Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications

Calin Glitia, Pierre Boulet, Eric Lenormand, Michel Barreteau

▶ **To cite this version:**

HAL Id: inria-00605069

https://hal.inria.fr/inria-00605069

Submitted on 30 Jun 2011

Repetitive Model Refactoring Strategies
for the Design Space Exploration
of Intensive Signal Processing Applications

Calin Glitia[a,b,c,d], Pierre Boulet[b,c,a,d], Eric Lenormand[e], Michel Barreteau[e]

[a]*INRIA Lille-Nord-Europe, F-59650 Villeneuve d'Ascq, France*
[b]*Univ Lille 1, LIFL, F-59655 Villeneuve d'Ascq, France*
[c]*CNRS, UMR8022, F-59655 Villeneuve d'Ascq, France*
[d]*Univ Lille Nord de France, F-59650 Villeneuve d'Ascq, France*
[e]*THALES Research & Technology, Embedded Systems Lab, F-91767 Palaiseau, France*

Abstract

The efficient design of computation intensive multidimensional signal processing application requires to deal with three kinds of constraints: those implied by the data dependencies, the non functional requirements (real-time, power consumption) and resources availability of the execution platform. We propose here a strategy to use a refactoring tool dedicated to this kind of applications to help explore the design space. This strategy is illustrated on an industrial radar application modeled using the Modeling and Analysis of Real-time and Embedded systems (MARTE) UML profile. It allows to find good trade-offs in the usage of storage and computation resources and in the parallelism (both task and data parallelism) exploitation.

Key words: specification language, parallelism, optimizations, high-level code transformations, design space exploration, refactoring strategy

1. Introduction

Computation intensive multidimensional signal processing applications are predominant in several application domains such as image and video processing or detection systems (radar, sonar). By multidimensional, we mean that they manipulate primarily multidimensional data structures such as arrays. An important part of the computations is represented by elementary data treatments (filters, FFTs, average values computations, etc.) that can be found in libraries, while the difficulty and the variety of this kind of applications comes mainly from the way these elementary functions access their input and output data as parts of multidimensional arrays.

The intensively repetitive structure expose the native logical parallelism available within such applications, while an important part of the problem of efficiently scheduling such applications refers to finding an optimized mapping

between the logical parallelism and the physical parallelism offered by the execution platforms.

The complex access patterns lead to difficulties to schedule these applications efficiently on parallel and distributed execution platforms such as multiprocessor systems-on-chip. Three kinds of scheduling constraints add up: data dependency constraints, environmental or non functional constraints (real-time, power consumption, etc.) and execution platform constraints (resource availability). A design exploration tool that would guaranty the respect of functional while allowing to improve on the others (usage of resources) would be very helpful.

The MARTE (Modeling and Analysis of Real-time and Embedded systems) UML profile [1, 2] is the emerging standard for modeling real-time and embedded systems. It is well suited to model intensive signal processing applications thanks to its Repetitive Structure Modeling extension of UML which is designed to provide ways of compact expression of systems having a repetitive structure or topology.

Gaspard2 (Graphical Array Specification for Parallel and Distributed Computing) [6, 7], an Integrated Development Environment dedicated to the visual co-design for embedded systems based on MARTE, by the extensive use of the RSM package of MARTE at specification level, allows to model multidimensional data accesses without compromising the usability of the specification and provides ways to statically schedule these applications on parallel hardware platforms.

Such a specification is conform with Array-OL (Array Oriented Language) model of computation (MoC), that defines, from a RSM specification, the whole data dependencies between the ensemble of data elements that can exist throughout an execution (time and space mix as dimensions of the data structures). This approach allows to separate the specification from the execution, leaving the user with more freedom at specification level to completely and correctly express the ensemble of data dependencies representing the functional constraints of an application.

For a RSM specification, these functional constraints remain unchanged regardless of the non functional and of the execution platform constraints. The search for an efficient scheduling for a RSM specification can be seen as an optimization problem that aims to maximize a predefined cost function by the projection in time (resource concurrency) and space (resource parallelism) of the multidimensional structures part of a RSM specification, while respecting the ensemble of constraints defined on the system.

The multidimensional structures express either data structures or parallel computations and, for the abstraction level of such descriptions, maximizing the amount of logical parallelism that is projected into physical parallelism while minimizing the memory size needed to store the data structures throughout the execution is a reasonably good cost function. Nonetheless, even such basic cost function defines two objectives that are concurrent (increase in parallel executions imply increase in memory size), and we are forced to talk about trade-offs between the usage of storage and computation resources and of strategies that allow us to find good trade-offs between different concurrent objectives. More

accurate (and more complex) scheduling techniques should take into account the constraints introduced by the targeted execution platform and should aim at optimizing the usage of the available resources.

A refactoring tool-box [3–5] (included in Gaspard2) is available for adapting the specification to the execution, by reducing array sizes or changing the granularity of the parallelism in order to allow a direct expression of an execution model. These tools are represented by high-level data-parallel code transformations that resemble loop transformations but at a higher level of specification, on data-parallel repetitions. These transformations are fully integrated into the environment as a Papyrus UML[1] plug-in and provide an interactive visual user-interface for the refactoring.

This paper is meant as an experimental validation of these transformations and aims to show how the refactoring tools can be used to adapt a specification to the execution, by design space exploration.

We propose here a strategy to use the refactoring tool supporting the design space exploration of repetitive MARTE models of intensive multidimensional signal processing applications and we experiment it on a typical radar processing application.

After briefly summarizing the principles of the modeling language in Section 2, the radar application used in the paper is described and modeled in Section 3. Finally we present the refactoring experiment and the strategy we propose in Section 4.

2. Multi-dimensional modeling

To model multi-dimensional signal processing applications, the language must support their core characteristics: multi-dimensional data structures, access to these data structures in a regular way, sliding windows, several sampling rates, cyclic data access (for cyclic space or frequency dimensions) and stateful computations. All these requirements are supported by the MARTE Repetitive Structure Modeling (RSM) package. Models using the MARTE profile represent visual abstractions that, by the extensive use of the RSM package, aim at expressing a maximum of the parallelism of computations, of targeted architectures and the repetitive distribution of computations on parallel execution units. Throughout the paper we will refer to such models as MARTE RSM models/specifications.

The repetitive specification provides ways to access multidimensional arrays capable of expressing all the previously mentioned accesses and more in the same specification and can express stateful computations such as recursive filters through uniform inter-repetition dependences proposed in [8].

To make such models statically scheduleable, rules are imposed on the specification, namely regarding the single assignment and the complete production of array elements. These rules and a complete formalism of the Array-OL model

---

[1] http://www.papyrusuml.org/

3

of computation (MoC), the theoretical basis of the MARTE RSM models, are available in [9].

A comparison to similar models of computation dedicated to signal processing, mainly in terms of allowed data structures and expressiveness, like synchronous data-flow languages (SDF [10] and its multidimensional extensions (G)MDSDF [11], WSDF [12]), functional languages (Alpha [13]) or imperative languages (Sisal [14], SaC [15]), is available in [8].

Formally, with MARTE RSM, an application is described as a set of *tasks* connected through *ports*, representing multi-dimensional arrays characterized by their *shape*, the number of elements on each of their dimensions[2] and their *direction*. The tasks are equivalent to mathematical functions reading data on their input ports and writing data on their output ports. The tasks are of three kinds: *elementary*, *compound* and *repetition*. An elementary task is atomic (a black box), it can come from a library for example. A compound is a dependence graph whose nodes are tasks connected via their ports and it allows to express task parallelism.

A repetition is a task expressing how a single sub-task is repeated, each instance of the repeated task operates with sub-arrays of the inputs and outputs of the repetition. Making the repetitions independent and therefore parallel by construction, it allows to express data parallelism. 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.

The *tiler* (one tiler per input or output) concept represents the mathematical expression of the elements of the patterns as tiles of the array, and is composed of: a *fitting* matrix F, whose column vectors represent the regular spacing between the elements of a pattern in the array; $\mathbf{o}$, the *origin* of the *reference pattern* (for the *reference repetition*) and a *paving* matrix P, whose column vectors represent the regular spacing between the patterns.

