Self-Timed Periodic Scheduling For Cyclo-Static DataFlow Model

Amira Dkhil, Xuankhanh Do, Paul Dubrulle, Stéphane Louise, Christine Rochange

▶ **To cite this version:**

Amira Dkhil, Xuankhanh Do, Paul Dubrulle, Stéphane Louise, Christine Rochange. Self-Timed Periodic Scheduling For Cyclo-Static DataFlow Model. Workshop on Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems - ALCHEMY 2014, Jun 2014, Cairns, Australia. pp.1134–1145, 10.1016/j.procs.2014.05.102 . hal-01136092

HAL Id: hal-01136092
https://hal.science/hal-01136092v1

Submitted on 26 Mar 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Open Archive TOULOUSE Archive Ouverte (OATAO)
OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible.

This is an author-deposited version published in : http://oatao.univ-toulouse.fr/
Eprints ID : 12956

To link to this article : DOI :10.1016/j.procs.2014.05.102
URL : http://dx.doi.org/10.1016/j.procs.2014.05.102

To cite this version : Dkhil, Amira and Do, Xuankhanh and Dubrulle, Paul and Louise, Stéphane and Rochange, Christine *Self-Timed Periodic Scheduling For Cyclo-Static DataFlow Model*. (2014) In: Workshop on Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems - ALCHEMY 2014, 10 June 2014 - 12 June 2014 (Cairns, Australia).

Any correspondance concerning this service should be sent to the repository administrator: staff-oatao@listes-diff.inp-toulouse.fr

Self-Timed Periodic Scheduling For a Cyclo-Static DataFlow Model

Amira Dkhil[1], XuanKhanh Do[1], Paul Dubrulle[1], Stéphane Louise[1], and Christine Rochange[2]

[1] CEA, LIST, PC172, 91191, Gif-sur-Yvette, France.
firstname.lastname@cea.fr
[2] IRIT, Université de Toulouse, 118, route de Narbonne, Toulouse cedex 9, France.
rochange@irit.fr

Abstract
Real-time and time-constrained applications programmed on many-core systems can suffer from unmet timing constraints even with correct-by-construction schedules. Such unexpected results are usually caused by unaccounted for delays due to resource sharing (*e.g.* the communication medium). In this paper we address the three main sources of unpredictable behaviors: First, we propose to use a deterministic Model of Computation (MoC), more specifically, the well-formed CSDF subset of process networks; Second, we propose a run-time management strategy of shared resources to avoid unpredictable timings; Third, we promote the use of a new scheduling policy, the so-said Self-Timed Periodic (STP) scheduling, to improve performance and decrease synchronization costs by taking into account resource sharing or resource constraints. This is a quantitative improvement above state-of-the-art scheduling policies which assumed fixed delays of inter-processor communication and did not take correctly into account subtle effects of synchronization.

Keywords: Many-core systems, Real-Time, Data-Flow, Scheduling, Latency, Guarantees

1 Introduction

Real-time embedded systems require both functionally correct and temporally predictable executions. Given the scale of new massively parallel systems, such as the SThorm chip from STMicroelectronics (64 cores) or the MPPA chip from Kalray (256 cores) [5], the scheduler design of tasks and communications becomes more complicated and its impact upon the entire system performance becomes more significant. Consequently, multiprocessor scheduling has been an active area and therefore many scheduling and resource management solutions was suggested. The Self-Timed Scheduling (STS) strategy (also known as as-soon-as-possible), is considered as the most appropriate policy for streaming applications [14, 15, 17]. In Dataflow models, each actor (*i.e.* process) firing starts as soon as their firing rules are satisfied.

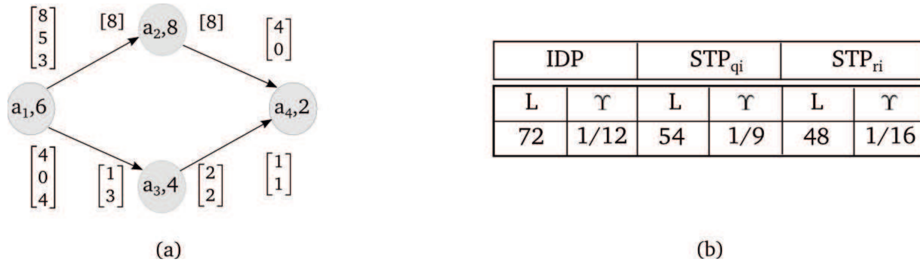
Such models of computation (MoC) offer a good visibility of datapath, actual parallelism in a given application and synchronization points, which make such paradigm especially well fitted. Programming languages like StreamIt [18], or ΣC [8] offer even more: An underlying MoC which is deterministic, and purely data-driven. For both, the MoC is a subset of Khan Process Network (KPN [9]), which can provably run in bounded memory. This is well-formed Static DataFlow (SDF [12]) for StreamIt and a superset of well-formed Cyclostatic DataFlow (CSDF [3]) for ΣC. The basic principle of KPN is simple: Processes are linked with FIFO channels for communication. All FIFO channels are read-blocking and write are non-blocking. In the case of SDF, for each firing of a process, a fixed amount of data tokens are expected on each input channel, and a fixed amount of data token are produced on each output channel. If any input channel has an insufficient amount of data tokens, the associated process cannot be fired. For CSDF, the principle is the same, but the number of tokens produced or consumed on each channel can vary in a cyclic way from one firing of the process to the next (see Figure 1).

