Design of Streaming Applications on MPSoCs using Abstract Clocks

Abdoulaye Gamatié
CNRS/LIFL - UMR 8022, Villeneuve D'Ascq, France. Email: abdoulaye.gamatie@lifl.fr

***Abstract*—This paper presents a cost-effective and formal approach to model and analyze streaming applications on multi-processor systems-on-chip (MPSoCs). This approach enables to address time requirements, mapping of applications on MPSoCs and system behavior correctness by using abstract clocks of synchronous languages. Compared to usual prototyping and simulation techniques, it is very fast and favors correctness-by-construction. No coding is needed to run and analyze a system, which avoids tedious debugging efforts. It is an ideal complement to existing techniques to deal with large system design spaces.**

I. Introduction

The increasing complexity and sophistication of streaming embedded systems is observable in nowadays consumer electronics, e.g., mobile phones, high-definition TV and other video/audio devices. The development of these systems brings challenges [1] such as time and energy requirements management, efficient application mapping on execution platforms and correctness of designs, which must be addressed within stringent time-to-market and low cost constraints. Multiprocessor system-on-chip (MPSoCs) provide high computing performance and parallelism required for efficient implementation of concerned applications. Prototyping and simulation techniques have appeared as the mainstream design solutions.

Prototyping and simulation techniques at a glance. Among low-level approaches, we mention on the one hand *hardware acceleration and emulation* [2] that involve field-programmable gate arrays (FPGAs) and require *Register transfer level* (RTL) descriptions; and on the other hand *physical prototyping* [3], which involves circuit board and SoC in the form of working silicon. While the major advantage of these approaches is the high accuracy, they require a long time or provide a limited flexibility into systems for an efficient design space exploration (DSE) of several architectures.

Several approaches, e.g. [4], adopt the *transactional level modeling* (TLM) as an abstraction level for a fast simulation. TLM abstracts away low-level communication protocol details by considering bus read/write transaction level ones. Approaches based on *instruction set simulator* (ISS) [3] for pre-silicon verification and debugging, execute applications on hosts equivalent to the processors of the target execution platform. They offer a good functional behavior. However, their simulation speed and timing accuracy are very limited. Virtual system prototypes allowing *cycle-accurate* (CA) simulations are often preferred to these approaches. Other approaches rely on *host-compiled model* [5], which uses back-annotations of timing estimates determined either statically or dynamically for a rapid yet accurate simulation. While the simulation speed is not affected by these notations, the accuracy of estimates quite depends on the ability to avoid possible pessimistic timing approximations obtained statically and unpredictable effects on timing approximations obtained dynamically.

From the previous glance at prototyping and simulation techniques, we observe that they are complementary regarding the rapidity and accuracy of performance and energy estimation. Concerning correctness, these techniques consider debugging and testing, which are tedious tasks and, with the ever increasing complexity of embedded systems they are even a nightmare [1]. New approaches defined at higher abstraction levels, adopting formal methods, offer a solution.

Contribution. This paper presents a modeling paradigm for combined software, hardware and environment specifications to overcome the design evaluation issues for streaming embedded systems. A means for evaluating design choices, w.r.t. correctness, execution time or energy consumption, typically during early DSE, is proposed by using static analysis. It is a useful cost-effective alternative to approaches that would require actual implementation of every design choice and then simulate/prototype it, even just for evaluation purposes. *Abstract clocks* defined in synchronous languages [6] play a key role here. They consist of discrete sets of logical instants denoting when events are observed in a system. The advocated approach benefits from the solid mathematical foundation of synchronous languages and their ability to favor correct-by-construction designs. It is an ideal complement to prototyping and simulation to deal with complex systems design.

Outline. In the next, Section II discusses related works. Section III introduces abstract clocks and application component modeling. Section IV presents environment constraint modeling and analysis on applications by using clocks. Section V deals with the deployment of applications on execution platform with clocks. Finally, Section VI gives the conclusion.

II. Related works

Kahn process networks (KPNs) [7] have been used for the design of streaming applications. They consist of processes communicating via FIFO channels. Read requests are blocking in empty channels while write requests are not, i.e., channels are unbounded. KPNs hold a mathematical semantics, which favors a formal reasoning. They specify deterministic functional behaviors. In [8], KPNs are used for DSE of multimedia applications on multiprocessor SoCs (MPSoCs).