We can summarize the pattern construction with one formula. For a given repetition index $\mathbf{r}, \mathbf{0} \leq \mathbf{r} < \mathbf{s}_{\mathrm{rep}}$ (the repetition space is unique for a repetition task and all its inputs and outputs) and a given index $\mathbf{i}, \mathbf{0} \leq \mathbf{i} < \mathbf{s}_{\mathrm{pattern}}$ in the pattern, the corresponding element in the array has the coordinates

$$\mathbf{o} + (P \ F) \cdot \begin{pmatrix} \mathbf{r} \\ \mathbf{i} \end{pmatrix} \bmod \mathbf{s}_{\mathrm{array}}, \tag{1}$$

where $\mathbf{s}_{\mathrm{array}}$ is the shape of the array, $\mathbf{s}_{\mathrm{pattern}}$ is the shape of the pattern, $\mathbf{s}_{\mathrm{rep}}$ is the shape of the repetition space.

The pattern construction proposed in MARTE RSM allows constructions ranging from simple block tiled patterns to complex uniform accesses where accesses non-parallel with the axes are combined with strides and modulo. Nonetheless, in real intensive signal processing applications, most of the access remain parallel with the axes, by block or with sliding windows:
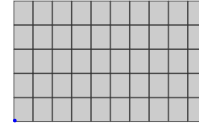
---

[2]A shape will be noted as a column vector or a comma-separated tuple of values indifferently.

Parallel with the axes. With such accesses, all the vectors of the paving and the fitting matrixes contain at most one non-zero value, making the correspondence between each of the repetition's or pattern's dimension and an array dimension.

Access by blocks. Uniform and perfectly tiled block-like patterns are accessed. The fitting and the paving matrix express parallel with the axes constructions where the patterns are continuous tiled blocks in the array (each fitting vector contains just one value equal to 1^3) and the blocks are perfectly tiled one after another (paving vectors have non-zero values equal to the dimension of the pattern on the corresponding array dimension).
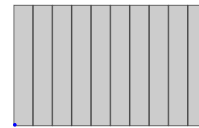
Example 1.

An array with a shape of $(80, 50)$ can be tiled into $(10, 5)$ blocks of $(10, 8)$ elements using the following tiler: fitting $= \left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$, paving $= \left(\begin{smallmatrix} 8 & 0 \\ 0 & 10 \end{smallmatrix}\right)$ as illustrated on the figure on the right.

Entire tiled dimensions. It is a special case of block access where pattern dimensions take entire array dimensions.

Example 2.

The same array with a shape of $(80, 50)$, tiled into (10) blocks of $(50, 8)$, using a fitting matrix of $\left(\begin{smallmatrix} 0 & 1 \\ 1 & 0 \end{smallmatrix}\right)$ and a paving matrix of $\left(\begin{smallmatrix} 8 \\ 0 \end{smallmatrix}\right)$.

Sliding windows. It represents a block access where the blocks do not perfectly tile on some dimensions, with overlapping blocks which cause array elements to be tiled in multiple patterns. On the dimensions of the sliding, paving vectors do not have non-zero values equal to the dimension of the pattern and the value represents the *step* of the window on this dimension.

Example 3.

Leaving unchanged the paving and the fitting matrix of the block access example and by enlarging the pattern to a block of $(10, 12)$, we have a sliding window access on the first dimension of the array, with a step of 8.
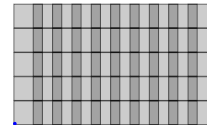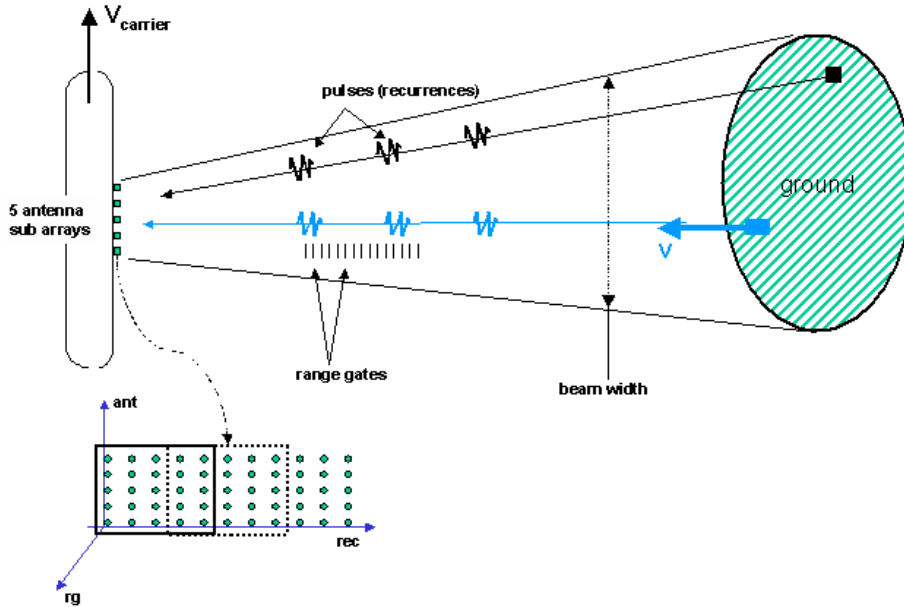
Figure 1: STAP

3. STAP radar application

3.1. Informal description

STAP is a Moving Target Indication (MTI) application, whose goal is to detect from an aircraft the objects that move on the ground, and especially move slowly among all the other generally still reflecting surfaces under the radar beam (ground clutter). This is done by receiving the echo from the ground of a periodic sequence of radar pulses (bursts). Radar processing permits to estimate both the position of a target through the delay between transmission of a signal (pulse) and reception of its echo, and its speed through the Doppler effect that affects echoes of several identical pulses sent periodically: the speed of the target results in a (small) variation of its distance from the radar, which is only visible as a phase shift on the radar signal. In this basic approach, Doppler processing consists in a bench of filters (e.g. a Fast Fourier Transform) each tuned towards a particular phase shift between successive echoes. This kind of Doppler processing is in some situations sufficient to separate reflecting objects on the basis of their speeds. When the beam is directed toward the ground, the largest part of the echoed energy is supposed to come from the still objects that compose the ground (called clutter), while the moving objects of interest send

---

[3]Pseudo-identity matrix, on each line or column we can find at most one non-zero value of 1, but the matrix is not mandatory square.

weak but phase-shifted echoes. However, the radar beam is not perfectly sharp and has a width of a few degrees, which results in giving to some still objects on the ground at the borders of the beam a relative speed with respect to the aircraft (due to the aircraft's own speed) and creating undesired interferences over the moving targets echoes: this creates an ambiguity between intrinsic speeds and azimuths of targets. Adaptive filtering techniques, where fixed filters are replaced by filters that are computed at run-time from the received signal itself, help to minimize at best the effects of an unwanted clutter signal due to the movement of the aircraft: in this MTI case, the Space Time Adaptive Processing (STAP) is used. In this method, a set of filters is computed at every burst, by solving linear systems whose right hand side terms are reference vectors of theoretical phase patterns expected on antenna sub-arrays at several consecutive pulses, each of which corresponding to a particular (velocity, angle) hypothesis of the target relatively to the aircraft. This is shown in Figure 1 where 2D patterns on dimensions antenna and pulse (rec) are considered to compute filters that remove the natural ambiguity between velocity and azimuth.

3.2. Implementation constraints

A large part of the application is data-flow, manipulating multi-dimensional arrays of data. The processing chain uses different operators, with in particular different needs in terms of precision and dynamic range. This processing chain is dynamic, and varies during run-time by frequent variation of algorithm parameters (sizes of arrays, loop bounds, . . .) while keeping roughly the same processing graph. This is generally called multi-moding in the radar community.

The computational load is high enough to clearly involve parallel computing hardware and real-time performance is one of the key requirements, both in terms of computation throughput and latency. This may result from some operational requirements and/or embedded architecture constraints such as memory limitations.