For timing concerns, compared to using worst-case execution times, self-timed scheduling will always do at least as well. Nonetheless, this result can only be true if synchronization times are negligible. Synchronization is a special form of communication, in which the data is an information for the control of the application. It has a dual role: (1) Enforcing the correct sequencing of actors firing, and (2) Ensuring the mutually exclusive access to certain shared data. To cope with such a requirement, STS introduces explicit synchronization checks whenever two processors communicate. In this case, each synchronization can cost up to four accesses to shared memory [16]. As a consequence, due to the complex and irregular dynamics of self-timed operations, in addition to the high synchronization overhead, many different assumptions were imposed, like uniform task execution times or contention-free communication. However, neglecting subtle effects of synchronization or considering uniform costs for communication operations like in [2, 15] is dangerous and not realistic with regards to actual systems and applications. Unless a special hardware for ordered communication transactions [17, 10] or contention free communications [14] (*e.g.*, Time Division Multiple Access (TDMA), Round-Robin (RR)) that maintains a predefined schedule of accesses to the shared memoryis employed, analysis and optimization of self-timed systems under real-time constraints remains challenging. Nowadays, periodic scheduling is receiving much more attention for streaming applications [2, 6, 15]. These algorithms provide many nice properties such as timing guarantees for applications, temporal isolation [4] and low complexity of the schedulability tests. It was shown that interprocessor communication overhead can be defined as a monotonically increasing function of the number of conflicting memory accesses in a given period of the schedule [6]. Nonetheless, periodic scheduling achieves optimal performance solely for matched I/O graphs *i.e* a graph where the product of actor's worst-case execution time and repetition is the same for all actors. As in the real world, execution time of processes can vary largely, it is difficult to prove that a graph is a matched I/O (or to build one).

In this paper, we show that it is possible to guarantee the matched I/O property for any Cyclo-Static DataFlow (CSDF) graph by using a new scheduling policy noted Self-Timed Periodic (STP) Schedule. STP is a hybrid execution model based on mixing Self-Timed schedule and periodic schedule while considering variable IPC times. To illustrate the impact of the STP model on performance, we present the following motivational example.

1.1 Motivational Example

For any consistent CSDF graph, we can show a periodical firing of actors which will return to a steady state (usually the initial state). This periodical firing is characterized by firing vectors,



IDP		STP_{qi}		STP_{ri}	
L	ϒ	L	ϒ	L	ϒ
72	1/12	54	1/9	48	1/16

Figure 1: (a) CSDF graph (b) Throughput and latency metrics for the mismatched I/O CSDFG

and can be calculated from the topology of the graph and the amount of tokens consumed and produced by each firing of actors.

For example, in Figure 1(a), we show a CSDF graph of 4 actors and 4 communication channels. The CSDF graph is characterized by two repetition vectors $\vec{q}$ and $\vec{r}$. $\vec{r}$ is the minimal set of actor firings returning the dataflow graph to its initial state. For the example depicted in Figure 1(a), $\vec{r} = [1, 2, 2, 4]$ and $\vec{q} = [3, 2, 4, 8]$. $\vec{q}$ is the minimal set of sub-tasks firings returning the dataflow graph to its initial state. In fact, each actor in the graph is executed through a periodically repeated sequence of subtasks. For example, if $r_1 = 1$ then $q_1 = 3$ because actor a_3 contains 3 subtasks (*i.e.* to get q_i, we multiply r_i by the length of the consumption and production rates of a_i). The worst-case computation and communication time of each actor is shown next to its name between round brackets (*e.g.* 6 for a_1). This graph is an example of a mismatched I/O graph since the product of actor execution times and repetition is not the same for all actors (*e.g.* $1 \times 6 \neq 2 \times 8$). Let Υ and L denote the throughput (*i.e.* rate) and latency of graphs G, respectively, derived in Figure 1(b) for the example of Figure 1(a). It has been shown that optimal throughput and latency of a matched I/O dataflow graph can be achieved under Implicit-Deadline Periodic (IDP) schedule [2]. However, for mismatched I/O graph, it pays a high price in terms of increased latency and decreased throughput. Instead, if the actors are to be scheduled as Self-timed Periodic (STP) tasks, then it is possible to have 25% to 40% improvement compared to the IDP schedule. In our contribution, we propose two granularities of scheduling. This depends on whether we use q_i or r_i as the basic repetition vector of CSDF. For the proposed example, including the subtasks of actors results in better performance for latency. However, for throughput STP_{q_i} gives better results[1].

