Automatic implementation of affine iterative algorithms: Design flow and communication synthesis

Alessandro Marongiu a,*, Paolo Palazzari b

a Electronic Engineering Department, University "La Sapienza", Via Eudossiana, 18 00184 Rome, Italy
b ENEA HPCN project, C.R. Casaccia, Via Anguillarese, 301 S.P. 100 00060 S. Maria di Galeria, Rome, Italy

Abstract

This work addresses the automatic generation of a parallel architecture, described by the skeleton of a VHDL program at the Register Transfer Level, starting from some high level expression specified by the LIGHTC language, a subset of the ANSI C. The synthesis methodology is related to the affine iterative algorithms, which arise in many scientific and technological applications and can be described by the System of Affine Recurrence Equations model (SARE). After reviewing basic theory on SARE and their synthesis, we present a new methodology to synthesize communications: the proposed communication synthesis methodology, based on a propagation technique which uses the concept of Integral Hilbert Vector basis, allows the generation of inexpensive (in terms of time and silicon area) communication patterns. The communication synthesis methodology is part of a global design flow: the main steps of the design flow are shortly reviewed and the whole design methodology is explained by a simple test case, the matrix–matrix product. © 2001 Published by Elsevier Science B.V.

1. Introduction

The numerical solution of many physical and engineering problems originates severe demands for today computational systems. Development of specialized parallel HW can be a viable solution to satisfy such huge computational demands (see the experiences of the APE [1] and GRAPE [2] projects launched, respectively, by Italian and Japanese physicists). The main drawback of this approach is connected to the long times spent in the design and implementation phases. Such development times are due to the necessity of designing from scratch a new specialized parallel architecture, discovering algorithm parallelism and solving all the issues connected to the low level details. Commercial CAD tools (Synopsys, Cadence) greatly support designers in the low level synthesis phase. On the contrary, extraction of parallelism is still left to the experience of the designer and its management (allocation, scheduling and synchronization) is partially supported by specialized CAD tools like, for example, Cadence Visual Architect which accepts algorithm description given by a data-flow graph and allocates and schedules it onto a given set of computing resources. As data-flow graphs are a powerful representation for a broad class of algorithms in control-dominated real-time applications, a lot of efforts have been spent in developing such type of CAD tools.

Another class of algorithms, orthogonal to the control-dominated, is constituted by the data-dominated applications, often arising in physics and engineering. The peculiarity of many ‘number crunching’
algorithms relies on the repetition of the same set of computations on large data domains (data-dominated iterative algorithms). Such problems, in real cases, are too large to be described by data-flow graphs: to give an idea, the multiplication of two $(1000 \times 1000)$ matrices generates a graph with $2 \times 10^9$ nodes. Data-flow graphs fail to deal with such algorithms because they are not able to exploit problem regularity: in fact, they need to explicitly represent each instance of each operation. As a consequence, CAD tools based on data-flow graphs cannot be definitively used for such data-dominated scientific applications.

A way to describe a restricted class of iterative algorithms, the affine iterative algorithms, is represented by the System of Affine Recurrence Equations (SARE) [3]. In affine iterative algorithms constant parameters can be introduced, the control is static, loop bounds are affine expressions of outer loop indices and algorithm parameters, indices of array variables are affine expressions of loop control indices and algorithm parameters. Affine iterative algorithms represent a large class of programs arising in such fields as linear algebra, numerical analysis, signal processing, neural networks, PDE integration. Due to its geometrical interpretation [3], SARE model allows a compact and closed form representation of affine iterative algorithms. The set of computations is described by equations defined over iteration spaces which are bounded sub-domains of the integer lattice.

The relevance of SARE model in scientific applications, along with the demand of high computational power, has driven the research attention toward automatic synthesis methods which, starting from high level description of an iterative algorithm, allow the automatic generation of the skeleton of a parallel digital circuit implementing the algorithm [4–9]. Particularly, our methodology [10], transforms the SARE algorithm into the architectural description of a synchronous system. VHDL [11] language is used to describe system architecture, by specifying: (a) the basic computing blocks (combinatorial functions); (b) their interconnection links with the associated delays used to enforce data synchronization; (c) I/O ports; (d) control Finite State Machine (FSM) to manage the whole system.

The synthesis of SARE algorithms, in the general case, has not been yet completely addressed. Among the other difficulties, one of the main obstacles to implement SARE model in its more general expression, is the generation of an efficient, low cost communication pattern among computing blocks. Only for the sub-class of Systems of Uniform Recurrence Equations (SURE) an efficient interconnection pattern can be easily and univocally defined [5]. In the general case of SARE algorithms, communication synthesis still suffers from lack of generality because communication pattern cannot be univocally defined.

The only attempt to give systematic results to the communication synthesis problem in the general SARE case was presented in [13–15] and will be discussed in detail in Section 6.

A solution to communication synthesis is given also in [9] for a specific iterative algorithm, but the general case is not faced.

It is a matter of fact that system design is affected by the communication pattern and, among the others, two main factors determine the efficiency of the communications. First, a regular communication pattern is always preferable to a non regular one because regular structures are easier to be implemented; secondly, communications among neighbor computing blocks (local communications) are less costly than non-local communications because routing is simpler and less silicon area is involved for their physical implementation. Due to previous considerations, architectures to be generated are often constrained to communicate data in a regular way only among neighbor computing blocks [9], thus significantly restricting both the design space and the class of affordable problems.

In this paper we describe the design flow to translate high level system specifics, expressed by means of a subset of the C language, into a VHDL description (at Register Transfer Level) of a parallel architecture which satisfies original specifics and is optimized according to some cost measure (silicon area, speed). Particularly, the focus of this work is on synthesis of communications in the general case of SARE algorithms, presenting a methodology that produces regular and local communication patterns. The proposed methodology is based on the individuation of the set of computations using the same data (data utilization set) and of the computation that generates such a data (data source computation).

The set of communications needed to transfer data from the source computation to the whole utilization
set is split into a pair of communication phases: the source value is transmitted to a computation in the data utilization set, then it is propagated by some elementary communications. The elementary communications involve neighbor computing blocks (locality) and must be able to propagate the source data to the whole data utilization set. The underlying theory, used to perform such a decomposition, is based on the Integer Hilbert Basis Vectors [16]. Such a basis is able to cover a set of integral points through a positive integral combination of the basis vectors (regularity).

The work is organized as in the following: in next section the global design flow is presented. After recalling some basic geometric definitions, the SARE model is reviewed and allocation and scheduling functions are introduced. The remaining parts of the paper are devoted to introduce a methodology to be used to synthesize communications. Finally the whole design flow is applied to a simple test case (the matrix–matrix product).

2. Design flow

In this section a brief review of the whole design flow described in [10] is presented (see Fig. 1). The proposed design flow drives the system designer from the high level specifics of the problem toward the physical realization. This is accomplished through a sequence of automatic steps that transform the high level specifics into a parallel system architecture. The output of the design flow is the skeleton of a VHDL (RTL) program that is processed by standard synthesis tools.

Transformation steps applied to the high level description are correct-by-construction, thus allowing an error-free design (once specifics are ensured to be error-free) and avoiding all error prone and cumbersome low level details.

The specifics are given by means of an affine iterative program written in the LIGHTC [17] language (a subset of the ANSI C). Standard C tools, such as C compilers and debuggers, may be used to verify, in a fast and straightforward way, the correctness of the specifics. As the subsequent transformation steps are proven to be correct by construction, the final design inherits the specifics correctness.

A LIGHTC program:
– is composed by a sequence of non perfectly nested loops;
– loop bounds (lower l and upper u) must be affine expressions of algorithm parameters and outer loop indices;
– the increment of each loop index is 1;
– data structures are multi-dimensional arrays;
– the only admissible instructions are assignment to a multi-dimensional variable;
– no conditional statements, goto and while instructions are allowed.

