Abstract Clocks for the DSE of Data-Intensive Applications on MPSoCs

Rosilde Corvino
Eindhoven University of Technology, The Netherlands
Email: r.corvino@tue.nl

Abdoulaye Gamatié
CNRS/LIFL,Villeneuve d'Ascq, France
Email: abdoulaye.gamatie@lifl.fr

***Abstract*—This paper presents an approach advocating abstract clocks to represent data-intensive applications executed on multiprocessor systems-on-chip (MPSoCs) for facilitating the exploration of large design spaces. By using abstract clocks, the advocated method characterizes applications defined by multiple loop nests, as well as, useful loop transformations that contribute to an efficient application execution. It combines the advantages of optimizations provided by loop transformations and the precision of information on scheduling captured by the abstract clocks. As a result, it favors a rapid, and yet accurate design space exploration (DSE) of data-intensive systems.**

***Keywords*-Abstract clocks, communicating nested loops, MPSoC design, data-intensive applications**

I. Introduction

Multi-processor systems-on-chip (MPSoCs) offer interesting performances to efficiently execute the increasing number of functionalities integrated in embedded systems. For data-intensive systems, data transfer and storage mechanisms have a strong impact on performance, power consumption and chip size, which heavily depend on number and type of memory accesses and buses activities, as well as, memory and communication architectures. As a consequence, it is crucial to provide designers with adequate solutions for easy and rapid design space exploration (DSE) of communication and memory sub-systems in MPSoCs.

Embedded system design frameworks consider various techniques [1] to cope with design space exploration, such as static analysis of application performance and resources utilization, SystemC or VHDL-based simulation, prototyping on FPGAs, etc. These techniques are applied at different abstraction levels and serve different purposes. For instance, static analysis, which is used in early phases of system design, is extremely rapid and competitively accurate with respect to simulation in the case of predictable applications.

a) Our contribution: The high complexity of MPSoCs realizing data-intensive applications calls for high level (static analysis) techniques that make it possible to rapidly explore large design spaces. In this paper, we advocate abstract clocks for the representation of systems and for the expression of DSE problems. Abstract clocks have been used for several decades to deal with issues such as synchronization in distributed systems [2] or optimized compilation of synchronous reactive programs [3] [4]. Here, they enable to simplify DSE and provide adapted expressivity to make rapid and precise decisions relevant for data-intensive systems design. They capture a rich set of architecture configurations and application mapping and scheduling.

b) Related works: Among domain-specific frameworks providing DSE facilities for embedded systems, we mention Milan [5], Metropolis [6] or Daedalus [7]. Milan uses Boolean expressions to represent a design space. The design elements and constraints are encoded with binary decision diagrams (BDDs) [8] for an efficient manipulation. Another alternative that is frequently used to encode and solve design optimization problems is integer constraint programming (ICP) as considered in [9]. Metropolis considers concurrent communicating processes as representation model. In Daedalus, the design is achieved with Kahn process networks (KPNs) [10]. Compared to all previous frameworks, our approach exploits abstract clocks for a well-suited characterizing of data-intensive predictable applications defined by communicating nested loops. Our approach also benefits from loop transformations, which are usually employed to optimally compile an application onto a fix processor architecture. Design representations based on SDFs [11] or KPNs are interesting for application graph transformations during DSE. However, in the specific case of data-intensive applications, they are less efficient than specifications as the polyhedral model supporting loop transformations, which are crucial for data managment mechanisms optimization. In this paper, we use loop transformations to design application specific hardware architectures. Similar existing works in literature only focuses on transformations for single loop nests [12] [13]. Our proposition exploits certain useful loop transformations featuring the optimization of applications with multiple communicating loops and the optimization of their hardware realization on MPSoCs.

c) Outline: In the remainder, Section II introduces background notions for describing data-intensive applications and MPSoCs. Section III defines our abstract clock framework. Section IV briefly discusses the benefits of this framework for DSE. Finally, Section V gives conclusions.

II. Design concepts: background

A. Actor oriented application model

Array-OL (Array-Oriented Language) is a formalism dedicated to the specification of data intensive applications manipulating multidimensional data arrays [14]. It allows to define functionally deterministic specifications in the form of data

dependencies. Such a specification is an oriented task graph in which three kinds of tasks are distinguished: *elementary*, *repetitive* and *composed* tasks. These tasks have input and output *ports*. An elementary task is an atomic function. A repetitive task expresses data-parallelism by specifying how a given task is repeated on different data subsets, referred to as *tiles* A repetitive task consumes and produces multidimensional input and outptu arrays. Such a task is illustrated in Figure 1. Its associated *repetition space* $\mathbf{r}$, a vector in which coordinates are iteration bounds, gives the total number of repetitions. Input and output arrays are conveyed by white square ports. Each tile, conveyed by a dark square port, is processed by a repeated task instance corresponding to an elementary task in Figure 1. All ports are associated with a *shape* information representing the static size/dimension of their conveyed arrays and tiles. In Figure 1, the shape of the unique input array port is a $(8, 7)$-matrix. The absolute coordinates of every tile within an array are computed via the information provided in a specific connector, called *tiler*, which connects an input/output array port to its corresponding tile port.

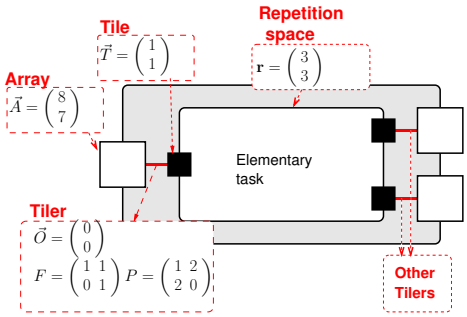

Fig. 1. Array-OL task specification and tilers.

