



High Level Loop Transformations for Systematic Signal Processing Embedded Applications

Calin Glitia, Pierre Boulet

► To cite this version:

Calin Glitia, Pierre Boulet. High Level Loop Transformations for Systematic Signal Processing Embedded Applications. [Research Report] RR-6469, INRIA. 2008, pp.27. inria-00262023v2

HAL Id: inria-00262023

<https://inria.hal.science/inria-00262023v2>

Submitted on 11 Mar 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

***Transformations de haut niveau pour application de
traitement systématique de signal sur systèmes
embarqués***

Calin GLITIA — Pierre BOULET

N° 6469 — version 2

version initiale Mars 2008 — version révisée Mars 2008

Thème COM

 ***rapport
de recherche***

Transformations de haut niveau pour application de traitement systématique de signal sur systèmes embarqués

Calin GLITIA , Pierre BOULET

Thème COM — Systèmes communicants
Équipe-Projet DaRT

Rapport de recherche n° 6469 — version 2 — version initiale Mars 2008 — version
révisée Mars 2008 — 27 pages

Résumé : Array-OL modèle de spécification est un langage combiné graphique textuel spécialisé dans la description d'applications de traitement du signal systématique. Le parallélisme de donnée et des tâches est spécifié directement dans l'application. Transformations de haut niveau sont défini sur ce modèle, permettant de refactoriser une application et en plus offrant directions d'optimisation. Les similitudes avec les connus est utilise transformations des boucles nous ont mené a essayer de prendre des concepts et résultats de cette domaine et voir comment de les mettre dans le contexte de Array-OL. On essay ici d'identifier liaisons entre ces deux domaines et aussi directions à l'avenir pour Array-OL, techniques d'optimisation en spécial.

Mots-clés : Flots de données multidimensionnels, traitement de signal, optimisation, transformations des boucles

High Level Loop Transformations for Systematic Signal Processing Embedded Applications

Abstract: Array-OL specification model is a mixed graphical-textual language designed especially 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 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. We try to identify the links between these two domains and also future directions for Array-OL, optimization techniques especially.

Key-words: Multidimensional Dataflow, signal processing, optimization, 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 major three directions for improving the performances are :

- increasing the instruction parallelism while multiplying the mechanism to allow the simultaneous execution of instructions and in the same time reducing as much as possible their latencies ;
- improving the speculative mechanisms that allow the prediction of programs local behavior ;
- the implementation of a complex memory hierarchy for exploiting as well as possible the time and space data localities.

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 can be used in two directions :

- increasing the instruction parallelism
- improving the data access regularity and locality and furthermore removing the system-level buffers of the application code.

“For” loop transformations change the way arrays are traversed all over the program. Although these transformations are extremely efficient and they were extensively researched, some major problems that need resolution still exist.

First of all, the usability issue ; the transformations can be used mostly in the case of code that contains extremely regular data treatment. Most of the research on source-to-source transformations was conducted on what we call “perfect loop-nests”. This reduces the domain of applications to a limited set, mainly of data-flow treatments. Some researchers tried the extension to “un-perfect” loop-nests or even further but the results were limited.

Another problem that many researchers encountered is the need for formalization. A traditional loop-nest as found in the source-code of an application has proven to be hard to manipulate. Therefore, the formalization is extremely important and the most important directions involved the use of concepts from Linear Algebra, Parametric Integer Programming or Polyhedral Abstraction [8, 6, 7].

And, last but not least, the optimization algorithms are of great importance to the source-to-source transformations topic. The traditional problem can be summarized as finding the optimum enchainment of transformations that applied on a given set of loop-nests will maximize its cost function (degree of parallelism, memory placement, memory access, etc.). It was shown that algorithms that are capable of finding the optimum solution, like the one proposed in [12], have a high complexity (*NP-complete*) ; therefore many algorithms, although capable of finding the optimal solution, are unusable due to their high complexity. Heuristics can be used as an alternative [11], many algorithms have been developed and still under research.

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 [4]. 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, the data dependencies between the arrays being represented by the graph of tasks.

Array-OL is just a modeling language ; it contains no details on the execution model. For this paper we will discuss only the execution model based on loop-nests. It is important because this is the starting point of the Array-OL transformations, which resemble a lot with the loop transformations.

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.

As we will see, Array-OL has some advantages that make it an excellent candidate for loop-transformations-like optimization algorithms.

We will start with a short introduction to loop transformation topic, followed with the presentation of Array-OL, insisting on the transformations and the resemblance with loop transformations. We will continue with a comparative analysis of the two and we will end with perspectives and conclusions.

2 Code optimization by loop transformations

Most transformations for optimizing programs for uni-processors reduce the number of instructions executed by the program using transformations based on the analysis of scalar quantities and data-flow techniques. In contrast, optimizations for high-performance superscalar, vector, and parallel processors maximize parallelism and memory locality with transformations that rely on tracking the properties of arrays using loop dependence analysis.

2.1 Loop transformations

An important early system level technique, the loop transformation technique, is aiming at improving the data access regularity and locality and removing the system-level buffers of the application codes. 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. This in turn reduces the memory size requirement. The improvement of data access regularity can also increase the degree of parallelism of the application.

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. This transformation could be used to increase the data locality or as a support for other transformations.
- **Loop merging (or fusion)** that groups several loops in a unique one. The transformation is used for eliminating intermediate arrays and in this way reduce the memory size.
- **Loop splitting** that realizes the reverse of merging. Loop splitting attempts to simplify a loop or eliminate dependencies by breaking it into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. It can also be used to separate the parallel instructions from the rest.

Local transformations explore more in depth the way loops are organized internally. They can modify memory locality and space requirement :

- **Loop tiling or partitioning** increases the nesting level of a loop. The effect is that the iteration space is cut into blocks that are processed sequentially. The partitioning of loop iteration space leads to partitioning of large array into smaller blocks, thus fitting accessed array elements into cache size, enhancing cache reuse and eliminating cache size requirements.
- **Loop unrolling** decreases the number of iterations by describing several times the same instruction in the loop body in order to decrease the number of times a loop condition is tested and the number of jumps, which hurt performance by impairing the instruction pipeline.
- **Loop pipelining or alignment or folding** shifts some instructions from one to several iterations within the loop body. This is used to increase data locality and it can also be used as a support for other transformations.
- **Loop collapsing** is the reverse of tiling. It can be used together with the tiling as a tool for manipulating the concept of data block in the applications, essential for data locality and parallelism.

These transformations usually are combined in order to achieve best performances. The choosing of the ones that are applied and their order is extremely important and may depend on the final goal of the transformation. As an observation, these are just some of the existing loop transformations ; the most common we could say.

2.2 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.

The first step is the most difficult and because analysis is expensive, engineering issues often constrain the optimization strategies that the compiler can perform. As processor architecture becomes more and more complex, the number of dimensions in which optimization is possible increases and this makes the decision process extremely complicated.

An major aspect that must be taken into account is the correctness of a transformation. We must make sure that by applying a transformation (or a series of transformations) that the functionality of the program is not affected. The restraints are usually represented by dependencies (data or control). This is the complicated part with optimization techniques. Implementing the transformations is the easy part, the complex part remains the identification of correct transformations (that do not change the data dependencies) and then choose the optimum chain of transformations for achieving our goal.

