

# High Level Loop Transformations for Systematic Signal Processing Embedded Applications

Calin Glitia and Pierre Boulet

Laboratoire d'Informatique Fondamentale de Lille  
Université des Sciences et Technologies de Lille  
INRIA Lille - Nord Europe  
59655 Villeneuve d'Ascq Cedex, France  
{glitia.calin, pierre.boulet}@lifl.fr

**Abstract.** Array-OL specification model is a mixed graphical-textual language designed to model multidimensional intensive signal processing applications. Data and task parallelism are specified directly in the model. High level transformations are defined on this model, allowing the refactoring of an application and furthermore providing directions for optimization. The resemblances between with the wide-known and used *Loop transformations* lead us to try taking concepts and results from this domain and see how they fit in Array-OL context.

**Keywords:** Multidimensional Dataflow, optimizations, loop transformations.

## 1 Introduction

In the last years, the gap between the performances claimed by the constructors and the ones achieved with real code has drastically increased. This is caused mainly by the brutal increase in processor complexity which brought with it a drastic degradation of the code generated by the compilers. The three major directions for improving the performances are: (1) increasing the instruction parallelism while multiplying the mechanism to allow the simultaneous execution of instructions; (2) improving the speculative mechanisms that allow the prediction of programs local behavior; (3) the implementation of a complex memory hierarchy for exploiting as well as possible the time and space data locality.

For all these directions, the source-to-source transformations techniques have a determinant role. Most of these techniques are represented by transformations applied on “for” loops which are efficient in the case of code that contains extremely regular data treatment.

Array-OL (*Array Oriented Language*) is a modeling language designed in order to conform to the needs for specification, standardization and efficiency of the multidimensional systematic signal processing [2]. This application domain is characterized by systematic, regular, and massively data-parallel computations. Array-OL relies on a graphical formalism in which the signal processing appears as a graph of tasks. Each task reads and writes multidimensional arrays in an extremely regular pattern.

In this paper we try to make a comparison between loop transformations and the Array-OL transformations, identify the resemblances and directions for using results from loop transformations optimization techniques to Array-OL.

## 2 Loop Transformations

An important early system level technique, the loop transformation technique, is aiming at improving the data access regularity and locality. Hence it reduces the overall memory size requirement and the access frequency to big and slow memories. This is vital to area, power consumption, and performance. Improved data access regularity and locality shorten the lifetimes of data elements and increases the memory location reuse ratio since memory locations can be reused for data elements with non-overlapping life-times.

Methods are divided into two classes: global methods which deal with each loop as atomic computation unit and local methods which change the way loops are organized internally. Here is a list of some of the global transformations that are useful for optimization. Global methods:

- **Code moving** that changes the execution order between two loops in the program without modifying the loops.
- **Loop fusion** that groups several loops in a unique one, used to reduce the size of intermediate arrays.
- **Loop splitting** that represents the reverse of merging. It attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which iterate over different contiguous portions of the index range.

Local transformations explore more in depth the way loops are organized internally:

- **Loop tiling** acts on partitioning of large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and reducing cache size requirements.
- **Loop pipelining** shifts some instructions from one to several iterations within the loop body. This is used to increase to data locality.
- **Loop collapsing** is the reverse of tiling.

These transformations usually are combined in order to achieve best performances. As an observation, these are just some of the existing loop transformations; the most common we could say.

### 2.1 Loop Optimization Techniques

Typically, applying a compiler optimization consists of three steps: decide upon a part of the program to optimize and the enchainment of transformations to be applied; verify the correctness of the optimization; and last, applying the transformations. As processor architectures become more and more complex, the number of dimensions in which optimizations are possible increase and this makes the decision process very complex.

The complexity of optimization algorithms is the reason why many compilers still use heuristics. This implies basically the use of the same chain of transformations, the one that proves to reach a relatively good result in most of the cases.

The complexity of the problem determined the need to introduce ways of representing the problem (constraints, transformations, cost function) using a more effective

formalism and which could facilitate the manipulation of concepts like correctness, data dependencies, cost function. Some approached the problem using Linear Algebra [4], Polyhedral Abstraction [6], graph theory algorithms or Integer Linear Programming [5]. The introduction of formalism is extremely important for the decision part of the optimization. Correct and complex optimization algorithms need to be designed around such formalisms.

