Models for Deterministic Execution of Real-time Multiprocessor Applications

Peter Poplavko[1] Dario Socci[1] Paraskevas Bourgos[1] Saddek Bensalem[1] Marius Bozga[2]
[1]*Univ. Grenoble Alpes, VERIMAG, Grenoble, F-38000, France*
[2]*CNRS, VERIMAG, Grenoble, F-38000, France*
{*Petro.Poplavko* | *Dario.Socci* | *Paraskevas.Bourgos* | *Saddek.Bensalem* | *Marius.Bozga*}*@imag.fr*

***Abstract*—With the proliferation of multi-cores in embedded real-time systems, many industrial applications are being (re-)targeted to multiprocessor platforms. However, exactly reproducible data values at the outputs as function of the data and timing of the inputs is less trivial to realize in multiprocessors, while it can be imperative for various practical reasons. Also for parallel platforms it is harder to evaluate the task utilization and ensure schedulability, especially for end-to-end communication timing constraints and aperiodic events. Based upon reactive system extensions of Kahn process networks, we propose a model of computation that employs synchronous events and event priority relations to ensure deterministic execution. For this model, we propose an online scheduling policy and establish a link to a well-developed scheduling theory. We also implement this model in publicly available prototype tools and evaluate them on state-of-the art multi-core hardware, with a streaming benchmark and an avionics case study.**

I. Introduction

A not so well known fact about fixed-priority scheduling is that it is commonly used in real-time systems not only for meeting the deadlines but also for ensuring functional determinism on uniprocessor platforms. This is so because the schedule priority define the *precedence* (*i.e.,* the relative execution order) of communicating tasks [1], [2]. However, when re-targeting the applications from single- to multi- processors this property of priority is lost, and hence alternative ways of ensuring 'schedulable' determinism is an important issue for industry [3], where multi-cores are considered an important target for next-generation real-time systems [3], [4]. Determinism is required for control stability and digital signal quality, for testing and fault-tolerance by triple-modular redundancy. Without deterministic communication it is impossible to define and guarantee end-to-end timing constraints.

Deterministic execution in any concurrent platform can be ensured by programming the application based on a deterministic model of computation, *i.e.,* a formal design language, and providing a correct implementation of the model semantics by safe synchronization between concurrent threads. Historically, in the academic research the *streaming/KPN(Kahn process network)*-based models of computation have gained a lot of popularity due to their affinity to signal/image processing, relative ease of multiprocessor implementation and applicability of well-established task graph scheduling and timing analysis theory [5]. In contrary, in industrial real-time applications *synchronous languages* have gained popularity due to their simple concept of timing through synchronous events, well-studied formal basis to define end-to-end precedence relationships between events (and hence their timing constraints) as well as their affinity to *'reactive-control'* applications.

Nowadays, with ever growing integration of various functionalities on shared resources it is practically relevant to consider hybrid streaming/reactive control applications. A step forward in this direction was combining KPNs with synchronous events in reactive process networks (RPNs) in [6]. However, so far no scheduling algorithms have been proposed for any proper subclass of RPN.

To close this gap and to help to address the industry needs in deterministic and schedulable multiprocessor models, in this paper we propose a subclass of RPNs called fixed-priority process networks (FPPNs). The model and its semantics is described in Section II. Then in Sections III and IV, for a quite general subclass of FPPNs, we propose a scheduling approach based on the scheduling theory of task graphs. In Section V we evaluate our publicly available FPPN code generation tools with a streaming (FFT) and a reactive control (avionics) applications. In the last section we discuss related work and present conclusions.

