cycle C of G. To build one of them, we have two possibilities. On one hand, if the number of elementary cycles is small, we can directly work with the cone generated by the weights of the cycles of G. The corresponding polar cone contains all candidate vectors $\vec{X}$. Then, to select the matrix M, optimization techniques such as in [2] can be used. On the other hand, if generating all the elementary cycles is too expensive, we can still build one solution in polynomial time, by choosing M as done in the proof of Lemma 5 (see [3] for more details). The key point is that a basis of U can be built in polynomial time using a *basis of cycles* of G'. Then a basis of $U^{\perp}$ that satisfies the conditions of Lemma 3 or Lemma 4 can also be built in polynomial time.

Note also that if the vector space defined by the weights of **all** cycles is not full-dimensional (of dimension $n' < n$), it is possible to build M so that the $n - n'$ outer loops are parallel.

As already noticed, no matter how M is completed into a square matrix of size n, each dependence that corresponds to an edge $e = (x, y)$ such that $M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x \geq \vec{0}$ and $M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x \neq \vec{0}$ will be satisfied as already carried by one of the loops corresponding to M. We still have to consider all other edges, those such that $M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x = \vec{0}$. The key point is that we can satisfy them recursively. Indeed, we can prove (see [3]) that the matrices M, which enable us to build a maximal set of fully permutable loops, are the matrices M, of full and maximal rank, satisfying both following properties:

1. there exist some vectors $\vec{\rho}_v$, $v \in V$, such that:

$$\begin{array}{rcl} (i)\ \forall e \notin G',\ M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x & \geq & \vec{0} \\ (ii)\ \forall e \in G',\ M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x & = & \vec{0} \end{array}$$

2. the subgraph G^* of G generated by the edges $e = (x, y)$ for which $M\vec{w}_e + \vec{\rho}_y - \vec{\rho}_x = \vec{0}$ is a forest of strongly connected components, and those with at least one edge are exactly the strongly connected components of G'.

The characterization above leads to a recursive construction of the whole n-dimensional transformation. As said before, each edge not in G^* is carried by one of the loops corresponding to M. Edges in G^*, but not in G', can be satisfied by a topological ordering of the strongly connected components of G^*. Finally, edges in G' will be satisfied through the recursive processing of the strongly connected components of G' which completes the matrix M already built. The construction of the new rows of M (which may be different for each strongly connected component of G') is done the same way. The only difference is that they must be chosen with an additional constraint: they have to be linearly independent with the existing rows of M. As in [5], the correctness of this recursive algorithm comes from the fact that G has no cycles of null weight. From a practical point of view, we point out that all statements do not necessarily have the same final n-dimensional matrix M. However, we can impose these matrices to be unimodular so as to get simpler codes.

We illustrate our technique on the following code:

```
for i=1 to N
  for j=1 to N
    for k=1 to N
      S1: a(i, j, k) = a(i, j-1, k) + b(i, j, k-1)
      S2: b(i, j, k) = b(i-1, j+i, k) + a(i, j-1, k+j)
    endfor
  endfor
endfor
```

Figure 6 shows the reduced dependence graph with direction vectors. Figure 7 shows the "uniformized" dependence graph G.

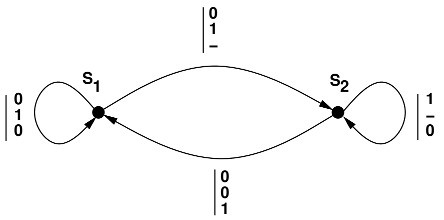

Fig. 6: RDG for third example.

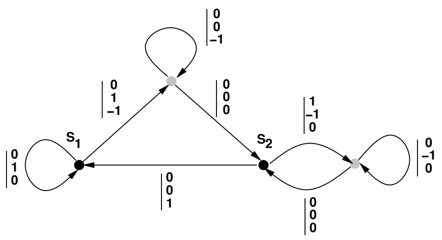

Fig. 7: "Uniformized" RDG (Example 3).