1.2 Paper Contributions

We propose two classes of STP schedules based on two different granularities. The first schedule is based on the repetition vector q_i without including the subtasks of actors. The second schedule has a finer granularity and includes the subtasks of actors. It is based on the repetition vector r_i. For mismatched I/O graphs, we show that it is possible to significantly decrease the latency and increase the throughput under the STP model for both granularities. We will show that our approach guarantees the property of matched I/O rates for any CSDF graph and consequently guarantees optimal performance.

The remainder of this paper is organized as follows. In Section 2, we represent a state of the art methods relative to the scheduling policies of Multi-Rate DataFlow (MRDF) graphs on

[1] Let's assume that an execution under STS has an additional cost of synchronization equal to 25% the computation time of actors, the latency will be equal to $L = 47$ units of time. Thus, for a mismatched graph example, the STP model is equally good to STS in terms of throughput and latency.

multiprocessor systems. The considered model is described in Section 3. Our main contribution is presented in Section 4. We finish with sections 5 and 6 where we present the case studies and we state the conclusions.

2 Related Work

The most prominent performance metrics of concurrent real-time applications are throughput and latency. Optimizing or analyzing latency of a stream program, in general, requires to find a good tasks scheduling among the computing units. A schedule is more efficient if it hides communication latencies whenever possible. In [10], Khandalia et al. explored the problem of efficiently ordering interprocessor communication operations in statically scheduled multiprocessors. Their method is based on finding a linear ordering of communication actors at compile-time which would minimize synchronization and arbitration costs, but at the expense of run-time flexibility. In [17], the author proposes to schedule all communications as well as all computations to eliminate shared resource contention. Their approach is based on using a hardware central transaction controller that maintains a predefined schedule of accesses to the shared memory. Another approach in [7] is based on Scenario-Aware Data-Flow (SADF). In such model, an application is modeled as a collection of SDF graphs, each representing individual scenarios of behavior, and a Finite State Machine (FSM) that specifies the possible orders of scenario occurrences. The paper provides techniques to analyze worst-case performance analysis (*i.e.* highest throughput and minimal latency) of such applications. SADF was primarily designed as a way of modeling behavior and not as a programming model. For this reason, the execution model is not explicit about scenario transition decisions and data-flow relation (*i.e.* there is no way of knowing where the decision to go from one scenario to another was taken). In [2], Bamakhrama and Stefanov present a complete framework for computing the periodic task parameters using an estimation of worst-case execution time. They assume that each write or read has constant execution time but this is often not true. Our approach is somewhat similar to [2] in using the periodic task model which allows applying a variety of proven hard-real-time scheduling algorithms for multiprocessors. However, it is different in the way of using the periodic behavior because actors will no longer be strictly periodic but self-timed assigned to periodic levels. In addition, we treat the case variable execution time of actors due to synchronization and contention in shared resources.

3 Model of Computation and Terminology

3.1 Timed Graph

The timed graph is a more accurate representation of the CSDF graph, that associates to each subtask or instance of an actor a computation time and a communication overhead. We consider the Timed graph $G = (A, E, \omega, f_\varphi)$. The set of actors is denoted by $A = \{a_1^1, a_1^2, ..., a_1^{J_1}, ..., a_n^1, a_n^2, ..., a_n^{J_n}\}$, where n is the total number of actors in the CSDF graph. The set of edges is denoted by $E = \{E_1, E_2, ..., E_v\}$. Each parallel actor a_i is represented by a Directed Acyclic Graph (DAG) and consists in a set of nodes and directed relations. The nodes represent the instances of an actor, while the directed relations show the FIFO buffers. Each instance a_i^j is viewed as executing through a periodically-repeating sequence of sub-tasks of length $\tau_i \in \mathbb{N}^\star$. The production and consumption behavior of an actor is constant for a given sub-task but may vary across the different sub-tasks.

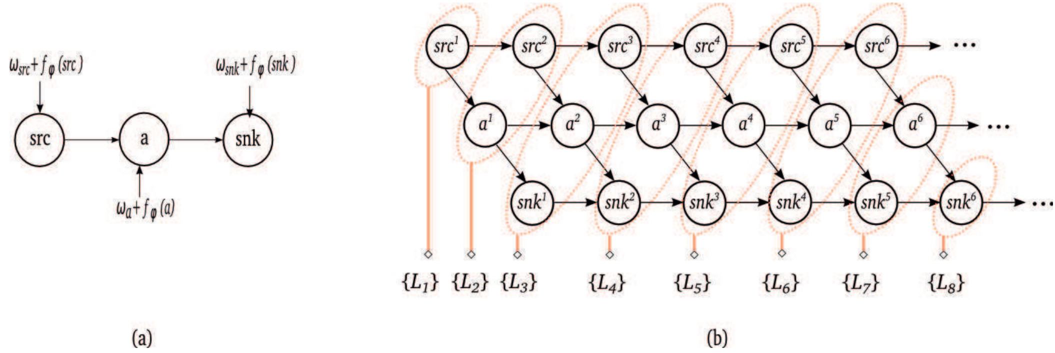


Figure 2: (a) Acyclic Timed dataflow graph of a pipeline example with timing parameters (b) Decomposition approach example

A real-time DAG of a_i is characterized by $(\tau_i, J_i, \omega_i, f_\varphi(i), D_i)$, where J_i is the unfolding factor of a_i, ω_i is the worst-case computation time, $f_\varphi(i)$ is its communication time according to schedule φ and D_i is the relative deadline.