The LIGHTC compiler is used to extract the SARE equivalent expression. LIGHTC compiler is based on the technique proposed in [18]. According to it, the LIGHTC program is transformed into a single assignment form by renaming left hand variables of the assignment statements and expanding the indices of renamed variables according to the number of surrounding loops. Once the program has been transformed into the single assignment form, an analytical dependence analysis is performed in order to write the recurrence equations, thus defining the equivalent SARE algorithm. The next step of design flow foresees the allocation and scheduling of SARE computations onto a set of Virtual Functional Units (VFU). Allocation and scheduling are given through the timing and allocation functions which are optimized through a heuristic approach similar to the one presented in [19]. The allocation step defines a set of computing resources (VFU) whose number matches the problem size (algorithm parameters). For real problems, the set of computing resources is too large to be physically implemented so, in a first phase, virtual computing resources are considered. An allocation function assigns SARE computations to different virtual computing resources, thus data communication paths must be provided to allow data passing among dependent computations. The communication pattern, derived through the methodology presented in this work, defines a set of local and regular interconnections. The set of interconnected VFU constitutes the system data path. Finally, scheduling allows the synchronization of the whole system and is enforced by a Finite State Machine (FSM) used to control the data path.

A successive clustering step, based either on the Locally Serial–Globally Parallel (LSGP) or the Locally Parallel–Globally Serial (LPGS) paradigms, compacts
the virtual resources onto a limited number of time-shared physical resources. Clustering step introduces a small set of interconnected Physical Functional Units (PFU) which define the system data path to be implemented. The scheduling of the PFU is derived from the VFU scheduling through the clustering technique. Details on clustering techniques can be found, for example, in [10,20]. In the last step the whole system (data path, control FSM, I/O ports) is automatically encoded through the VHDL language.

3. Notations and basic definitions

Throughout the paper the following definitions and notations will be used.

The identity matrix is denoted as $Id$. The following definition and properties can be found in [16,21–23].

**Definition 1** (Span and kernel). Given a matrix $A$ – the span of $A$ $(\text{span}(A))$ is the set of point spanned by its columns;
– the kernel of $A$ ($\text{Ker}(A)$) is the set of points $x$ such that $Ax = 0$.

Definition 2 (Convex combination). Given a set of vectors $\{x_i\}$ and a set of scalar coefficients $\{a_i\}$, the convex combination of the given vectors $x_i$, is defined as $\sum_i a_i x_i$ with $a_i \geq 0$ and $\sum_i a_i = 1$.

Definition 3 (Vertex). A vertex of a given set of points $K$ is a point which cannot be expressed as a convex combination of other points in $K$.

Definition 4 (Parameterized polyhedron). A $N$-dimensional parameterized polyhedron is defined as $\{z \in \mathbb{Z}^N \mid Az + Pp + b \geq 0\}$ where $A$ and $P$ are integral constant matrices, $b$ is an integral constant vector and $p$ is a parameter vector. A parameterized polyhedron is the set of integer $N$-vectors $z$ which satisfy the set of inequalities $Az + Pp + b \geq 0$. Every parameterized polyhedron is convex.

Definition 5 (Parameterized polytope). A $N$-dimensional parameterized polytope is a bounded parameterized polyhedron.

Property 1 (Dual representation of a parameterized polytope). A $N$-dimensional parameterized polytope has a dual representation defined as the convex combination of its vertices $V_i$, i.e. $\{z \in \mathbb{Z}^N \mid z = \sum_i v_i \text{ with } v_i \geq 0 \text{ and } \sum_i v_i = 1\}$.

Property 2 (Vertices of parameterized polytope). The vertices $V$ of a parameterized polytope are affine expression of its parameters, i.e. $V = Dp + g$ where $D$ is a constant matrix and $g$ is a constant vector.

Definition 6 (Cone). A set $C$ is a cone with apex 0 if for all $x \in C$, $\lambda x \in C$ with $\lambda \geq 0$. Given a set of vectors $\{h_i\}$, the set $\{x \mid x = \sum_i a_i h_i, \text{ with } a_i \geq 0\}$ is the cone, with apex 0, generated by $\{h_i\}$. A set $C$ is a cone with apex $x_0$ if $\{y \mid y = x - x_0 \text{ with } x \in C\}$ is a cone with apex 0. A cone $C$ with apex $x_0$ is pointed if $x_0$ is a vertex of $C$.

4. SARE model and its synthesis: a theoretical review

This section briefly presents SARE model and its synthesis methodology. We follow the general framework of [3–8].

4.1. SARE model

A SARE is described by $k$ equations:

$$\begin{align*}
\ldots
X(z) &= f_i(\ldots, Y[\rho(z)], \ldots) \quad \text{with } z \in I_i \\
\ldots
i &= 1, 2, \ldots, k,
\end{align*}$$

where:

– $X$ and $Y$ are multi-dimensional array variables. Variables appearing on the left side of (1) are intermediate/final results of the algorithm and their values are computed at run-time. Result variables are always $N$-dimensional arrays. Variables appearing only on the right side of (1) are the input variables of the algorithm.

– $z = (z_1, z_2, \ldots, z_N)^T$ is the $N \times 1$ iteration vector. $X(z)$ represents $X[z_1, z_2, \ldots, z_N]$ value of $X$ array. The computation $C_X(z)$, which computes $X(z)$, is represented by $z$.

– $I_i$ is the iteration space related to equation $i$, i.e. it is the set of indices $z$ where Eq. (1) is defined. $I_i$ is described through a $N$-dimensional parameterized polytope, i.e. $I_i = \{z \in \mathbb{Z}^N \mid A_i z + P_i p + b_i \geq 0\}$, where $p$ is the algorithm parameter vector. The union of all the $I_i$ is the global iteration space polytope $I$. SARE is a single assignment model, so equations assigning the same array variable, i.e. equations having the same left-hand side variable, must be defined onto disjointed iteration spaces.

– $\rho(z) = Rz + r$ is the index mapping function. It defines a flow dependence between $\rho(z)$ and $z$.

– $f_i$ is a single valued function which computes $X(z)$.

Eqs. (1) completely describe the iterative algorithm. For each $z$, i.e. for each iteration, $X(z)$ is computed using $f_i$ which needs the operands $(\ldots, Y[\rho(z)], \ldots)$. If $Y$ is an input variable, the value $Y[\rho(z)]$ must be supplied from the outside. If $Y$ is a result variable, the
The value $Y[\rho(z)]$ is supplied from $C_Y[\rho(z)]$, i.e. from the computation of $Y$ occurred in iteration $\rho(z)$. In such a case, $X(z)$ depends on $Y[\rho(z)]$.

Algorithm dependencies are modeled using the index mapping function $\rho(z)$ which, for each computation point $z$, returns the indices of the operands of $f$ needed to compute $C_X(z)$. Only result variables introduce dependencies among computing points. Input variables do not introduce any dependence because they are supplied from the outside. Dependence relations occurring in a SARE algorithm are modeled through dependence vectors:

**Definition 7 (Dependence vectors).** Given a SARE equation (1), dependence vectors associated to the result variable $Y$ are:

$$
d_{Y,\rho}(z) = z - \rho(z) = z - Rz - r = (Id - R)z - r \quad \forall z \in I_i. \quad (2)
$$

Dependence relation between $X(z)$ and $Y[\rho(z)]$ is given by the dependence vector, i.e. by the difference between the coordinate of the current point $z$ and the point $\rho(z)$ where $Y[\rho(z)]$ has been computed. Dependence vector connects the source computation point $\rho(z)$ with the destination computation point $z$.

In the general case, dependence vectors depend on $z$. The special case of uniform dependence vectors is obtained when they do not depend on $z$, i.e. when they are constant. From (2), it is easy to verify that uniform dependence vectors arise when $R = Id$.

As a consequence, a System of Affine Recurrence Equations is uniform (System of Uniform Recurrence Equations — SURE) when the equality $R = Id$ is verified for all the index mapping functions. In such a case, from (2), $d_{Y,\rho}(z) = -r$.