A perfect optimization algorithm like the one proposed by McKinley and Kennedy for maximal reuse by loop fusion it has been proven to be extremely costly (complexity, time and resources) [12] – *NP-complete* problem – and this led to the extensive use of heuristics ; most of the optimizing compilers embody a set of heuristic decisions as to transformation orderings likely to work best for the target machine(s). Optimization can take place in distinct phases of program compilation, there is no definitive organization, different architectures dictate different designs, and opinions differ on which is the best order. Such a design is presented as an example in [2] to give an idea of how the transformations fit together.

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. This is the simple case, more complex compilers have more chains of transformations from where to choose, according to application characteristics.

2.3 Functionality

For a better understanding the usage of loop transformations, an example might be more appropriate. The classic example below presents how the *Loop Fusion* works. Here an array is written in a loop and is read in another. Between these loops the arrays must be kept in memory.

```

1: for  $i = 1$  to  $n$  do
2:    $A[i] = expr1$ 
3: end for
4: for  $i = 1$  to  $n$  do
5:    $expr2 = f(A[i])$ 
6: end for

```

The *Loop Fusion* technique practically merges several loops into one. The result of this merging is seen below.

```

1: for  $i = 1$  to  $n$  do
2:    $A[i] = expr1$ 
3:    $expr2 = f(A[i])$ 
4: end for

```

The merging had no impact on the memory consumption. This is because the *Loop Fusion* does not work alone. It is combined with other transformations, in this example the *Scalar Replacement*. This technique can remove entire arrays from memory, by replacing them with scalars, like below. This replacement is correct only if the values of array **A** produced in the first loop are not used elsewhere.

```

1: for  $i = 1$  to  $n$  do
2:    $a = expr1$ 
3:    $expr2 = f(a)$ 
4: end for

```

Dependencies can be more complicated, thus not allowing the removal of an entire array. In this case other techniques can be applied, like *Intra-array storage order optimization*, which can calculate an address reference window and then one may choose to “fold” the arrays. How it works is clearer in the example presented below. The following code

```

1: for  $i = 1$  to  $n$  do
2:    $A[i] = expr1$ 
3: end for
4: for  $i = 1$  to  $n$  do
5:    $expr2 = f(A[i - 1], A[i])$ 
6: end for

```

is transformed into

```

1: for  $i = 1$  to  $n$  do
2:    $a[i\%2] = expr1$ 
3:    $expr2 = f(a[i\%2 - 1], a[i\%2])$ 
4: end for

```

by loop fusion with intra-array storage order optimization.

As clearly seen in the previous examples, *Loop Fusion* can be a powerful tool if combined with other transformations in order to reduce the memory consumption. The examples above are extremely simples ; in real applications the things are usually much more complicated but the homogenous array manipulations in data-flow applications like multimedia applications make this technique extremely efficient for this type of applications.

The Loop optimization problem for a section of the source code can

To find an optimal chain of loop transformations that applied will maximize the cost function associated and in the same time will not affect its correctness.

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 [6], Polyhedral Abstraction [8], graph theory algorithms or Integer Linear Programming [7]. All these approaches have in common, besides the fact that can obtain an almost-optimum solution, their high complexity.

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. Indeed, most of the elementary functions are sums, dot products or Fourier transforms, which are well known and often available as library functions. The difficulty and the variety of these intensive signal processing applications come from the way these elementary functions access their input and output data as parts of multidimensional arrays. The complex access patterns lead to difficulties to schedule these applications efficiently on parallel and distributed execution platforms. 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, defining the minimal order of the tasks.
- 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. The data dependencies expressed at a level (between arrays) are approximations of the precise dependencies of the sub-levels (between patterns).
- 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. No data element is ever written twice. It can be read several times, though. Array-OL can be considered as a first order functional language.
- The spatial and temporal dimensions are treated equally in the arrays. In particular, time is expanded as a dimension (or several) of the arrays.
- The arrays are seen as tori. Indeed, some spatial dimensions may represent some physical tori (think about some hydrophones around a submarine) and some frequency domains obtained by FFTs are toroidal.

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.

As a simplifying hypothesis, the application domain of Array-OL is restricted. No complex control is expressible and the control is independent of the value of the data. This is realistic in the given application domain, which is mainly data flow. Some efforts to couple control flows and data flows expressed in Array-OL have been done in [14] but are outside the scope of this paper.

The usual model for dependence based algorithm description is the dependence graph where nodes represent statements and edges dependencies. Various flavors of these graphs have been defined. The expanded dependence graphs represent the task parallelism available in the application. In order to represent complex applications, a common extension of these graphs is the hierarchy. A node can itself be a graph. 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. As said above, the Array-OL model is single assignment. Time is thus represented as one (or several) dimension of the data arrays. For example, an array representing a video is three-dimensional of shape (width of frame, height of frame, frame number).

There is only one limitation on the dimensions : there must be at most one infinite dimension by array. Most of the time, this infinite dimension is used to represent the time, so having only one is quite sufficient.

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 (1920×1080 at the input and 720×640 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.

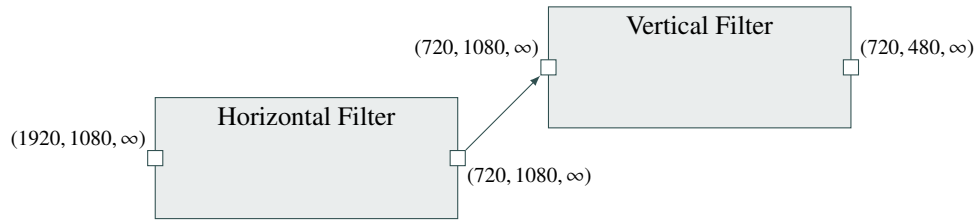


FIG. 1 – Downscaler application – task dependence

Each execution of a task reads one full array on its inputs and writes the full output arrays. It's not possible to read more than one array per port to write one. *The graph is a dependence graph, not a data flow graph.*

The compound description expresses only the task parallelism, not the data parallelism of the application.

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 parallel¹. 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.

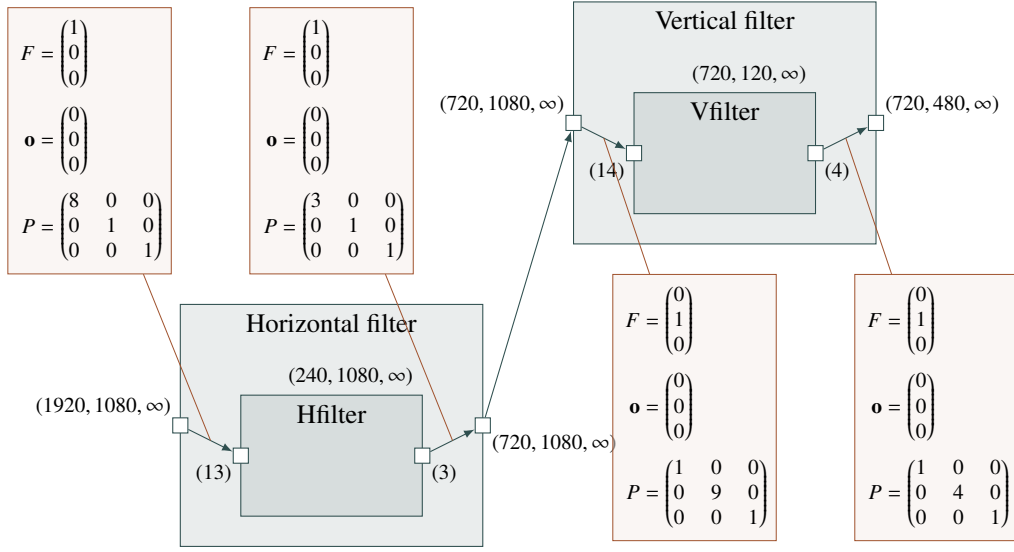
As these sub-arrays are conform, they are called *patterns* when considered as the input arrays of the repeated task and *tiles* when considered as a set of elements of the arrays of the repetition task. In order to give all the information needed to create these patterns, a *tiler* is associated to 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 (whose meaning will be explained shortly) :

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

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

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*.