978-3-9810801-8-6/DATE12/©2012 EDAA

While the unboundedness assumption of KPN channels has been pointed out as a limitation, e.g. to deal with memory dimensioning [9], a synchronous variant [10] based on periodic abstract clocks, has been defined with N-bounded channels for synchronizability analysis between processes.

Synchronous dataflows (SDFs) [11] are similar models specifying nodes exchanging data tokens via oriented edges. The token consumption and production rates are defined statically. SDFs are statically analyzable and schedulable based on balance equations on the produced/consumed token rates. This is not the case of KPNs, which are scheduled dynamically in general. The StreamIt language [12] shares several features with SDFs. It proposes a compiler enabling optimizations for streaming applications. In [13], SDFs are used as in an optimization of streaming applications on heterogeneous execution platforms, mixing FPGA and CPUs. In [14], they are used in DSE for multimedia applications. SDFs are not explicitly clocked, which is a limitation for expressing multi-clock behaviors in combined software, hardware and environment specifications. For this reason in [9], authors consider a translation from SDF models to synchronous models.

Synchronous languages [6], e.g. Esterel, Lustre and Signal, enable the reliable development of safety critical embedded systems by promoting correct-by-construction designs. The "synchrony" hypothesis of these languages abstracts the quantitative time properties of embedded systems by considering that a program is faster than its environment. While it is very suitable for correctness analysis, it imposes *in fine* the validation of quantitative time aspects by finding execution platforms that guarantee the hypothesis. Our approach enriches the synchronous model with quantitative time via abstract clocks. The resulting model therefore provides a support for design assessment w.r.t. quantitative properties.

III. Abstract clocks for application modeling

We define basic notions about abstract clocks and affine clock relations. The definition of these notions is inspired by [15]. The proposed clock model is advocated as an internal model for the analysis of specifications defined, e.g. in SDFs, synchronous languages or the recent *clock constraint specification language* (CCSL) [16]. The translation from SDFs can be defined based on periodic behavior traces corresponding to their self-time scheduling as illustrated in [9], while this is quite straightforward from the other formalisms since they manipulate very similar concepts.

A. Abstract clocks (component activations)

Let us consider the following sets: $\mathbb{X}$ is a countable set of variables; $\mathbb{V}$ is a value domain; and $\mathbb{T}$ is a dense set equipped with a partial order $\leq$, with a lower bound. The elements of $\mathbb{T}$ are called *tags* or *logical instants*.

Definition 1 (Observation points): The tags of a set $\mathcal{T} \subset \mathbb{T}$ are observation points if: *i)* $\mathcal{T}$ is countable; *ii)* $\mathcal{T}$ holds a lower bound for the $\leq$ relation; *iii)* $\leq$ is well-founded on $\mathcal{T}$, i.e., there exists no infinite series (t_n) s. t. $\forall n \in \mathbb{N}, t_{n+1} \leq t_n$. □

The set $\mathcal{T}$ provides a discrete time dimension that corresponds to logical instants according to which the presence and absence of events can be observed during a system execution.

A *chain* $C \subseteq \mathcal{T}$ is a totally ordered set admitting a lower bound. The set of all chains is denoted by $\mathcal{C}$ and the set of all possible chains in a set of observation points $\mathcal{T}$ is noted $\mathcal{C}_\mathcal{T}$.

Definition 2 (Events, Signals): Given a set of observation points $\mathcal{T}$, an event e is a pair $(t, v) \in \mathcal{T} \times \mathbb{V}$. A signal s is a partial function $C \rightharpoonup \mathcal{V}$ that associates values with observation points belonging to a chain $C \in \mathcal{C}_\mathcal{T}$. □

We denote by $tags(s)$ the set of tags associated with a signal s, i.e. the domain of the signal s.

Definition 3 (Clocks): Given a signal s, its corresponding *abstract clock* is the totally ordered set $tags(s)$. □

Clocks can be combined by considering the usual set operations: intersection, union and difference. In order to determine the relative distance from any tag to its neighbors, a reference time, such as $\mathbb{N}$ the set of natural numbers, is required. We define *affine clocks* as ordered sets of instants with positions identified by an affine enumeration w. r. t. a reference time.