3.3. STAP model

The STAP application was modeled in Papyrus UML using the MARTE Profile[4].

Starting from the top level, the application is successively depicted using a compound or repetitive decomposition, until the wanted detail level is reached, represented by elementary tasks that can be deployed on library functions. The top level of the specification, illustrated on Figure 2, describes the global functionality and the interaction with the environment.

The infinite dimension of the arrays processed by *GlobalSTAP* represents the abstraction of time and Figure 3 describes the data-flow level of the application, the infinity repetition (time) of a single STAP data treatment.

Figure 4 illustrates the compound decomposition of the repeated task of Figure 3 into successive repetitive filters, with array sizes and repetition spaces

---

[4]Throughout the paper, figures represent real manipulable UML models.
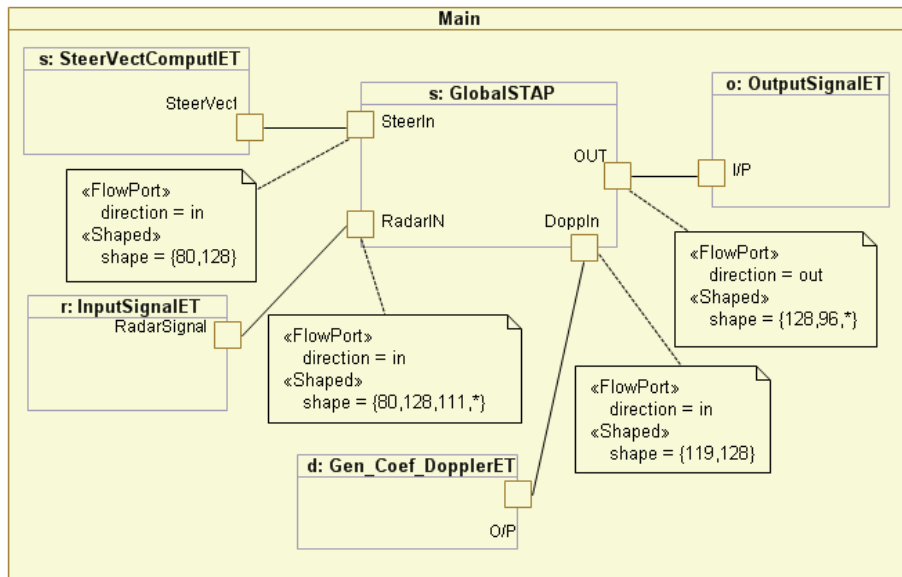
The main STAP application, *GlobalSTAP* takes as input an multidimensional infinite array of data from the sensors (*InputSignalET*) and, using the steer vectors provided by *SteerVectCompIET* and the Doppler coefficients provided by *Gen_Coef_DopplerET*, processes the data and provides the radar detection results as an infinite array, which are consumed by *OutputSignalET* for display, storage or post-treatment.

Figure 2: STAP: top level.
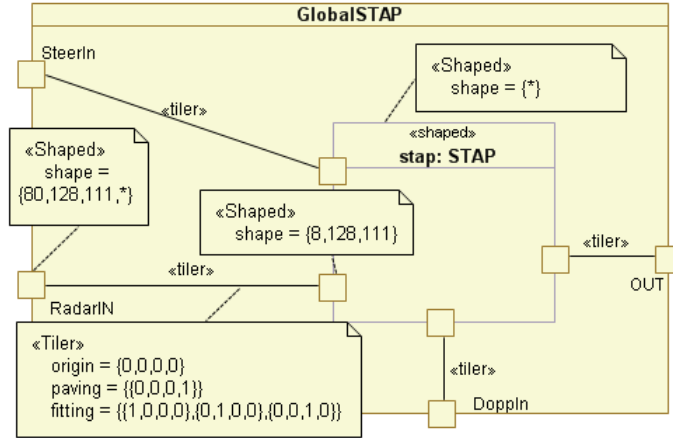
Just one sample tiler is shown, expressing how the infinite radar signals, represented by an array of shape $\{8, 128, 111, *\}$ is decomposed into an infinity, $\{*\}$, of patterns of shape $\{8, 128, 111\}$. The paving matrix of $\{\{0, 0, 0, 1\}\}$ expresses the correspondence between the infinite repetition and the last dimension of the array, while the fitting matrix of $\{\{1, 0, 0, 0\}, \{0, 1, 0, 0\}, \{0, 0, 1, 0\}\}$ expresses the correspondence between the pattern dimensions and the first three dimensions of the array.

Figure 3: STAP: data-flow level

shown on the figure. The tilers that express the pattern/tiles construction for each repetition are not made visible.[5]

Each repetitive filter has a different functionality and different data access patterns:

PulseCompression takes sliding windows with a step of $\{1\}$ of $\{16\}$ values on the third dimension of the input array and computes an average value for each of the pattern.

CovMatrixEstim takes $\{119, 96\}$ blocks of $\{10, 8\}$, with a sliding window with a step of $\{1\}$ on the second dimension of the input array and expands them into blocs of $\{80, 80\}$ which will be arranged into an array of size $\{80, 80, 119, 96\}$.

AverageCovar eliminates the third dimension of the arrays by computing an average of $\{119\}$ values.

MatInv takes $\{96\}$ square matrices of $\{80, 80\}$ and inverses them.

---

[5]A complete model is available at `http://gforge.inria.fr/frs/download.php/5755/examples_papyrus.zip`.

STAP

«FlowPort»
direction = in
«Shaped»
shape = {80,128}

cov: Gbl_CovMatrixEstim

«Shaped»
shape = {119,96}

«shaped»
me: CovMat...

av: Gbl_AverageCovar

«Shaped»
shape = {80,80,96}

«shaped»
a: Average...

inv: Gbl_Mat_Inv

«Shaped»
shape = {96}

«shaped»
mi: Mat_InvE...

filter: Gbl_STAP_Filter

«shaped»
f: STAP_Filt...

«Shaped»
shape = {128,96,80}

«FlowPort»
direction = in
«Shaped»
shape = {8,128,111}

«Shaped»
shape = {80,80,119,96}

«Shaped»
shape = {80,80,96}

«Shaped»
shape = {80,80,96}

«FlowPort»
direction = out
«Shaped»
shape = {128,96}

pulse: Gbl_PulseCompres...

«Shaped»
shape = {8,128,96}

«shaped»
pc: PulseCo...

app: Gbl_STAP_Application

«shaped»
appli: STAP_A...

«Shaped»
shape = {119,128,96}

«Shaped»
shape = {80,128,96}

dopp: Gbl_Int_Doppler

«Shaped»
shape = {128,96}

«shaped»
dotproduct: I...