The downscaler is decomposed into two successive filters, one that scales an image on the horizontal and the other on the vertical. Each of the filter has a repetitive functionality that is described by the repetition of an elementary component. This repetition 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.

To illustrate this link between the inputs and the outputs, we show below several repetitions of the horizontal filter repetition (*Figure 3*). In order to simplify the figure and as the treatment is made frame by frame, only the first two dimensions are represented². The sizes of the arrays have also been reduced by a factor of 60 in each dimension for readability reasons.

3.4 Enforcing determinism by construction

The basic design decision that enforces determinism is the fact that Array-OL only expresses data dependencies. To ease the manipulation of the values, the formalism is single assignment. Thus each array element has to be written only once. To simplify the verification of this, the constraint that each task produces all the elements of its output arrays is built into the model. An array has to be fully produced even if some elements are not read by other tasks. Enforcing this rule for all the tasks at all the levels of the hierarchy also allows to compose tasks easily.

A direct consequence of this full production rule is that a repetition has to tile exactly its output arrays. In other words each element of an output array has to belong to exactly one tile. Verifying this can be done by using polyhedra computations. But this is not the point here. The user has to respect the rule. The compiler can ensure it or assume it.

3.5 Projection onto an execution model

The Array-OL language expresses the minimal order of execution that leads to the correct computation. This is a design intension and lots of decisions can and have to be taken when mapping an Array-OL specification onto an execution platform : how to map the various repetition dimensions to time and space, how to place the arrays in memory, how to schedule parallel tasks on the same processing element, how to schedule the communications between the processing elements ?

3.5.1 Space-time mapping

One of the basic questions one has to answer is : What dimensions of a repetition should be mapped to different processors or to a sequence of steps ? To be able to answer this question, one has to look at the environment with which the Array-OL specification interacts. If a dimension of an array is produced sequentially, it has to be projected to time, at least partially. Some of the inputs could be buffered and treated in parallel. On the contrary, if a dimension is produced in parallel (e.g. by different sensors), it is natural to map it to different processors. But one can also group some repetitions on a smaller number of processors and execute these groups sequentially. The decision is thus also influenced by the available hardware platform.

²Indeed, the third dimension of the input and output arrays is infinite, the third dimension of the repetition space is also infinite, the patterns do not cross this dimension and the only paving vector having a non null third element is $\begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}$ along the infinite repetition space dimension.

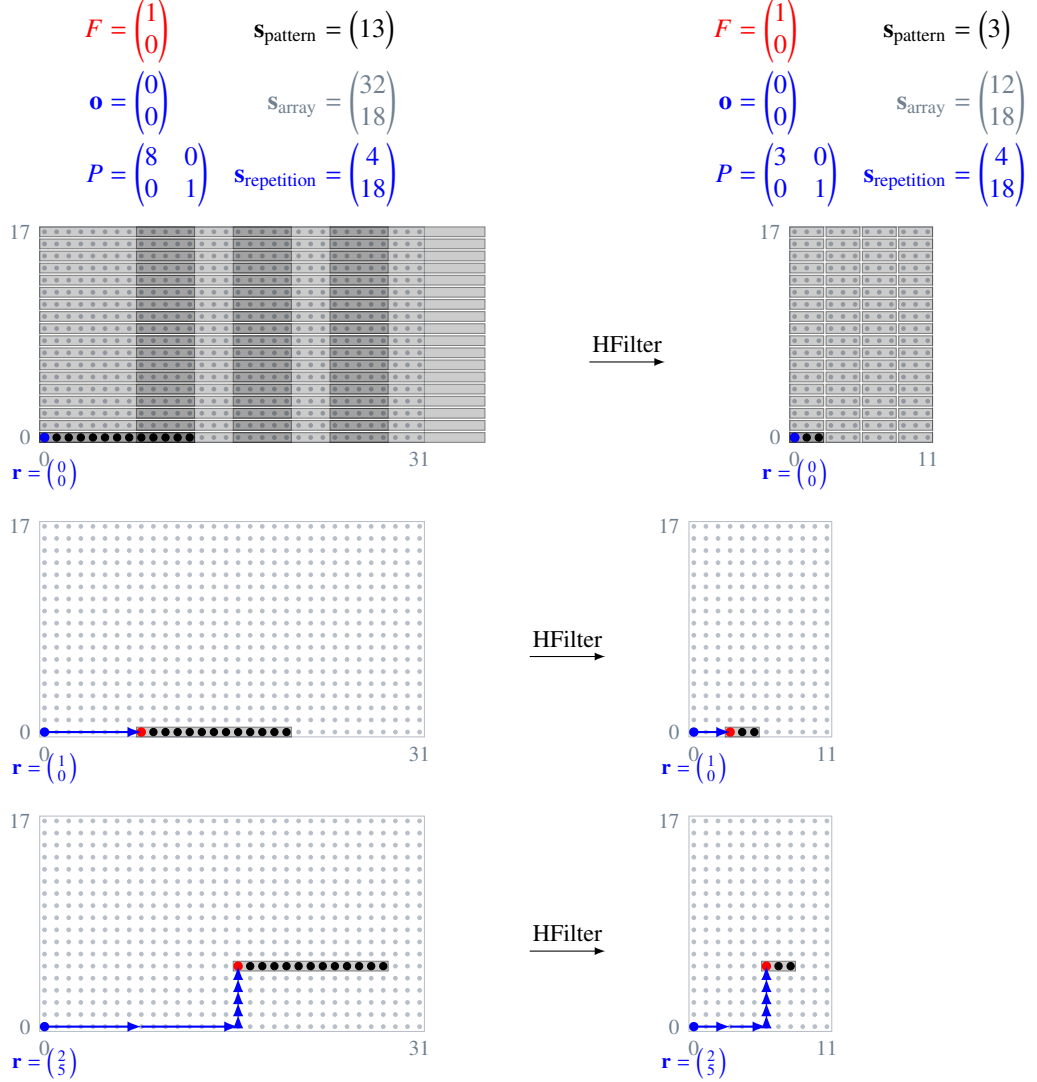


FIG. 3 – Repetition example

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 [9, 10]. 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).

3.6 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 [17] and Philippe Dumont [5].

3.6.1 Necessity

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. This is done by creating a special kind of hierarchical structure in the applications, using the Array-OL transformations. By creating such a 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.