### 3 Array-OL Model of Specification

The initial goal of Array-OL is to give a mixed graphical-textual language to express multidimensional intensive signal processing applications. These applications work on multidimensional arrays and their complexity does not come from the elementary functions they combine, but from their combination of the ways they access the intermediate arrays. As these applications handle huge amounts of data under tight real-time constraints, the efficient use of the potential parallelism of the application on parallel hardware is mandatory.

#### 3.1 Principles

Form these needs, we can state the basic principles that underly the language:

- Array-OL is a *data dependence expression* language. Only the true data dependencies are expressed in order to express the full parallelism of the application.
- Data access is done through sub arrays, called patterns.
- The language is *hierarchical* to allow descriptions at different granularity levels and to handle the complexity of the applications.
- All the potential parallelism in the application should be available in the specification, both *task parallelism* and *data parallelism*.
- It is a *single assignment* formalism.
- The spatial and temporal dimensions are treated equally in the arrays.
- The arrays are seen has tori.

*The semantics of Array-OL is that of a first order functional language manipulating multidimensional arrays. It is not a data flow language but can be projected on such a language.*

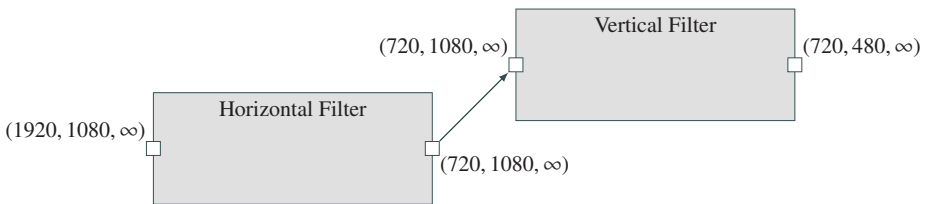
The usual model for dependence based algorithm description is the dependence graph where nodes represent statements and edges dependencies. In order to represent complex applications, a common extension of these graphs is the hierarchy. Array-OL builds upon such hierarchical dependence graphs and adds a special kind of node to represent the data-parallelism of the application: repetition nodes.

Formally, an Array-OL application is a set of *components* connected through *ports*. The components are equivalent to mathematical functions reading data on their input ports and writing data on their output ports. The components are of three kinds: *elementary*, *compound* and *repetition*. An *elementary* component is atomic (a black box). A *compound* is a dependence graph whose nodes are components connected via their ports. A *repetition* is a component expressing how a single sub-component is repeated.

All the data exchanged between the components are arrays. These arrays are multidimensional and are characterized by their *shape*, the number of elements on each of their dimension. Each port is thus characterized by the shape and the type of the elements of the array it reads from or writes to.

### 3.2 Tasks Parallelism

For a better understanding, in the rest of the study we will use to illustrate the Array-OL concepts on an application that scales an high definition TV signal down to a standard definition TV signal, called *downscaler*. Both signals are represented as a three dimensional array; the first two dimensions represent the frame resolutions ( $1920 \times 1080$  at the input and  $720 \times 480$  at the output) while the third represents the flow of frames (in time). The application's task dependence is presented in *Figure 1*. The application is constituted from two filters, the horizontal and the vertical filter.



**Fig. 1.** Downscaler application – task dependence

Each execution of a task reads one full array on its inputs and writes the full output arrays. *The graph is a dependence graph, not a data flow graph.*

### 3.3 Data Parallelism

A data-parallel repetition of a task is specified in a repetition task. The basic hypothesis is that all the repetitions of this repeated task are independent. They can be scheduled in any order, even in parallel<sup>1</sup>. The second one is that each instance of the repeated task operates with sub-arrays of the inputs and outputs of the repetition. For a given input or output, all the sub-array instances have the same shape, are composed of regularly spaced elements and are regularly placed in the array. This hypothesis allows a compact representation of the repetition and is coherent with the application domain of Array-OL which describes very regular algorithms.