A tiler specifies the following information: the coordinates $\vec{O}$, referred to as *origin vector* of the data array; a *paving matrix* P used to compute the absolute coordinates of tile origins in an array; and a *fitting matrix* F used to compute data coordinates within a tile.

A composed task is a hierarchical composition of elementary and repetitive tasks, allowing for task parallelism representation. Figure 2 shows an example of Array-OL specification, composed of four interconnected repetitive tasks, each repeating an elementary task.

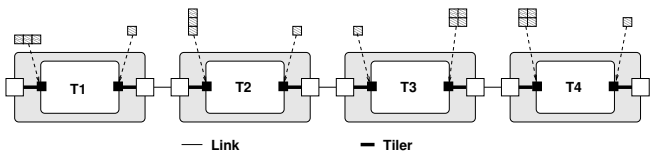

Fig. 2. An Array-OL specification composed of four tasks.

Loop-like transformations can be used to modify Array-OL specifications [14]. Here, the main transformations considered are: *fusion*, *tiling* and *paving change*. Figure 3 intuitively shows an Array-OL specification derived by applying the following transformations to the specification of Figure 2: *i)* fusion of tasks `T1` and `T2`, where the two elementary tasks `T1` and `T2` are merged in a common repetition space; *ii)* tiling of task `T3`, where a new repetition hierarchy level is created and the repetitions are distributed between the hierarchy levels; and *iii)* paving change of task `T4`, where the size of the consumed and produced tiles are increased.

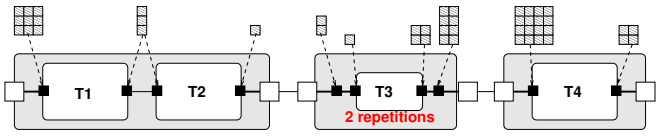

Fig. 3. Specification of Figure 2 after 1) fusion of tasks `T1` and `T2`; 2) tiling of task `T3` and 3) paving change of task `T4`.

B. A generic tile-based hardware architecture

We consider a generic hardware architecture model, two explanatory instances of which are given in Figure 4 and Figure 5. Such a model is a simplified representation of a tile-based MPSoC [15]. Each processing tile (Proc Tile i) contains a processing element (T_i), local memories (light gray squares) and a local control for data access (CTRL). Thanks to a double buffering mechanism, i.e. two local buffers alternatively read and written by the processing elements and the CTRL of a processing tile, data accesses and computations can be performed in parallel. Furthermore, a processing tile executes task repetitions in a pipelined fashion due to the usage of pipelined computing units.

Different tiles communicate through point-to-point links if they frequently exchange small amounts of data, or through a shared bus if they exchange large amounts of data.

Application/architecture mapping and scheduling rules make each Array-OL repetitive task correspond to a processing tile of the MPSoC and each tile of data to a local double buffer. Figure 4 represents the customized architectures associated with the specification of Figure 2, which contains four repetitive tasks exchanging (large) arrays, as a consequence four processing tiles are instantiated to execute the four tasks. The local memories of the instantiated processing tiles are able to store the data tiles of these tasks and use a double buffering mechanism to mask data access to the external memory that is performed through a shared bus.

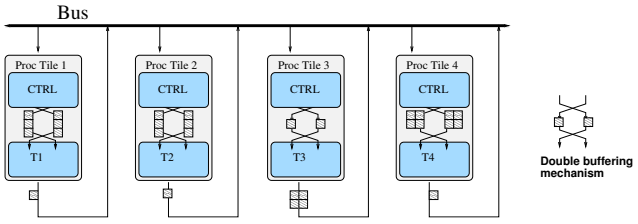

Fig. 4. Architecture associated with the Array-OL model of Figure 2.

Loop-like transformations directly set specific mapping and scheduling rules on the considered tile-based hardware architecture, as follows: 1) The task *fusion* determines the

communication structure. Indeed, when two tasks are merged they repeatedly exchange smaller data blocks. Thus, they are mapped onto a pipeline of processing tiles with point-to-point connections. They benefit from parallel read and write accesses to local double buffers. By contrast, two unmerged tasks exchange larger data blocks that cannot be stored in local buffers. They are mapped onto processing tiles communicating via the shared bus with exclusive read/write accesses. 2) The *paving-change* determines different sizes of local double buffers. 3) The *tiling* determines different parallelism levels multiplying the number of processing elements within each processing tile.

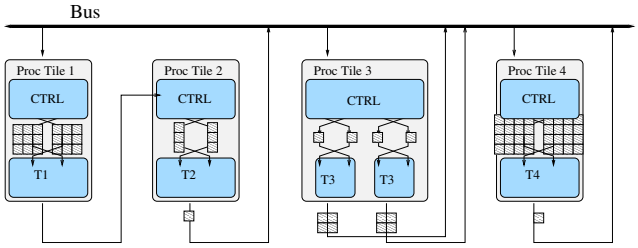

Fig. 5. Architecture associated with the Array-OL model of Figure 3.

Figure 5 represents the customized architecture associated with the specification of Figure 3. The tasks `T1` and `T2` are merged by fusion. They are mapped onto processing tiles that communicate directly and realize a pipeline in our MPSoC architecture. Proc Tile 1 can copy data directly into the local memory of Proc Tile 2. The task `T3` is tiled with two repetitions moved to the inner repetition level. Its corresponding processing tile implements two parallel processing elements with own local buffers, that can process different data tiles in parallel. Proc Tile 3 uses a single shared controller to copy data into its local double buffers in order to reduce chip area overhead due data parallelism increase.