The question now is how we can use the techniques from loop optimizations in the context of Array-OL. An Array-OL transformation called *Fusion* could provide an answer. This transformation practically merges two tasks by creating a hierarchy. When applying such a transformation, the intermediate arrays descend one level of hierarchy but they usually lose the infinite dimension. This resembles very much with the use of loop fusion combined with scalar replacement or intra-array storage optimization presented earlier in the paper.

We can imagine a series of Array-OL transformation that could transform an application into a special hierarchical application, where all the infinite array dimension remains on the top level. In this way we can have a pseudo separation between the spatial and temporal dimensions and we can easier manipulate the application.

Most of the transformations were formalized by Julien Soula [17] and Philippe Dumont [5] using the ODT formalism and they are implemented in a development tool called *ODTTransfos* [15].

3.6.2 ODT formalism

This formalism was necessary because we must assure that the application’s functionality is not modified when a transformation is applied. The restrictions are determined by the data dependencies and the correct manipulation of these dependencies is the real challenge when designing the Array-OL transformations.

The ODT formalism³ was proposed by Alain Demeure [3] to represent the dependencies between the operand and the resultant part of an Array-OL task. It is based on relations that define the connection between two \mathbb{Z}^n spaces. We can link these operators and these spaces with the help of the usual relation composition law. To translate in the ODT formalism an Array-OL task we must link the array elements to their coordinates and apply the operators on these coordinates. We obtain in this way a sequence of

³“Opérateurs de Distribution de Tableau” in French

operators that express the dependencies and on which we can apply a certain number of computations.

The ODT formalism is much too complex to be presented in detail here. What is important is how this formalism is used in the context of our transformations. Each task described in Array-OL can be represented using the ODT representation consisting in a series of operators that define the data dependencies. The representation has the same form for any task, the difference is the operands on which these operators are applied, and which are combinations of the matrixes and vectors that describe the task-array associations (pattern, fitting, paving, origin). Also, each ODT representation of a task can be transformed to a normal Array-OL form.

Typically, when applying a transformation, all the tasks involved are transformed to the ODT representation, then ODT transformations are applied to them and finally the resulted tasks are re-transformed to Array-OL representation and later inserted at their appropriate place in the application.

Next we will present a list with the Array-OL transformations but we will not insist on the details of implementation, just basic characteristics, functionality and, for a better understanding, how they work on the downscaler application. More details can be found in the PhD thesis of Julien Soula [17] and Philippe Dumont [5]

3.6.3 Fusion

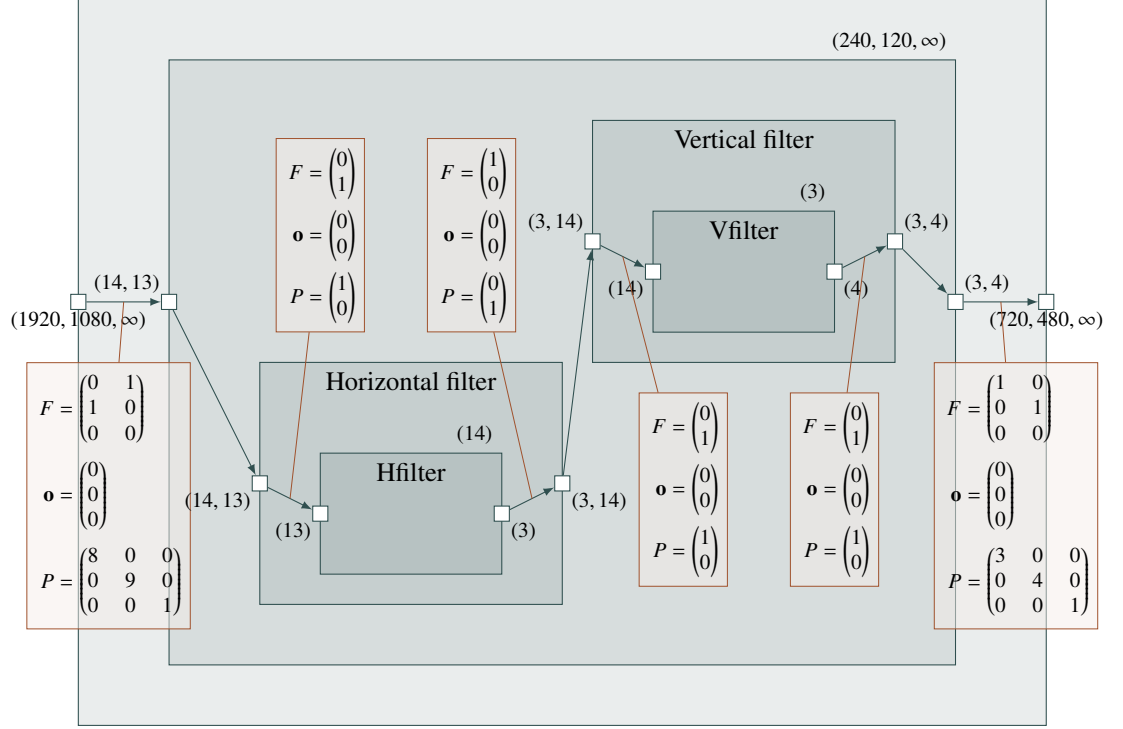
This transformation basically takes two components that have at least one common array (a common array means that the first component produces an array that is consumed by the second component) and these two components are merged into a single compound component that contains the previous two. This means that the result of fusion is the creation of a hierarchy level. 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 important 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.

The fusion is the most complicated transformation because of all the different scenarios that can appear in the design of an application for the two components intended for fusion (more common arrays, more exterior arrays, intermediate but not common arrays, etc.) ; but in the same time it is the most useful.

In *Figure 4* 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 filters but these filters consume different arrays, the infinite dimension remaining at the top level.

We can use the fusion transformation only on two components that have at least one common array. This reduces the domain of applicability ; for example if we want to fuse three or more tasks we have to apply more times the transformation, this leading to complications, especially the “abyssal hierarchies” phenomenon that we will discuss later in the paper.



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. The fusion chose to take each time the smallest possible pattern to pass to the filters but assuring that it will not block the application (in our case patterns of (14,13) pixels formed by sliding the horizontal and vertical dimension by 8, respectively 9 pixels). This structure allows an pipeline execution of the two filters and also for each filter a parallel execution of it's repetitions. As for the top level components, it's execution in a sequential or parallel manner depends only on the data availability.

FIG. 4 – Downscaler after fusion transformation

3.6.4 Change paving

The **Change paving** transformation is the second transformation and it was introduced with the purpose of reducing the re-calculations introduced in some cases by the fusion.

This problem can appear in some special cases, after the fusion. This phenomenon can be seen in the first component of the created compound component, in the case where the 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 array.

The re-calculations phenomenon can be seen on our downscaler example. As seen on the *Figure 2* and *Figure 4*, before the fusion we have the first *HFilter* component executed over the vertical dimension 1080 times, while after the transformations it must

be executed $120 \times 14 = 1440$ times⁴. The extra executions are caused by the fact that the patterns are taken in an overlapping way (*Figure 3*). Practically, different repetitions of the top level component will determine the lower level filters to compute same values (same inputs on the original arrays – before the fusion).

