# A Computation Core for Communication Refinement of Digital Signal Processing Algorithms

Sylvain Huet, Emmanuel Casseau, Olivier Pasquier

# A Computation Core for Communication Refinement of Digital Signal Processing Algorithms

Sylvain Huet
Université de Bretagne Sud
LESTER Lab CNRS FRE 2734
56321 Lorient Cedex, France
sylvain.huet@univ-ubs.fr

Emmanuel Casseau
Université de Bretagne Sud
LESTER Lab CNRS FRE 2734
56321 Lorient Cedex, France
emmanuel.casseau@univ-ubs.fr

Olivier Pasquier
Polytech'Nantes
IREENA
BP50609, 44306 Nantes Cedex 3, France
olivier.pasquier@polytech.univ-nantes.fr

## Abstract

*The most popular Moore's law formulation, which states the number of transistors on integrated circuits doubles every 18 months, is said to hold for at least another two decades. According to this prediction, if we want to take advantage of technological evolutions, designer's productivity has to increase in the same proportions. To take up this challenge, system level design solutions have been set up, but many efforts have still to be done on system modelling and synthesis. In this paper we propose a computation core synthesis methodology that can be integrated on the communication refinement steps of electronic system level design tools. In the proposed approach, computation cores used for digital signal processing application specifications relying on coarse grain communications and synchronizations (e.g. matrix) can be refined into computation cores which can handle fine grain communications and synchronizations (e.g. scalar). Its originality is its ability to synthesize computation cores which can handle fine grain data consumptions and productions which respect the intrinsic partial orders of the algorithms while preserving their original functionalities. Such cores can be used to model fine grain input output overlapping or iteration pipelining. Our flow is based on the analysis of a fine grain signal flow graph used to extract fine grain synchronizations and algorithmic expressions.*

## 1 Introduction

### 1.1 Problem formulation

The classical approach for designing complex Digital Signal Processing (DSP) applications is based on a top-down refinement flow where an initial abstract specification of the application is progressively and hierarchically decomposed into interacting subsystems.
There are many paths leading from the specification of a system to its implementation. To mark it out, some researchers have established graphical taxonomies which can help the designers to analyse their designs or to define the path which best suits their needs. The Y chart model due to Gajski [5] classifies the abstractions of a system on structural, behavioural and geometrical axes. Nevertheless, it lacks of a representation of time and data abstractions. This is the reason why Ecker et al. introduce the Design Cube [3], where a design flow can be categorized according to the three following axes: timing, values, view. Seven years later, Jantsch et al. introduced the Rugby Model [6]: they extend the notions of timing, values, view and add a fourth dimension, communication, to be able to classify modern hardware software design flows. The idea of the Design Cube has also been updated by Thabet et al. [9] with the aim at defining a communication refinement flow which includes data type refinements. This dimension is especially important in the context of hardware refinement.

Actually, DSP application specifications often rely on abstract data types (e.g. matrix) whereas their hardware implementations work on scalar arithmetic operators in a parallel way. It results parallel scalar communications between the actors of the systems. From the implementation point of view, hardware communications can be managed according to these two levels of abstraction leading to the two following design paradigms. (1) For each abstract data type communication, two memories of a size of the abstract data type can instantiated and will alternatively be used as a buffer memory and a working memory: the receiver reads data in the working memory whereas the sender writes data in the buffer memory; when the receiver has finished to consume data located in the working memory and the buffer memory is full, the roles are reversed. This scheme can be used to implement communications with abstract data types synchronizations but it can lead to an higher memory cost and data latency than the following technique. (2) The communication patterns of the receivers and senders are adapted at the scalar grain with the view to minimize the memory costs and the latency. Filo et al. [4], Coussy et al. [2] work on that problematic in the context of High Level Synthesis (HLS). However, it would be interesting to introduce the analysis of the fine grain communication patterns before the synthesis process. This can be achieved by having simulation models which can be synchronized at fine grain. In this paper we propose a computation core synthesis methodology which allows to generate such models: it allows to model fine grain Input Output (IO) overlapping, while preserving the original algorithm functionality. We call this computation core a Fine Grain Synchronized Computation Core (FGSCC).

### 1.2 Paper organisation

Section two presents some useful prerequisites to the comprehension of the next sections. Section three presents the FGSCC and the design flow used to synthesize it. Section four illustrates the synchronization and fine grain computation aspects. Section five concludes and gives an overview of our current work.