Definition 4 (Affine clocks w. r. t. $\mathbb{N}$): An abstract clock c is said to be affine if its associated tags can be characterized with an affine function according to a reference time $\mathbb{N}$: $c = \{\pi\tau + \phi \mid \pi \in \mathbb{N}\backslash\{0\}, \phi \in \mathbb{Z}, \tau \in \mathbb{N}\}$. □

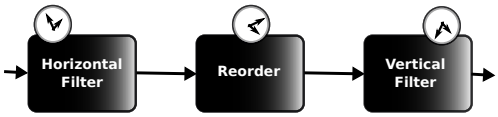


Fig. 1. Image downscaling process.

Let us consider a model of image downscaling process [10] as shown in Fig. 1. It re-sizes images by reducing the size first horizontally, then reordering the result, and finally reducing the size vertically. The three components that achieve the global functionality are assumed to be associated with abstract clocks c_1, c_2 and c_3. For the sake of simplicity, we will consider this very simple model as an illustrative example along the paper.

Fig. 2 shows an example of trace w.r.t. a reference time, characterizing c_1, c_2 and c_3 defined by the sets: $c_1 = \{\tau \mid \tau \in \mathbb{N}\}, c_2 = \{2\tau + 1 \mid \tau \in \mathbb{N}\}$ and $c_3 = \{6\tau + 3 \mid \tau \in \mathbb{N}\}$.

ref. time	0	1	2	3	4	5	6	7	8	9	10	...
c_1	•	•	•	•	•	•	•	•	•	•	•	...
c_2		•		•		•		•		•		...
c_3				•						•		...

Fig. 2. Trace of affine clocks c_1, c_2 and c_3.

Affine clocks are particularly suitable for expressing streaming algorithms, which are often defined in a regular and synchronized manner. For specification convenience, an abstract clock can be represented by a binary word [16] [10]. Given a clock c and a chain $C \in \mathcal{C}_\mathcal{T}$ s. t. $c \subseteq C$, the *binary encoding* of c is defined by a function $b : C \to \{0, 1\}$ as follows:

$$\forall t \in C, b(t) = \begin{cases} 1 & \text{if } t \in C \cap c \text{ (i.e., event presence)} \\ 0 & \text{if } t \in C\backslash c \text{ (i.e., event absence).} \end{cases}$$

For instance, compact binary representations of c_1, c_2 and c_3 in Fig. 2 are respectively $(1)^\omega, 0(10)^\omega$ *and* $000(100000)^\omega$

where $\omega \in \mathbb{N}$. E.g., in $0(10)^\omega$, the first 0 is the phase and (10) is the period. Fig. 3 illustrates a corresponding trace.

ref. time	0	1	2	3	4	5	6	7	8	9	10	...
c_1	1	1	1	1	1	1	1	1	1	1	1	...
c_2	0	1	0	1	0	1	0	1	0	1	0	...
c_3	0	0	0	1	0	0	0	0	0	1	0	...

Fig. 3. Binary trace of affine clocks c_1, c_2 and c_3.

The multi-clock feature of the above clock modeling is an important ingredient to capture the combinatorics of components interaction in complex streaming applications. Such interactions are quite easily specified with clock relations.

B. Abstract clock relations (components interaction)

Since abstract clocks are sets of instants, the usual sets comparison naturally applies to them: equality, inclusion and exclusion. Further relations such as alternation of instants belonging to different clocks can be also specified [16]. In the following, we mainly focus on the (n, ϕ, π)-*affine clock relation* ($n, \pi \in \mathbb{N}\backslash\{0\}, \phi \in \mathbb{Z}$) between two clocks c_1 and c_2.

Definition 5 (Affine clock relation): Two abstract clocks c_1 and c_2 are said to be in (n, ϕ, π)-affine relation, noted $c_1 \overset{(n,\phi,\pi)}{\rightarrow} c_2$, if they satisfy the following: by inserting $(n-1)$ tags between any two successive tags of c_1, then c_2 is composed of each π^{th} tag in the extended c_1, starting from the $(\phi+1)^{th}$ tag, where $n, \pi \in \mathbb{N}\backslash\{0\}$ and $\phi \in \mathbb{Z}$. □

In this paper, we only consider affine (n, ϕ, π)-relations where $n = 1$. Such relations hold for the clocks of shown in Fig. 2:

$$c_1 \overset{(1,\phi_1,\pi_1)}{\rightarrow} c_2 \quad \text{and} \quad c_2 \overset{(1,\phi_2,\pi_2)}{\rightarrow} c_3 \tag{1}$$

where $(\phi_1, \pi_1) = (1, 2)$ and $(\phi_2, \pi_2) = (1, 3)$.