For many non-terminating dataflow graphs, the execution can be divided into finite iterations. An iteration is a minimal set of actor firings returning the dataflow graph to its initial state. $q_i > 0$ [3] represents the number of invocations of an actor a_i in one iteration of G and $\overrightarrow{q} = [q_1, q_2, ..., q_n]^T$, is the basic repetition vector of G.

In order to exploit inter-iteration parallelism more effectively, J iterations can be scheduled together, where J represents the unfolding factor of G. If the graph is unfolded J times, each actor a_i is executed $J_i = J \times q_i$ times. According to the DAG model, the execution flow of subtasks is constrained by their directed relations between instances and sub-tasks which is particular to CSDF model.

So, scheduling depends on actors and sub-tasks of actors. Each sub-task in $a_i^j = \{a_i^j(1), ..., a_i^j(\tau_i)\}, \ \forall \ j \in [1, ..., J_i]$, can be treated as a single unit denoted by its total computation time and its total communication time. However, ignoring this finer granularity offered by CSDF model can result in a pessimistic analysis, as with the example presented in section 1.1.

In this work, we consider only acyclic CSDF graphs. An acyclic graph G has a number of levels, denoted by L. Assigning actors to levels is based on passing through the *directed-acyclic graph (DAG)* of the MRDF application at compile-time. Different graph traversals types exist like topological, breadth-first, etc. Actors will be assigned to a set of levels $L = \{L_1, L_2, ..., L_\alpha\}$. Authors in [2], proposed a method based on assigning the actors in the graph according to precedence constraints. In order to guarantee bounded resource execution, additional precedence edges can be added to the timed graph. As depicted in Figure 2, the DAG is decomposed into a set of levels executed sequentially.

Definition 1. *Consistency [11]:* A CSDF graph is called consistent if and only if it has a non-trivial repetition vector. For a consistent graph, there is a unique smallest non-trivial repetition vector which is designated as the repetition vector of the CSDF graph. A repetition vector is called non-trivial if and only if $q_i > 0$ for all $a_i \in A$.

Theorem 1. In a CSDF graph, a repetition vector $\overrightarrow{q} = [q_1, q_2, ..., q_n]^T$ is given by [3]:

$$\overrightarrow{q} = P \cdot \overrightarrow{r} \ , with \quad P = P_{jk} = \begin{cases} \tau_j & , if \ \ j = k \\ 0 & , otherwise \end{cases} \tag{1}$$

And, $\overrightarrow{r} = [r_1, r_2, ..., r_n]^T$, where $r_i \in \mathbb{N}^\star$, is a solution of the balance equation:

$$\Gamma \cdot \overrightarrow{r} = 0, \tag{2}$$

The topology matrix Γ specifies the connections between edges in directed multigraph. As an example of this edge-vector topology matrix, a matrix entry Γ_{ui} would be 0 if edge e_u does not connect to actor a_i, p_i if actor a_i is the source actor of edge e_u, and $-c_i$ if actor a_i is the sink actor of e_u.

3.2 System Model and Schedulability

The system consists of a set $\Pi = \{\Pi_1, \Pi_2, ..., \Pi_m\}$ of m homogeneous processors. The processors execute a levels set $L = \{L_1, L_2, ..., L_\alpha\}$ of α periodic levels. A periodic level $L_i \in L$ is defined as $L_i = (S_i, \omega_{l_i}, f_\varphi(l_i), \phi_i, D_i)$, where S_i is the start time of L_i, $\omega_i \in \mathbb{N}^*$ is the worst-case computation time, $f_{STP}(i) \in \mathbb{N}$ is the worst-case communication time of L_i under STP schedule, $\phi_i \geq \omega_i + f_{STP}(i)$ is the level period and D_i is the relative deadline of L_i where $D_i = \max\limits_{k=1\rightarrow\beta_j} D_k$. $\beta_j \in \mathbb{N}^*$ represents the number of actors in each level.

A periodic level L_i is invoked at time instant $t = S_1 + (i-1)\phi$ and has to finish execution before time $t = S_1 + (i-1)\phi + \omega_i + f_\varphi(i)$. If $D_i = \phi$, then L_i is said to have implicit-deadline. If $D_i < \phi$, then L_i is said to have constrained-deadline. Actors (*i.e.* tasks) in G are scheduled as implicit-deadline periodic tasks and assigned to levels. At run-time , they are executed in a self-timed manner. This is possible because actors of level $k+1$ consume the data produced in level k.

The utilization of a task is $U_i = \frac{\omega_i}{\phi_i}$. For each level L_i, the total utilization of is $U_{L_i} = \sum_{j=1}^{\beta_i} \frac{\omega_j}{\phi_j}$. A scheduling algorithm is said to be optimal *iff* it can schedule any feasible task set A on Π such that $U_i \leq m$. Several global and hybrid algorithms were proven optimal for scheduling asynchronous sets of implicit-deadline periodic tasks [4]. We restrict our attention to consistent and live CSDF graphs. In fact, any acyclic graph is live [11].