## 2 Definitions

This section gives some definitions and basic notions that are used throughout the paper.

**Definition 2.1** (Fine grain data)**.** *A fine grain data is a data which is operand or result of an operator used for the hardware implementation of an algorithm. In the context of this paper, a fine grain data is a scalar.*

**Definition 2.2** (Coarse grain data)**.** *A coarse grain data is a conceptual clustering of fine grain data. Matrix, vectors, are common coarse grain data used to specify DSP applications.*

**Definition 2.3** (Algorithm)**.** *An algorithm is a finite set of well-defined instructions for accomplishing some task which, given an initial state, will terminate in a corresponding recognizable end-state. From the DSP application modelling point of view an algorithm is an indivisible process that is fired when all its inputs are available and that will then produce all its outputs. For example, the algorithm presented on figure 1 is fired when A and B are available and then produces C.*

```
Algorithm PRODMAT with
        constant N = 10;
        input:   A[1:N][1:N] of integer;
        input:   B[1:N][1:N] of integer;
        output : C[1:N][1:N] of integer;
begin
        //matrix product code
end
```

**Figure 1. matrix product algorithm**

**Definition 2.4** (Interface)**.** *An interface is the place where an algorithm communicates with its environment. It is composed of oriented communication ports. In the context of DSP algorithm, we classify the interfaces into two classes, the input interfaces used to receive information, the output interfaces used to send information to their environment. The algorithm presented on figure 1 has one input interface composed of the ports A and B and one output interface composed of the port C.*

**Definition 2.5** (Fine grain interface)**.** *A fine grain interface is an interface where all the ports exchanged fine grain data.*

**Definition 2.6** (Coarse grain interface)**.** *A coarse grain interface is an interface where at least one port exchange coarse grain data.*

**Definition 2.7** (Algorithmic iteration)**.** *An algorithmic iteration is composed of the set of the input values and the set of the resulting output values obtained by the algorithm firing. An algorithmic iteration is identified by an iteration number which is incremented by one each firing, consider the algorithm $f$.*
*Let $I^i = \left\{i_1^i, i_2^i, \ldots, i_m^i\right\}$ be the set of the values of the*

*inputs of the algorithm $f$ at the iteration $i$.*
*Let $O^i = \{o_1^i, o_2^i, \ldots, o_n^i\}$ be the set of the values of the outputs of the algorithm $f$ at the iteration $i$, obtained by the relation $O^i = f(I^i)$.*
*The algorithmic iteration number $i$ of $f$ is the couple $(I^i, O^i)$*

**Definition 2.8** (Fine grain input output overlapping)**.** *Fine grain input output overlapping refers to the interleaving of consumptions and productions of fine grain data. For example, if we consider figure 1, C[1][1] can be produced as soon as the first row of A and the first column of B are available. In such a case, we can say that we have an overlapping of the consumption of the first row of A and the first column of B with the production of C[1][1].*

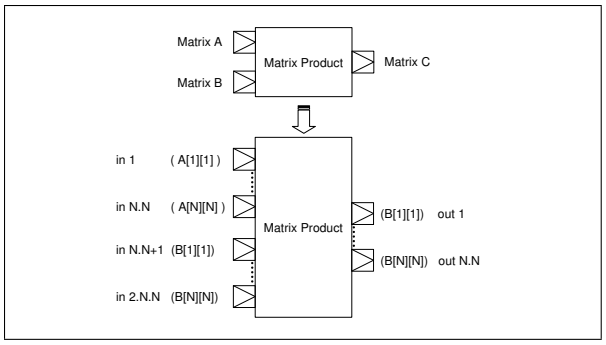


**Figure 2. coarse grain to fine grain algorithm interface transformation**

# 3 Computation Core Synthesis

The first subsection presents the formal model of execution of our FGSCC. Subsection two shows how we implement it. At last, subsection three presents the design flow we put into practice to generate a FGSCC.

## 3.1 The formal model of execution

The first step of the synthesis of our FGSCC consists in transforming the original coarse grain interfaces of the algorithm in fine grain interfaces. The second step consists in refining the original indivisible algorithm into multiple algorithms that allow to compute fine grain outputs according to the fine grain inputs and to extract the fine grain synchronizations.

### 3.1.1 coarse grain to fine grain algorithm interface refinement

As illustrated on figure 2, coarse grain to fine grain interface transformation is a trivial refinement. It consists in defining bijections to split coarse grain interfaces into fine grain interfaces.