SARE, in a very compact way, contains all the information of the data-flow graph model:
- each computing point $z$ of the SARE algorithm defines a node of the data-flow graph;
- each dependence vector defines an edge between the two corresponding graph nodes.

As explanatory example, we consider the following SARE:

1. $A(z_1, z_2) = f_1(B(z_1))$ with $z = (z_1, z_2) \in I_1 = [(z_1, z_2) \in Z^2 | 0 \leq z_1 \leq 3, z_2 = 0]$;
2. $A(z_1, z_2) = f_2(A(z_1, z_2 - 1), A(z_2 - 1, z_2 - 1))$ with $z = (z_1, z_2) \in I_2 = [(z_1, z_2) \in Z^2 | 1 \leq z_1 \leq 3, 1 \leq z_2 \leq 1]$.

Previous SARE involves two variables: the one-dimensional input variable $B$ and the two-dimensional result variable $A$. Computations performed by the SARE are connected to the points of the polytope defined on the integer two-dimensional lattice. Eq. (1) does not introduce any dependence because only the input variable $B(z_1)$ is involved. Eq. (2) introduces two dependence relations caused by $A(z_1, z_2 - 1)$ and $A(z_2 - 1, z_2 - 1)$. The first dependence introduces the following index mapping function

$$
\begin{pmatrix}
z_1 \\
z_2 - 1
\end{pmatrix}
= \begin{pmatrix}
1 & 0 & 0 \\
0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
z_1 \\
z_2
\end{pmatrix}
+ \begin{pmatrix}
0 \\
-1
\end{pmatrix}
= Idz + r
$$

and it is uniform because $R = Id$. The corresponding uniform dependence vector is $d = (0 1)^T$. The second dependence introduces the following index mapping function

$$
\begin{pmatrix}
z_2 - 1 \\
z_2 - 1
\end{pmatrix}
= \begin{pmatrix}
0 & 1 & 0 \\
0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
z_1 \\
z_2
\end{pmatrix}
+ \begin{pmatrix}
-1 \\
-1
\end{pmatrix}
= Rz + r
$$

and is a general affine dependence which defines dependence vectors depending on $z$. Fig. 2 depicts the iteration space along with the dependence vectors. It is worthwhile to note the similarity with a data-flow graph.
4.2. SARE synthesis

Synthesizing a SARE requires
– the definition of a set of computing resources;
– the allocation of the computations onto the computing resources;
– the scheduling of the allocated computations.
A lot of methodologies have been developed to schedule and allocate a given set of computation onto a given set of computing resources (e.g., allocation and scheduling of data-flow graphs [24–28]) and they can be used, with little or no changes, also in the SARE case.

The geometrical nature of the SARE model has allowed the adoption of different methodologies which fully exploit the characteristics of SARE. Since the work of Lamport [29], synthesis of iterative algorithms has been performed as a geometrical mapping from the iteration space to a processor-time space [8]. Such projective methodologies evolved together with the advances on integer linear programming theory.

SARE synthesis is thus performed through a space-time transformation which, for each computation $C_X(z)$, i.e. for each $z$ in the iteration space, gives the time step when (scheduling) and the computing resource where (allocation) $C_X(z)$ is executed.

As SARE computations are defined on a subset of the integer $N$-dimensional lattice, the space-time transformation is a mapping from $Z^N$ to time and space coordinates.

Allocation and scheduling require the definition of the set of computing resources that perform computations. We assume a set of Virtual Functional Units (VFU) regularly distributed on the $m$-dimensional integer lattice. Each VFU is univocally individuated by a set of $m$ coordinates $vfu = (vfu_1, vfu_2, \ldots, vfu_m)^T$. We consider Virtual FU because their number is usually too large to allow the implementation of each VFU as a Physical FU (PFU). According to the design flow, the successive clustering step collapses several VFU on the same PFU.

Space-time transformation is composed of:
– the timing function $t = t(z)$, which returns the time $t$ when $C_X(z)$ starts its execution;
– the allocation function $vfu = vfu(z)$, which returns the spatial coordinates $vfu$ of the VFU where $C_X(z)$ is executed.

$t(z)$ and $vfu(z)$ must preserve the semantics of the algorithm, i.e. dependence relations have to be respected, and must be compatible, i.e. no more than one computation must be scheduled at the same time on the same VFU.

We choose $t(z)$ as a $n$-dimensional affine integer function. We have:

$$t = (t_1 t_2 \ldots t_n)^T = t(z) = A \cdot z + \alpha,$$

where $A$ is an integral $n \times N$ matrix and $\alpha$ is an integer $n \times 1$ vector. $t(z)$ values are $n$-vectors representing a $n$-dimensional time and they must be totally ordered to univocally determine the execution time of a given $C_X(z)$. We adopt the lexicographical ordering ($\gg$).

The following theorem, demonstrated by the authors in [4], states the constraint on timing function to preserve dependence relations.

**Theorem 1** ($n$-dimensional timing function admissibility). A $n$-dimensional $t(z)$ is admissible if and only if, for every dependence vector $d$, $\Delta d \gg 0$.

Allocation function must allocate a given set of computations onto the VFU set. Allocation function $vfu(z)$ must return the coordinates of the VFU executing computation $C_X(z)$. We choose $vfu(z)$ as a $m$-dimensional integral affine function:

$$vfu = (vfu_1, vfu_2, \ldots, vfu_m)^T = vfu(z) = \Sigma \cdot z + \beta,$$

where $\Sigma$ is an integral $m \times N$ matrix and $\beta$ is an integral $m \times 1$ vector.

From the allocation function definition, the VFU set is given by

$$\{ VFU(vfu) \mid vfu = \Sigma z + \beta \ \forall z \in I \}.$$
signals are generated through the data path control circuitry which is explained below. As an example, Fig. 3 depicts a VFU implementing the following skeleton of SARE:

- As in [5,8,30], timing function dimension $n$ and allocation function dimension $m$ must respect the constraint $n + m = N$.
- Composition of $t(z)$ and $vfu(z)$ leads to the space-time transformation function $T(z)$:
  \[
  \begin{pmatrix}
  t \\
  vfu
  \end{pmatrix} = T(z) = \begin{pmatrix}
  \Lambda \\
  \Sigma
  \end{pmatrix} z + \begin{pmatrix}
  \alpha \\
  \beta
  \end{pmatrix}.
  \]
  \[
  T(z) \text{ is an integral } N \times N \text{ matrix (composition of } \Lambda \text{ and } \Sigma \text{) and } \gamma \text{ is an integral } N \times 1 \text{ vector (composition of } \alpha \text{ and } \beta \text{).}
  \]

- Compatibility between timing and allocation functions is assured if $T$ is not singular (see [8] for the proof).

- Allocation and scheduling of the SARE is performed through the application of the space-time transformation $T(z)$ to the iteration space polytope $I_i$, which is transformed into the target iteration space polytope $I_{T,i}$, defined as:
  \[
  I_{T,i} = \left\{ \begin{pmatrix}
  t \\
  vfu
  \end{pmatrix} \in T(z), \forall z \in I_i \right\}.
  \]

- As the application of space-time transformation maps original computation points $z$ onto space-time points $(z_{vfu})$, index mapping functions are also transformed according to $T(z)$ as in the following:

**Theorem 2.** Given an index mapping function $\rho : z \rightarrow Rz + r$ it is transformed by $T(z)$ into

\[
\rho_{T} : \begin{pmatrix}
  t \\
  vfu
  \end{pmatrix} \rightarrow R_{T} \begin{pmatrix}
  t \\
  vfu
  \end{pmatrix} + r_{T},
\]

where:
- $R_{T} = TRT^{-1}$ and $r_{T} = -TRT^{-1} \gamma + Tr + \gamma$ if $\rho(z)$ is related to a result variable;
- $R_{T} = RT^{-1}$ and $r_{T} = -RT^{-1} \gamma + r$ if $\rho(z)$ is related to an input variable.