II. Fixed Priority Process Networks

A. Preliminaries

Functional determinism requires that the data sequences and time stamps at the outputs should be a well-defined function of the data sequences and time stamps at the inputs. Among deterministic models, the KPN (Kahn process networks) have gained popularity in the research on multiprocessor scheduling. They are deterministic due to the blocking of the reads from the empty channels. Reactive process networks (RPN) [6] extend KPN by events. Simultaneous occurrence of events can lead to non-determinism, but [6] suggest that determinism can be ensured by priorities between events. This suggestion is exploited in our model, Fixed Priority Process Network (FPPN). Our model differs from KPN and RPN by assuming blocking access of processes to events and non-blocking one to the data channels. Nevertheless, any FPPN can be directly translated to an equivalent RPN where processes never have to block for data [7].

We ensure determinism by so-called *functional priorities*, whose effect is equivalent to the effect of fixed priorities on a set of tasks under uniprocessor fixed-priority scheduling with zero task execution times. The order in which such tasks execute is defined first of all by the time stamps when the

Research supported by the European ICT Collaborative Project no. 288175 (CERTAINTY) and no. 332987 (ARROWHEAD).

tasks are released (we say, 'invoked') and secondly by the task priorities. Controlling the execution order implies determinism. This property extends from zero-delay to conventional tasks provided their periods and deadlines have some restrictions [1], [2]. Modeling their behavior makes FPPN functionally equivalent to such real-time systems. However, we do not put any restrictions on periods and deadlines. We use the priorities not directly in scheduling, but rather in the definition of *model semantics*. FPPN can be scheduled on single or multiple processors by scheduling policies with and without priorities, provided that the semantics is respected.

To define the semantics, we introduce some preliminary definitions. We assume a set of variables, each variable initialized at start. The set of variables is divided into the set of local variables, X, and channels. The latter are subdivided into **internal channels** C, *i.e.,* those shared between pairs of processes, and the **external inputs and outputs**, denoted I and O. Access to the channels is determined by channel type, which define the effect of read and write actions. We define two default channel types: a **FIFO** (first-in-first-out) and a **blackboard**. The FIFO has a semantics of a queue. The blackboard remembers the last written value, and it can be read multiple times. Reading from an empty FIFO or a non-initialized blackboard returns an indicator of non-availability of data. An action of writing variable $x \in X$ to channel $c \in C$ is denoted $x!c$. An action of reading is denoted $x?c$.

An **event generator** e is defined by the set of possible sequences of time stamps τ_k that it can produce online. A generator e is characterized by deadline d_e and a partitioned subset I_e and O_e of external channels. $[\tau_k, \tau_k + d_e]$ define the time interval when k-th sample in I_e and O_e can be read resp. written. The corresponding actions are denoted $x?_{[k]}I_{ei}$ and $x!_{[k]}O_{ej}$. We define two types of event generators: **multi-periodic** and **sporadic**. Both are parameterized by m_e, the burst size, and T_e, the period. Bursts of m_e periodic events occur at times 0, T_e, $2T_e$, *etc.*. For sporadic events, at most m_e events can occur in any half-closed interval of length T_e.

Next to write and read actions, we define variable assignment and waiting until time stamp τ: $\mathbf{w}(\tau)$. The actions are assumed to have zero delay. The set of all **actions** is denoted Act. **Execution trace** $\alpha \in Act^*$ is a sequence of actions, *e.g.,*

$$\alpha = \mathbf{w}(0),\ x?_{[1]}I_1,\ x := x^2,\ x!c_1, \mathbf{w}(100),\ y?c_1,\ O_1!_{[2]}y$$

In this example, at time 0 we read data from I_1 sample [1] and compute its square. Then we write to channel c_1. At time 100 we read from c_1 and write to output O_1 sample [2].