### 3.1.2 coarse grain synchronized algorithm to fine grain synchronized algorithm

The previous step is not sufficient to to generate a FGSCC. Indeed it can be used to model fine grain communications, but the computations remain coarse grain synchronized. To be able to synchronize them with fine grain data, it is necessary to have the algorithmic expressions of the fine grain outputs according to the fine grain inputs. Moreover, in this paper we focus on DSP algorithms. These ones can have an inter-iterations memory effect, that is to say the algorithm can use past information to compute the present, this is the consequence of the $z^{-1}$ operators. We introduce the possibility to model this memory effect with fine grain variables which we call ageing variables. The reader can consult subsection 4.1 to have a practical example of an ageing variable use. Our model of execution is formalised below. Consider the algorithm $f$.
Let $I^i = \{i_1^i, i_2^i, \ldots, i_m^i\}$ be the set of the values of the fine grain inputs of the algorithm $f$ at the iteration $i$.
Let $O^i = \{o_1^i, o_2^i, \ldots, o_n^i\}$ be the set of the values of the fine grain outputs of the algorithm $f$ at the iteration $i$.
Let $A^i = \{a_1^i, a_2^i, \ldots, a_k^i\}$ be the set of the values of the values of the ageing variables of the algorithm $f$ at the iteration $i$.
Let $F_a = \{f_{a,1}, f_{a,2}, \ldots, f_{a,k}\}$ be the set of the functions which compute respectively $\{a_1^{i+1}, a_2^{i+1}, \ldots, a_k^{i+1}\}$ according to $I^i$ and $A^i$.
Let $F_o = \{f_{o,1}, f_{o,2}, \ldots, f_{o,n}\}$ be the set of the functions which compute respectively $\{o_1^i, o_2^i, \ldots, o_n^i\}$ according to $I^i$ and $A^i$.
The finest synchronization grain is obtained for the functions of $F_a$ and $F_o$ which have the smallest start space. For example, let consider C[1][1] of figure 1. It is possible to find an algorithmic expression which computes C[1][1] according to the whole matrices A and B. Nevertheless C[1][1] can be computed in function of the first row of A and the first column of B, these constitute the smallest start space to compute C[1][1] and thus offer the finest synchronization grain on C[1][1].
From a practical point of view, we obtain such functions through the analysis of a fine grain Signal Flow Graph (SFG) representation of the algorithms.

### 3.2 The computer model of execution

To build a computer model of the previous formal model of execution, we define (1) an `iteration` object which is used to model the iterations (2) an `iteration_vector` object which is used to manipulate the FGSCC.

#### 3.2.1 The iteration object

The figure 3 presents the `iteration` object. The two following paragraphs present its attributes and methods.

```
Class iteration< type >

Private Attributes
        matrix< bool > *        m_dep
        matrix< bool > *        v_in_pre
        matrix< type > *        v_in_val
        matrix< bool > *        v_out_pre
        matrix< type > *        v_out_val
        matrix< bool > *        v_out_con

Public Member Functions
        bool    in_exists (int ref)
        bool    out_exists (int ref)
        bool    put (type val, int ref)
        type    get (int ref)
        bool    is_consumed (int ref)
        bool    is_consumed ()
        bool    is_in_ageing (int ref)
        bool    is_out_ageing (int ref)
        bool    refresh ()
```

**Figure 3. iteration object**

$$\texttt{m\_dep}^i = \begin{array}{c|cccccc} & o_1^i & \dots & o_n^i & a_1^{i+1} & \dots & a_k^{i+1} \\ i_1^i & bool & \dots & bool & bool & \dots & bool \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ i_m^i & bool & \dots & bool & bool & \dots & bool \\ \hdashline a_1^i & bool & \dots & bool & bool & \dots & bool \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ a_k^i & bool & \dots & bool & bool & \dots & bool \end{array}$$

**Figure 4. dependencies matrix**

**Attributes** - The dependencies matrix, (figure 4)
The dependencies matrix, `m_dep`, is composed of booleans which represent (1) the dependencies of the outputs of the current algorithmic iteration according to the inputs and the ageing variables of the current algorithmic iteration (2) the dependencies of the of ageing variables of the next algorithmic iteration according to the inputs and the ageing variables of the current algorithmic iteration. True represents a dependency, false a non dependency.