Such affine clock relations are composable [17]. The following property characterizes this composition.

Property 1 (Affine relation composition): Given three abstract clocks c, c' and c'' s. t. $c \overset{(1,\phi,\pi)}{\rightarrow} c'$ and $c' \overset{(1,\phi',\pi')}{\rightarrow} c''$ (where $\phi' \in \mathbb{N}$), the composition of the two affine relations is an affine relation between c and c'': $c \overset{(1,\phi+\pi\phi',\pi\pi')}{\rightarrow} c''$. □

Another useful notion is synchronizability, which allows to guarantee the existence of a dataflow-preserving way to make two affine clocks synchronous. In other words, a finite-size buffer protocol can be defined to synchronize such clocks.

Property 2 (Affine clock synchronizability): Given four clocks c, c', c'' and c_σ related by relations s. t. $c \overset{(1,\phi,\pi)}{\rightarrow} c'$, $c' \overset{(1,\phi',\pi')}{\rightarrow} c''$ and $c \overset{(1,\phi_\sigma,\pi_\sigma)}{\rightarrow} c_\sigma$, the clocks c'' and c_σ are synchronizable iff: *i)* $\phi + \pi\phi' = \phi_\sigma$ and *ii)* $\pi\pi' = \pi_\sigma$. □

In the sequel, we show how to reason about application timing behavior w. r. t. environment constraints.

IV. Modeling of application environment

We use abstract clocks to capture the way an application interacts with its environment. Typically for the downscaler, let us assume a pixel sensor that gathers images and sends them to the downscaler in the form of pixel flows. Then, the downscaler transforms the received pixels in order to present re-sized images to a display screen at some rate. Here, the sensor and the display play the role of the environment for the downscaler application. Let us denote by c_s and c_d the respective logical clocks of the above pixel sensor and display screen. They respectively represent pixel arrival rate in the sensor and image display rate on the screen. The whole model works as follows: the sensor produces pixel by pixel; the downscaler periodically performs an operation whenever it receives from the sensor a given number of pixels; and the screen periodically displays images whenever a certain number of transformed pixel blocs is received from the downscaler. As a result, there are affine clock relations on the one hand between the sensor and the downscaler, and on the other hand between the downscaler and the display screen as follows:

$$c_s \overset{(1,\phi_s,\pi_s)}{\rightarrow} c_1 \tag{2}$$

$$c_3 \overset{(1,\phi_3,\pi_3)}{\rightarrow} c_d \tag{3}$$

where $(1, \phi_s, \pi_s)$ and $(1, \phi_3, \pi_3)$ are the affine relation parameters. An affine clock relation can therefore be inferred directly between the sensor and the vertical filter via the horizontal filter and the reorder component. This is achieved by composing all intermediate clock relations, i.e. (2) and (1):

$$c_s \overset{(1,\phi_s+\pi_s\phi_1+\pi_s\pi_1\phi_2,\pi_s\pi_1\pi_2)}{\rightarrow} c_3. \tag{4}$$

The above clock relations (3) and (4) define the resulting property of the application under design. Now, some expected quality of service (QoS) requirements can be addressed in order to check whether or not the clock rates considered for the different components meet them. Typically, for given rate values in the design, a specific image display rate may be required to comply with a product characteristics for user convenience. Let us specify such a QoS requirement as a new affine relation between the clock c_s of the sensor component and a clock c_{QoS} denoting the required rate for image display:

$$c_s \overset{(1,\phi_q,\pi_q)}{\rightarrow} c_{QoS}. \tag{5}$$

Checking whether or not relation (5) is achievable with chosen rate values in the design amounts to show the synchronizability of the clocks c_d and c_{QoS}. Hence, by applying Property 2, one has to ensure that the parameters of involved affine clock relations satisfy the specified properties, i.e.:

$$\begin{cases} (\phi_s + \pi_s\phi_1 + \pi_s\pi_1\phi_2) + (\pi_s\pi_1\pi_2)\phi_3 = \phi_q \\ (\pi_s\pi_1\pi_2)\pi_3 = \pi_q \end{cases} \tag{6}$$

Thanks to this clock based reasoning, a designer can get useful insights about the synchronizability of application components w.r.t. its environment constraints. Another important design issue concerns the execution platform, which must be configured so as to provide the performance level required by an application. This is addressed in the next section.