«Shaped»
shape = {8,128,96}

«Shaped»
shape = {119,128,96}

«FlowPort»
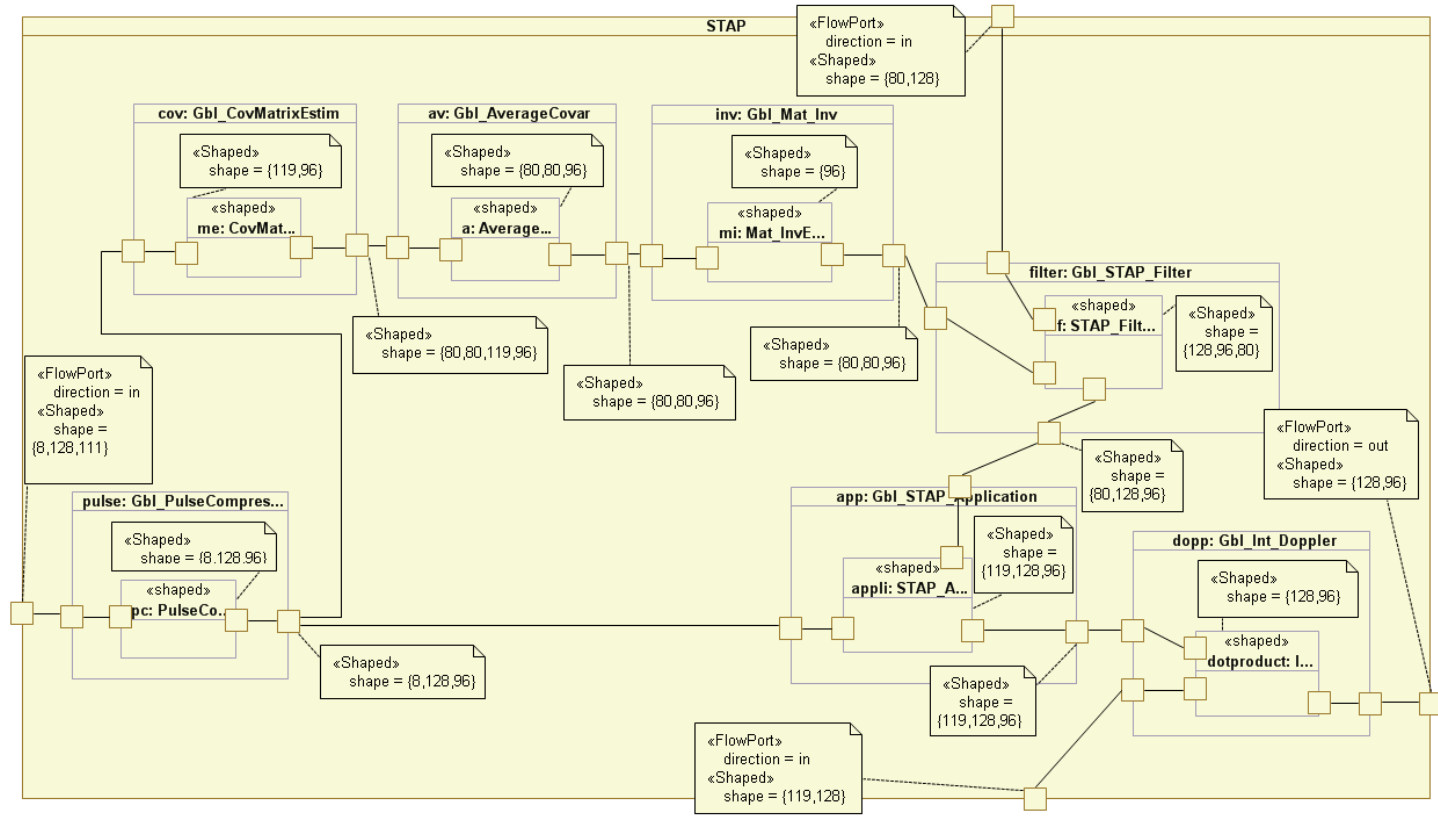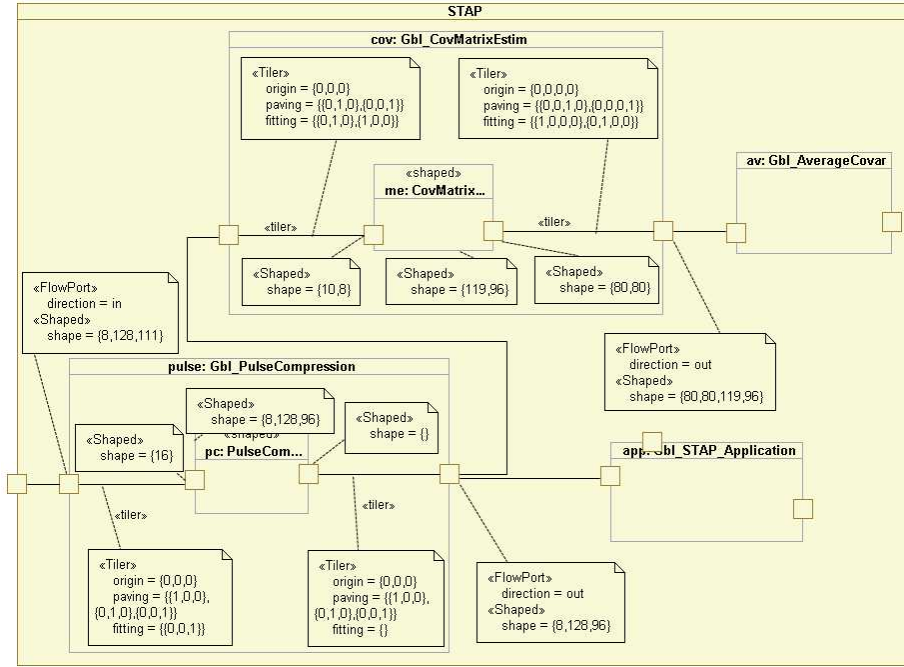direction = in
«Shaped»
shape = {119,128}

Figure 4: STAP decomposition

For the *PulseCompression* repetition, starting from each element of the input array (identity paving matrix), a pattern of {16} elements on the third dimension is used to compute a single value, {}, values rearranged into a tree dimensional array with a shape of {8, 128, 96}. The difference of size between the input and the output array on the third dimensions ($111 - 96 = 15$) is caused by the sliding window accesses, for the last 15 elements the access window will exit the array dimensions. *CovMatrixEstim* repetition has a similar sliding window functionality, this time 2D blocks with a shape of {10, 8} sliding on the second dimension with a step of 1. These blocks are expanded into {119, 96} square blocks of {80, 80}.

Figure 5: First two filters: tiler construction

Stap_Filter applies the {128} steering vectors of dimension {80} to each line of the inverted matrices, producing an array of size {128, 96, 80}.

Stap_Application compares the filtered values with the initial compressed pulses, by blocks of {8, 10} to identify the movements.

Int_Doppler applies the {128} Doppler vectors of size {119} to identify the speeds of the moving targets.

Each repetition is characterized by different repetitive and uniform pattern accesses, data consumption or production, expressed by the tiler informations of each connection inside the repetition, accordingly to equation 1. Figure 5 illustrates the tiler construction for two of the STAP filters, *PulseCompression*

and *CovMatrixEstim.*

At filter level, each repeated task has an elementary functionality which can be deployed on library functions: matrix inversion, average computations, etc.

MARTE RSM models are static and therefore we have chosen to implement a single mode of the multi-moding functionality mentioned in Section 3.2. Multiple modes can be modeled with the use of control structures based on mode automata have been proposed by Labbani et al. [16, 17] or in the upcoming beta3 or 1.0 version of MARTE (no public reference yet available at the time of writing) but they are outside the scope of this study.

4. Refactoring

We present here succinctly the refactoring strategy by working it on the STAP application.

The efficient scheduling for a RSM specification can be seen as an optimization problem for:

> The projection in time (resource concurrency) and space (resource parallelism) of the multidimensional structures part of a RSM specification, while respecting the ensemble of constraints defined on the system, aiming to maximize a predefined cost function.

The chosen cost function characterizes the researched efficient schedule and directly influences the optimization algorithm and its complexity. For a RSM specification, where multidimensional structures express either data structures or parallel repetitions (as we will see in Section 2 where we will present in more details the principles of a repetitive specification) and considering only the functional aspects, the choice of a cost function that

> Maximizes the amount of logical parallelism that is projected into physical parallelism while minimizing the memory size needed to store the data structures throughout the execution,

could be considered as a rather accurate cost function[6].

Nonetheless, even such basic cost function defines two objectives that are concurrent (increase in parallel executions imply increase in memory size), and we are forced to talk about a trade-offs between the usage of storage and computation resources and of strategies that allow us to find good trade-offs between different concurrent objectives.

A more accurate cost function, taken into account the constraints introduced by the execution platform, could be:

> Maximizing the amount of logical parallelism that is projected into the available physical parallelism while limiting the amount of memory needed to store the data structures to the available memory size of the execution platform.

---

[6]relatively to the abstraction level within a RSM specification.

4.1. Specification model vs execution model

A MARTE RSM *specification* expresses the maximum of parallelism through the use of the repetitive decomposition. It describes the data dependencies between the elements of the arrays and, as a direct consequence, a strict partial order between the calls of the tasks. A valid specification must be statically defined and therefore defines a strict partial order. Indeed, any schedule that respects this order will compute the same output values from the same inputs. The construction rules that assure that a specification is valid are described and proved in [9].