See [8] for the proof.

Application of the space-time transformation maps the original SARE defined on $I$ into a target SARE defined over $I_{T}$:

\[
X(vfu) = fi,...,Y[\rho_{T}(vfu)]....
\]

- with $(t vfu) \in I_{T,i}$

\[
\ldots
\]

\[
i = 1, 2, \ldots, k.
\]

Coming back to the explanatory example, we transform the original SARE with:

- a 1-dimensional timing function $t = t(z) = (1, 1)z$;
- a 1-dimensional allocation function $vfu = vfu(z) = (1, 0)z$.

The resulting space-time transformation is obtained by composition of timing and allocation function, i.e.

\[
\begin{pmatrix}
  t \\
  vfu
  \end{pmatrix} = \begin{pmatrix}
  t_{1} \ vfu_{1}
  \end{pmatrix} = T(z) = Tz + \gamma
\]

\[
= \begin{pmatrix}
  1 & 1 \\
  1 & 0
  \end{pmatrix} \begin{pmatrix}
  z_{1} \\
  z_{2}
  \end{pmatrix} + \begin{pmatrix}
  0 \\
  0
  \end{pmatrix}.
\]

The resulting target SARE, after application of previous space-time transformation, is:
We choose \( CX \) formed into target SARE is shown in Fig. 4.

1. \( A(t_1, vfu_1) = f_1(B(vfu_1)) \) with
   \[
   \begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}
   = \begin{pmatrix}
   t_1 \\
   vfu_1
   \end{pmatrix} \in I_T
   = \begin{cases}
   \begin{pmatrix}
   t_1 \\
   vfu_1
   \end{pmatrix} & 0 \leq vfu_1 \leq 3, \ t_1 = vfu_1
   \end{cases}.
   \]

2. \( A(t_1, vfu_1) = f_2(A(t_1 - 1, vfu_2), A(2t_1 - 2vfu_1 - 2, t_1 - vfu_1 - 1)) \) with
   \[
   \begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}
   = \begin{pmatrix}
   t_1 \\
   vfu_1
   \end{pmatrix} \in I_T
   = \begin{cases}
   \begin{pmatrix}
   t_1 \\
   vfu_1
   \end{pmatrix} & 1 \leq vfu_1 \leq 3, \ vfu_1 + 1 \leq t_1 \leq 2vfu_1
   \end{cases}.
   \]

The space-time transformation defines 4 VFUs; the target SARE is shown in Fig. 4.

Because \( T(z) \) maps \( z \) into \( \begin{pmatrix}
   t \\
   vfu
   \end{pmatrix} \), \( C_X(z) \) is transformed into \( C_X(\begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}) \); computation \( C_X(z) \) is executed by VFU(\( \begin{pmatrix}
   t \\
   vfu
   \end{pmatrix} \)) scheduled at time \( t = (t_1t_2 \ldots t_n)^T \).

In order to scan multidimensional time in a lexicographical way, we use a scalar \( tick(t) \) function with the following property:

\[ tick(t_1) < tick(t_2) \Leftrightarrow t_1 \ll t_2. \]

We choose \( tick(t) \) as in [30]:

\[
tick(t) = t_n + c_n t_{n-1} + c_n c_{n-1} t_{n-2} + \cdots + c_n c_{n-1} \cdots c_2 t_1,
\]

where

\[ c_i = \max(t_i) - \min(t_i) + 1 \quad \text{for} \ i = 1, 2, \ldots, n. \quad (11)\]

Scanning \( tick(t) \) in the natural integer order causes time \( t \) to be lexicographically scanned. From \( tick \) definition (10), the completion time \( (CT) \) of the circuit is

\[ CT = c_1c_2 \ldots c_n. \quad (12)\]

A conceptual scheme of a circuit to convert \( tick(t) \) into \( t \) is shown in Fig. 5, where each block is a mod(\( c_i \)) counter which is incremented at each clock pulse when \( EN = 1 \). When the counter reaches the value \( c_i \), \( END = 1 \) and the counter is reset.

5. Communication synthesis

In the SARE model computations are not mutually independent. After applying the space-time transformation, computations are distributed among different VFU which must communicate. Communicating values among VFU involves both links, connecting the communicating VFU, and synchronizations. In fact, source data is produced at time \( t_0 \) in VFU(\( vfu_0 \)) and is used at a later time \( t_1 \) in VFU(\( vfu_1 \)): a link connecting VFU(\( vfu_0 \)) and VFU(\( vfu_1 \)) with an associated synchronizing delay \( tick(t_1 - t_0) \) must be provided.

The following definition of communication is given:

Definition 8 (Communication). Communication is defined as the couple \((\text{interconnection–delay})\) used to transfer and synchronize a data value from the source/producer VFU to the destination/consumer VFU.

Here

- interconnection defines the link between source and destination VFU;
- time delay provides the synchronization between the producing and consuming time.

As communications arise from computation data dependencies, they are related to the transformed index mapping functions \( \rho_T(\begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}) \), as defined in (8).

Given the transformed SARE, computation of \( X(\begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}) \) is performed in VFU(\( vfu_1 \)) and occurs at time \( t \). In order to perform such a computation, its operands \((\ldots , Y(\rho_T(\begin{pmatrix}
   t \\
   vfu
   \end{pmatrix})), \ldots ) \) are needed. \( \rho_T(\begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}) \) returns:

- the time \( t_0 \) and the VFU(\( vfu_0 \)) in which the result value \( Y(\begin{pmatrix}
   t \\
   vfu
   \end{pmatrix}) \) has been computed. In this case
a communication is needed to transfer $Y(t_0vfu_0)$, produced in VFU(vfu) at time $t_0$, to VFU(vfu) at time $t$. So the communication is given by the link from $vfu_0$ to $vfu$ and the delay $tick(t-t_0)$.

- the index $z_0$ of the input value $Y(z_0)$. In this case a communication is needed to transfer the input value $Y(z_0)$ from the input port of $Y$ to VFU(vfu) at time $t$. So the communication is given by the link from the input port to VFU(vfu) with a null associated delay.

A formal definition of the sets of source and consumer points is given in [3] and is recalled in the following:

**Definition 9 (Emission set).** The emission set related to the use of a result variable $Y$ is defined as:

\[
\text{Emit}_i(Y) = \left\{ \left( t_{vfu}, t_0vfu_0 \right) \mid \exists \left( t_{vfu}, t_0vfu_0 \right) \text{ s.t. } \rho_T(t_{vfu}) = \left( t_{vfu}, t_0vfu_0 \right), \left( t_{vfu}, t_0vfu_0 \right) \in I_{T,i} \right\} \equiv \rho_T(I_{T,i}).
\]

The emission set related to the use of an input variable $Y$ is:

\[
\text{Emit}_i(Y) = \left\{ (z_0) \mid \exists \left( t_{vfu}, t_0vfu_0 \right) \text{ s.t. } \rho_T(t_{vfu}) = z_0, \left( t_{vfu}, t_0vfu_0 \right) \in I_{T,i}, \right\} \equiv \rho_T(I_{T,i}).
\]

The emission set, related to the transformed index mapping function $\rho_T(t_{vfu})$, defines the set of all the computations which produce a value that is used as operand in $f_1(...) Y(\rho_T(t_{vfu}))[...].$ In a similar way, when $Y$ is an input variable, the emission set, related to the transformed index mapping function $\rho_T(t_{vfu})$, defines the set of all the indices addressing the input ports containing the values used as operand in $f_1(...) Y(\rho_T(t_{vfu}))[...].$

**Definition 10 (Utilization set).** The utilization set related to the use of a result variable $Y(t_0vfu_0)$ is defined as:

\[
\text{Util}_i(Y(t_0vfu_0)) = \left\{ \left( t_{vfu}, t_0vfu_0 \right) \in I_{T,i} \mid \rho_T(t_{vfu}) = t_0vfu_0 \right\}.
\]

The utilization set related to the use of an input variable $Y(z_0)$ is defined as:

\[
\text{Util}_i(Y, z_0) = \left\{ \left( t_{vfu}, t_0vfu_0 \right) \in I_{T,i} \mid \rho_T(t_{vfu}) = z_0 \right\}.
\]

The utilization set related to a result (input) variable contains all the points $(t_{vfu}, t_0vfu_0)$ which use $Y(t_0vfu_0)(Y(z_0))$ to perform the computation.