III. Abstract clock reasoning framework

Abstract clocks describe how the data consumption and production of repetitive tasks are synchronized when executed on MPSoC processing tiles according to the previous mapping rules. An abstract clock is associated with each input and output data tile. *An abstract clock is a periodic binary word, which has a phase Φ and a period Π repeated r times.* The value 0 denotes a synchronization cycle and 1 denotes a read or write data access. By convention, we consider $0^0 \equiv 1^0 \equiv empty\ word$.

$$
\begin{array}{rl}
clk(i0) = & \overbrace{0000001\ \ 0000001\ \ 0000001\ \ 0000001}^{r}\ \ldots \\
clk(i1) = & 0001111\ \ 0001111\ \ .\ \ .\ \ \ldots \\
clk(o) = & \underbrace{0000000}_{\Phi}\ \underbrace{0000011}_{\Pi}\ \underbrace{0000011}_{\Pi}\ \ .\ \ \ldots
\end{array}
$$

Fig. 6. Abstract clocks associated with a repetitive task having two inputs $i0$ and $i1$, and one output o.

Figure 6 gives the abstract clocks describing the behavior of a repetitive task when executed onto a processing tile. The considered task has two inputs $i0$ and $i1$, and one output o. The numbers of elementary data items in each data tile are respectively 1, 4 and 2 for $i0$, $i1$, and o. The clock associated with the port o, named $clk(o)$, has a phase Φ, marked by 0's, to synchronize its first output firing to the input availability. The clocks of ports of a task have the same period Π during which the rhythm of data consumption and production is marked by 1's. The clock cycles marked with 0's within a clock period or a clock phase can denote the latency due to an external memory accesses or a synchronization.

In our approach, when a task is executed onto a processing tile, it is referred to as an *actor*. Such an execution is captured via abstract clocks as detailed in the next sections.

A. A clock modeling of actors

We first define the *cardinality of a port* in an actor, which corresponds to the number of elementary data contained in each data tile conveyed by the port.

Definition 1: (Cardinality) Given an actor α, the cardinality of a port p with a shape $[d_0, ..., d_q]$ in α, denoted by $|p|$ is the product $d_0 \times ... \times d_q$ of its shape dimensions. □

Similarly to [16], we consider the *initiation interval* of an actor, as the minimum latency between the start of two firings or repetitions pipelined on the same processing tile.

Definition 2: (Initiation interval of an actor) Given an actor α and its associated sets of input and output ports $\mathbf{I} = \{\mathbf{i}_k\}$ and $\mathbf{O} = \{\mathbf{o}_l\}$, the initiation interval of α is:

$$\Upsilon(\alpha) = max\left\{max\{|\mathbf{i}_k|\}, max\{|\mathbf{o}_l|\}\right\}_{\forall \mathbf{i}_k, \mathbf{o}_l \in \mathbf{I}, \mathbf{O}}. \qquad \square$$

We decompose the repetition space $\mathbf{r}$ of an actor α into two components: a multidimensional finite spatial sub-space s and a monodimensional infinite temporal sub-space t, such that $|\mathbf{r}| = |s| \times |t|$. Here, the cardinality of a repetition space represented by a vector denotes the total number of repetition instances executed in the space, obtained in a similar way as in Definition 1. The behavior of α is completely defined over s and is periodically repeated over t. Let us consider a video application in which a flow of frames is processed. The spatial component is the sub-space of repetitions that the processor needs to compute either a part of a frame, or a whole frame, or a set of frames. The temporal component is ∞ and gives the time dimension of the flow.

The behavior of an actor over its repetition space is periodic. This periodicity is determined by actor's *synchronization points*, defined below.

Definition 3: (Synchronization points of an actor) Given an actor α with a repetition space $\mathbf{r}$, the set of its synchronization points is:

$$sync(\alpha) = \{\sigma_i \mid \sigma_i = i \times \Upsilon(\alpha), 0 \leq i \leq |\mathbf{r}|\} \qquad \square$$

Intuitively, each synchronization point σ_i of an actor marks the end of the production of output data on each output port an actor α. The duration between two successive synchronization

points is $\Upsilon(\alpha)$. Using the synchronization points, we can synchronize the activities on all actor ports and determine a scheduling for a repetitive task execution. Indeed, from their definition, we derive the fact that within each period of $\Upsilon(\alpha)$ cycles, all ports of an actor α have to perform their associated read/write data access. This allows us to define the periodic *abstract clock associated with a port of an actor*. For an actor α with $\mathbf{r}$ as repetition space, the generic form of a clock associated with a port p of α is:

$$clk_p = \Phi(\Pi_1\Pi_2)^{|\mathbf{r}|}$$

where Φ is the clock phase; Π_1 and Π_2 are the components of the clock period (repeated $|\mathbf{r}|$ times) respectively denoting synchronization and activity periods. A port receives or emits data only during activity periods.