A static schedule [17] of a consistent and live CSDF graph is valid if satisfies the precedence constraints specified by the edges. Authors in [14], introduced a theorem that states the sufficient and necessary conditions for a valid schedule. However, this result was established for Synchronous Dataflow graphs where actors have constant execution times. The test in Equation 3 is a novel contribution of this paper. This allows the timing of firing respects the firing rules of actors.

Theorem 2. *Valid Schedule of CSDFG:* A schedule S is valid if and only if for any edge $e_{\,i \rightsquigarrow j}$, for any instance $k \in \mathbb{N}$ and for any sub-task $\tau \in [1, ..., \tau_j]$:

$$s(j, k, \tau) \geq s(i, k + k_\epsilon, \tau_\epsilon) + \omega_i + \max[f_\varphi(a_i, \zeta_i)] \tag{3}$$

The schedule function $s(i, k, \tau) \in \mathbb{Z}^+$ represents the time at which the τth sub-task of the kth instance of actor a_i starts execution. k_ϵ is defined as:

$$k_\epsilon = \begin{cases} \sum_{x=1}^{\tau} c_j^x \ \ div \ \ (p_i + d_{i,j}) & , \ if \ \ \sum_{x=1}^{\tau} c_j^x \ \ mod \ \ (p_i + d_{i,j}) \neq 0 \\ \sum_{x=1}^{\tau} c_j^x \ \ div \ \ (p_i + d_{i,j}) - 1 & , \ else \end{cases} \tag{4}$$

$d_{i,j}$ are the tokens already present on buffer(i, j). Similarly, τ_ϵ is the smallest integer that can verify the following equation:

$$\frac{\sum_{x=1}^{\tau_\epsilon} p_i^x + d_{i,j} - \sum_{x=1}^{\tau-1} c_j^x}{c_j^\tau - k_\epsilon p_i} \geq 1 \tag{5}$$

Proof 1. The firing rule of each actor allows precedence constraints not to be violated, it follows that:

$$s(j,k,\tau) = t \ \Leftrightarrow \text{buff}(i,j) \geq c_j^\tau \tag{6}$$

The number of tokens stored on buffer buff(i,j) of edge $e_{\,i \frown j}$ should be greater than or at least equal to the number of consumed tokens for actor a_j. Since CSDF is monotonic:

$$\text{buff}(i,j) \sqsubseteq \text{buff}'(i,j) \Longrightarrow s(j,k,\tau) \sqsubseteq s(j,k',\tau'), \ \forall \ k' \geq k \tag{7}$$

FIFO property of communication buffers suggests that two consecutive executions of an actor will always produce output tokens in the order of their firing. FIFO ordering of tokens can be maintained, if each actor has a constant execution time, or has a self cycle with one token:

$$s(j,k,\tau) \geq end(i,k',\tau') \tag{8}$$

Then, If actor a_i have a variable execution time T_i

$$s(j,k,\tau) \geq s(i,k',\tau') + T_i, \ \forall \ T_i \in \mathbb{R}^* \tag{9}$$

Lemma 1 (FIFO property for DAG)**.** *: Any Directed Acyclic Graph (DAG) with actors that have variable execution times can conserve FIFO property*

Let $T_i = \omega_i + \max[f_\varphi(a_i, \zeta_i)]$, $k' = k + k_\epsilon$ and $\tau' = \tau_{epsilon}$, k_ϵ and $\tau_{epsilon}$ are obtained according to formula 6.

4 Self-Timed Periodic Model

4.1 Assumptions and Definitions

A graph G refers to an acyclic consistent CSDF graph. A consistent graph can be executed with bounded memory buffers and no deadlock. We base our analysis on the following assumptions:

A1. External sources in dataflow: The model is accomplished with interfaces to the outside world in order to explicitly model inputs and outputs (I/Os). A source and a sink nodes can be integrated as closures since they define limits for a portion of an application. A graph G has a set of input streams $I = \{I_1, I_2, ..., I_\Delta\}$ connected to the input actors of G, and a set of output streams $O = \{O_1, O_2, ..., O_\Lambda\}$ processed from the output actors of G. An actor $a_i \in A$ is defined, inter alia, with $E_{a_i} = (E_{a_i}^{in}$ and $E_{a_i}^{out})$ the sets of its input and output edges. These special nodes are defined as follows:

- 1. Nodes uniqueness: $src \in A$, $E_{src}^{in} = \emptyset$ and $E_{src}^{out} = \{I_1, I_2, ..., I_\Delta\}$, $snk \in A$, $E_{snk}^{in} = \{O_1, O_2, ..., O_\Lambda\}$ and $E_{snk}^{out} = \emptyset$.
- 2. Samples arrival: the first samples of source actor arrive prior or at the same time when the actors of G start execution. They are characterized by a minimum inter-arrival time assumed to be controlled by the designer to match the periods of the intervals.