From Definition 10, utilization set is the intersection between the iteration space $I_{T,i}$ and the solutions of the equation

\[
\rho_T(t_{vfu}) = R_T(t_{vfu}) + r_T = t_0vfu_0.
\]

When $R_T$ has full rank, previous equation has one solution and, hence, utilization set contains exactly one point. If $R_T$ has not full rank, utilization set contains more than one point; particularly, utilization set lies on $\text{Ker}(R_T)$. Basis vectors of $\text{Ker}(R_T)$ are called utilization vectors.

In Figs. 6 and 7 the emission and the utilization sets of the SARE considered in Section 4 are depicted.
Each computing point in the emission set (source point) univocally defines the corresponding utilization set (consumer points). Utilization set is a parameterized polytope with parameters $p$ (the algorithm constant parameters) and $(t_0, vfu_0)$, so each point $(t_0, vfu_0)$ defines a different utilization set. Because vertices of a parameterized polytope are affine expressions of its parameters (see Property 2), vertices of utilization set are affine expression of both $(t_0, vfu_0)$ and $p$ as in

\[
V = D\begin{pmatrix} t_0 \\ vfu_0 \end{pmatrix} + Ep + g.
\]

According to space-time transformation, $V$ can be written as

\[
V = \begin{pmatrix} V^t \\ V^{vfu} \end{pmatrix} = \begin{pmatrix} D^t \\ D^{vfu} \end{pmatrix}\begin{pmatrix} t_0 \\ vfu_0 \end{pmatrix} + \begin{pmatrix} E^t \\ E^{vfu} \end{pmatrix}p + \begin{pmatrix} g^t \\ g^{vfu} \end{pmatrix},
\]

where $D^t$ is an $n \times N$ matrix and $D^{vfu}$ a $m \times N$ matrix.

Previous considerations and definitions are valid also in the case of utilization set related to an input variable with the obvious substitution of $(t_0, vfu_0)$ with $z_0$.

Following Definition 7, we introduce dependence vectors in the transformed space-time coordinates:

**Definition 12 (Transformed dependence vector).** The transformed dependence vector associated to a result variable $Y$ is

\[
d_{Y, \rho T}(\begin{pmatrix} t \\ vfu \end{pmatrix}) = \begin{pmatrix} t \\ vfu \end{pmatrix} - \rho T\begin{pmatrix} t \\ vfu \end{pmatrix} = (Id - R_T)\begin{pmatrix} t \\ vfu \end{pmatrix} - r_T
\]

\[
\forall (\begin{pmatrix} t \\ vfu \end{pmatrix}) \in I_{T,i}.
\]

Given $(\begin{pmatrix} t \\ vfu \end{pmatrix}) \in I_{T,i}$, $d_{Y, \rho T}(\begin{pmatrix} t \\ vfu \end{pmatrix})$ returns the dependence vector between computations $C_X(\begin{pmatrix} t \\ vfu \end{pmatrix})$ and $C_Y(\rho T(\begin{pmatrix} t \\ vfu \end{pmatrix}))$. According to the definitions of emission and utilization sets, dependence vectors connect the source points $\rho T(\begin{pmatrix} t \\ vfu \end{pmatrix}) = (\begin{pmatrix} t \\ vfu \end{pmatrix}) \in \text{Emit}_i(Y)$ to consumer points $(\begin{pmatrix} t \\ vfu \end{pmatrix}) \in \text{Util}_i(Y, (\begin{pmatrix} t_0 \\ vfu_0 \end{pmatrix}))$. In the following, when there is not ambiguity, $d(\begin{pmatrix} t \\ vfu \end{pmatrix})$ will be used instead of the extended notation $d_{Y, \rho T}(\begin{pmatrix} t \\ vfu \end{pmatrix})$.

According to space-time transformation, a dependence vector $d(\begin{pmatrix} t \\ vfu \end{pmatrix})$ can be decomposed as $d(\begin{pmatrix} t \\ vfu \end{pmatrix}) = (d^t, d^{vfu})$, where $d^t$ is an $n$-vector time distance and $d^{vfu}$ is an $m$-vector space distance.

Dependence vectors can be used to design communications between source points and consumer points:

- $d^{vfu}$ generates the interconnection between the source VFU($vfu_0$) and the consumer VFU($vfu_0 + d^{vfu}$).
– $d^t$ generates the delay between the computation of $Y(\mathbf{vfu}_0)$ and its use at time $t = t_0 + d^t$. Being the $n$-vector time $t$ scanned by $\text{tick}(t)$, the scalar delay associated to $d^t$ is $\text{tick}(d^t)$. In fact, from the $\text{tick}$ definition (10), $\text{tick}(t_0 + d^t) - \text{tick}(t_0) = \text{tick}(d^t)$.

Due to Definition 8 a dependence vector $d(\mathbf{vfu}) = (d_{vfu})$ is sufficient to generate a communication, because it gives the information to connect $\text{VFU(}vfu_0\text{)}$ to $\text{VFU(}vfu\text{)}$ with associated delay $\text{tick}(d^t)$. Due to admissibility constraint Theorem 1, $d^t \geq 0 \Rightarrow \text{tick}(d^t) > 0$.

Fig. 8 shows the implementation of a communication starting from a dependence vector. In the general case, dependence vectors depend on $(d_{vfu})$. The special case of uniform dependence vectors is obtained when they do not depend on $(d_{vfu})$, i.e., when they are constant. From Definition 12, it is easy to verify that uniform dependence vectors arise when $R_T = I_d$.

**Definition 13** (Uniform dependence vector). A transformed dependence vector is uniform if it does not depend on $(d_{vfu})$; so $d(\mathbf{vfu})$ is uniform if $R_T = I_d$; in such a case $d(\mathbf{vfu}) = -r_T$.

Because $R_T = T R T^{-1}$ (see Theorem 2), $R_T = I_d$ if and only if $R = I_d$, thus uniform dependence vectors are closed under the application of the space-time transformation.

A uniform dependence vector individuates a one-point utilization set because $I_d$ has full rank. In such a case, for each point $(t_{\mathbf{vfu}})$ in the emission set, the corresponding utilization point is

$$\begin{bmatrix} t \\ \mathbf{vfu} \end{bmatrix} = \begin{bmatrix} t_0 \\ \mathbf{vfu}_0 \end{bmatrix} + \begin{bmatrix} d^t \\ d_{\mathbf{vfu}} \end{bmatrix},$$

Communication is performed through a link connecting $\text{VFU}(vfu_0)$ to $\text{VFU}(vfu)$ (thus implying a spatially regular connection because $d_{\mathbf{vfu}}$ is constant) and a delay $\text{tick}(d^t)$ which is the same for all the links.

When $R_T \neq I_d$ two situations may arise:

– $R_T$ is a full rank matrix. In this case utilization set contains one point, as in the uniform case, but the related dependence vector is not uniform: the distance between each emission and utilization point is a function of utilization point coordinates. For this case, a technique to transform non-uniform into quasi-uniform dependencies is presented in [31]. A quasi-uniform dependence is almost always uniform, with the only exception of the points at the border of the iteration space.

– $R_T$ has not full rank. In this case utilization set has several consuming points and so there are several non-uniform dependence vectors. Communicating $Y(\mathbf{vfu}_0)$ to the points $(\mathbf{vfu}) \in \text{Util}(Y(\mathbf{vfu}_0))$ requires a set of different links with different delays. Fig. 9 depicts the implementation of communications via dependence vectors. The edges represent the links needed to transfer data and the numbers represent the delay associated to the link. It is worthwhile to note that even for this simple example the communication patterns is quite complex.