Definition 4: (Abstract clocks of actor ports) Given an actor α with $\mathbf{r}$ as repetition space, and its associated sets of input and output ports $\mathbf{I} = \{\mathbf{i}_k\}$ and $\mathbf{O} = \{\mathbf{o}_l\}$, the components of the abstract clocks associated with its input and output ports are characterized as follows:

- $clk_{\mathbf{i}_k, \forall \mathbf{i}_k \in \mathbf{I}}$,	$\Phi = empty$	$\Pi_1 = 0^{\Upsilon(\alpha)-\lvert\mathbf{i}_k\rvert}$	$\Pi_2 = (1)^{\lvert\mathbf{i}_k\rvert}$
- $clk_{\mathbf{o}_l, \forall \mathbf{o}_l \in \mathbf{O}}$,	$\Phi = 0^{\Upsilon(\alpha)}$	$\Pi_1 = 0^{\Upsilon(\alpha)-\lvert\mathbf{o}_l\rvert}$	$\Pi_2 = (1)^{\lvert\mathbf{o}_l\rvert}$

For each clock, the periodic part $(\Pi_1\Pi_2)$ is repeated $|\mathbf{r}|$ times. □

An example of abstract clocks associated with ports of an actor α is shown in Figure 7. The actor is represented by an Array-OL repetitive task to execute on a processing tile. Let us consider the repetition space of α is $\mathbf{r}$. The associated port cardinalities are mentioned (on the left part). The actor has two input ports (i and $i1$) and an output port (o).

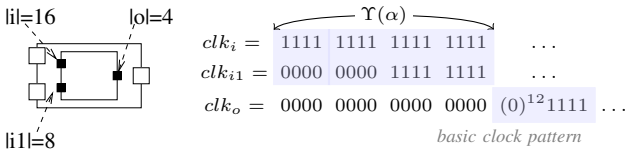

Fig. 7. Sketch of a repetitive actor with its corresponding clock trace.

By applying Definition 2, it follows that: $\Upsilon(\alpha) = |i| = 16$, and from Definition 4, we compute the following abstract clocks for each port:

- $clk_i = (0^{\Upsilon(\alpha)-|i|}(1)^{|i|})^{|r|} \quad \Rightarrow \quad ((1)^{16})^{|\mathbf{r}|}$
- $clk_{i1} = (0^{\Upsilon(\alpha)-|i1|}(1)^{|i1|})^{|\mathbf{r}|} \quad \Rightarrow \quad (0^8(1)^8)^{|\mathbf{r}|}$
- $clk_o = 0^{\Upsilon(\alpha)}(0^{\Upsilon(\alpha)-|o|}(1)^{|o|})^{|\mathbf{r}|} \quad \Rightarrow \quad 0^{16}(0^{12}(1)^4)^{|\mathbf{r}|}$

Figure 7 shows the corresponding clock trace (on the right part). Thanks to the assumptions on *the double buffering*, an actor is able to produce and receive data in parallel. Furthermore, due to the *pipelined computing units*, the actor's repetitions can be executed in a pipeline fashion on the same processing tile ensuring that an output data is produced at each cycle. This scheduling is similar to the modulo scheduling (or software pipeline) of the actor repetitions. During each repetition of a basic clock pattern (highlighted with blue colored background in Figure 7) the actor produces an output data block of size $|o|$ on its output ports. By repeating $|s|$ times the basic clock pattern, where s is the finite spatial component of $\mathbf{r}$, the actor produces a whole output data array on its output port.

Figure 8 illustrates another example of clock trace corresponding to the specifications of Figure 2 and the architecture of Figure 4. The trace contains abstract clocks associated with each task Ti and its input or output ports, i and o respectively. Each clock is a periodic, binary word having a phase $\Phi_k(.)$ and a period Π_k repeated r_k times. A period is common to all the clocks of a task and can contain synchronization cycles marked by 0's and activity cycles marked by 1's. A phase describes the behavior of a single port and can only contain synchronization cycles. In this example, the actors communicate through a shared external memory. Thus, there is no pipeline between the execution of different actors. However, thanks to the double buffering mechanism, the latency to get access to the external memory can be hidden by the computation execution.

B. Clock characterization of loop transformations

The loop-like transformations mentioned previously modify the clock characterization of application execution on our reference architecture. We define these modifications on the clocks of actor ports.

d) Fusion: Fusion is applied to merge two connected actors: a producer and a consumer. It depends on data granularity and on production/consumption order of data. Let α_1 and α_2 be two actors, communicating respectively through the ports $\mathbf{o}_1$ and $\mathbf{i}_2$ (indexes refer to actors). Their fusion computes a minimum common macro-block of data m that ensures the coherency of the production/consumption order. As a result, after fusion, the cardinalities of the ports $\mathbf{o}_1$ and $\mathbf{i}_2$ is $|m|$. The fusion transformation can be described by a tuples of positive integers, called fusion factors: $k_1 = \frac{|m|}{|\mathbf{o_1}|}$ and $k_2 = \frac{|m|}{|\mathbf{i_2}|}$. These factors are used to compute how the fusion changes the cardinality of all task ports as follows: $new\_\mathbf{i}_1 = k_1 \times \mathbf{i}_1$; $new\_\mathbf{i}_2 = m$; $new\_\mathbf{o}_1 = m$ and $new\_\mathbf{o}_2 = k_2 \times \mathbf{o}_2$. Finally, the fusion modifies the initial clock traces of merged actors and adds a global repetition space on top of both the producer and the consumer actor elementary tasks.

e) Tiling: Tiling is applied to a single repeated actor α to change the dimensions of its repetition space. Let $clk = \Phi(\Pi_1\Pi_2)^e$ be the generic abstract clock form for any port of α. Let k be an integer that exactly partitions the repetition space of α. The tiling of α with respect to k modifies the abstract clock of its ports as follows: $clk' = \Phi((\Pi_1\Pi_2)^k)^{\frac{e}{k}}$ where k is referred to as tiling factor. In an Array-OL model, no order is specified *a priori* for executing the repetitions of a task. The execution of the corresponding actor is captured by abstract clocks, which make explicit the sequential/parallel execution of the actor repetitions. The tiling is used to redistribute these repetitions in spatial and temporal components, executed in parallel or sequentially respectively. Here, a tiling of α with respect to k means the utilization of k parallel actor pipelines whose port abstract clocks are: $clk'' = \Phi(\Pi_1\Pi_2)^{\frac{e}{k}}$.