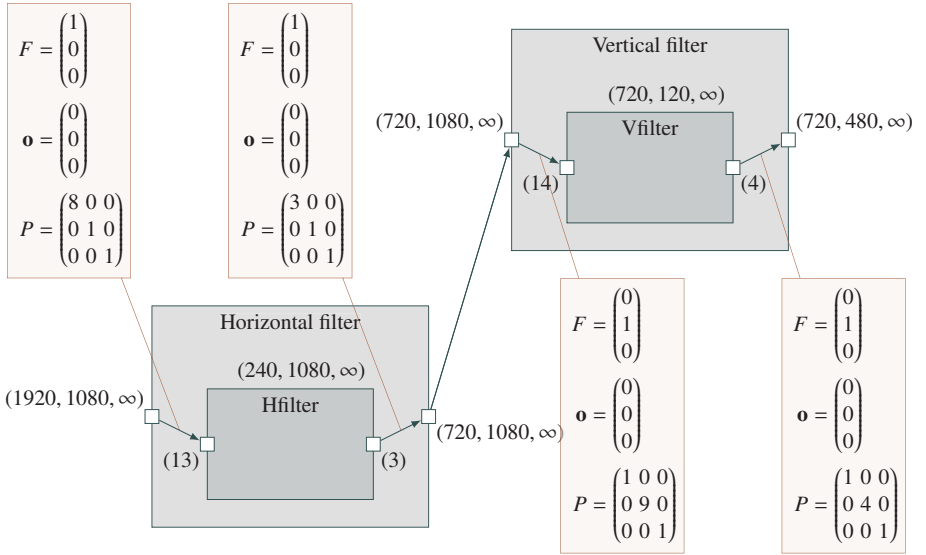
As these sub-arrays conform, they are called *patterns*. In order to give all the information needed to create these patterns, a *tiler* is associated to each array (ie each edge). A tiler is able to build the patterns from an input array, or to store the patterns in an output array. It describes the coordinates of the elements of the tiles from the coordinates of the elements of the patterns. It contains the following information:

<sup>1</sup> This is why we talk of *repetitions* and not *iterations* which convey a sequential semantics.

- $F$ : a *fitting* matrix.
- $\mathbf{o}$ : the *origin* of the *reference pattern* (for the *reference repetition*).
- $P$ : a *paving* matrix.

The shapes of the arrays and patterns are, as in the compound description, noted on the ports. The *repetition space* indicating the number of repetitions is defined itself as an multidimensional array with a shape. Each dimension of this repetition space can be seen as a parallel loop and the shape of the repetition space gives the bounds of the loop indices of the nested parallel loops.

In the downscaler application, each of the two filters has a repetitive functionality, so this means we can represent them by using repetition components. Thus the complete representation is presented in *Figure 2*.



Each of the filter has a repetitive functionality that is described with the tilers. For example, the horizontal filter's elementary component takes a window of 13 elements that slides with 8 elements on each line of each image frame and produces 3 elements.

**Fig.2.** Complete specification of the downscaler application

Returning now to the Array-OL specifications, for each repetition, one needs to design the reference elements of the input and output tiles and the elements of these tiles. The reference elements of the reference repetition are given by the *origin* vector,  $\mathbf{o}$ , of each tiler. The reference elements of the other repetitions are built relatively to this one. Their coordinates are built as a linear combination of the vectors of the *paving* matrix as follows

$$\forall \mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\text{repetition}}, \text{ref}_{\mathbf{r}} = \mathbf{o} + P \times \mathbf{r} \mod \mathbf{s}_{\text{array}} \quad (1)$$

where  $\mathbf{s}_{\text{repetition}}$  is the shape of the repetition space,  $P$  the paving matrix and  $\mathbf{s}_{\text{array}}$  the shape of the array. The elements of the tile of repetition  $r$  are built relatively to the reference element of this tile using a linear combination of the vectors of the *fitting* matrix as follows

$$\forall \mathbf{i}, 0 \leq \mathbf{i} < \mathbf{s}_{\text{pattern}}, \mathbf{e}_{\mathbf{i}} = \text{ref}_{\mathbf{r}} + F \times \mathbf{i} \mod \mathbf{s}_{\text{array}} \quad (2)$$

where  $\mathbf{s}_{\text{pattern}}$  is the shape of the pattern.