V. Modeling of execution platforms

We show how abstract clocks allow one to model and reason about the execution of applications on hardware platforms [18]. We only assume abstract clocks with finite size. This is implemented in a prototype tool, called *Clock Analysis System*.

A. Clock model of platform behavior

a) Execution platform: We consider a generic parallel architecture model composed of processors operating synchronously according to a global clock and communicating via a shared memory. For instance, let us consider an architecture with three processors P_1, P_2 and P_3 with the initial frequencies $f_1 = 100MHz, f_2 = 50MHz$ and $f_3 = 40MHz$ respectively. We use the periods $1/f_i$ of processes to define their activation instants. For synchronization purpose, we also consider a reference clock κ with a period of $1/LCM(f_1, f_2, f_3)$, where LCM denotes the Least Common Multiple.

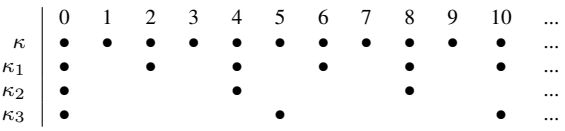


Fig. 4. Clock trace of processors.

Fig. 4 depicts the periodic activation rates of processors P_1, P_2 and P_3 according to their periods. We refer to these activations as *processor clocks* κ_j.

b) Clock projection: To capture mapping and scheduling choices with abstract clocks, we define a projection of application clocks c_i on processor clocks κ_j (see Algorithm 1). We use this projection to describe the scheduling of tasks on processors. Two parameters δ and ρ represent respectively the number of processor cycles corresponding to a task activation (i.e., value 1 in associated clock c_i) and the synchronization delay corresponding to each logical instant where a task is waiting and is not active (i.e., value 0 in its clock c_i). These values can be determined statically according to the target processors. Worst-case estimations may also be considered.

The projection of application clocks on processor clocks is central. It is a refinement of the synchrony hypothesis assumed in application clocks, where each activation instant denotes an "instantaneous" execution of a function. The projection splits such an instant into corresponding processor cycles. The result captures a temporal behavior of a task, which is closer to its actual execution. In order to preserve the ordering of events characterized by an application clock, a projection must be monotonic (increasing) [19]. The same must be guaranteed for events from different application clocks.

In Algorithm 1, the occurrence of 1 at an instant in a clock c'_i indicates that the processor corresponding to κ_i is *active* at that instant. The value 0 indicates that the processor is in the *Nop* state. The meaning of -1 is contextual: when a sequence of such a value is immediately preceded by 1, then it denotes *potentially active at those instants*; otherwise, it denotes *idle*.

Fig. 5 illustrates projections of application clocks c_1, c_2 and c_3 shown in Fig. 3 on processor clocks κ_1, κ_2 and κ_3 shown in Fig. 4, where a task activation consists of $\delta_1 = 3, \delta_2 = 2$ and $\delta_3 = 1$ processor cycles respectively and $\rho = 1$. We can observe the monotonicity of the projection as follows:

- $c_1[0] \mapsto c'_1[0], c_1[1] \mapsto c'_1[6], c_1[2] \mapsto c'_1[12]$, ...
- $c_2[0] \mapsto c'_2[0], c_2[1] \mapsto c'_2[4], c_2[2] \mapsto c'_2[12]$, ...
- $c_3[0] \mapsto c'_3[0], c_3[1] \mapsto c'_3[5], c_3[2] \mapsto c'_3[10]$, ...

Algorithm 1 Clock projection

Inputs: let c_i and κ_j be two clocks with sizes (i.e., number of instants) s. t.: $|\kappa_j| \geq |c_i| \geq 1$. Parameters δ and ρ are given.
Local: pos stores positions of c_i's tags on κ_j after projection.
Output: the result of the projection is a clock c'_i

```
c'_i := κ_j;
for all α ∈ [1..|c'_i|] do
    c'_i[α] := −1;
end for
pos[0] := 0;
for all α ∈ [1..|c_i|] do
    if c_i[α − 1] = 1 then
        pos[α] := pos[α − 1] + δ;
    else if c_i[α − 1] = 0 then
        pos[α] := pos[α − 1] + ρ;
    end if
end for
for all α ∈ [1..|c_i|] do
    c'_i[pos[α]] := c_i[α];
end for
return c'_i;
```

c) Scheduling of tasks on processors: We distinguish several task schedulings on an execution platform:

- *Mono-task scheduling on a processor.* Each processor executes one task (itself executed only on this processor). For each task associated with application clock c_i, its scheduling on a processor associated with clock κ_j is defined by a projection of c_i on κ_j (given δ and ρ).
- *Multi-task scheduling on a processor.* To schedule several tasks on a single processor, we consider two alternatives:
  - *Non-preemptive static scheduling*: tasks are scheduled according to the data dependency specified in an application. When a task is scheduled, it runs until completion before another task gets scheduled. E.g., if the horizontal filter and reorder components of the downscaler are scheduled on the same processor, the clock c_1 associated with the former component is first projected on the considered processor clock κ_j, then the clock c_2 of the latter is projected on κ_j.
  - *Time division multiple access (TDMA)*: tasks are allocated time slots during which they can execute on the processor. A time slicing is applied to the application clocks of all tasks mapped onto the processor. For each clock c_i, this yields a set of subsequences of c_i, referred to as slices s_i^k. Then, we alternate the projections of these *slices* from a task to another on the considered processor clock κ_j. Fig. 6 illustrates such a scheduling where two tasks are executed on a processor associated with clock κ_1.

Beyond the above two schedulings, one may also need to *schedule either multiple tasks on multiple processors, or one task on several processors* (useful for a parallel execution of a task). These last cases are solved in a similar way as above.

B. Architecture choice assessment

We assess the design choices resulting from the previous section, w.r.t. the correctness of application clock properties,

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	...
κ	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	...
κ_1	•		•		•		•		•		•		•		•		•		•		•		...
c'_1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	...
κ_2	•				•				•				•				•				•		...
c'_2	0	-1	-1	-1	1	-1	-1	-1	-1	-1	-1	-1	0	-1	-1	-1	1	-1	-1	-1	-1	-1	...
κ_3	•					•					•					•					•		...
c'_3	0	-1	-1	-1	-1	0	-1	-1	-1	-1	0	-1	-1	-1	-1	1	-1	-1	-1	-1	0	-1	...

Fig. 5. Trace after a projection of clocks c_1, c_2 and c_3 of Fig. 2 on κ_1, κ_2 and κ_3 respectively ($\delta_{c_1} = 3, \delta_{c_2} = 2$ and $\delta_{c_3} = 1$ and $\rho_{c_1} = \rho_{c_2} = \rho_{c_3} = 1$).

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	...
κ_1	•		•		•		•		•		•		•		•		•		•		•		•	...
c'_1	1	-1	-1	-1	-1	-1	1	-1	-1	-1	-1	-1									1	-1	-1	...
c'_2													0	-1	1	-1	-1	-1	-1	-1				...

Fig. 6. TDMA-scheduling of clocks c_1 and c_2 of Fig. 2 on κ_1 ($\delta_{c_1} = 3, \delta_{c_2} = 2$ and $\rho_{c_1} = \rho_{c_2} = 1$). Slices are $s_1 = (11)$ for c_1 and $s_2 = (01)$ for c_2.

performance and energy consumption.

d) Application clock properties (i.e., causality): It is important to preserve application properties after the mapping and scheduling on processors. In other words, the global causality relations between application clocks c_i must be guaranteed after their projection on processor clocks κ_j. This is achieved according to the following parameters:

- the synchronization delay ρ: its value approximation should be defined such that the number of processor cycles corresponding to each 0 value preceding activations, i.e. value 1 in c_i, is high enough to meet all synchronization requirements before any activation;
- processor frequency values: a global causality problem (between different projected clocks) also arises when a processor P_2 executing a data consuming task, is active very often before the required activations of a processor P_1 executing a data producing task. Thus, the choice of processor frequencies must satisfy the global causality.

Notice that the modification of processor frequency values can be used to correct a pessimistic estimate of the synchronization delay ρ that does not ensure the causality property.

e) Temporal performance: To evaluate the performance of a system, we compute the workload of each processor as the total number of executed processor cycles after mapping and scheduling. It corresponds to the length of the clocks resulting from projections, e.g., c'_1, c'_2 and c'_3 in Fig. 5. If a processor has a frequency f and performs Δ cycles during application execution, then its corresponding execution time is: $\frac{\Delta}{f}$. For instance, in Fig. 5, the execution of the first two activations in c_1 takes $\frac{6}{100} = 0.06\mu s$ when P_1 runs at $100MHz$.