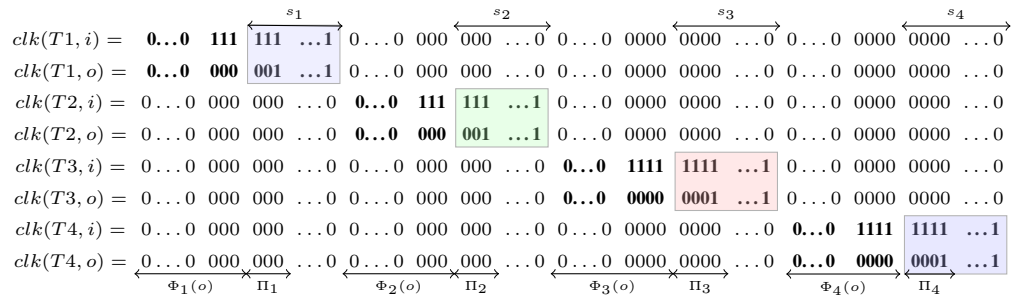

Fig. 8. Abstract clocks associated with the Array-OL model in Fig. 2 and the architecture in Fig. 4. The basic clock patterns are highlighted. The binary pattern $0\ldots0$, found in a phase or a period, is the latency due to external memory accesses. s_i is the spatial repetition space of an actor.

f) Paving change: Paving change is applied to a single repeated actor α to change the data granularity in α. Let $clk = \Phi(\Pi_1\Pi_2)^e$ and k be the generic abstract clock form for any port of α, and a real number. The paving change modifies clk as $clk' = \Phi(\Pi_1\Pi_2)^{\frac{e}{k}}$ where k is referred to as paving change factor. The data granularity is decreased[1] when $k \geq 1$ and increased when $k \leq 1$.

To apply loop-like transformations, one has to set the factors that describe them. The tiling and paving change factors can be set *a priori*. On the contrary, the fusion factors, are determined only after applying the fusion to an initial Array-OL model of the application.

Figure 9 shows the port abstract clock traces associated with the specification of Figure 3 and the architecture of Figure 5. These clocks capture the effects of loop transformations. For instance, when the task fusion processing pipeline are applied, the throughput of the implemented application increases in the corresponding clock trace. The latency and the energy consumption of the application execution are also reduced after the reduction of external memory accesses, which are expensive compared to local memory accesses. The clock traces of Figure 8 and Figure 9 exemplify the reduction of external memory accesses. In contrast with energy and execution time reduction, the task fusion requires the usage of larger double buffering w.r.t those used without fusion. This increases the internal memory size. As a consequence, a trade-off is needed between the usage of communication through internal and external memories.

C. Clock characterization of actor networks

Let us consider a network $N\{\alpha_1, ..., \alpha_k\}$ of k communicating actors, all associated with their own spatial and temporal repetition spaces, $\mathbf{r}_1$=$[s_1, t]$, ..., $\mathbf{r}_k$=$[s_k, t]$. In order to define a procedure that computes the synchronization points of N, we first distinguish two specific cases, in which, all actors execute either concurrently using point-to-point communications, or sequentially using a shared bus. We model concurrent executed actors in N, as actors within the context of a fusion, such that a common repetition hierarchy level exists on top of them (e.g., see Figure 3). Given the network N, such a fusion provides a new model consisting of a global repeated actor in the top level of the hierarchy, with $[s, t]$ as repetition space. Each repetition instance of this actor, is a pipelined execution of actors of N on reduced repetition spaces $\{s'_1, ..., s'_k\}$. In other words, each initial spatial repetition space s_i has been decomposed into s for the top level and s'_i for the low level of the task hierarchy after fusion. We refer to $\{\alpha'_1, ..., \alpha'_k\}$ as the actors of N associated with the reduced repetition spaces in the next.

Provided the above fusion transformation, the *delay between two successive synchronization points* of N is:

- for concurrent execution: the maximum of values computed as a product between the initiation interval of each actor α'_i and its reduced repetition space s'_i;
- for sequential execution: the sum of values computed as a product between the initiation interval of each actor α_i and its spatial repetition space s'_i.

The next definition summarizes the description of synchronization points in these two cases.

Definition 5: (Synchronization points of concurrent/sequential actors) Let us consider a network N of k communicating actors $\{\alpha_1, ..., \alpha_k\}$, respectively associated with repetition spaces, $\mathbf{r}_1$=$[s_1, t]$, ..., $\mathbf{r}_k$=$[s_k, t]$ respectively considered on the same temporal referential. The synchronization points of N are defined according to the following two cases:

- if all α_i's are concurrent and execute in pipeline after fusion: $\{\sigma_i \mid \sigma_i = i \times max\{|s'_1| \times \Upsilon(\alpha'_1), ..., |s'_k| \times \Upsilon(\alpha'_k)\}, 0 \leq i \leq |s| \times |t|\}$
- if all α_i's are sequential and communicate via a shared bus: $\{\sigma_i \mid \sigma_i = i \times v, v = (|s_1| \times \Upsilon(\alpha_1) + ... + |s_j| \times \Upsilon(\alpha_j)), 2 \leq j \leq k, 0 \leq i \leq |t|\}$

where $\{\alpha'_1, ..., \alpha'_k\}$ are respectively the actors corresponding to $\{\alpha_1, ..., \alpha_k\}$ after fusion. They are associated with $\{s'_1, ..., s'_k\}$ as repetition sub-spaces, within a shared global repetition space $[s, t]$ resulting from the fusion. □

Given Definition 5, we give now a general procedure (Procedure 1) describing how synchronization points are determined for any network of actors. For this purpose, we consider a few auxiliary functions in order to simplify the procedure description: *Extract_elementary_tasks_of(A)* produces the list of all elementary tasks contained in an application specification A in Array-OL; *Pop_element_from(E)* returns a task from a list E

[1]When $k \geq 1$, k must exactly partition the repetition space $\mathbf{r}$.