### 3.4 Projection onto an Execution Model

It is a strength of Array-OL that the space-time mapping decision is separated from the functional specification. This allows to build functional component libraries for reuse and to carry out some architecture exploration with the least restrictions possible. Mapping compounds is not specially difficult. The problem comes when mapping repetitions. This problem is discussed in details in [1] where the authors study the projection of Array-OL onto Kahn process networks [7]. The key point is that some repetitions can be transformed to flows. In that case, the execution of the repetitions is sequentialized (or pipelined) and the patterns are read and written as a flow of tokens (each token carrying a pattern).

### 3.5 Array-OL Transformations

A set of Array-OL code transformations has been designed to allow to adapt the application to the execution, allowing to choose the granularity of the flows and a simple expression of the mapping by tagging each repetition by its execution mode: data-parallel or sequential. This paper is not meant to give a complete presentation of the Array-OL transformations; the topic is much too complex. More details can be found in the PhD thesis of Julien Soula [9] and Philippe Dumont [3].

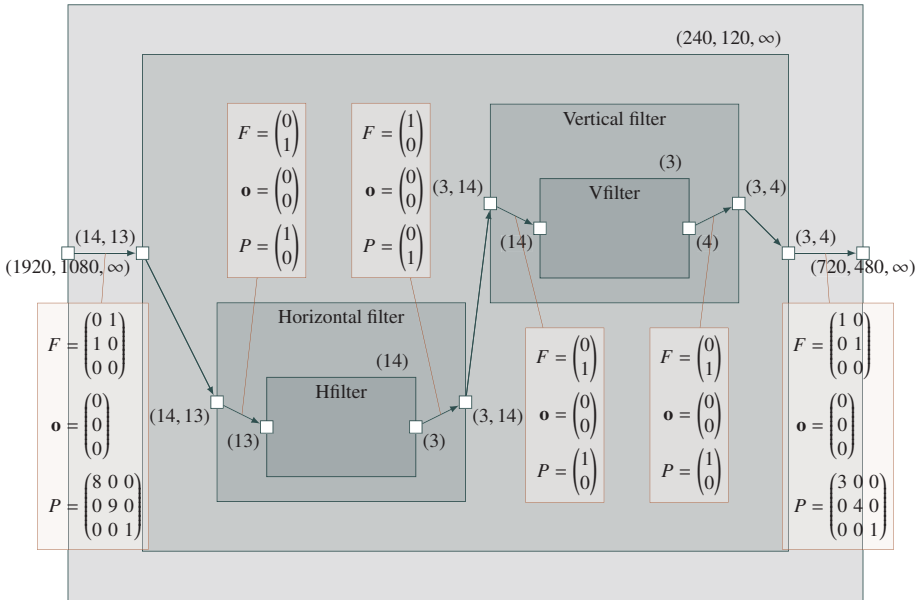
A major problem for designing an execution model for Array-OL is introduced by the so called “synchronization barriers” between the components. Such a barrier is created by the data dependencies. A task cannot begin its execution until all its input arrays are entirely produced. A sequential execution is, by consequence, not appropriate; the presence of any intermediary array that contains an infinite dimension would cause the execution to be stalled in that point. A solution could be a pipelined execution by refactoring the application using the Array-OL transformations. Using the hierarchy, we intend to isolate the infinite dimensions at the top hierarchical level of the application (which will represent the data-flow), while in the lower levels we can choose a pipelined execution.

All the Array-OL transformations are based on a mathematical formalism that ensures their correctness, but which will not be presented due to limited paper size. Details can be found in the bibliography.

**Fusion** This transformation basically takes two components that have at least one common array (the first component produces an array consumed by the second component) and these two components are merged into a single compound component containing the previous two. The result of fusion is the creation of a hierarchy level, with a common repetition and sub-repetitions on the lower level hierarchy. The components keep

their functionality after the fusion but the difference is that the arrays that they work on are different (parts of the original arrays). The question is how the parts of the original arrays are chosen and why? In our implementation the fusion was designed in such a way that the created compound component takes the smallest possible patterns from the input arrays that can produce at least one element of each output arrays.