**6. Propagation technique**

When $R_T$ has not full rank the communication pattern, individuated as in the previous section, is costly to implement, because the connecting links have different lengths and may intersect.

To the best of our knowledge, the only attempts to afford communication synthesis in the case of $R_T$ having not full rank are presented in [12–15]. In these works, the propagation technique is introduced;
particularly [15] uses branching over kernel basis of matrix $R_T$ to find the (near) optimal solution. The set of dependence vectors, arising when $R_T$ has not full rank, originates communications which are synthesized in two phases:

- the first step foresees the communication of $Y(t_0^{vfu_0})$ to a point of the corresponding utilization set;
- the second step propagates $Y(t_0^{vfu_0})$ to the whole utilization set.

The main focus of [13,14] relies on the optimization of the first step. Affine per variable transformations (each variable is projected with a different space-time transformation) are used to enforce communication locality. The second phase of the communication synthesis, propagation of $Y(t_0^{vfu_0})$ to the whole utilization set, is performed by using utilization vectors: $Y(t_0^{vfu_0})$ is propagated to all the points in the utilization set with communications between adjacent points, i.e. points whose difference gives one of the utilization vectors.

As in [13,14]

- we require that $Y(t_0^{vfu_0})$ is first communicated to a vertex $V = (V^t V^{vfu_0})$ of $Util_i(Y, (t_0^{vfu_0}))$ by means of the dependence vector $d(V^t V^{vfu_0})$;
- starting from $(V^t V^{vfu_0})$, $Y(t_0^{vfu_0})$ is propagated within $Util_i(Y, (t_0^{vfu_0}))$ by means of some constant elementary vectors $h_j = (h^t h^{vfu_0})$.

Vectors $h_j$ must span the utilization set through an integral combination; in [13] this fact is not ensured because utilization vectors are only constrained to be a base for $Ker(R_T)$; in [14] utilization vectors are chosen to be the columns of an unimodular matrix, thus ensuring the spanning of utilization set through an integral combination.

Each $h_j = (h^t h^{vfu_0})$ must respect $h^t \geq 0$. This condition ensures that no point in the utilization set is required to propagate a not yet received value.

When choosing a basis for the utilization set, we put the further constraint that its points must be spanned through non-negative integral combinations of $\{h_j\}$, i.e. $(V^{vfu}) = (V^t V^{vfu}) + \sum_i \alpha_i (h^t_i h^{vfu}_i)$ with $\alpha_i 

\geq 0$. In fact, if a combination contains a negative coefficient, a not causal communication is generated ($h^t_j \preceq 0$).

The main drawback of the technique in [13,14] is that propagation obtained through the use of the utilization vectors is not sufficient to guarantee the synthesis of causal communications, because there is no explicit constraint enforcing the coefficients of the combination to be non-negative. Fig. 10 shows a possible case arising when using the technique proposed in [14]. Not all the points are reachable via positive combination: vector $u_1$ must be used. In this case if $u_1$ defines a causal communication, $-u_1$ defines a non-causal communication (or vice-versa).

Furthermore, if we succeed in finding utilization vectors which yield to integral positive combinations, we are not guaranteed that communications arising from such vectors are causal. In fact, causal propagation is ensured by relation $g^t \gg 0$, where $(g^{vfu})$ are the vectors which generate the smallest cone with apex in $V$ and containing utilization set. This condition only ensures that utilization points can be reached with an (eventually not integral) positive combination of the $g$ vectors. We are not guaranteed that utilization vectors yield causal communications, because they may be ob-
tained as a negative combination of the $g_i$ vectors as in Fig. 11. Here $u_2$ may be not casual because it is derived from $-g_1$ which is not causal ($g_1$ is causal from condition $(g^t \gg 0)$.

On the basis of previous considerations, we require the set $\{h_j\}$ to satisfy the following properties:

1. each $h_j$ must be an integral vector;
2. every point of the utilization set must be obtained by the vertex $V$ and a non-negative integral combination of the $h_j$ vectors;
3. each $h_j = \left( \frac{h_j^t}{h_j^w} \right)$ must fulfill $h_j^t \gg 0$.

A set of vectors satisfying conditions (1) and (2) is an integral Hilbert basis of the smallest pointed cone with apex in $V$ and containing utilization set, as in the following

**Definition 14 (Integral Hilbert basis).** [16, p. 232].

A finite set of integral vectors $\{h_j\}$ is an integral Hilbert basis for $C$ if each integral vector in the cone $C$ generated by $\{h_j\}$ is a not negative integral combination of $\{h_j\}$. $\{h_j\}$ is minimal if no vector can be obtained as integral not negative combination of other vectors in $\{h_j\}$. If $C$ is pointed (i.e. its apex is a vertex) a unique minimal integral Hilbert basis exists.

**Definition 15 (Utilization set integral Hilbert basis).** Utilization set integral Hilbert basis is an integral Hilbert basis of the smallest pointed cone $C_V$ with apex in $V$ and containing Util$_i(Y, \{h_i^o\})$ (see Fig. 12).

Determination of the minimal integral Hilbert basis is a NP-hard problem. A methodology to find a (not necessarily minimal) integral Hilbert basis is given in [16, p. 233], where Hilbert vectors are found as points of $\{h_j \in \mathbb{Z}_N^k | h_j = \sum_i \mu_i g_i, 0 \leq \mu_i < 1\}$, being $g_i$ the generating vectors of the cone.

As we are interested in a minimal Hilbert basis, the further constraint $\sum_i \mu_i \leq 1$ is introduced. Such a constraint is aimed at diminishing the number of the resulting Hilbert vectors and is justified by simple geometric considerations. Since we are considering the restricted class of integer cones, the pointed ones, on the basis of our experience we conjecture that the Hilbert basis given by

$$\{ h_j \in \mathbb{Z}_N^k | h_j = \sum_i \mu_i g_i, 0 \leq \mu_i < 1, \sum_i \mu_i \leq 1 \}$$

is minimal.

In Fig. 12 the Hilbert vectors arising from the previous example are shown. The number of Hilbert vectors, needed to cover the utilization set with a non-negative combination, is greater than, or equal to, the dimension of the utilization set.
As Hilbert vectors form a basis of $\text{Util}_i(Y, (t_0_{vfu}))$, they allow to propagate $Y(t_0_{vfu})$ within the utilization set. Given $Y(t_0_{vfu})$, received and used in VFU($vfu$) at time $t$, it is propagated toward VFU($vfu + h_{j,vfu}$) where it is used at time $t + h_j$, being both $Y(t_{vfu})$ and $(t + h_j)_{vfu}$ belonging to $\text{Util}_i(Y, (t_0_{vfu}))$.

Verification of condition 3 ($h_{j,vfu} \gg 0$) is ensured by the following

**Theorem 3.** $h_{j,vfu} \gg 0$ if the initial vertex $V$ is the lexicographical minimum vertex $V_m$ in $\text{Util}_i(Y, (t_0_{vfu}))$.

**Proof.** Any point $(t_{vfu})$ of $\text{Util}_i(Y, (t_0_{vfu}))$ is obtained as
\[
(t_{vfu}) = (V_{m,vfu}^t) + \sum_i a_i (h_i^t_{vfu}), \quad \text{with } a_i \in \mathbb{N}.
\]
As $(t_{vfu}) \gg (V_{m,vfu}^t)$,
\[
(t_{vfu}) = (V_{m,vfu}^t) + \sum_i a_i (h_i^t_{vfu}) \Rightarrow \sum_i a_i (h_i^t_{vfu}) \gg 0.
\]
Being $a_i \geq 0$, due to the generality of $(t_{vfu})$, $h_i = (h_i^t_{vfu}) \gg 0$ and hence $h_{j,vfu} \gg 0$.