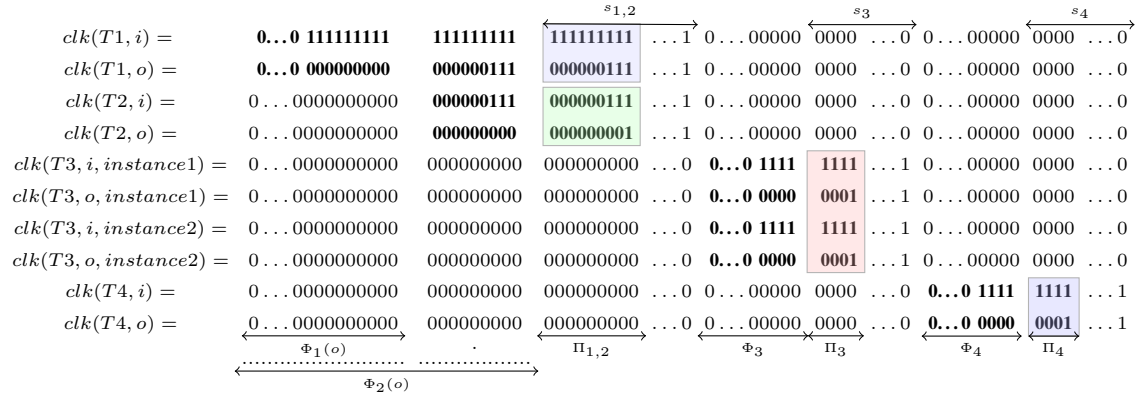

Fig. 9. Abstract clocks associated with the Array-OL specification of Figure 3 and to the architecture of Figure 5. The basic clock patterns are highlighted. For the pipelined processing tiles, the binary pattern $0\ldots0$, which represents the latency due to external memory access, is only present at the beginning of the first processing tile iterations. Indeed, the second processing tile accesses the local buffers. s_i is the spatial repetition space of an actor.

of elementary tasks; and *Parent_hierarchy_level_of(e)* returns the list of all elementary tasks contained in the hierarchy level (if it exists) above the level of an elementary task e.

```
1  //Data structure
   //A: input Array-OL specification
   //E: list of all elementary tasks e
4  //K: list of already processed elementary tasks
   //h: hierarchy level of a task

7  //σ_E: list of synchronization points corresponding
   // to an elementary task, when executed in
   // different single repetitions or in a fusion
10
   //global variable
   static K = ∅;
13
   // main Procedure
   [σ_E] = Compute_sync_points_of(A){
16   E = Extract_elementary_tasks_of(A);
     σ_E = ∅;
     while( E is not ∅ ){
19     e = Pop_element_from(E);
       while( h = Parent_hierarchy_level_of(e)
                  exists and e ∉ K ){
22       σ_Ẽ = ∅
         Ẽ = Extract_elementary_tasks_of(h);
         E = E \ (Ẽ \ K);
25       forall e_k ∈ Ẽ {
           if( e_k is a direct
               child of the hierarchy level h)
28           if( e_k ∉ K ) {
               compute σ_{e_k} w.r.t. the repetition
               h and according to Def 3;
31             σ_Ẽ.addall(σ_{e_k});
               K.add(e_k);}
           else
34           σ_Ẽ.addall( Compute_sync_points_of(h) );
         }
         if( h is a Fusion )
37         update σ_Ẽ according to Def 5 item 1;
         else if ( h is a Tiling ){
             //i.e. a repetition of a repetitive task
40           multiply the sync points and the
             corresponding clocks times the number
             of repetitions
43       }
         else
           update σ_Ẽ according to Def 5 item 2;
46       σ_E.add(σ_Ẽ);
       }
     }
49   return σ_E;
   }
```

Procedure 1. Procedure to compute the synchronization points of a generic network of actors.

Procedure 1 allows to compute the synchronization points of a generic Array-OL specification associated with our architecture model and including loop-like transformations. Such a procedure uses recursion to deal with hierarchy levels; it employs Definition 3 to compute the synchronization points of repeated elementary tasks and Definition 5 to update the computed synchronization points for each type of hierarchy level, i.e. fusion, tiling, etc. From the found synchronization points, it is possible to compute the abstract clocks, corresponding to Array-OL specifications as follows.