A complete solution to this phenomenon, without eliminating the previous fusion is not always possible. What we can do is reducing the amount of re-calculation by extending the pattern of the compound component so it will include more. This will reduce the re-calculations ; in the extreme case if we extend to the maximum the pattern in some of the paving vectors on which the re-calculation appears we may even eliminate the re-calculation phenomenon. Still, this is not possible in the case where the re-calculation is present on the infinite dimension without eliminating the principal role of the fusion, that of reducing the intermediate array sizes. A special case of this reduction is to handle the infinite dimension only at the top level of the hierarchy.

As result, we have two ways (two transformations) that act on extending the pattern of the compound component. The first is much simpler, it can be applied in any case, but as we will see, it does not reduce the re-calculations. The second one is more complex, it is designed exactly in the purpose of reducing the re-calculations, it does that but it can be applied only when re-calculations appear.

Change paving by adding dimensions The first transformation, as its name indicates, extends the pattern by the use of an extra dimension, that will have the size of the number of previous patterns that will be included in this new pattern. Practically, we reduce the number of times the sub-components are executed but without doing anything on the way the patterns are calculating, so without reducing the re-calculations.

This transformation can be used in the case where one wants to change the granularity of the application by distributing repetitions through the hierarchy levels.

It can be seen how this works on the downscaler in *Figure 5* containing the application after applying a change paving by adding dimensions to the post-fusion structure. It increased the pattern on the horizontal by a factor of 240, the maximum possible, and this leads to the possibility of removing the horizontal paving vectors from the compound component.

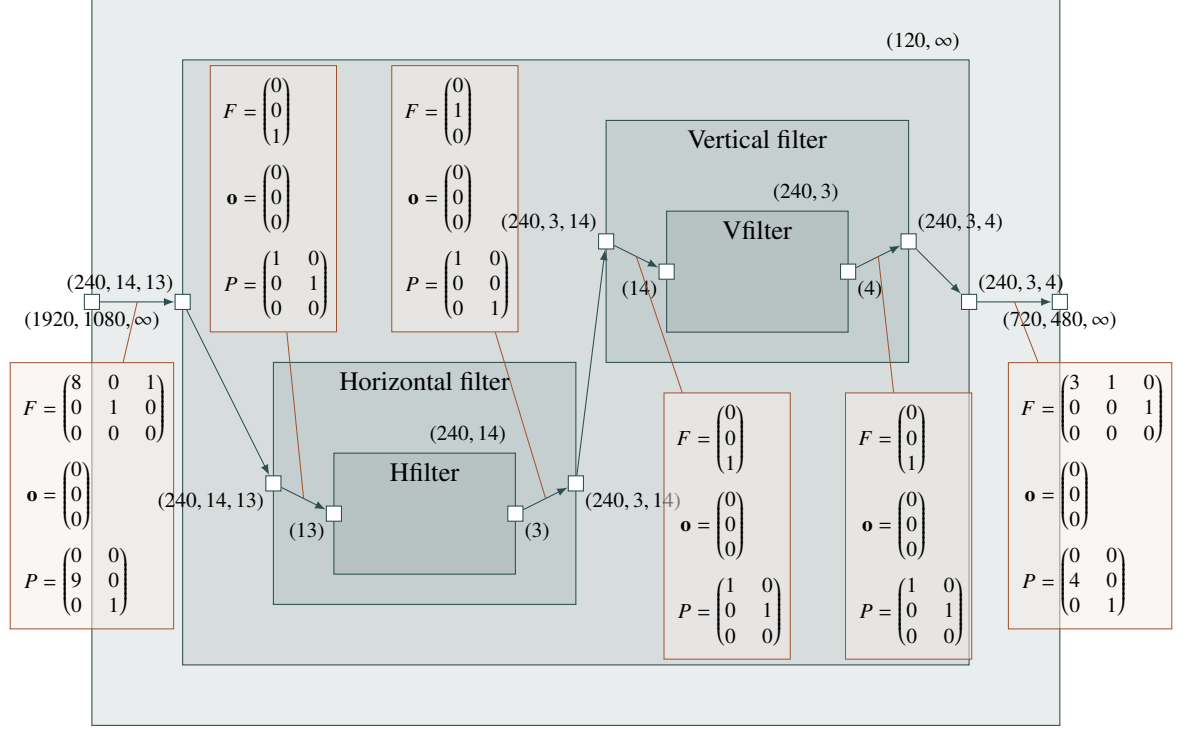
The major problem with this transformation is the “introduction of extra dimensions”. The use of this transformation leads to the change of array dimensions in the application. For example, we might start with a three dimensional array and after some transformations we get a four or five dimensional array that has some inconveniences ; it changes the original arrays and also as the number of dimension grows it gets harder and harder to be manipulated by the user.

Change paving by linear growth The second change paving transformation is called linear growth and, as we mentioned earlier, is designed specially to reduce the re-calculations and so it can be applied only on tasks that have a special structure, the one obtained after a fusion.

What this transformation does different than the other is to calculate a surrounding pattern and so the transformation can be used to reduce the re-calculations.

The transformation has three steps. First, because the transformation is proven to be correct only in the case of a re-calculate, we must find out if such a phenomenon appears. For this an algorithm that returns a list of all correct transformations exists. For each possible transformation we can calculate using a formula the degree in which the

⁴by multiplying the repetitions from the two hierarchy levels



The add dimension transformation on the horizontal paving vector determined the splitting of the repetition domain of the horizontal paving vector on the top level component, one part of the repetition domain passing to the lower hierarchical level, by enlarging the repetition domains of the components on this level with the *change paving factor* and causing the introduction of another dimension to the patterns at this level). The difference is that the top component now will execute less times but it will take more consecutive patterns at one time. In our case of maximal change paving this lead to the possibility of removing the now redundant paving vector from the top level. The result is that now the second level will consume 240 patterns of (14,13) and generate at the end 240 patterns of (3,4).