$$\texttt{v\_in\_val}^i = \begin{array}{c|c} i_1^i & fgdt \\ \dots & \dots \\ i_m^i & fgdt \\ \hdashline a_1^i & fgdt \\ \dots & \dots \\ a_k^i & fgdt \end{array}, \texttt{v\_out\_val}^i = \begin{array}{c|c} o_1^i & fgdt \\ \dots & \dots \\ o_n^i & fgdt \\ \hdashline a_1^{i+1} & fgdt \\ \dots & \dots \\ a_k^{i+1} & fgdt \end{array}$$

**Figure 5. input and output values vector**

$$\texttt{v\_in\_pre}^i = \begin{array}{c|c} i_1^i & bool \\ \dots & \dots \\ i_m^i & bool \\ \hdashline a_1^i & bool \\ \dots & \dots \\ a_k^i & bool \end{array}, \texttt{v\_out\_pre}^i = \begin{array}{c|c} o_1^i & bool \\ \dots & \dots \\ o_n^i & bool \\ \hdashline a_1^{i+1} & bool \\ \dots & \dots \\ a_k^{i+1} & bool \end{array}$$

**Figure 6. input and output presences vector**

- The values vector, (figure 5)
The input values vector, `v_in_val`, contains the values of the inputs and the ageing variables of the current algorithmic iteration. The output values vector, `v_out_val`, contains the values of the outputs of the current algorithmic iteration. The data types of theses values are fine grain data type, referred as fgdt on the figure.
- The presences vector, (figure 6)
The input presences vector, `v_in_pre`, is composed of booleans which represent the presences of the inputs and the ageing variables for the current algorithmic iteration. If true the corresponding input or ageing variable is present in the current algorithmic iteration. That is to say the corresponding value in `v_in_pre` is valid. The output presences vector, `v_out_pre`, is composed of booleans which indicate if the corresponding output value in `v_out_val` is valid.
- The output consumption vector, (figure 7)

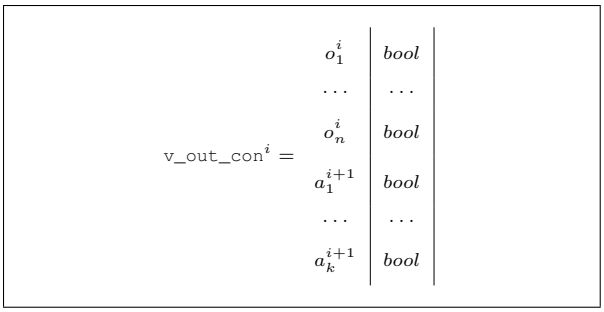


**Figure 7. output consumption vector**

The output consumption vector `v_out_con` is composed of booleans which indicates which values of `v_out_val` has been consumed.

**Methods** `-bool in_exists (int ref)`
This method is used to know if the element at row ref in `v_in_val` is already present. If yes return true, else return false.
`-bool out_exists (int ref)`
This method is used to know if the element at row ref in `v_out_val` is valid, i.e. is computed. If yes return true, else return false.
`-bool put (type val, int ref)`
This method is used to put the value val at row ref in `v_in_val`. Then the boolean at row ref in `v_in_pre` is set to true. Return true if the value is put, i.e. if row ref of `v_in_pre` equals false, else return true.
`-type get (int ref)`
This method is used to get the value at row ref of `v_out_val`. When called, the boolean at row ref of `v_out_con` is set to true.
`-bool is_consumed (int ref)`
This method is used to know if the value of `v_out_val` at row ref has been consumed.
`-bool is_consumed ()`
The method is used to know if all the values of the iteration have been consumed.
`-bool is_in_ageing (int ref)`
The method is used to know if the value at row ref in `v_in_val` is an ageing variable. If yes return true, else return false.
`-bool is_out_ageing (int ref)`
The method is used to know if the value at row ref in `v_out_val` is an ageing variable. If yes return true, else return false.
`-bool refresh ()`
This method computes the the output and ageing variable. Its algorithm can be summarized as follow: for each column of the `m_dep`, if the corresponding output or ageing variable is not computed and `v_in_pre` equals the column of `m_dep`, compute it, put the result in `v_out_val` and set the corresponding boolean to true in `v_out_pre`.