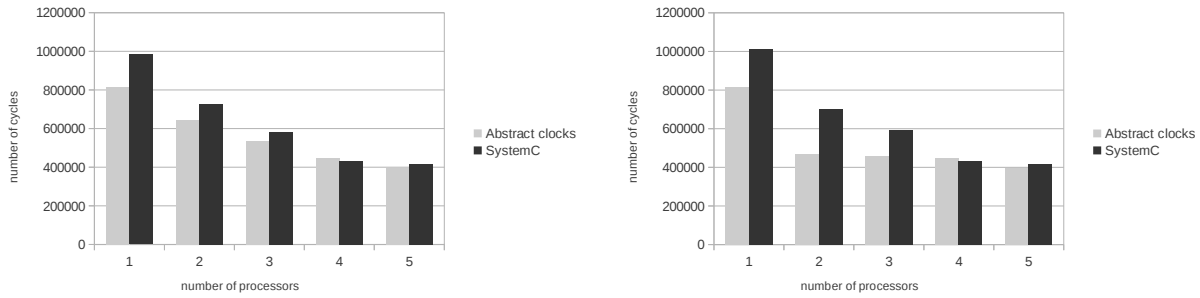
Fig. 7 shows a comparison of our abstract clock based approach with a SystemC cycle-accurate simulation for a JPEG decoder in the SoClib simulation environment [20]. The considered execution architecture includes a network-on-chip (NoC) connecting five MIPS processors with shared (multibank) memory. The JPEG decoder is composed of five pipelined tasks activated thirty six times. We first measured with SoClib, for each task T, an average number of processor cycles corresponding to a single activation. Then, we used this number as δ_T during the associated clock projection. The value of ρ parameter has been determined manually in such a way that all causality relations are satisfied.

We applied our clock based approach and compared it with equivalent simulations in SoClib. We observe that both approaches lead to the same tendency regarding the number of cycles corresponding to different mapping scenarios (see Fig. 7). Note that in the TDMA scheduling, the assignment of slices has an impact on the run-time of an application. Here, we consider a particular TDMA scheduling settings that associate with each task, a slice corresponding to the processing of 8×8 pixels, equivalent to $(\frac{1}{36})^{th}$ of considered images.

Of course, the SoClib simulation is unsurprisingly more precise. But, the important thing to notice here is that the same conclusions about the relative comparison between the different mapping scenarios can be obtained coherently with both approaches. To achieve the whole experiment (i.e., configure, execute and report), our approach requires a few minutes while SoClib necessitates a few hours. Since it is faster and more flexible due to its high abstraction level, it can be considered for an early rapid exploration to reduce a design space, before applying simulation and prototyping.

f) Energy consumption: During an execution, the *slack time* of a task is the difference between its completion time and its associated deadline. By modifying processor frequencies, this completion time can be either shortened or stretched. In particular, by reducing the frequency value in order to keep the slack time the shorter possible, the energy consumption is reduced, while the functional properties are still verified. Furthermore, such a frequency reduction can benefit to data storage optimization since processors executing faster than required may produce a high number of data, which necessitate large storage buffers [1]. Another way to deal with energy E in our framework is to consider the execution time T calculated previously, together with pre-determined power consumption W of every task by using tools such as Sim-Panalyzer [21], as follows: $E = T \times W$.

From a global point of view, the clock-based approach advocated in this work is a cost-effective and relevant means to facilitate the early analysis of complex design choices. It scales well for periodic clocks, which adequately abstract the computation regularity inherent to streaming applications. More generally, it is well-suited for time-triggered applications. Our approach is suitable for environments such as those adopting platform-based design [22], where high-level specifications of



(a) Non-preemptive static scheduling

(b) TDMA scheduling

Fig. 7. Execution time estimation for JPEG decoder of one image: abstract clocks *vs* SystemC simulation.

application functionality and hardware architecture are refined with well-characterized intellectual properties (IPs) and analyzed so as to rapidly converge towards design requirements.

VI. Conclusions

In this paper, we proposed a high-level formal modeling paradigm for combined software, hardware and environment specifications to overcome the design validation issues for streaming embedded systems, based on abstract clocks defined in synchronous languages. We showed how to check time requirements imposed by an environment on an streaming application. We also addressed the assessment of architecture choice for application the execution. Beyond the possibility to deal with design correctness while avoiding the usual tedious debugging and testing tasks, our solution offers a simple and fast alternative to explore and reduce complex design spaces before applying simulation and prototyping. It does not aim to replace completely these techniques since its accuracy is limited due to the high abstraction; instead, it is an ideal complement to them.