G has five cycles, three self-dependences with weights $(0, 1, 0)$, $(0, 0, -1)$, and $(0, -1, 0)$, and two other cycles with weights $(0, 1, 0)$ and $(1, -1, 0)$. Therefore, G'

is the graph generated by the cycles whose weights are $(0,1,0)$ and $(0,-1,0)$. The dimension of U is 1. We can thus build $3-1=2$ outermost permutable loops. Here, we see directly that the two canonical vectors $\vec{X_1}=(1,0,0)$ and $\vec{X_2}=(0,0,-1)$ belong to $U^\perp$, and that they satisfy $\vec{X_i}.\vec{w}_C \geq 0$ for all other cycles. Thus, we can choose them as the rows of M. We just have to find the corresponding shift vectors. We get $\vec{\rho}_{S_1}=(0,1)$, and $\vec{\rho}_{S_2}=(0,0)$. Here, $G^*=G'$, no topological sort is required. We consider the strongly connected component that contains S_1 and S_2, and we look for a vector $\vec{X_3}$, linearly independent with $\vec{X_1}$ and $\vec{X_2}$, such that $\vec{X_3}.(0,1,0) \geq 0$, e.g. $\vec{X_3}=(0,1,0)$. No shift in this dimension is required, however S_2 has to be textually ordered before S_1. To rewrite the code, we use the function *codegen* of the software Petit [9]. Our final transformation can be expressed in Petit's framework as $S_1(i,j,k) \rightarrow (i,-k+1,j,1)$ and $S_2(i,j,k) \rightarrow (i,-k,j,0)$, and we get the code of Figure 8 in which tiling can be performed on the two outermost loops. Note that, in this example, permutable loops cannot be detected by Wolf and Lam's algorithm.

```
for i=1 to N
  for k=-N to 0
    for j=1 to N
      S2: if (k < 0) then
            b(i, j, -k) = b(i-1, j+i, -k) + a(i, j-1, -k+j)
      S1: if (k > -N) then
            a(i, j, -k+1) = a(i, j-1, -k+1) + b(i, j, -k)
    endfor
  endfor
endfor
```

Fig. 8: Final permutable code for third example.

5 Conclusion

In this paper, we enlarge the set of codes that can be generated by standard linear scheduling techniques, and that expose either parallel loops or permutable loops. Our method exploits the structure of the dependence graph by combining graph retiming and scheduling techniques.

For fine-grain parallelism detection, we are now able to generate codes with parallel loops that contain loop independent dependences. This can be useful for minimizing communications and/or synchronizations.

For medium-grain parallelism detection, we generalize Wolf and Lam's algorithm to the case of loops with multiple statements. We generate maximal sets of fully permutable loops that are essential for tiling.

We still have some open problems: how to define a criterion of optimality for the detection of permutable loops? How to handle non perfectly nested loops? How to choose the size and shape of a tile? How to map data with respect to the chosen tiling? Our future work will address these problems.

References

1. J.R. Allen and K. Kennedy. Automatic translations of Fortran programs to vector form. *ACM Toplas*, 9:491–542, 1987.
2. Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.
3. Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. Technical Report 96-34, LIP, ENS-Lyon, France, November 1996.
4. Alain Darte and Frédéric Vivien. Revisiting the decomposition of Karp, Miller, and Winograd. *Parallel Processing Letters*, 5(4):551–562, December 1995.
5. Alain Darte and Frédéric Vivien. Optimal fine and medium grain parallelism in polyhedral reduced dependence graphs. In *Proceedings of PACT'96*, Boston, MA, October 1996. IEEE Computer Society Press.
6. Paul Feautrier. Some efficient solutions to the affine scheduling problem, part II, multi-dimensional time. *Int. J. Parallel Programming*, 21(6):389–420, December 1992.
7. F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.
8. R.M. Karp, R.E. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.
9. W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *New user interface for Petit and other interfaces: user guide.* University of Maryland, June 1995.
10. M.S. Lam and M. E. Wolf. Automatic blocking by a compiler. In J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 537–542, 1992.
11. Leslie Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.
12. 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.
13. J. Ramanujam. A linear algebraic view of loop transformations and their interaction. In J. Dongarra, K. Kennedy, P. Messina, D. C. Sorensen, and R. G. Voigt, editors, *Proc. of the fifth SIAM Conference on Parallel Processing for Scientific Computing*, pages 543–548, 1992.
14. R. Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report 90-38, The University of Tennessee, Knoxville, TN, August 1990.
15. Alexander Schrijver. *Theory of Linear and Integer Programming.* John Wiley and Sons, New York, 1986.
16. B. Sinharoy and N. Szymanski. Finding optimum wavefront of parallel computation. In H. El-Rewini, T. Lewis, and B. D. Shriver, editors, *Proc. of the Twenty-Sixth Annual Hawaii International Conference on System Sciences*, volume 2, pages 225–234. IEEE Computer Society Press, 1993.
17. 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.