The *execution model* represents the abstraction of the execution and it must respect the strict partial order defined by the specification. It is a design intention that, by expressing the minimal order of execution, lots of decisions can and have to be taken when mapping a MARTE RSM 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? It is a strength of this specification 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.

We have chosen to make the transition from a MARTE RSM specification to an execution model the most straight-forward possible. Indeed the execution model should reflect the specification. This choice has as the advantage that the execution model remains similar to the structure of the specification (parallelism, granularity), at least until the code generation. A representative example is the generation of VHDL code that will be used for FPGA synthesis[7] and where the component structure of the specification can be found in the VHDL modules and even on the FPGA layout. It comes down to the constraint that a component may starts its execution when all its input arrays are available.

This constraint introduces a major problem, represented by synchronization barriers between the components. Such a barrier is created by the data dependencies; a extreme illustration of the problem is the presence of any intermediate array that contains an infinite dimension, which would cause the execution to be stalled in that point. As a solution, some repetitions can be transformed to flows, the execution of the repetitions is pipelined and the sub-arrays are read and written as a flow of tokens. Using the hierarchy, we intend to isolate the infinite dimensions at the top hierarchical level of the application (which will represent the data-flow).

4.2. High-level data-parallel transformations

The data-parallel code transformations can be used to adapt the application to the execution, allowing to choose the granularity of the flows and a

---

[7]available within the Gaspard2 framework.

simple expression of the mapping by tagging each repetition by its execution mode: data-parallel or pipeline. A great care has been taken with these transformations to ensure that they do not modify the precise element to element dependences [3, 4]. A comparative study between these transformations and the loop transformations in the context of program optimizations can be found in [5].

Loop transformations [18, 19] are used intensively for optimizations at compilation time but the complexity of optimization algorithms is one reason why many compilers still use heuristics.

Our transformations work as a refactoring tool, their result is translated into model changes at the same level of abstraction and the main optimization targets are the reduction of array sizes and the parallelism management.

A specification expresses the maximum of parallelism and the transformations can be used to change its granularity for a better mapping on a parallel execution platform. The specification, the refactoring and the transition to an execution model are separate stages. The same specification can be adapted differently to the execution, accordingly to platform and execution constraints.

Our high-level transformations are designed as tools at the specification level that can be used for adapting the application to the execution on parallel embedded platforms and for design space exploration, leading to some interesting characteristics:

- The transformations work as a refactoring tool, their result is translated into model changes at the same level of abstraction.

- The application keeps its exact functionality, the same output values are computed for the same inputs.

- As optimization targets, the reduction of array sizes and the parallelism management are the priority. A specification expresses the maximum of parallelism and the transformations can be used to change its granularity for a better mapping on a parallel execution platform.

- The specification, the refactoring and the transition to an execution model are separate stages. The same specification can be adapted differently to the execution, accordingly to platform and execution constraints.

Regardless of their role, all these transformations have a similar impact on an application by redistributing repetitions through the hierarchy levels with the creation or suppression of hierarchy levels.

Taking each transformation one by one, we have:

- *Fusion* takes two successive repetitive tasks, creates a superior hierarchy level for the computed common repetition, while what is left of the initial repetitions is placed on the inferior hierarchy level, minimizing the size of the intermediate array.

- *Tiling* splits a repetition into blocks, by creating a hierarchy level.

14

- *Change paving* (either by dimension creation or by linear growth) acts on redistributing repetitions through successive levels of hierarchy and modifying the granularity of the application.

- *Collapse*, by being the opposite of fusion and tiling, suppresses the superior hierarchy level, its repetitions being added to each of the inferior level repetitions.

4.3. Refactoring objectives

Next, we will see how the STAP specification can be adapted to the execution using the refactoring and how a heuristic of transformation chaining can be deduced.

Isolating the infinite dimensions.

The presence of infinite dimensions (of arrays or repetitions) throughout the application represents the first obstacle when executing a MARTE RSM specification.

An infinite repetition will block the execution in that point and an array with an infinite dimension cannot be placed in a fixed size memory. As the infinity is used to express the time (or the data flow), a solution is to isolate the infinity values at a high level of the hierarchy and consider this level as the data-flow level at the execution: the arrays will not be allocated entirely in memory (just the patterns) and an pipeline execution will be chosen for the repetition. By the use of the *fusion* transformation on successive infinite repetitions, a common repetition can be computed and will represent the data-flow level at execution[8].

The STAP application was modeled already having all the infinity values at one hierarchy level, Figure 3, and therefore this step can be skipped for this particular example. A specification where the infinity values would have been at the filter decomposition level (Figure 4) would express the same exact functionality but would necessitate the isolation of the infinity at a data-flow level to enable an execution without blocking points.

Granularity change.

Execution constraints may impose changes in the *granularity* of an application. Taken for instance, if we have 4 processors and want to execute the repetition of Figure 3 in parallel, a *tiling* transformation can be used to split the repetition into blocks of 4 repetitions that will be placed on the 4 processors and executed in parallel, while the sequence of blocks will represent the data-flow level. It is what we call changing the granularity of a repetition and the result of such transformation is shown on Figure 6.

---

[8]The fusion of two infinite repetitions that fails to isolate the infinity at top level indicates an invalid specification.
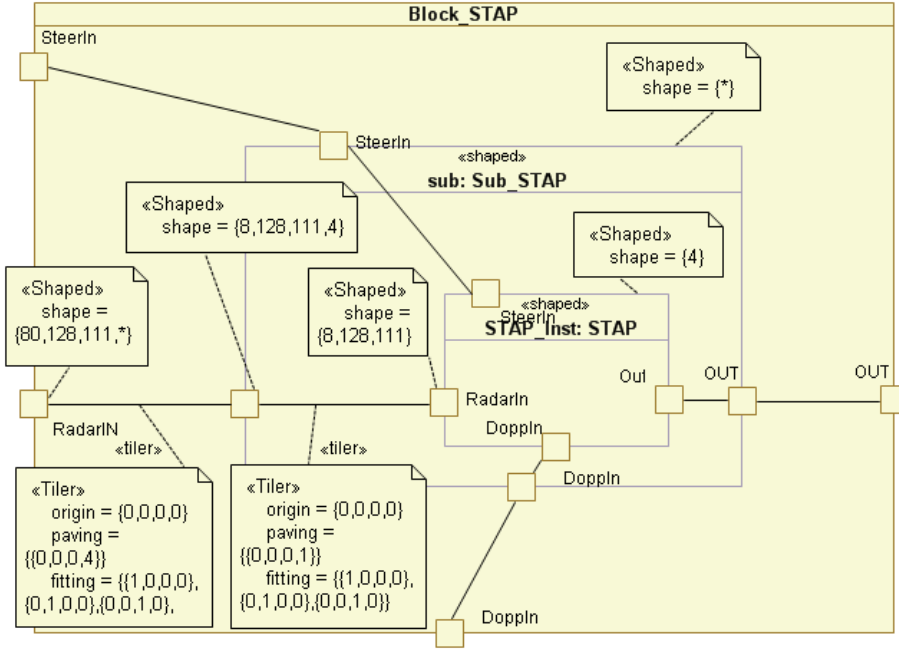
Figure 6: Tiling into blocks of shape {4}

Reduction of the size of the intermediate arrays.

The reduction of the size of the intermediate arrays can be achieved by the use of the fusion transformation. An elementary fusion computes the minimum intermediate array between two successive repetitions. The next definitions will be illustrated on the refactoring of STAP in Section 4.4.