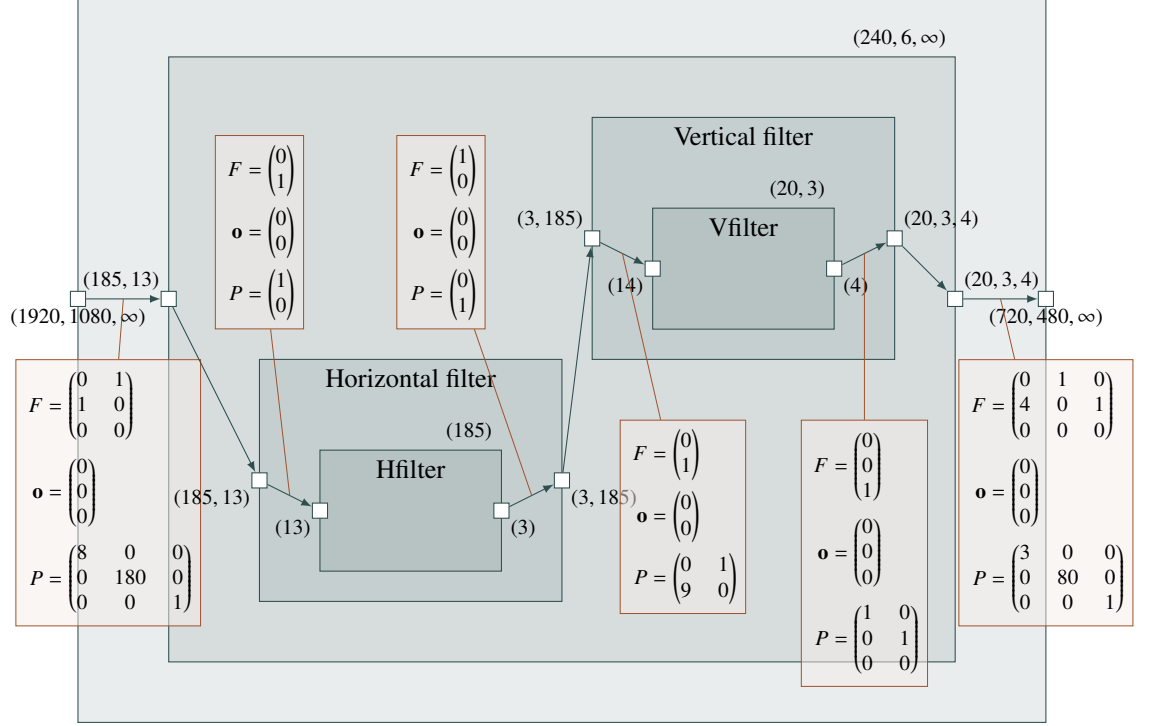
FIG. 5 – Downscaler after maximal add dimension on horizontal paving vector

re-calculation is reduced. In this way one may choose the appropriate transformation. After choosing the transformation, if such exists, we can choose to apply it.

The limitation of this transformation is that it can be applied only when re-calculations appear. We proposed its extension to the case where we don't have re-calculations but the patterns are "glued" together but this is also limited by the ODT formalism.

The result of such a transformation can be seen in Figure 6.

Transformation to post-fusion structure Another problem with the previous transformation is that it cannot be used only after a fusion. In order to fix this inconvenient, with the use of an Array-OL concept named *shortcut*, we introduce another transfor-



The difference between linear growth transformation and the add dimension presented in Figure 5 is that, although this one also determines the filters to consume more patterns at one time, the change paving factor does not appear as another dimension but rather as an extension of the previous pattern (and repetition domain) so it will contain all these patterns. In our case, due to the fact that the vertical paving vector slides on the vertical dimension with 9 elements every time, a linear growth with a factor of 20 on the vertical paving vector will determine the pattern to grow on this dimension, to the size of (185,13) ($185 = (20 - 1) \times 9 + 14$). We can notice in this case that because it is not a maximum change paving transformation, just a part of the repetition space associated to the vertical paving vector passes to the lower level of hierarchy.

FIG. 6 – Downscaler after linear growth on vertical dimension by the factor of 20

information that takes as input an ordinary compound component and transforms it to the special structure of a post-fusion component (if possible⁵), on which we can apply now the previous transformation (linear growth).

This new transformation is based on the shortcut, which was designed for expressing in a compound component the link between the coordinates of a lower level sub-component's elements and the top-level component's.

Using this concept we can practically compose two or more tilers and in this way we can eliminate the need for multiple tiler computations when dealing with hierarchical

⁵the transformation to post-fusion structure can be used only on a compound component

components. This is extremely important when passing to an execution model, the tiler computations being one of the main parts of an Array-OL application. In the case of our transformations, we will use the shortcut to combine two tilers (a tiler from the top level component and the tiler from the lower level hierarchy that connects to the patterns produced by the first) and the resulting shortcut tiler will be placed at the top level, while for the lower level we practically do not need any pattern computations, but in order to remain in the Array-OL domain we must introduce a tiler that has just the role a copying patterns between hierarchy levels (this tiler will have the paving and the fitting matrixes pseudo-identity – not identity because they are not square). For better understanding we can see the downscaler after the fusion in *Figure 4* where the structure is one similar to the one after applying the shortcut. As we can see, all the patterns are computed by the top-level tilers, while the tilers from the second level just copy patterns.

The shortcut can be regarded as a transformation but it does not work on an entire component ; just on half of a hierarchical structure. What we can do is take the two shortcuts and combine them in order to obtain the transformation of the entire structure.

The first shortcut will be constituted from an operand tiler of the compound component and the operand tiler of the first sub-component. The second shortcut will contain the resultant tiler of the compound component and the resultant tiler of the last sub-component of the hierarchy. Now, applying the transformation to these two shortcuts we will achieve the actual transformation that we were searching for.

An important plus of this transformation is that by the fact that it modifies only the first and the last sub-components in the hierarchy, it can be applied to any compound component, regardless of the number of sub-components.

3.6.5 Tiling

The **Tiling** was designed in order to allow the introduction of *granularity degree* concept in an application.

The concept of *granularity degree*, introduced in the context of control, allows to delimitate the different *execution cycles* but also it allows the introduction of data flow semantics in the description of Array-OL applications.

More details on this topic can be found in the PhD thesis of Ouassila Labbani [13] (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 subset is defined using the same Array-OL concept of origin, fitting, paving, etc. It can be seen as the introduction of a hierarchy level in the application.

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.

The transformation resembles the fusion transformation but it is applied to only one component and the block size is specified by the user.

3.6.6 Collapse

The transformation named collapse is important for refactoring an Array-OL application.

A problem that we mentioned earlier with the fusion transformation is that it can work only on two tasks at a time. In we want for example to fuse three or more tasks we must apply the fusion more times and this will lead to the creation of what we call

“abyssal hierarchies” that are spread on more layers, which are not easy to manipulate not from the application’s point of view and not from the user’s perspective.

The solution is the collapse transformation that practically is represented by a series of maximum change paving transformations that practically have the role of extending the patterns of the compound component so it contains all the original patterns and in this way this component is useless and it can be eliminated by replacing it with its sub-components, which will “climb” a level in the hierarchy.

In this way, by applying a certain number of transformations, fusion and collapse mainly, we can change the structure of an Array-OL application without modifying its functionality. One can use these transformations to reshape the application to handle various constraints (timing, hardware platform mapping, memory optimization).

3.7 Downscaler

To show how we can use the Array-OL transformations we have chosen to illustrate them on the downscaler application. The intermediate array between the two filters presented in *Figure 2* contains an infinite dimension and this represents a blocking point for execution. In order to be able to project this application onto an execution platform, one possibility is to make a flow of the time dimension and to allow pipelining of the space repetitions. A way to do that is to transform the application by using the fusion transformation to add a hierarchical level. The top level can then be transformed into a flow and the sub-level can be pipelined. The result will not be presented due to paper-size limitations. What is important is that after the transformation, a hierarchical level has been created that is repeated $(240, 120, \infty)$ times, with an input pattern a block of $(13, 14)$ elements. The intermediate array between the filters has been reduced to the minimal size that respects the dependencies. If the inserted level is executed sequentially and if the two filters are executed on different processors, the execution can be pipelined.

This form of the application takes into account internal constraints : how to chain the computations. Next we can now propose a new form of the downscaler application taking the environment constraint into account by extending the top-level patterns to include full rows. This is done by applying a maximal linear growth on the horizontal paving vector. The result is presented in *Figure 8*.