In order to propagate a given value $Y(t_{vfu})$ within $\text{Util}_i(Y, (t_0_{vfu}))$, a communication must be provided between the emission point $(t_{vfu})$ and the lexicographical minimum vertex $(V_{m,vfu}^t) \in \text{Util}_i(Y, (t_0_{vfu}))$.

This communication is implemented by an interconnection between VFU($vfu$) and VFU($V_{m,vfu}^t$) with associated delay $\text{tick}(V_{m,vfu}^t - t_0)$. $Y(t_{vfu})$ is then propagated from $(V_{m,vfu}^t)$ to any other point of $\text{Util}_i(Y, (t_0_{vfu}))$ through the Hilbert basis vectors. As $\{h_j\}$ is a basis, each point of $\text{Util}_i(Y, (t_0_{vfu}))$, with the exception of $(V_{m,vfu}^t)$, is connected through a Hilbert vector with another point of $\text{Util}_i(Y, (t_0_{vfu}))$. Thus, given a point $(t_{vfu}) \in \text{Util}_i(Y, (t_0_{vfu}))$, it can receive the value $Y(t_{vfu})$ through the Hilbert vector $h_j$ if and only if $(t_{vfu}) - h_j \in \text{Util}_i(Y, (t_0_{vfu}))$.

In order to individuate the whole set of points which can be reached by the Hilbert vector $h_j$, we introduce the cover set of $h_j$:

**Definition 16** (Hilbert vector cover set). Given an Hilbert vector $h_j$ the Cover Set of $h_j$, denoted by $\text{Cover}(h_j)$, is the set of points that can receive the value $Y(t_{vfu})$ via $h_j$:
\[
\text{Cover}(h_j) = \left\{ \left( t_{vfu} \right) \in \text{Util}_i \mid \left( t_{vfu} \right) - h_j \in \text{Util}_i \right\}.
\]

In Figs. 13, 14 and 15 the cover sets related to the three Hilbert vectors found in previous example are depicted.

Hilbert vectors are our basic tool to synthesize communications. Depending on the time and space
parts, they can be classified according to the following definitions:

**Definition 17** (Local Hilbert vector). $h_j = \left( \begin{array}{c} h_{t_j} \\ h_{vfu_j} \end{array} \right)$ is local if $h_{vfu_j}$ is null, i.e. $\left( \begin{array}{c} 0 \\ h_{t_j} \end{array} \right)$.

A local Hilbert vector takes into account the local reusing of $Y(t_0^{vfu})$: $Y(t_0^{vfu})$, used at time $t$ in VFU(vfu), is reused at time $t + h_{t_j}$, i.e. after $\text{tick}(h_{t_j})$ time steps, in the same VFU(vfu). A local Hilbert vector defines an interconnection between VFU(vfu) and itself with an associated delay $\text{tick}(h_{t_j})$. A multiplexer must be provided to allow the selection between data from another VFU (first communication) and the local reuse of the same data (successive communications). Fig. 16 depicts the implementation of a local Hilbert vector.

**Definition 18** (Pipeline Hilbert vector). $h_j = \left( \begin{array}{c} h_{t_j} \\ h_{vfu_j} \end{array} \right)$ is pipeline if $h_{t_j}$ and $h_{vfu_j}$ are both not null.

A pipeline Hilbert vector takes into account the reusing of a value $Y(t_0^{vfu})$: $Y(t_0^{vfu})$, used at time $t$ in VFU(vfu), is reused at time $t + h_{t_j}$, i.e. after $\text{tick}(h_{t_j})$, in VFU(vfu + $h_{vfu}$). A pipeline Hilbert vector defines an interconnection between VFU(vfu) and the VFU(vfu + $h_{vfu}$) with an associated delay $\text{tick}(h_{t_j})$. Fig. 17 depicts the implementation of a communication related to a pipeline Hilbert vector.

**Definition 19** (Broadcast Hilbert vector). $h_j = \left( \begin{array}{c} h_{t_j} \\ h_{vfu_j} \end{array} \right)$ is broadcast if $h_{t_j}$ is null, i.e. $h_j = \left( \begin{array}{c} 0 \\ h_{vfu_j} \end{array} \right)$.

A broadcast Hilbert vector takes into account the concurrent use of $Y(t_0^{vfu})$: $Y(t_0^{vfu})$, used at time $t$ in VFU(vfu), is also used, at the same time $t$, in VFU(vfu + $h_{vfu}$). A broadcast Hilbert vector defines an interconnection between VFU(vfu) and VFU(vfu + $h_{vfu}$) with an associated delay $\text{tick}(0) = 0$ (i.e. a bus). Fig. 18 shows the communication implementing a broadcast Hilbert vector.

The propagation technique illustrated above can also be applied to the case of input variables (with the obvious substitution of $(t_0^{vfu})$ with $(z_0)$).

### 7. Communication synthesis design

Generally, $\text{Cover}(h_1) \cap \text{Cover}(h_j) \neq \emptyset$. This fact implies that the same point of the utilization set can
be reached by different Hilbert vectors. The choice of the most convenient $h_j$ depends on the cost of the associated communication (silicon area required to implement wiring and delays).

To implement the less costly communication pattern, Hilbert vectors must be ordered (<) according to increasing implementation cost. We assume broadcast < pipeline < local vectors because of wiring, memory and multiplexing requirements. Different orderings can be chosen, depending on the actual synthesis costs.

An algorithm, based on the following items, has been developed to generate a (near) optimal communication pattern:

- connect as much as possible VFUs through the less costly vector $h_1$ and mark them as connected. The set of points reached through $h_1$ is $\text{Cover}(h_1)$;
- iterate previous step by connecting the not yet marked VFUs through the $j$th Hilbert vector $h_j$; the set of points reached through $h_j$ is $\text{Cover}(h_j) - \text{Cover}(h_{j-1}) - \cdots - \text{Cover}(h_1)$.

We give an algorithm which, receiving the transformed SARE description, synthesizes communications (i.e. interconnection paths and delays).

**Communication Design Algorithm**

Foreach equation $i$

Foreach dependence $\rho_T(1_{\text{vfu}})$

Find $\text{Util}(Y, (1_{\text{vfu}}))$

Find $V_m$ set

Find the corresponding Hilbert basis $\{h_j\}$

Order $\{h_j\}$ in increasing implementation cost

Foreach $V_m$

If $Y$ is a result variable then

connect input $Y$ of $\text{VFU}(V_{\text{vfu}})$ to output $Y$ of $\text{VFU}(V_{\text{vfu}}) - d_{\text{vfu}}(V_m))$ with delay $\text{tick}(d_{\text{vfu}}(V_m))$; this connection is selected when $(1_{\text{vfu}}) = V_m$

If $Y$ is an input variable then

connect input $Y$ of $\text{FU}(V_{\text{vfu}})$ with the $Y$ input port with delay 0. This connection is selected when $(1_{\text{vfu}}) = V_m$

Endforeach

8. Control FSM synthesis and clustering

The set of interconnected VFU, derived from allocation and scheduling phases and from the communication synthesis step, defines the system data path. Data path is controlled by selection signals which are driven by a FSM that is composed by the $\text{tick}$ conversion circuit and by a combinatorial circuit. Such a combinatorial circuit depends on the timing function and generates selection signals for the communication and VFU multiplexers according to the current value of vector time $t$, to the communication selection derived from previous algorithm and to transformed iteration spaces. The whole structure of the control FSM is given in Fig. 19.

The clustering step is needed because the VFU array, generated by the application of the timing and allocation functions, is usually too large to be directly implemented on the limited set of physical resources.

The clustering methodology we use is based on either the Local Parallel–Global Serial (LPGS) or the Local Serial–Global Parallel (LSGP) approaches implemented as described in [10]; different implementation of LSGP/LPGS clustering can be found in [20]. The chosen clustering formulations ensure an analytical closed form solution both for allocation and scheduling. An array of Physical Functional Units

(PFU) is individuated; PFU and VFU arrays have the same spatial dimensionality \(m\) but the PFU array contains a number of functional units significantly smaller than VFU.