Definition 2. For a graph G under periodic schedule, the worst-case communication overhead $f_{STP}^{L_j}$ of any level $L_j \in L$ depends on the maximum number of accesses to memory m_{β_j} processed in the time interval $[(j-1) \times \phi, j \times \phi[$:

$$f_{STP}^{L_j} = \uparrow f(m_{\beta_j}), \quad \forall L_j \in L \tag{10}$$

In [6], we proved that f is a monotonic increasing function of the number of conflicting memory accesses.

A2. For periodic schedules, synchronization cost is constant, because periodic behavior guarantees that an actor $a_i \in L_j$, $\forall i \in [1, ..., \beta_j]$, will consume tokens produced at level $(j-1)$ [6]. The latter implies that actors of the same level can start firing immediately in the beginning of a given period because all the necessary tokens have already been produced.

Definition 3. Matched Input/Output rates property:
According to [2], a graph G is a matched input/output (I/O) rates graph if and only if:

$$\eta \, mod \, Q = 0, \quad where \quad \eta = \max_{a_i \in A}(\omega_i q_i) \tag{11}$$

$Q = lcm(q_1, q_2, ..., q_n)$ (lcm denotes the least-common-multiple operator). If formula 11 does not hold, then G is a mis-matched I/O rates graph. If $\eta \, mod \, Q = 0$, then there exists at least a single actor in the graph fully utilizing the processor (*i.e.* which represents the minimal level period) on which it runs which allows the graph to achieve a maximum throughput.

4.2 Latency Analysis under STP Schedule

A self-timed schedule does not impose any extra latency on the actors. This leads us to the following result:

Definition 4. (STP_{q_i}) For a graph G, a period ϕ, where $\phi \in \mathbb{Z}^+$, represents the period, measured in time-units, of the levels in G. If we consider $\vec{q}$ as the basic repetition vector of G, then ϕ is given by the solution to:

$$\begin{cases} \phi \geq \max\limits_{j=1 \to \alpha} \max\limits_{k=1 \to \beta_j} \sum\limits_{i=1}^{\nu_{j,p_k}} (q_i \omega_i + f_\phi(i)), & \text{iff } \nu_{j,p_i} > 1 \\ \phi \geq \max\limits_{j=1 \to \alpha} \max\limits_{k=1 \to \beta_j} (q_k \omega_k + f_\phi(k)), & \text{iff } \nu_{j,p_i} = 1 \end{cases} \tag{12}$$

$\forall i \in [1, \beta_j]$ *and* $\forall j \in [1, \alpha]$.

The levels period ϕ is defined as the maximum execution time of all levels.

Definition 4 implies an equal period for all the levels. Similarly, we define the schedule function for the finer granularity of CSDF characterized by the repetition vector $\vec{r}$ as follows:

Definition 5. (STP_{r_i}) If we consider $\vec{r}$ as the basic repetition vector of G, then ϕ is given by the solution to:

$$\begin{cases} \phi \geq \max\limits_{j=1 \to \alpha'} \max\limits_{k=1 \to \beta'_j} \sum\limits_{i=1}^{\nu_{j,p_k}} (r_i \omega_i + f'_\phi(i)), & \text{iff } \nu_{j,p_i} > 1 \\ \phi \geq \max\limits_{j=1 \to \alpha'} \max\limits_{k=1 \to \beta'_j} (r_k \omega_k + f'_\phi(k)), & \text{iff } \nu_{j,p_i} = 1 \end{cases} \tag{13}$$

$\forall i \in [1, \beta_j]$ *and* $\forall j \in [1, \alpha]$.

The earliest start time of actors under the STP model is given by the following lemma.

Lemma 2. For a graph G, the earliest start time of an actor $a_i \in L_j$, denoted by $s_{i,j}$, under a strictly periodic schedule is given by:

$$s(i,j) = \begin{cases} 0 & , if \quad j = 1 \\ (j-1)\phi & , if \quad j > 1 \end{cases} \tag{14}$$

Latency is defined as the maximum time elapsed between the first firing of *src* actor and the last firing of *snk* actor. Using Equations 14, 13 and 12, it is possible to compute the latency under the STP model for any acyclic CSDF graph:

$$L_{STP_{q_i/r_i}} = \alpha \times \phi \tag{15}$$

Self-Timed Periodic (STP) Schedule is the main piece of our method. It's a new hybrid schedule based on the well-known periodic and self-timed schedules for streaming applications. The first step in STP is based on passing through the *directed-acyclic graph (DAG)* of the MRDF application at compile-time. Different graph traversals types exist like topological, breadth-first, etc. Actors will be assigned to a set of levels $L = \{L_1, L_2, ..., L_\alpha\}$. The best-case level structure is composed of m tasks which can be executed in parallel by fully utilizing the m processors provided by the platform, with a cumulative utilization of the task set that does not exceed m. The level structure does not only depend on precedence constraints between tasks defined by formula 3, but also on the degree of parallelism and of the architecture.