Definition 6: (Abstract clocks of communicating actors) Given p in a set $N = \{\alpha_1, \ldots, \alpha_N\}$ of actors. Let I^N and O^N be the set of input ports of all actors in N producing data for α_j and the set of output ports of all actors in N consuming data from α_j. The basic clock pattern of any actor α in N are:

$$\begin{array}{ll} \forall\ input\ \mathbf{i}\ \text{of}\ \alpha_j & \Phi = empty \qquad \Pi_1 = (1)^{\Upsilon(i)} \\ & \Pi_2 = 0^{\Upsilon(\alpha_j)-\Upsilon(\mathbf{i})+\Delta_\sigma-\Upsilon(\alpha_j)} \\ \forall\ output\ \mathbf{o}\ \text{of}\ \alpha_j & \Phi = 0^{\Upsilon(\alpha_j)} \qquad \Pi_1 = (1)^{\Upsilon(o)} \\ & \Pi_2 = 0^{\Upsilon(\alpha_j)-\Upsilon(\mathbf{o})+\Delta_\sigma-\Upsilon(\alpha_j)} \end{array}$$

where Δ_σ denotes the duration between any two successive synchronization points of the network $\{\alpha_1, \ldots, \alpha_N\}$. □

In the given model, the synchronization of several actors corresponds to their execution either in pipeline or in a mutual exclusion. This synchronization has an impact on the parallelism level and throughput of the target system. It is possible to use loop transformations to modify the parallelism level and the throughput of the actors. As a consequence, a design space exploration is performed by observing the effects of loop transformations on the clocks.

IV. Application to design space exploration

From our abstract clock characterization, we can define quality parameters to assess various system design solutions during DSE. Examples of parameters are given below as well as some results obtained on an application.

1) Amount of internal memory:

$$IM = \sum_t \left\{ \sum_i \{2 \times |\Pi_t(i)|_1\} \right\} + \sum_{t \to sink} \left\{ \sum_o \{|\Pi_t(o)|_1\} \right\}$$

where $|\Pi_t(i)|_1$ (respectively $|\Pi_t(o)|_1$) indicates the number of 1's in the period Π_t of an input port i (respectively output port o) of a task t. The factor $|\Pi_t(i)|_1$ represents the number of input data needed to be available on the port i so that the task t can fire $|\Pi_t(o)|_1$ outputs. The factor 2 is due to the double buffering mechanism. The index $t \to sink$ indicates all tasks t communicating with a sink.

2) Throughput:

$$T = \frac{\sum_{t \to sink} \{\sum_o \{|\Pi_t(o)|_1\}\}}{\sum_p \left\{ r \times \Delta_{\Pi_p} \right\}}$$

where the numerator is the total number of produced output data and the denominator is the latency of the computations needed to produce the total output data.

In the numerator, o indicates an output port of a task t communicating with a sink. In the denominator, Δ_{Π_p} is a latency computed as $\Pi_p \times r$, where Π_p is the period of tasks merged in a pipeline p and r is the number of times this period is repeated until the output data of p are produced.

3) Energy consumption due to the external memory accesses.

$$E = E_{read} \times \sum_{t \leftarrow source} \sum_i |\Pi_t(i)|_1 + E_{write} \times \sum_{t \to sink} \sum_o |\Pi_t(o)|_1$$

where $|\Pi_t(.)|_1$ is defined as in above formulas. E_{read} and E_{write} are the energy consumption per read and write depending on the used technology and the size of the external memory. The size EM of the external memory is the sum of all input and output array sizes, for all the tasks communicating with the source and sink tasks. It depends on the transformations applied to the specification.

A. Advantages of the proposed method

Our approach exploits the principle of abstraction in order to abstract away irrelevant specification details that are not in the scope of the addressed design space exploration. For this purpose, we use a data-oriented representation which has a precise but yet a concise representation of data related issues, i.e. data bandwidth, storage requirement, data parallelism, etc. With respect to other method using data flow representation, e.g. SDF-based design flows, our method has much more exploration power. Indeed, it uses a representation that is equivalent to the polyhedral model and, as a consequence, it benefits form loop-like transformations in order to systematically explore the space of possible data-oriented architectures and the space of corresponding application specifications. With respect to the polyhedral model, our method includes the usage of abstract clocks, which enable the possibility to concisely, yet precisely, express the scheduling information.

In [9], we considered a formalization using integer variables that represent required local buffer sizes and data parallelism respectively. In order to optimize design objectives, such as minimization of local buffers size and improvement of system temporal performance, some design constraints are formulated on these variables taking into account the chosen architecture/execution model. We noticed that while a variable only expresses buffer size information, the related DSE constraints involve more aspects such as the time balancing between data access time and output computation time, which is inadequately dealt with through local buffer sizes. For instance, to balance an output computation time of 4 cycles, one needs a data access time less or equal to 4 cycles, which, by using the integer variables representing buffer sizes, is achieved by filling buffers of size less or equal to 4 or $4 * m$ where m is the memory bandwidth.

The clock formalization given in this paper is more accurate than [9] by replacing the value of integer variables with binary words that encode in addition the order of data accesses, the time needed to read data and synchronizations. In particular, the usage of synchronization periods allows to de-correlate buffer sizes and buffer access times. This provides a more adequate way to divide the data access time between synchronization and actual data accesses. Furthermore, abstract clocks provide a more formal and uniform way to describe DSE constraints and objectives. This favors a simpler formalization of the optimization problem and thus opens the possibility to implement faster algorithms, based on either ILP or genetic algorithms, to solve design optimization problems.

Exploiting the abstract clocks formalism, we have developed a design space exploration tool based on a genetic algorithm. Such a tool, described in [17], explores hardware implementations of a given application by applying loop-like transformations. In this tool, the abstract clocks are used to precisely yet efficiently describe the mapping and temporal behavior of explored hardware implementations.