VFU array is scheduled and allocated onto the smaller PFU array. Given a VFU(\(vfu\)), the clustering phase returns the \(m\)-dimensional auxiliary time \(t_a\) (internal PFU scheduling) and the \(m\)-dimensional \(pfu\) coordinates (allocation). All the auxiliary time steps (scanned through the auxiliary time \(t_a\)) must be contained within a single \(t\) step. A computation allocated on VFU(\(vfu\)) and scheduled at time \(t\) is executed on PFU(\(pfu\)) at time \(t_a\) within the \(t\) time step. Simple clustering relations allow to determine, for each VFU(\(vfu\)), the PFU where, and the auxiliary time when, VFU is emulated.

Finally, VHDL code is generated to describe the designed parallel architecture.


In order to illustrate the presented design methodology, we choose a simple but significant example: the matrix–matrix product. Its simplicity allows to avoid cumbersome details, notations and figures; at the same time, we are dealing with a surely significant case study, representing the basic building block for nearly all the linear algebra operations (BLAS-3 routines).

In the following, we show each step of the proposed design flow.

The initial specific is given through the LIGHTC program reported in Fig. 20 which implements the multiplication of two \((q \times q)\) matrices; we choose to synthesize the system with \(q = 32\).

Once the correctness of the algorithm has been verified at this very high level of abstraction, the LIGHTC program is automatically converted by the LIGHTC compiler into the semantically equivalent SARE formulation, shown in Fig. 21. SARE is expressed by means of the SIMPLE (SARE IMPElementation) language [17].

The obtained SARE must be allocated and scheduled through timing and allocation functions. The chosen timing function is

\[
t(z) = (1\ 0\ 1)z + 0
\]

which originates the \(\text{tick}\) function

\[
\text{tick}(t) = t_1
\]

with

\[
c_1 = \max(t_1) - \min(t_1) + 1 = 2q - 1 = 63.
\]

The circuit in Fig. 22 performs the \(\text{tick}\) conversion.

The SARE is allocated through the following allocation function

\[
\text{vf}(z) = \Sigma z + \beta = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} z + \begin{pmatrix} 0 \\ 0 \end{pmatrix}.
\]

The \(32 \times 32\) VFU array in Fig. 23 is obtained.

Blocks of type 0 perform the multiplication of the input values \(A\) and \(B\) and corresponds to the first statement of the LIGHTC program in Fig. 20. Blocks of type 1 perform the multiplication of the input values \(A\) and \(B\) and the sum with the input value \(C\).

The choice of \(t(z)\) and \(\text{vf}(z)\) is performed through an optimization process [19] which is driven by
main ()
{
    /* variable declaration */
    ...
    /* memory allocation and initialization */
    ...
    /* matrix-matrix product */
    for (i=0; i<=q-1; i++) {
        for (j=0; j<=q-1; j++) {
            for (k=0; k<=q-1; k++) {
                c[i][j] = f0(a[i][k], b[k][j]);
                /* f0: a[i][k]*b[k][j] */
            }
            for (k=1; k<=q-1; k++) {
                c[i][j]=f1(c[i][j], a[i][k], b[k][j]);
                /* f1:c[i][j]+a[i][k]*b[k][j] */
            }
        }
    }
    /* output result */
    for(i=0; i<=q-1 ;i++) {
        for(j=0; j<=q-1; j++) {
            put_output(f_out, c[i][j]);
        }
    }
    /* freeing memory */
    ...
}

Fig. 20.

a cost function taking into account both the hardware implementation requirements and the computing time. In this example, we skip the optimization phase and we choose allocation and timing functions in order to emphasize the different communication types arising in the communication synthesis phase.

The communication synthesis leads to three types of communications:
- a uniform communication connecting the output of a given VFU(vfu1, vfu2) with the input C of VFU(vfu1, vfu2 + 1); the delay associated to this communication is 1 clock cycle.
- a broadcast communication connecting the input B of a given VFU(vfu1, vfu2) with the input B of VFU(vfu1 + 1, vfu2);
- a local communication connecting the input A of a given VFU(vfu1, vfu2) with the input A of the same VFU; the delay associated to this communication is 1 clock cycle.

The resulting virtual system is shown in Fig. 24, where the small boxes are unitary delays; the multiplexers are selected by the signals from the selection generator and distributed as sketched through the dotted lines.

As the number of VFU is too large to be implemented (nearly one thousand of multipliers and adders are needed) previous VFU array is clustered onto a $2 \times 2$ PFU array, obtaining the circuit in Fig. 25. The number in the square boxes represents the communication delay expressed in clock cycles. The four blocks with $s_{ij}$ input data perform the multiply and add operations and produce intermediate or final results computed through the expressions shown near the outputs. Multiplexers are selected by the signals from the selection generator and distributed as sketched through the dotted lines. The tick conversion circuit is composed by three counters, to take into account the auxiliary time vector generated in the clustering step. The completion time of the circuit is $CT = 16 * 16 * 63 = 16384$ clock cycles. As the number of atomic operations $(AB + C)$ necessary to implement the $32 \times 32$ matrix product is $32768 (32^3)$, the efficiency of the
\begin{verbatim}
Ind[z1,z2,z3];
Par[q]
{ q-2>=0 };
Input s0[z1,z2] /* A matrix */
{z1>=0, z2>=0, -z1+q-1>=0, -z2+q-1>=0, 1>=0};
Input s1[z1,z2] /* B matrix */
{z1>=0, z2>=0, -z1+q-1>=0, -z2+q-1>=0, 1>=0); Result s2; /* C matrix */
Result s3; /* C matrix */
S2[z1,z2,z3]=f0(s0[z1,z3],s1[z3,z2] ) ;
{z3=0, 0<=z1<=q-1, 0<=z2<=q-1};
S3[z1,z2,z3]=f1(s2[z1,z2,0], s0[z1,z3],s1[z3,z2] );
{z3=1, 0<=z1<=q-1, 0<=z2<=q-1};
s3[z1,z2,z3]=f1( s3[z1,z2,z3-1],s0[z1,z3],s1[z3,z2] );
{2<=z3<=q-1,0<=z1<=q-1, 0<=z2<=q-1} ;
Write(s3[z1,z2,z3] );
{z3=q-1, 0<=z1<=q-1, 0<=z2<=q-1} ;
\end{verbatim}

Fig. 21.

Fig. 22. tick conversion circuit.

system is $\eta = 32768/(16384 \times 4) = 0.5$. If we use the timing function $t(z) = (0 \ 0 \ 1)^T$, originating one uniform and two broadcast communications, completion time is $CT = 8192$ and the resulting efficiency is $\eta = 32768/(8192 \times 4) = 1$.

10. Conclusion

In this work a design flow, used to synthesize a parallel architecture starting from high level specifics given by a LIGHTC program, has been presented. The proposed design methodology is able to automatically generate the skeleton of a VHDL program, describing the final parallel architecture, in the case of problems which can be solved by means of affine iterative algorithms. The LIGHTC language, a subset of the ANSI C, is the language designed and used to describe affine iterative algorithms.

Affine iterative algorithms can be represented through the System of Affine Recurrence Equations model (SARE). After reviewing basic theory and definitions about SARE model, we recalled the synthesis steps which allow to automatically generate a parallel architecture starting from the SARE description of the algorithm.

Special emphasis has been given to the communication synthesis, being this work mainly devoted to the presentation of the theory which allows an efficient design of communications in the general case of SARE algorithms. The theory uses the concept of Integral Hilbert Vector basis to implement a communication technique which propagates data from the source operations to all the consuming operations. The commu-
Fig. 23. VFU array.

Fig. 24. Virtual system.
nication patterns achievable through the presented theory are characterized by high regularity, thus allowing the generation of efficient interconnection schemes.

In order to make clear the theoretical concepts introduced in the paper, the whole design flow has been presented by means of a simple explanatory example, the matrix–matrix product.

References