3.8 Array-OL transformations – implementation

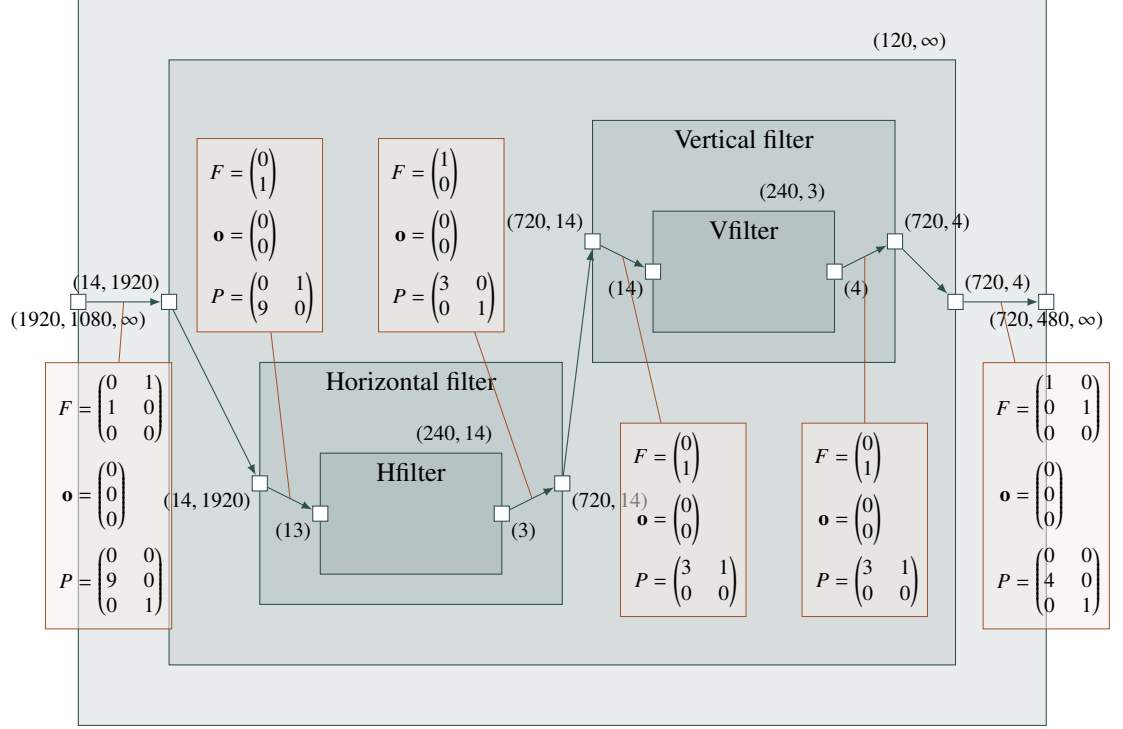
All the transformations presented are currently implemented in the *ODTTransfos* tool⁶. The implementation respects the ODT formalism ; meaning that it is structured on two levels, the Array-OL level and the ODT level. At the ODT level were implemented all the ODT operators and also the operations characteristic to the ODT representation of Array-OL. At this level the actual transformations are implemented. The Array-OL level works mostly as an interface between the Array-OL language and the ODT formalism.

This tool is also included in the Gaspard2 co-modelling environment [16] as a transformation module.

3.9 Array-OL transformations – extension

Possible extensions to these transformations are still in research. The extension of linear growth transformation is one of them. The limitations of its applicability is

⁶available at <http://www.lifl.fr/west/aoltools/>



The top-level repetition now works with tiles containing full rows of the images. Less parallelism is expressed at that level but as the images arrive in the system row by row, the buffering mechanism is simplified and the full parallelism is still available at the lowest level.

FIG. 7 – Downscaler model that respects the data flow constraints

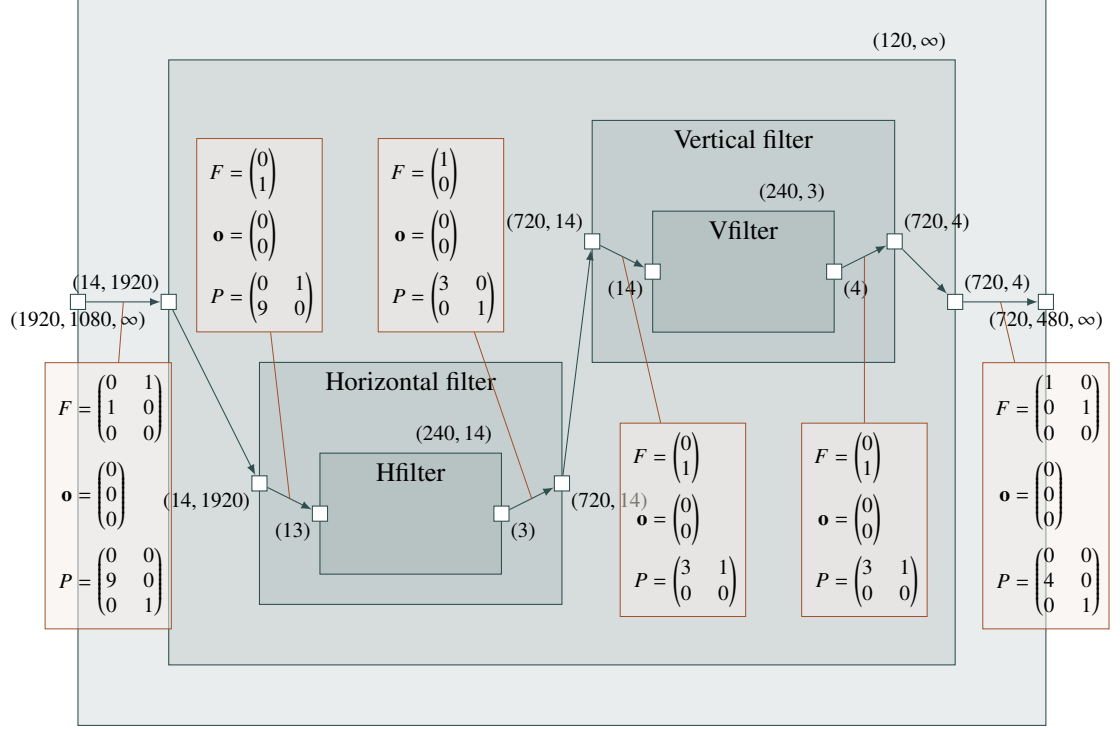
problematic, forcing the use of the change paving by add dimensions transformation which as we saw leads to the introduction of extra dimensions phenomenon.

Also, we could imagine more complex transformations using these basic transformations, like for example a transformation that could transform an entire application (or a part of an application) to the two-levels structure by using an enchainment of fusion and collapse transformations. More complex transformation can be also designed around the concept of granularity degree.

3.10 Conclusions on the Array-OL transformations

Probably the biggest problem with Array-OL is the impossibility of expressing constraints directly in our Array-OL specification model. There are three major types of possible constraints : internal computation chaining constraints, those of the environment and the ones introduced by the hardware's limitations.

For our downscaler application we have the constrain representing the concept of data flow ; in our case the data arrives as a pixel flow, row after row. These constraints are extremely important when passing to an execution model and mapping the application on an architecture. We can see this clearly in the case of the downscaler that it is not



The top-level repetition now works with tiles containing full rows of the images. Less parallelism is expressed at that level but as the images arrive in the system row by row, the buffering mechanism is simplified and the full parallelism is still available at the lowest level.