The second step consists in assigning priorities to actors of the same level using the self-timed (*i.e.* As-Soon As-Possible (ASAP) start-time) strategy. At run-time, actors are scheduled according to the assigned priorities. The final step assigns to each level a global period ϕ according to following formulas:

5 Evaluation

5.1 ΣC: a new programming language for embedded manycores

We use the ΣC [8, 1], a language designed in order to ensure programmability and efficiency on many cores. Close and familiar with C, it minimizes the specific syntaxes, while making explicit the construction of parallelism. As a programming language, ΣC relates to StreamIt [18], another programming language developed by Massachusetts Institute of Technology and specially engineered for modern streaming systems. Whereas the way to build Networks of Processes in StreamIt is by using the semantic of the code, which limits the topology of the associated Network (StreamIt topology is hierarchical, and is mostly limited to series-parallel graphs with nonetheless the important addition of feed-back loops; special features like teleport-messaging are required to overcome this limitation, see [18]), the way it is done in ΣC, is through a two-step compilation: The first step, is an off-line compilation to build the network of so-called agents (individual tasks in the stream model) and a communication interconnect called subgraph. This two-step compilation process has the advantage to permit any kind of topology, because it proceeds to an off-line execution of the first-stage compilation to build the process network.

5.2 MPPA Platform

We consider the $MPPA - 256$ [5] clustered architecure, from Kalray, comprising 256 user cores (*i.e.* cores with fully processing power provided to the programmer for computing tasks) organized as 16 (4×4) clusters tied by a Network-on-Chip (NoC) with a torus topology. Each cluster has 16 user processors connected to a shared memory. There are also 2 DMA engines (one in Rx, one out Tx) for communication with the NoC, and one special processor called *Ressource manager* which makes the role of orchestra conductor and provides OS-like services. Each processing core PE_i and the RM are fitted with two-way associative instruction and data caches (i.e., each location in main memory can be cached in either of two locations in the cache). In addition to the 16 clusters, there are 4 I/O clusters that provide access to external DRAM memory or interfaces, etc. The shared memory of a given compute cluster is a modular memory system. The memory system consists of M memory modules numbered $1, 2, 3, ...$ M-1, M, among which the addresses are distributed cyclically, that is, if i is the address of a memory location, then $j \equiv i(modM)$ is the address of the module containing the location. For the MPPA case, the memory system contains 16 memory modules of $128KB$, so $2MB$ per cluster. Each module has a memory controller connected to each pair of user processors (*i.e.* via a bus). The memory is implemented as a multi-bus approach [5]: it provides the same functionality as a full crossbar with lower impact on surface occupation and power consumption [13].

5.3 Evaluation Results

We evaluate our proposed framework in section 4 by performing an experiment on a set of 5 real-life streaming applications. The objective of the experiment is to compare the worscase end-to-end latency of streaming applications when scheduled using our self-timed periodic scheduling to their worst-case achievable latency obtained via strictly periodic scheduling. After that, we discuss the implication of our theoretical results from section 4 and the latency comparison experiment. The streaming applications used in the experiment are real-life applications coming from different domains (*e.g.* signal processing, audio processing, *etc.*). Some of these programs have been developed at CEA LaSTRE and some of them are StreamIt benchmarks. We use the ΣC language to implement the streaming applications on MPPA platform. Each application is executed with a set of input data to generate an execution trace. From these results, we derive the number of shared memory requests as well as the execution time of computation operations for each actor. The generated number of memory accesses is then used in the analytical memory access model presented in [6] in order to have an upper bound of communication and synchronization overhead. In fact, we can use the same model presented in [6] because STP schedule conserves the periodic behavior. Since we have a computation time and a communication overhead of actors, we apply the STP scheduling strategy in order to delimit the different levels of execution and calculate levels period.

5.4 Discussion

In this experiment, we compare worst-case end-to-end latency resulting from our STP approach to the IDP model. For latency, we report the graph maximum latency according to Formula 15. For IDP schedule, we used the minimum period vector given in [2]. For STP schedule, we used the minimum period vector given by Definition 12. ΣC tool-set defines the repetition vector q_i. Consequently, we derive only experimental results for one granularity. We will extend these results in the future for the finer granularity STP model based on the repetition vector r_i. Now, Figure 3 shows the results of comparing the latency of one iteration in the graph under both

Table 1: Presentation of each application

Application	Description	Source
Moving Average	Calculate the average	CEA LaSTRE and MIT
DCT	Functions that implement Discrete Cosine Transforms and Inverse DCT	CEA LaSTRE and MIT
BeamFormer	Template application to perform beam-forming on a set of inputs	CEA LaSTRE and MIT
AudioBeam	Appication to do real-time beamforming on a microphone input array	CEA LaSTRE and MIT
Laplacian	Laplace operator	CEA LaSTRE and MIT