Definition 1 (Minimum intermediate array). *Between two successive repetition tasks[9], a minimal intermediate array is represented by a minimum group of patterns produces by the first task that allows the second task to execute at least once, therefore producing elements and in consequence allowing a non-blocking execution.*

Remark 1. *The fusion computes this minimum group of intermediate patterns and transforms the application by creating a common repetition with two sub-repetitions and an intermediate array reduced to the minimum group of patterns.*

Example 4. *Figure 7 shows the result of fusion for the repetitions of Figure 5. A common repetition with a shape of* {119, 96} *is computed and two sub-repetition of* {10, 8} *and respectively* {} *on the second level of hierarchy.*

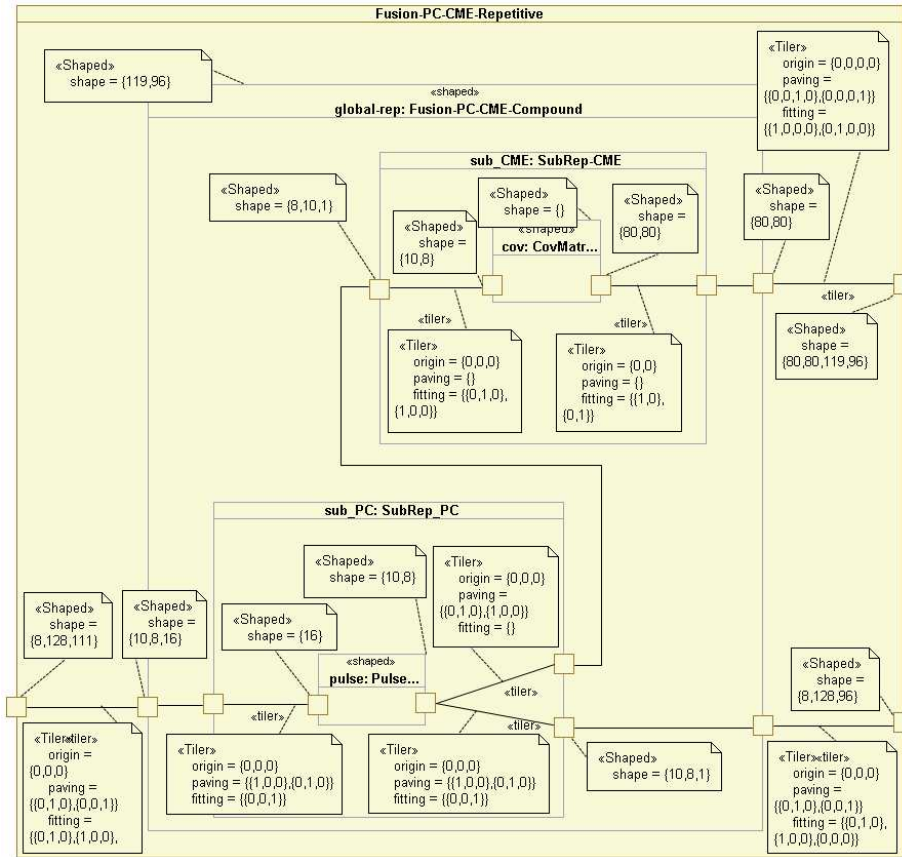[9]The first task produces an array consumed by the second one.

Figure 7: Repetitions of first two tasks after the fusion

The minimum group of patterns produces by the first sub-repetition that allow the execution at least once[10] of the second sub-repetition is of $\{10, 8, 1\}$, and therefore the intermediate array is reduced from the shape of $\{8, 128, 96\}$ ($8 \times 128 \times 96 = 98304$ memory elements reduced to $10 \times 8 \times 1 = 80$, with a factor of $98304/80 = 1228.8$ times).

Definition 2 (Multiple fusion). *A fusion between several connected repetitive tasks can be achieved by the chaining of elementary fusion and collapse transformations. Following the dependence order between tasks, each task is fusioned with the result of the previous fusion[11], while a collapse transformation limits the number of hierarchy levels.*

---

[10]In this case exactly once, {}.

[11]We start by the fusion of two repetition.

Remark 2. *By a multiple fusion, just the last[12] intermediate array is minimized. The other intermediate arrays represent the minimum group of patterns needed for the execution at least once of the last repeated task, and not of the repetitive task that consumes this array.*
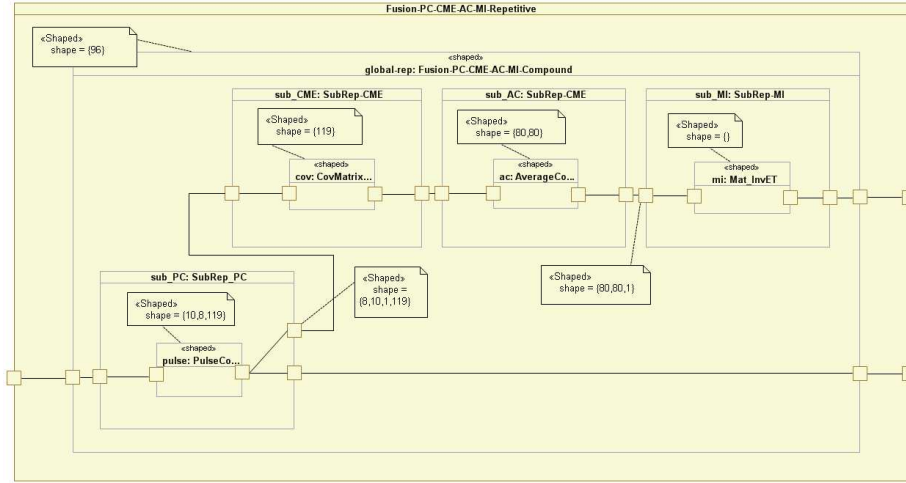

Figure 8: Repetitions of first four tasks after the fusion

Example 5. *Figure 8 shows the fusion of the repetitions* PulseCompression, CovMatrixEstim, AverageCovar *and* MatInv. *The first three sub-repetitions represent minimum execution that allow the last sub-repetition to execute at least once and therefore minimizing the last intermediate array, but not the first two. With such fusion, the first intermediate array is reduces to a size of* $\{10, 8, 1, 119\}$, 119 *times bigger than the minimal reduction reached by the fusion of just the first two repetitions of Figure 7.*

Another factor that comes into play is the amount of *re-computations* introduced into the application by the fusion.

Definition 3 (Re-computation). *Re-computations are represented by an increase of the repetitions for the first task involved in a fusion, caused by the pattern production/consumption between the two tasks.*

Remark 3. *The complete repetition of a task is given by concatenating all the shapes of the hierarchical repetitions starting at the level of the task and climbing the hierarchy levels to the top level[13]. Re-computations are represented by an*

---

[12]The order of tasks in a multiple fusion is given by the strict partial order between these tasks.

[13]The infinite repetition at data-flow level (representing the time) is neglected in out paper.

increase in the complete (taking into account all the levels of hierarchy) repetition of a task after the fusion.

After the fusion, sliding window accesses with some initial elements found in several pattern groups implies that the first task has to compute them several times. In the case of re-computations, the decrease of intermediate array size has as side effect an increase of computations.

Example 6. *On Figure 7, the complete repetition of the first repeated task is, by concatenating the repetition spaces of the two hierarchy level, of $\{119, 96, 10, 8\}$, while the initial repetition was of $\{8, 128, 96\}$ and therefore the number of repetitions increase with a factor of $119 \times 96 \times 10 \times 8/(8 \times 128 \times 96) = 9.29$.*

Thus, in the case of intermediate values that are used several times, the designer has to make a trade-off between the storage in memory or multiple computations of these intermediate values.

4.4. Reduction of intermediate arrays on STAP

The reduction of the size of the intermediate arrays is the principal use of the fusion transformation. Having several successive repetitive tasks, we need a way of reducing the size of the intermediate arrays while avoiding blocking points in the execution and limiting as much as possible the re-computations introduced by the fusions. Different mappings might impose slightly different heuristics, like forbidding re-computations or the priority to minimizing the array sizes in the detriment of re-computations. On the STAP application of Figure 4, we have chosen as objective to minimize the intermediate arrays while limiting the re-computations.

Complete fusion.