FIG. 8 – Downscaler model that respects the data flow constraints

enough to reach a two-level structure in order to be able to propose for example a pipelined execution model that respects these constraints. For now, the designer must verify that the constraints are respected. The solution in the case of downscaler is to extend the top-level patterns after the fusion so it would include the entire row. This can be done using a change paving transformation on the horizontal paving vector (Figure 8).

The introduction of concepts for expressing constraints is essential when passing to an execution model. The most important concept is the one of **order** which is currently under research with the introduction of “clock” concepts.

The introduction of new concepts in Array-OL would give new directions for the evolution of Array-OL transformations, execution models and possibly optimization techniques.

4 Array-OL vs. Loop transformations

An important early system level technique, the loop transformation technique, is aiming at improving the data access regularity and locality and removing the system-

level buffers of the application codes. It is not the goal of this paper to present the extensively known loop transformations like the loop *fusion*, *splitting*, *unrolling*, *shifting*, *tiling* and so on. More details on the topic can be found in other papers like [2] or [7]. What is important to this paper is that these transformations are most efficient on code that contains extremely regular data treatment (perfectly-nested loops). This is exactly the domain of Array-OL and as we have shown earlier in the paper, translating to an execution model based on loops is trivial. Also the evident resemblances between the two types of transformations (the Array-OL transformations were designed around the loop transformations, reason why, for each transformation its name was taken from its correspondent) lead us to try identify the connections between the two domains and furthermore to investigate the domain of optimization techniques based on loop transformations and its possible connections with optimizations in 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 must also denote that the Array-OL transformations cannot handle for the moment the *inter repetition dependencies*, a concept from Array-OL used for expressing dependencies between repetitions of the same component.

The complexity of optimization algorithms determined the necessity to introduce ways of representing the problem (constraints, transformations, cost function) using a more efficient formalism and which could facilitate the manipulation of concepts like correctness, data dependencies, cost function. Some approached the problem using Linear Algebra [6], Polyhedral Abstraction [8], graph theory algorithms or Integer Linear Programming [7]. All these approaches have in common, besides the fact that can obtain an almost-optimum solution, their high complexity. The introduction of a formalism is extremely important for the decision part of the optimization. Correct and complex optimization algorithms need to be designed around such formalisms. Array-OL has the advantage that it is built around such a formalism and also all the transformations are specified and their correctness proven by the use of the ODT formalism [3] based on Linear Algebra.

We will not compare separately each pair of transformations, as mentioned, each Array-OL transformations resembles in functionality with it's 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. 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 special 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 constrains, like the case of our example.

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 constrains. None the less, transformations change the structure of an application and this implies changes to the parallelism.

Considering the resemblance between the two types of transformations, what we can try to do is to take results from loop optimization techniques, that we might say that they have reached a maturity level due to the extended research done on the topic, and use these results in Array-OL context. What we must point out is that algorithms that can give the optimum solution for memory optimization are not practical, due to complexity issues. Most of the times heuristics that are proven to give good results in most of the cases 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 two-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.

Acknowledgements

The authors would like to thank Julien Soula, Philippe Dumont and Ouassila Labbani for their work on various aspects of the Array-OL transformations.

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.

Table des matières

1	Introduction	3
2	Code optimization by loop transformations	4
2.1	Loop transformations	4
2.2	Loop optimization techniques	5
2.3	Functionality	6
3	Array-OL model of specification	7
3.1	Principles	7
3.2	Tasks parallelism	9
3.3	Data parallelism	9
3.4	Enforcing determinism by construction	11
3.5	Projection onto an execution model	11
3.5.1	Space-time mapping	11
3.6	Array-OL transformations	13
3.6.1	Necessity	13
3.6.2	ODT formalism	13
3.6.3	Fusion	14
3.6.4	Change paving	15
3.6.5	Tiling	19
3.6.6	Collapse	19
3.7	Downscaler	20
3.8	Array-OL transformations – implementation	20
3.9	Array-OL transformations – extension	20
3.10	Conclusions on the Array-OL transformations	21
4	Array-OL vs. Loop transformations	22
5	Conclusions	24

Références

- [1] Abdelkader Amar, Pierre Boulet, and Philippe Dumont. 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] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4) :345–420, 1994.
- [3] Alain Demeure. Les ODT : propositions de notation pour décrire des opérateurs de distribution de tableaux. Technical report, Thomson Marconi Sonar, Sophia-Antipolis, France, 1998.
- [4] Alain Demeure, Anne Lafarge, Emmanuel Boutillon, Didier Rozzonelli, Jean-Claude Dufourd, and Jean-Louis Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Gretsi*, Juan-Les-Pins, France, September 1995.
- [5] Philippe Dumont. "Spécification Multidimensionnelle pour le traitement du signal systématique". Phd thesis, Laboratoire d'informatique fondamentale de Lille, 2005.
- [6] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1) :23–53, 1991.
- [7] Antoine Fraboulet. *Optimisation de la mémoire et de la consommation des systèmes multimédia embarqué*. Phd thesis, LIP, November 2001.
- [8] Sylvain Girbal. *Optimisation d'applications - Composition de transformations de programme : modèle et outils*. PhD thesis, University Paris 11, Orsay, France, September 2005.
- [9] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74 : Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland, August 1974.
- [10] Gilles Kahn and David B. MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing 77 : Proceedings of the IFIP Congress 77*, pages 993–998. North-Holland, 1977.
- [11] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. Improving locality using loop and data transformations in an integrated framework. In *MICRO 31 : Proceedings of the 31st annual ACM/IEEE international symposium on Microarchitecture*, pages 285–297, Los Alamitos, CA, USA, 1998. IEEE Computer Society Press.
- [12] Ken Kennedy and Kathryn S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *1993 Workshop on Languages and Compilers for Parallel Computing*, number 768, pages 301–320, Portland, Ore., 1993. Berlin : Springer Verlag.
- [13] Ouassila Labbani. "Modélisation à haut niveau du contrôle dans des applications de traitement systématique à parallélisme massif". Phd thesis, Laboratoire d'informatique fondamentale de Lille, 2006.
- [14] Ouassila Labbani, Jean-Luc Dekeyser, Pierre Boulet, and Éric Rutten. Introducing control in the gaspard2 data-parallel metamodel : Synchronous approach. *International Workshop MARTES : Modeling and Analysis of Real-Time and*

-
- Embedded Systems (in conjunction with 8th International Conference on Model Driven Engineering Languages and Systems, MoDELS/UML 2005)*, October 2005.
- [15] USTL Laboratoire d'informatique fondamentale de Lille. Array-ol tools. <http://www.lifl.fr/west/aoltools/>, 2006.
 - [16] USTL Laboratoire d'informatique fondamentale de Lille. Gaspard home page. <http://www.lifl.fr/west/gaspard/>, 2006.
 - [17] Julien Soula. *Principe de Compilation d'un Langage de Traitement de Signal*. Phd thesis, Laboratoire d'informatique fondamentale de Lille, December 2001.



Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex

Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
<http://www.inria.fr>
ISSN 0249-6399