### 3.2.2 The iteration_vector object

The iteration vector object is the interface of the FGSCC. The figure 8 presents the iteration object. The two following paragraphs present its attributes and methods.

```
Class iteration_vector< type >

Private Attributes
        iteration< type > *     iteration_list

Public Member Functions
        bool    put (type val, int ref)
        bool    exists (int ref)
        type    get (int ref)
```

**Figure 8. iteration_vector object**

**Attributes** `-iteration_list`
It is an ordered list which contains the instances of alive iterations. An alive iteration is an iteration which has not consumed and produced all its fine grain inputs and outputs.

**Methods** `-bool put (type val, int ref)`
This method is used to put the input value val at position ref in the FGSCC. Its algorithm can be summarized as follow:

1. get the older iteration which has no input value at position ref
2. if a such iteration does not exist create a new iteration and add it at the end of the list
3. put the value val at position ref in the iteration
4. refresh the iteration
5. if ageing variables has been computed put them in the next iteration. To perform that task the present algorithm is applied in a recursive way.

`-bool exists (int ref)`
This method is used to check if at least one iteration contains a value at position ref.

`-type  get (int ref)`
This method is used to get the older value at position ref in the `iteration_list`. Its algorithm can be summarized as follow:

1. get the older iteration which has not consumed output value at position ref
2. if a such iteration does not exist return 0
3. else get the value in the found iteration
4. if this iteration is now entirely consumed, remove it from the iteration_list

### 3.3 The automated flow

Figure 9 presents the automated flow we use to generate our FGSCC. The initial algorithmic description is transformed into a fine grain SFG thanks to the GAUT [7] SFG generator. It also performs the coarse grain to fine grain interface transformation. The dependencies matrix and the functions contained in the sets $F_a$ and $F_o$ are extracted from the analysis of the SFG. Then an eXtensible Markup Language (XML) generator transforms these information into an XML format. This representation is then transformed according to an eXtensible Stylesheet Language (XSL) transformation. The use of the couple XML, XSL allows to have a flexible code generation mechanism. At the moment we have an XSL transformation which generates a C code FGSCC, i.e. which generate the `iteration_list` and `iteration` objects. For more information about XML and XSL the reader is invited to consult [8].

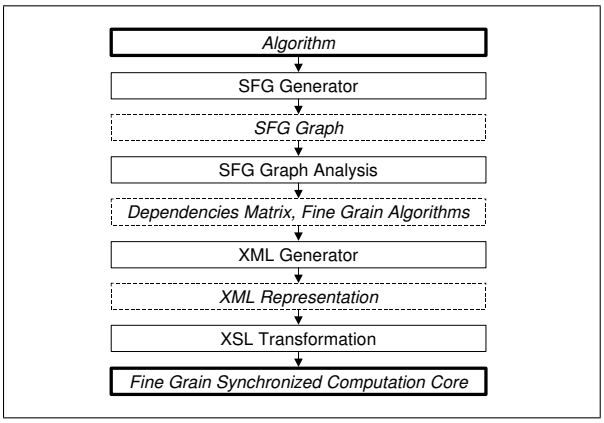


**Figure 9. the automated flow**

## 4 Examples

To illustrate the two key concepts of our FGSCC, (1) fine grain synchronization, i.e. the dependencie matrix (cf. figure 4), (2) the fine grain algorithmic functions of $F_a$ and $F_o$, i.e. fine grain computations (cf. sub section 3.1) , this section presents two pedagogical examples. The first one has no interest in the context of coarse grain to fine grain refinement but illustrates the ageing variable concept. The second one deals with the matrix product presented in figures 1 and 2. In a last third sub-section, we present the insertion of a matrix product FGSCC in an Electronic System Level (ESL) design tool named Cofluent Studio [1].

### 4.1 FIR Filter example

A N taps FIR filter has one input $x_n$, one output $y_n$ and N coefficients $h_i$. It has a N-1 memory effect. Its algorithmic expression is:

$$y_n = \sum_{i=0}^{N-1} h_i x_{n-i}$$

where $x_{n-i}$ is the value of the input $i$ algorithmic iterations before.
Let consider the case of a four taps FIR filter. The obtained dependencies matrix is:

$$\texttt{m\_dep}^i = \begin{array}{c|c:c:cc} & y_n^i & x_{n-1}^{i+1} & x_{n-2}^{i+1} & x_{n-3}^{i+1} \\ \hdashline x_n^i & true & true & false & false \\ \hdashline x_{n-1}^i & true & false & true & false \\ x_{n-2}^i & true & false & false & true \\ x_{n-3}^i & true & false & false & false \end{array}$$