A *complete* fusion (the fusion of all the repetitions at a hierarchy level) is not appropriate in this case, as shown in Table 1[14] (containing informations on the repetitions before and after the fusion, the introduced re-computations and the size reduction of the intermediate arrays), where we find ourselves with a great amount of re-computations. The fusion of multiple repetitions minimizes only the last intermediate array (Figure 2). Indeed, the table shows how just two intermediate arrays have a reduced size, as for the others we have even an increase in array size, caused by the multidimensional re-arrangement of patterns as result of the succession of fusion/collapse transformations[15].

Nonetheless, such a transformation provides useful informations that can be used for finding the best sequence of fusions for our objective. First of all, it provides the fusion order representing the strict partial order between the

_____

[14]PulseCompression=PC, CovMatrixEstim=CME, AverageCovar=AC, MatInv=MI, StapFilter=SF, StapApplication=SA, IntDoppler=ID.

[15]Initial overlapping accesses are expanded through multiple dimensions with element duplication.

Table 1: Repetitions for the complete fusion

Task	Rep. before		Rep. after	Re-compu-tation($\times$)
PC	$8 \times 128 \times 96$		$8 \times 128 \times 119$	119×128
CME	119×96		119×119	119×128
AC	$80 \times 80 \times 96$		$80 \times 80 \times 119$	119×128
MI	96	$128 \times 96 \times$	119	119×128
SF	$128 \times 96 \times 80$		80×119	119
SA	$119 \times 128 \times 96$		119	1
ID	128×96		1	1

The hierarchy of repetitions is represented by brackets in the table. After a complete fusion of the repetitions, a common repetition of $\{128, 96\}$ is computed and the total repetition for each sub-repetitions is given by the multiplication with this common repetition. The re-computation is computed as the division of the repetition after and before the refactoring.

Table 2: Array sizes for the complete fusion

Array	Size before	Size after	Reduction
PC→CME	$8 \times 128 \times 96$	$8 \times 128 \times 119$	0.807
CME→AC	$80 \times 80 \times 119 \times 96$	$80 \times 80 \times 119 \times 119$	0.807
AC→MI	$80 \times 80 \times 96$	$80 \times 80 \times 119$	0.807
MI→SF	$80 \times 80 \times 96$	$80 \times 80 \times 119$	0.807
SF→SA	$80 \times 128 \times 96$	80×119	103.26
PC→SA	$8 \times 128 \times 96$	$8 \times 128 \times 119$	0.807
SA→ID	$119 \times 128 \times 96$	119	12288

Array size in memory is given in memory units of the data type of array elements, by the multiplication of the multidimensional dimensions of its shape. In the table, arrays are identified by the tasks that produce/consume the array (Producer→Consumer) and the reduction in size is given by the division of array sizes before and after the refactoring.

repetition tasks, in this case: *PulseCompression < CovMatrixEstim < AverageCovar < MatInv < StapFilter < StapApplication < IntDoppler*. Secondly, on Table 1 we can identify where changes in the re-computations appear and therefore the fusions that introduce re-computations, in our case: *MatInv < StapFilter* and *StapFilter < StapApplication*. In our goal to reduce at minimum the re-computation, it would suggest that the grouping of the first four repetitions and of the last two for fusion, while isolation *StapFilter*, would not introduce any re-computations.

Other fusions that provide useful informations are the fusions of each two successive repetitions. The results contain informations on the minimal intermediate array achievable between any two repetitions and on the re-computations introduced by the fusions, as shown on Table 3.

Table 3: Two-by-two fusion

Fusion	Reduction ($/$)	Re-comp.
PC, CME	$128 \times 96/10 = 1228.8$	9.29
CME, AC	96	1
AC, MI	96	1
MI, SF	96	128
SF, SA	$128 \times 96 = 112288$	119
PC, SA	$128 \times 96/10 = 1228.8$	$119 \times 10 = 1190$
SA, ID	$128 \times 96 = 12288$	1

The maximum reduction of the intermediate array together with the re-computations introduces on the first sub-repetition, achieved by the fusion of each two successive repetitions.

Best choice fusion.

The informations provided by the complete and the two-by-two fusions can guide the choice of a sequence of fusions that achieves the best results for our chosen objective of reducing at maximum the intermediate arrays while limiting the re-computations. The complete fusion suggests the grouping of the first four repetitions and of the last two but the two-by-two fusions indicate that the fusion of the first two repetitions introduces re-computations with a factor of nearly 10 for a reduction of array size with a factor of 1228.8. We have two alternatives for the first group of repetitions:

- If the re-computation factor of 10 is acceptable, we can group the first four repetitions for the fusion.

- Otherwise, just repetitions two to four will be fusioned, with no re-computations.

For illustration, we have chosen the first alternative, grouping the first four tasks and the last two. The result shows a reduction to maximum (accordingly to Table 3) of arrays: *CovMatEst→AvCov*, *AvCov→MatInv* and *StapApp→IntDopp* but a non-maximal reduction for *PulCompr→CovMatEst*. On the second level of the hierarchy, by the fusion of the two sub-repetitions, *PulseCompresion* and *CovMatrixEstim*, we can further reduce (to maximum) this array, with as result the presence of three levels of repetitions for these two repetitions. The array-size reductions are shown in Table 4, while the repetitions and re-computations in Table 5.

A part of the transformed application accordingly to the previous algorithm are illustrated on Figure 9, the transformed filters level, and on Figure 10 the sub-repetitions for the last two tasks.

21

Table 4: Array sizes for the best choice fusion

Array	Size before	Size after	Reduction
PC→CME	$8 \times 128 \times 96$	8×10	1228.8
CME→AC	$80 \times 80 \times 119 \times 96$	$80 \times 80 \times 119$	96
AC→MI	$80 \times 80 \times 96$	80×80	96
MI→SF	$80 \times 80 \times 96$	$80 \times 80 \times 96$	1
SF→SA	$80 \times 128 \times 96$	$80 \times 128 \times 96$	1
PC→SA	$8 \times 128 \times 96$	$8 \times 128 \times 96$	1
SA→ID	$119 \times 128 \times 96$	119	12288

Table 5: Repetitions for the best choice fusion

Task	Rep. before	Rep. after	Re-comp.
PC	$8 \times 128 \times 96$	$96 \times \left(119 \times \begin{pmatrix} 10 \times 8 \\ 1 \end{pmatrix} \right)$	9.29
CME	119×96		1
AC	$80 \times 80 \times 96$	$96 \times \begin{pmatrix} 80 \times 80 \end{pmatrix}$	1
MI	96	$96 \times \begin{pmatrix} 1 \end{pmatrix}$	1
SF	$128 \times 96 \times 80$	$128 \times 96 \times 80$	1
SA	$119 \times 128 \times 96$	$128 \times 96 \times \begin{pmatrix} 119 \\ 1 \end{pmatrix}$	1
ID	128×96		1

4.5. Result analysis

In the previous section we have seen how the high-level transformations can be used to adapt a specification to the execution. Optimizations can be of two kinds, platform dependent or general-purpose. The reduction of intermediate arrays can be considered as a general-purpose optimization but, as shown, execution constrains can guide the choice of fusions that lead to the reduction of intermediate arrays. The granularity changes, through the use of transformations like change paving, tiling or collapse, are more platform-dependent optimizations, aiming at the adaptation of the specification to the execution platform and the optimization of the placement of repetitions on parallel architectures.