In Figure 3 we can see the result of the fusion on the downscaler application. We can see that after the transformation the two initial filters are merged into a single component which contains the initial filters that now consume different arrays, the infinite dimension remaining at the top level.



After the fusion, a hierarchy level was introduced in the application, the original filters were merged into a single compound component that passes now just parts of the initial arrays to the filters.

**Fig. 3.** Downscaler after fusion transformation

**Change paving** transformations can be used to change the granularity of the applications or of parts of the application by redistributing repetitions between hierarchy levels. As a direct consequence it can be used to reduce the redundant computations (called *re-calculations*) generated in some cases by the fusion. This problem can appear after the fusion, if the first component before the fusion produced overlapping patterns. This will cause the first sub-component after the fusion to compute multiple times the same elements of the original arrays. What we can do is reducing the amount of re-calculations by extending the pattern of the compound component so it will include more. In the extreme case, if we extend to the maximum the pattern on all the paving vectors which

cause re-calculations, we may even eliminate the phenomenon. Still, this is not possible in the case where the re-calculations are present on the infinite dimension without eliminating a major role of the fusion, that of isolating the infinite dimension on the top level of the hierarchy.

*Change paving by adding dimensions*, as its name indicates, extends the pattern by the use of an extra dimension, having the size of the change paving *level*. Parts of the repetition of the top level descend a level of hierarchy as new dimensions of the repetition spaces of sub-components. *Change paving by linear growth* transformation is designed specially to reduce the re-calculations and so it can be applied only on tasks that contain re-calculations. What this transformation does different than the previous is to calculate a surrounding pattern and in this way the transformation can be used to reduce the re-calculations. We extended the use of the transformation to the case where the patterns are “glued” together, even if there are no re-calculations. It’s advantage is that it does not introduces extra dimensions to the arrays involved in the transformation.

**Tiling** transformation was designed in order to allow the introduction of *granularity degree* concept in an application. This concept, introduced in the context of control, allows to delimitate different *execution cycles*. More details on this topic can be found in the PhD thesis of Ouassila Labbani [8] (chapter 7.3). A *granularity degree* basically defines a subset of the repetition domain that corresponds at the execution to a controlled Array-OL component. The result of such a transformation is similar to the loop tiling and is basically the separation in functional blocks that have as an important characteristic the increased locality.

**Collapse** The fusion transformation can work only on two tasks at a time. In we want to fusion three or more tasks we must apply the fusion multiple times and this will lead to the creation of what we call “abyssal hierarchies”, applications that are spread on multiple hierarchy layers. The solution is the collapse transformation, represented by a series of maximum change paving transformations that have the effect of extending the patterns of the compound component so it contains all the original patterns and in this way this component can be eliminated by replacing it with its sub-components, which will “climb” a level in the hierarchy.

By applying a certain number of transformations we can change the structure of an Array-OL application without modifying it’s functionality. One can use these transformations to refactor the application to respect various constraints (timing, hardware mapping, memory optimization).

## 4 Array-OL vs. Loop Transformations

Loop transformations are most efficient on code that contains extremely regular data treatment (perfectly-nested loops) which is exactly the domain of Array-OL.

We start with some important observations on these transformations. First, Array-OL transformations have a major advantage over loop transformations that are usually local optimizations while the Array-OL ones can be applied at any level of the hierarchy thanks to the pattern based data accesses. The pattern based data accesses make the Array-OL access structure more visible and much easier to manipulate, differently from

the complex formulas manipulating the loop indices. There are also disadvantages with Array-OL; the most important is introduced by the limitations of the language, one of them being the extreme regularity. This restrains the domain of applications that can be specified with Array-OL to a limited set.