The obtained $F_o$ set is:

$$y_n^i = h_0 x_n^i + h_1 x_{n-1}^i + h_2 x_{n-2}^i + h_3 x_{n-3}^i$$

The obtained $F_a$ set is:

$$\begin{cases} x_{n-1}^{i+1} = x_n^i \\ x_{n-2}^{i+1} = x_{n-1}^i \\ x_{n-3}^{i+1} = x_{n-2}^i \end{cases}$$

From the analysis of $M_d^i$ we can conclude (1) that the FIR filter computation core is able to compute the output $y_n^i$ as soon as $x_n^i, x_{n-1}^i, x_{n-2}^i, x_{n-3}^i$ are available (2) that the ageing variables $x_{n-1}^i, x_{n-2}^i, x_{n-3}^i$ are available as soon as $M_d^{i-1}$ is computed that is to say as soon as $x_n^{i-1}$ arrives, and so on. The reader can point

out the initialization problem: to have a working computation core, the ageing variables have an initial value in $\texttt{v\_in\_val}^0$ and their corresponding boolean in the presences vector $\texttt{v\_in\_pre}^0$ are set to true.

### 4.2 Matrix product example

The matrix product example (cf. figures 1 and 2) is an interesting example since it is a simple example of potential fine grain IO overlapping. Let consider the algorithm of figure 1 with $N = 2$. The obtained dependencies matrix is:

$$\texttt{m\_dep}^i = \begin{array}{c|cccc} & c_{11}^i & c_{12}^i & c_{21}^i & c_{22}^i \\ \hline a_{11}^i & true & true & false & false \\ a_{12}^i & true & true & false & false \\ a_{21}^i & false & false & true & true \\ a_{22}^i & false & false & true & true \\ b_{11}^i & true & false & true & false \\ b_{12}^i & false & true & false & true \\ b_{21}^i & true & false & true & false \\ b_{22}^i & false & true & false & true \end{array}$$

The obtained $F_a$ set is empty. The obtained $F_o$ set is:

$$\begin{cases} c_{11}^i = a_{11}^i b_{11}^i + a_{12}^i b_{21}^i \\ c_{12}^i = a_{11}^i b_{12}^i + a_{12}^i b_{22}^i \\ c_{21}^i = a_{21}^i b_{11}^i + a_{22}^i b_{21}^i \\ c_{22}^i = a_{21}^i b_{12}^i + a_{22}^i b_{22}^i \end{cases}$$

The analysis of the dependencies matrix and the set $F_o$ shows that the computations of $c_{11}^i, c_{12}^i, c_{21}^i, c_{22}^i$ are now synchronized on rows of the matrix A and columns of the matrix B. For example our fine grain computation core allow to produce $c_{11}$ as soon as the first row of A and the first column of B are available.

### 4.3 Matrix product integrated in an ESL design tool

Let consider the following refinement of a NxM matrix product C=A.B:

- it has two input ports: `a1_to_pm_refinement`, `a2_to_pm_refinement`, which carry vectors of size M. On the first one it receives the rows of the left hand side operand, on the second one it receives the columns of the right hand side operand.
- it has two output ports: `pm_refinement_01_to_c`, `pm_refinement_02_to_c`, which carry scalars. On the first one it produces the scalars of the result matrix which have an even row index, starting by column one, on the second one, it produces the scalars of the result matrix which have an odd row index, starting by the first column.

Let $r_i, i = 0..N-1$ be the lines of the left hand side operand, let $c_i, i = 0..N-1$ be the columns of the right hand side operand, let $s_{ij}, i = 0..N-1, j = 0..N-1$ be the scalar results of the matrix product.
In a traditional communication refinement approach, the refinement of a coarse grain matrix product to the matrix product specified above consists in refining the communication interfaces, i.e. matrix are sliced into vectors and scalars without refining the initial algorithm specification. Thus the computations remain coarse grain synchronized. Figure 10 presents a time line we obtained with Cofluent Studio [1] of a such refinement for N=3, M=5. We can point out there is no fine grain input output overlapping.
With our approach we can model all the potential fine grain synchronizations of the matrix product algorithm. Figure 11 presents a time line obtained in the same conditions than before, that is to say we emit the data in the following order: $r_0, c_0, r_1, c_1, r_2, c_2$. We can now observe the fine grain input output overlapping and thus can do finest input output order and timing analysis.