Acknowledgment

The author would like to thank the anonymous reviewers for their interesting feedback on this work. He also would like to thank his colleagues A. Abdallah, J.-L. Dekeyser, R. Ben Atitallah and S. Boumedien who contributed to the vision exposed here via several insightful discussions.

References

[1] M. Duranton, "The challenges for high performance embedded systems," in *DSD'06*. IEEE Computer Society, 2006, pp. 3–7.

[2] D. Hedde, P.-H. Horrein, F. Petrot, R. Rolland, and F. Rousseau, "A mp-soc prototyping platform for flexible radio applications," in *Euromicro DSD'09*. IEEE Computer Society, 2009, pp. 559–566.

[3] B. Bailey and G. Martin, *ESL Models and their Application: Electronic System Level Design and Verification in Practice*. Springer Publishing Company, Incorporated, 2010.

[4] S. Abdi, Y. Hwang, L. Yu, G. Schirner, and D. D. Gajski, "Automatic TLM Generation for Early Validation of Multicore Systems," *IEEE Design and Test of Computers*, vol. 28, pp. 10–19, 2011.

[5] A. Gerstlauer, "Host-compiled simulation of multi-core platforms," in *Proceedings of the 21st IEEE International Symposium on Rapid System Prototyping, RSP 2010, Fairfax, VA, USA,*, 2010, pp. 1–6.

[6] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone, "The synchronous languages twelve years later," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 64–83, January 2003.

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

[8] 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.

[9] J. Zhu, I. Sander, and A. Jantsch, "Energy efficient streaming applications with guaranteed throughput on mpsocs," in *EMSOFT'08, Atlanta, GA, USA*, 2008, pp. 119–128.

[10] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet, "N-sychronous Kahn networks," in *ACM Symp. on Principles of Programming Languages (PoPL'06)*, Charleston, South Carolina, USA, January 2006.

[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] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe, "Streamit: A compiler for streaming applications," MIT, Cambridge, MA, Tech. Memo LCS-TM-622, December 2001. [Online]. Available: http://cag.lcs.mit.edu/commit/papers/01/StreamIt-TM-622.pdf

[13] J. Zhu, I. Sander, and A. Jantsch, "Pareto efficient design for reconfigurable streaming applications on cpu/fpgas," in *DATE'2010*, 2010, pp. 1035–1040.

[14] Y. Yang, M. Geilen, T. Basten, S. Stuijk, and H. Corporaal, "Automated bottleneck-driven design-space exploration of media processing systems," in *DATE'2010*, 2010, pp. 1041–1046.

[15] P. Le Guernic, J.-P. Talpin, and J.-C. Le Lann, "Polychrony for system design," *Journal for Circuits, Systems and Computers*, vol. 12, pp. 261–304, 2002.

[16] C. André and F. Mallet, "Clock Constraints in UML/MARTE CCSL," INRIA, Research Report RR-6540, 2008. [Online]. Available: http://hal.inria.fr/inria-00280941/PDF/rr-6540.pdf

[17] I. Smarandache, T. Gautier, and P. Le Guernic, "Validation of Mixed Signal-Alpha Real-Time Systems through Affine Calculus on Clock Synchronisation Constraints," in *World Congress on Formal Methods (2)*, 1999, pp. 1364–1383.

[18] A. Abdallah, A. Gamatié, and J.-L. Dekeyser, "Correct and Energy-Efficient Design of SoCs: the H.264 Encoder Case Study," in *Proceedings of the International Symposium on System-on-Chip (SoC'10)*, 2010.

[19] T. Melham, *VLSI Specification, Verification and Synthesis*. Kluwer Academic Publishers, 1987, ch. Abstraction Mechanisms for Hardware Verification, pp. 129–157.

[20] "The SoClib Project," 2011, http://www.soclib.fr.

[21] "The SimpleScalar-ARM Power Modeling Project," 2011, http://www.eecs.umich.edu/~panalyzer.

[22] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi, "Benefits and challenges for platform-based design," in *Proceedings of the 41st annual Design Automation Conference*, ser. DAC'04, 2004, pp. 409–414.