We will not compare separately each pair of transformations, each Array-OL transformation resembles in functionality with its homonym, but rather try to identify the role of each transformation and its possible usage. When passing to an execution model in Array-OL there are a set of key concepts that must be carefully analyzed. First, we must isolate as much as possible the infinite dimension but in the same time respect the internal constraints introduced by the data dependencies and avoid any blocking points in the execution. All these are done by the use of the fusion that has three major effects: it isolates the infinite dimension on the top hierarchy level, it minimizes the intermediate arrays and guaranties a non-blocking structure. As the loop fusion, they both have the role of merging two dependent entities (Array-OL components in the first case and loop-nest in the other) with the purpose of eliminating or at least reducing intermediate data size. An advantage of Array-OL fusion is that it automatically does the array resize, while the loop transformation needs other transformations in order to achieve this, like the scalar replacement or intra-array storage order optimization. The fusion in Array-OL can be used to reach a multi-level application structure where all the infinite dimensions are left on the top level that will represent the data-flow. The collapse transformation has an important role in connection with the fusion, for avoiding the apparition of “abyssal hierarchies” created by chaining fusions.

The change paving, resembles with the loop unrolling. They both act on redistributing the iterations between levels (hierarchy levels or nest levels). In the context of Array-OL we can use this type of transformation for example to restructurate the application so it respects the environment constraints.

The Array-OL tiling corresponds to the loop tiling or partitioning transformation; the first introduces a level of hierarchy while the second introduces a nesting level to the loop-nest. The both have the role of splitting the iteration space into functional blocks which has a positive influence on the data locality.

We must note that in the context of Array-OL optimizations we don't need to search to increase the parallelism of the application, the parallelism is evident, it was one of the starting point of Array-OL to produce a specification language where the parallelism is fully expressed in the specifications. What we are most interested in is memory optimizations (static and dynamic), but by respecting the application constraints. None the less, transformations change the structure of an application and this implies changes to the parallelism.

Algorithms based on loop transformations that can give the optimum solution for memory optimizations are not practical, due to complexity issues. Most of the times heuristics are used. In the context of Array-OL we can also use as a starting point a heuristic, the one that involved the transformation of an application to the multi-levels structure, which has proved extremely useful.

As said in the introduction, the Array-OL language presents some advantages. The application defined in Array-OL is extremely regular and this regularity is contained directly in the language; also the parallelism is evident so this is another thing that we

don't have to worry about. Another advantage is brought by the ODT formalism, which guaranties the correctness of the transformations as regarding the data dependencies.

## 5 Conclusions

Array-OL transformations have a determinant role in the context of Array-OL. They can be used not only for optimization but also as a tool for refactoring the application. For now it is just an instrument in the hands of the designer but in the future, after the needed concepts will be introduced to Array-OL, optimization algorithms using the presented transformations will be designed and implemented. These optimizations also depend on the execution model chosen for the Array-OL model and they will evolve in parallel with the evolution of the execution models.

## References

1. Amar, A., Boulet, P., Dumont, P.: Projection of the Array-OL specification language onto the Kahn process network computation model. In: International Symposium on Parallel Architectures, Algorithms, and Networks, Las Vegas, Nevada, USA (December 2005)
2. Demeure, A., Lafarge, A., Boutillon, E., Rozzonelli, D., Dufourd, J.-C., Marro, J.-L.: Array-OL: Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In: Grets, Juan-Les-Pins, France (September 1995)
3. Dumont, P.: Spécification Multidimensionnelle pour le traitement du signal systématique. Phd thesis, Laboratoire d'informatique fondamentale de Lille (2005)
4. Feautrier, P.: Dataflow analysis of array and scalar references. *International Journal of Parallel Programming* 20(1), 23–53 (1991)
5. Fraboulet, A.: Optimisation de la mémoire et de la consommation des systèmes multimédia embarqué. Phd thesis, LIP (November 2001)
6. Girbal, S.: Optimisation d'applications - Composition de transformations de programme: modèle et outils. Phd thesis, University Paris 11, Orsay, France (September 2005)
7. Kahn, G., MacQueen, D.B.: Coroutines and networks of parallel processes. In: Gilchrist, B. (ed.) *Information Processing 77: Proceedings of the IFIP Congress 77*, pp. 993–998 (1977)
8. Labbani, O.: Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif. Phd thesis, Laboratoire d'informatique fondamentale de Lille (2006).
9. Soula, J.: Principe de Compilation d'un Langage de Traitement de Signal. Phd thesis, Laboratoire d'informatique fondamentale de Lille (December 2001)