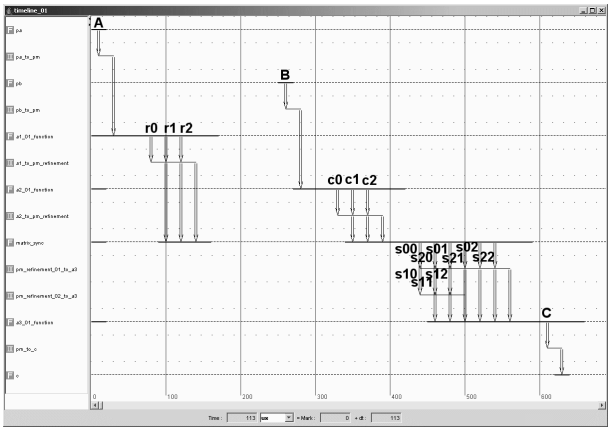


**Figure 10. time line refined matrix product**

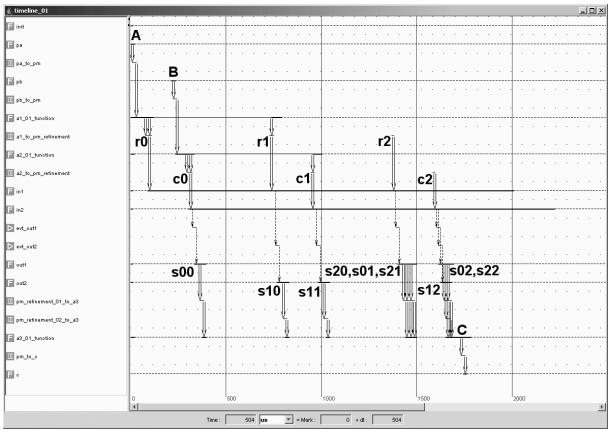


**Figure 11. time line refined matrix product with FGSCC**

## 5 Conclusions and Work in progress

In this paper we propose a computation core which can be used to model fin grain IO overlapping and iteration pipelining while preserving the initial algorithm functionality. A design flow has been developed to automate the generation of such core. We are now working on the integration of a such refinement in the ESL design tool Colfuent Studio [1] with the two following objectives: (1) interconnect fine grain synchronized algorithms to extract fine grain IO constraints which can be used to constrain an HLS tool [2] (2) model Register Transfert Level (RTL) components with an higher level of abstraction for the computations but with Cycle Accurate, Bus Accurate (CABA) interfaces.

## 6 Acronyms

**DSP** Digital Signal Processing

**RTL** Register Transfert Level

**HLS** High Level Synthesis

**IO** Input Output

**FGSCC** Fine Grain Synchronized Computation Core

**SFG** Signal Flow Graph

**XML** eXtensible Markup Language

**XSL** eXtensible Stylesheet Language

**ESL** Electronic System Level

**CABA** Cycle Accurate, Bus Accurate

## References

[1] Cofluent design, cofluent studio, $http : //www.cofluentdesign.com$..

[2] P. Coussy, A. Baganne, and E. Martin. Communication and timing constraints analysis for ip design and integration. In *VLSI-SOC*, pages 38–43, 2003.

[3] W. Ecker and M. Hofmeister. The design cube: a new model for vhdl designflow representation. In *EURO-DAC '92: Proceedings of the conference on European design automation*, pages 752–757, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[4] D. Filo, D. Ku, C. Coelho, and G. D. Micheli. Interface optimization for concurrent systems under timing constraint. In *IEEE Transactions on VLSI Systems*, 1993.

[5] D. Gajski. The structure of a silicon compiler. In *Proceedings of IEEE ICCD*, 1987.

[6] A. Jantsch, S. Kumar, and A. Hemanimumi. The rugby model: a conceptual frame for the study of modelling, analysis and synthesis concepts of electronic systems. In *DATE '99: Proceedings of the conference on Design, automation and test in Europe*, page 54. ACM Press, 1999.

[7] E. Martin, O. Sentieys, H. Dubois, and J. Philippe. Gaut: An architectural synthesis tool for dedicated signal processors. In *EURO-DAC*, 1993.

[8] A. Skonnard. *Essential XML Quick Reference: A Programmer's Reference to XML, XPath, XSLT, XML Schema, SOAP, and More*. Addison-Wesley, 2001.

[9] F. Thabet, J. L. Goff, P. Coussy, and E. Martin. A methodology for timing and structural communication refinement in dsp systems. In *International Conference on Microelectronics (ICM)*, 2004.