We have experimented our clock-based DSE implementation (available on demand from: http://www.es.ele.tue.nl/~rcorvino/tools.html) on the hydrophone monitoring application case study considered in [9]. With the clock-based approach, we obtained Pareto solutions that reduce by a factor of 22 the cumulative size of all internal memories. This improvement comes from the fact that with clocks and their associated synchronization points, we can use local buffers that are smaller enough to ensure time balancing requirement regarding application functionality correctness. On the other hand, thanks to its expressivity, the clock-based approach enabled us to observe additional solutions that are optimized w.r.t. more quality parameters. Typically, it makes it possible to deal with both dynamic and static power consumption while the method in [9] only supports the static part that depends on the size and shape of memories. The dynamic part requires the number of (local and external) memory accesses and the number of computations.

Detailed case studies, based on our abstract clocks formalism and concerning the exploration of four data-intensive applications, such as JPEG encoder, radar application [18], hydrophone monitoring [19] and low pass spatial filter [20], are reported in [17].

V. Conclusion

We presented an abstract clock characterization of data intensive applications executed on MPSoC platforms in order to propose an adequate high level support for design space exploration (DSE). We showed how abstract clocks provide adapted expressivity to make DSE precise and relevant regarding design concerns such as communication and energy consumption issues in MPSoCs. They capture a rich set of execution architecture configurations as well as application task mappings and schedulings. A prototype DSE tool has been defined based on this proposal, which shows very promising results. Potential design frameworks that can benefit from such a tool are those devoted to the development of data-intensive applications on MPSoCs such as Gaspard2 [21] and SDF3 [22].

VI. Acknowledgement

This paper has been partially supported by ASAM project of the ARTEMIS Research Program.

References

[1] B. Bailey, G. Martin, and A. Piziali, *ESL Design and Verification: A Prescription for Electronic System Level Methodology*. Elsevier Morgan Kaufmann, 2007.

[2] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[3] P. Amagbégnon, L. Besnard, and P. L. Guernic, "Implementation of the data-flow synchronous language signal," in *PLDI*, 1995, pp. 163–173.

[4] D. Biernacki, J.-L. Colaço, G. Hamon, and M. Pouzet, "Clock-directed modular code generation for synchronous data-flow languages," in *LCTES*, K. Flautner and J. Regehr, Eds. ACM, 2008, pp. 121–130.

[5] A. Bakshi, V. K. Prasanna, and Á. Lédeczi, "Milan: A model based integrated simulation framework for desgin of embedded suystems," in *LCTES*. ACM, 2001, pp. 82–87.

[6] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. L. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *IEEE Computer*, vol. 36, no. 4, pp. 45–52, 2003.

[7] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere, "A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs," in *CODES+ISSS'07*. New York, NY, USA: ACM, 2007, pp. 9–14.

[8] R. E. Bryant, "Symbolic manipulation of boolean functions using a graphical representation," in *DAC*, H. Ofek and L. A. O'Neill, Eds. ACM, 1985, pp. 688–694.

[9] R. Corvino, A. Gamatié, and P. Boulet, "Design space exploration for efficient data intensive computing on socs," in *Handbook of Data Intensive Computing*, B. Furht and A. Escalante, Eds. Springer New York, 2011, pp. 581–616.

[10] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

[11] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow: Describing signal processing algorithm for parallel computation." in *COMPCON*, 1987, pp. 310–315.

[12] J. Park, P. C. Diniz, and K. R. Shesha Shayee, "Performance and area modeling of complete FPGA designs in the presence of loop transformations," *IEEE Trans. Comput.*, vol. 53, pp. 1420–1435, November 2004.

[13] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau, "Spark : A high-level synthesis framework for applying parallelizing compiler transformations," in *Proc. of the 16th International Conference on VLSI Design, 2003*.

[14] C. Glitia, P. Boulet, E. Lenormand, and M. Barreteau, "Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications," *Journal of Systems Architecture*, vol. 57, no. 9, pp. 815–829, 2011.

[15] L. Benini and G. De Micheli, "Networks on chips: a new soc paradigm," *Computer*, vol. 35, pp. 70 –78, jan 2002.

[16] R. Schreiber, S. Aditya, S. Mahlke, V. Kathail, B. Rau, D. Cronquist, and M. Sivaraman, "Pico-npa: High-level synthesis of nonprogrammable hardware accelerators," *J. of VLSI Signal Proc.*, vol. 31, no. 2, pp. 127–142, Jun 2002.

[17] R. Corvino, A. Gamatie, M. Geilen, and L. Jozwiak, "Design space exploration in application-specific hardware synthesis for multiple communicating nested loops." in *SAMOS 2012*, 2012.

[18] C. Glitia, P. Boulet, E. Lenormand, and M. Barreteau, "Repetitive model refactoring strategy for the design space exploration of intensive signal processing applications," *J. of Systems Architecture*, vol. 57, pp. 815 – 829, Oct. 2011.

[19] P. Boulet, J.-L. Dekeyser, J.-L. Levaire, P. Marquet, J. Soula, and A. Demeure, "Visual data-parallel programming for signal processing applications," in *Proc. of Ninth Euromicro Workshop on Parallel and Distributed Processing, 2001*.

[20] R. Corvino, A. Gamatié, and P. Boulet, "Architecture exploration for efficient data transfer and storage in data-parallel applications," in *Euro-Par 2010 - Parallel Processing*, ser. Lecture Notes in Computer Science, P. DAmbra, M. Guarracino, and D. Talia, Eds. Springer Berlin / Heidelberg, 2010, vol. 6271, pp. 101–116.

[21] A. Gamatié, S. L. Beux, É. Piel, R. B. Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser, "A model-driven design framework for massively parallel embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 4, p. 39, 2011.

[22] S. Stuijk, M. Geilen, and T. Basten, "Sdf3: Sdf for free," in *International Conference on Application of Concurrency to System Design (ACSD 2006)*, june 2006, pp. 276 – 278.