Signal processing applications are often represented as successive repetitive filters, as the case of the STAP application of Figure 4. The complex pattern accesses makes it impossible to reduce at maximum every intermediate array and sometimes only with the introduction of additional computations. We have shown how, by exploring some fusion configurations, we can extract informations that lead us to the best choice of chain of transformations. The complete fusion provides informations on the strict partial order and on where changes in re-computations appear, while the two-by-two fusions provide the values of the maximum achievable reductions and on the re-computations introduced by such a reduction. These informations were used to separate the repetitions into groups by forbidding fusions that introduce considerable re-computations. Different objectives and different applications might determine the alternative use of fusions, taken for instance the presence of an infinite dimension intermediate array, where the elimination of the infinite dimension has priority on the
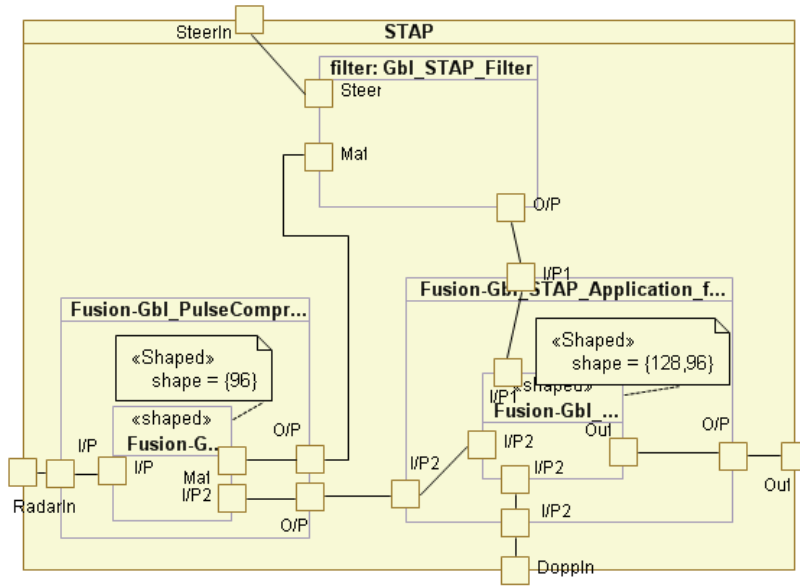
The first four repetitions are grouped into a common repetition of {96} and the last two into a repetition of {128, 96} of a compound containing two sub-repetitions of the initial tasks, shown on Figure 10.

Figure 9: Top level transformed application.

introduced re-computations.

5. Conclusions

The present article shows how high-level data-parallel transformations can be used to explore the design space of multi-dimensional computation intensive signal processing applications with the objective of adapting a specification to the execution. In the context of the Gaspard2 environment [6], using a visual UML interface and the MARTE profile, multi-dimensional intensive processing applications can be modeled, together with repetitive architectures and repetitive placement of the applications on such architectures. Before the code generation, a MARTE RSM specification must be adapted to the execution platform by reducing array sizes and adapting the application's granularity to the architectural and execution constraints. Starting from an application model, different strategies can be chosen to adapt the specification to the execution. These strategies
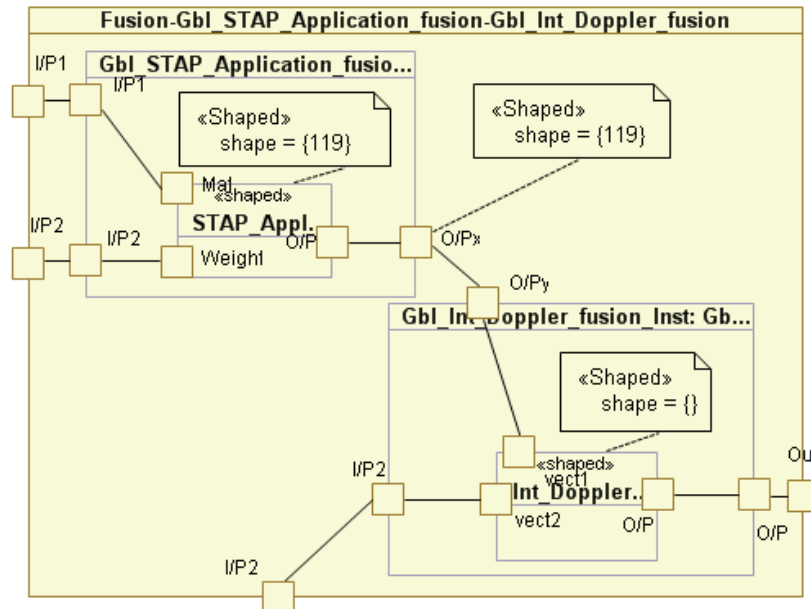
One of the reduction of the intermediate array *StapApp→IntDopp* to a size of {119} can be observed.

Figure 10: Sub-repetitions of *StapApplication* and *IntDoppler*

together with different mappings can be easily explored with possible feedback from the code generation, simulation or execution. As future work, these different strategies could be implemented and integrated in the environment together with a semi-automatic refactoring: proposition to the user of different strategies with computed gains in array size reduction and re-computations.

References

[1] Object Management Group, A UML profile for MARTE, `http://www.omgmarte.org` (2007).

[2] ProMarte partners, UML Profile for MARTE, Beta 2, `http://www.omgmarte.org/Documents/Specifications/08-06-09.pdf` (Jun. 2008).

[3] J. Soula, Principe de compilation d'un langage de traitement de signal, Thèse de doctorat (PhD Thesis), Laboratoire d'informatique fondamentale de Lille, Université des sciences et technologies de Lille, (In French) (Dec. 2001).

[4] P. Dumont, Spécification multidimensionnelle pour le traitement du signal systématique, Thèse de doctorat (PhD Thesis), Laboratoire d'informatique

fondamentale de Lille, Université des sciences et technologies de Lille (Dec. 2005).

[5] C. Glitia, P. Boulet, High level loop transformations for multidimensional signal processing embedded applications, in: SAMOS 2008 Workshop, Samos, Greece, 2008.

[6] DaRT Team, Graphical Array Specification for Parallel and Distributed Computing (GASPARD2), `http://www.gaspard2.org/` (2009).

[7] E. Piel, R. Ben Atitallah, P. Marquet, S. Meftali, S. Niar, A. Etien, J.-L. Dekeyser, P. Boulet, Gaspard2: from MARTE to SystemC simulation, in: Modeling and Analyzis of Real-Time and Embedded Systems with the MARTE UML profile DATE'08 Workshop, 2008.
URL `http://www2.lifl.fr/MARTEworkshop/`

[8] C. Glitia, P. Dumont, P. Boulet, Array-OL with delays, a domain specific specification language for multidimensional intensive signal processing, Multidimensional Systems and Signal Processing.

[9] P. Boulet, Formal semantics of Array-OL, a domain specific language for intensive multidimensional signal processing, Research Report RR-6467, INRIA (Mar. 2008).
URL `http://hal.inria.fr/inria-00261178/en/`

[10] E. A. Lee, D. G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. on Computers.

[11] P. K. Murthy, E. A. Lee, Multidimensional synchronous dataflow, IEEE Transactions on Signal Processing 50 (8) (2002) 2064–2079.

[12] J. Keinert, C. Haubelt, J. Teich, Modeling and analysis of windowed synchronous algorithms, in: International Conference on Acoustics, Speech and Signal Processing (ICASSP), 2006, pp. III–892– III–895.

[13] R. M. Karp, R. E. Miller, S. Winograd, The organization of computations for uniform recurrence equations, J. ACM 14 (3) (1967) 563–590.

[14] I. Attali, D. Caromel, Y. syau Chen, J. luc Gaudiot, A. L. Wendelborn, A formal semantics for sisal arrays, in: Proc. Joint Conf. Infor. Sci., 1995.

[15] S.-B. Scholz, Single assignment c: efficient support for high-level array operations in a functional setting, J. Funct. Program. 13 (6) (2003) 1005–1059. doi:http://dx.doi.org/10.1017/S0956796802004458.

[16] O. Labbani, J.-L. Dekeyser, P. Boulet, E. Rutten, UML2 profile for modeling controlled data parallel applications, in: FDL'06: Forum on Specification and Design Languages, Darmstadt, Germany, 2006.

[17] O. Labbani, J.-L. Dekeyser, P. Boulet, E. 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).

[18] R. Allen, K. Kennedy, Optimizing Compilers for Modern Architectures: A Dependence-based Approach, Morgan Kaufmann Publishers, 2001, `http://www.mkp.com/books_catalog/catalog.asp?ISBN=1-55860-286-0`.

[19] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, O. Temam, Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies, Int. J. Parallel Program. 34 (3) (2006) 261–317. doi:http://dx.doi.org/10.1007/s10766-006-0012-3.

[20] Ter@ops-emb project.
URL `http://teraops-emb.ief.u-psud.fr/`