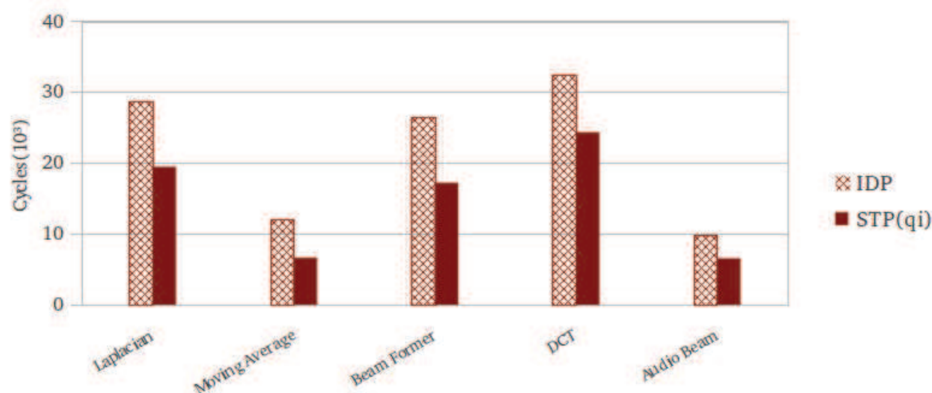


Figure 3: Results of the latency comparison

IDP and STP_{q_i}. We clearly see that our STP model delivers an improvement of 25% to 35% compared to IDP model. For all the applications, the latency was improved without verifying if the graph is matched I/O or not.

6 Conclusion and Perspectives

We prove that the actors of a streaming application modeled as CSDF graph, can be scheduled as self-timed periodic tasks. As a result, we conserve the properties of a periodic scheduling and in the same time improve its performance. We also show how the different granularities offered by CSDF model can be explored to decrease latency. We present an analytical framework for computing the periodic task parameters while taking into account inter-processor communication and synchronization overhead. As a future work, we will compare self-timed periodic schedule and self-timed schedule in the presence of non negligeable IPC overhead. We also want to improve our scheduling test for STP model with a finer granularity.

References

[1] P. Aubry, P.E. Beaucamps, F. Blanc, B. Bodin, S. Carpov, L. Cudennec, V. David, P. Dore, P. Dubrulle, B. D. de Dinechin, F. Galea, T. Goubier, M. Harrand, S. Jones, J. D. Lesage, S Louise, N. M. Chaisemartin, T. H. Nguyen, X. Raynaud, and R. Sirdey. Extended cyclostatic dataflow

program compilation and execution for an integrated manycore processor. *Propedia Computer Science, International Conference on Computational Science (ICCS), Alchemy Workshop*, 2013.

[2] MohamedA. Bamakhrama and TodorP. Stefanov. On the hard-real-time scheduling of embedded streaming applications. *Design Automation for Embedded Systems*, 2012.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J.A. Peperstraete. Cyclo-static data flow. In *Acoustics, Speech, and Signal Processing, ICASSP-95*, 1995.

[4] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 2011.

[5] Benoît Dupont de Dinechin, Pierre Guironnet de Massas, Guillaume Lager, Clément Léger, Benjamin Orgogozo, Jérôme Reybert, and Thierry Strudel. A distributed run-time environment for the kalray mppa-256 integrated manycore processor. *Procedia Computer Science*, 2013.

[6] Amira Dkhil, Stéphane Louise, and Christine Rochange. Worst-Case Communication Overhead in a Many-Core based Shared-Memory Model. In *Junior Researcher Workshop on Real-Time Computing, Nice*, 2013.

[7] M. Geilen and S. Stuijk. Worst-case performance analysis of synchronous dataflow scenarios. In *Hardware/software Codesign and System Synthesis, 2010 IEEE/ACM/IFIP International Conference on*, CODES+ISSS, USA, 2010.

[8] Thierry Goubier, Renaud Sirdey, Stéphane Louise, and Vincent David. ΣC: A programming model and language for embedded manycores. In *ICA3PP (1)*. Springer, 2011.

[9] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[10] M. Khandelia, N.K. Bambha, and S.S. Bhattacharyya. Contention-conscious transaction ordering in multiprocessor dsp systems. *Signal Processing, IEEE Transactions on*, 2006.

[11] E. A. Lee. Consistency in dataflow graphs. *IEEE Trans. Parallel Distrib. Syst.*, 1991.

[12] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, sept. 1987.

[13] Stephane Louise. A formal evaluation of mean-time access latencies for interleaved on-chip shared banked memory in manycores. In *Embedded Multicore Socs (MCSoC), 2013 IEEE 7th International Symposium on*, pages 19–24, Sept 2013.

[14] Orlando Moreira. Temporal analysis and scheduling of hard real-time radios running on a multiprocessor. PHD Thesis, Technische Universiteit Eindhoven, 2012.

[15] Orlando M Moreira and Marco JG Bekooij. Self-timed scheduling analysis for real-time applications. *EURASIP Journal on Advances in Signal Processing*, 2007.

[16] P. K. Murthy and S. S. Bhattacharyya. Memory management for synthesis of dsp software. 2006.

[17] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded Multiprocessors: Scheduling and Synchronization.* Marcel Dekker, Inc., 2nd edition, 2009.

[18] William Thies, Michal Karczmarek, and Saman P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, ICCC'02, 2002.