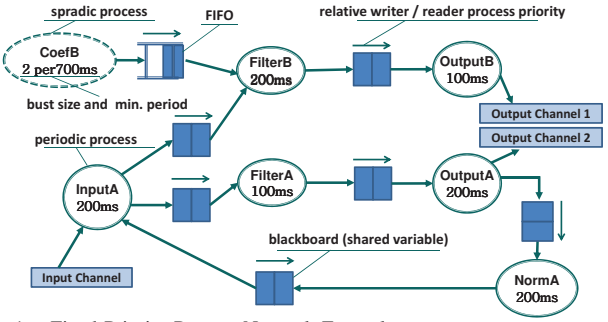

Fig. 1. Fixed Priority Process Network Example

Definition 2.1 (FPPN): An *FPPN* is a tuple $\mathcal{PN} = (P, C, \mathcal{FP}, e_p, I_e, O_e, d_e, \Sigma_c, CT_c)$ where P is a set of processes, $C \subseteq P \times P$ is a set of internal channels, so (P, C) is a directed graph. In addition to process network graph, which can be cyclic, we define a directed graph $(P, \mathcal{FP})$, $\mathcal{FP} \subset P \times P$, called functional priority graph, which must be acyclic (a DAG). We use notation $p_1 \to p_2$ for $(p_1, p_2) \in \mathcal{FP}$. The functional priority should be defined at least for the processes accessing the same channel: $(p_1, p_2) \in C \Rightarrow p_1 \to p_2 \vee p_2 \to p_1$. e_p is a mapping from process p to a unique event generator, whereas I_e and O_e are mappings from event generator to a (possibly empty) partition subset of external input and output (I/O) channels. d_e defines the relative deadline for accessing the I/O channels of generator e. Σ_c defines alphabets for internal and external I/O channels. CT_c defines the channel types.

By abuse of notation $c \in C$ is a channel (state variable) and at the same time a pair of writer and reader, (p_1, p_2). For p_1 the channel is said to be an output and for p_2 an input.

Because processes p are related one-to-one to event generators e_p, we associate the generator attributes with the processes, and use notations T_p, m_p, d_p, I_p and O_p.

An example is shown in Figure 1. It represents an imaginary signal processing application with input sample period 200ms, reconfigurable filter coefficients and a feedback loop.

Definition 2.2 (Process): Each process p is associated with a deterministic automaton $({\ell_p}^0, L_p, X_p, {X_p}^0, \mathcal{I}_p, \mathcal{O}_p, A_p, \mathcal{T}_p)$. where L_p is a set of locations, ${\ell_p}^0$ is initial location, X_p is set of internal variables, ${X_p}^0$ are initial values of variables. $\mathcal{I}_p, \mathcal{O}_p$ are (both internal and external) input and output channels: $I_p \subseteq \mathcal{I}_p$, $O_p \subseteq \mathcal{O}_p$. A_p is a set actions, which consists of variable assignments for X_p, reads from $\mathcal{I}_p$, and writes to $\mathcal{O}_p$. $\mathcal{T}_p$ is transition relation $\mathcal{T}_p : L_p \times G_p \times A_p \times L_p$, where G_p is the set of predicates (guarding conditions) defined on the variables from X_p.

Informally, a process represents a software subroutine with a given set of locations (source-code line numbers), variables and transitions (data and operators). The latter include the current location (line number), the guard on variables ('if' condition), the action (operator body) and the next location. A **job execution run** of a process automaton is a non-empty sequence of automaton steps (executed lines of code) that brings it back to its initial location (as a subroutine). We assume that at k-th job execution run the external inputs I_p and outputs O_p are read/written only at sample index $[k]$.

We give two definitions of FPPN semantics. An imperative requirement for the execution of FPPN is synchronous arrival of all simultaneous event invocations.

The zero-delay semantics can be defined in terms of rules to construct the execution trace of FPPN for a given sequence $(t_1, \mathbf{P}^1), (t_2, \mathbf{P}^2) \ldots$ where $t_1 < t_2 < \ldots$ are time stamps and $\mathbf{P}^i$ is the multiset of processes invoked at time t_i by their event generators. The execution trace has the form:

$$Trace(\mathcal{PN}) = \mathbf{w}(t_1) \circ \alpha^1 \circ \mathbf{w}(t_2) \circ \alpha^2 \ldots$$

where α^i is a concatenation of job execution runs for the processes in $\mathbf{P}^i$ included in an order such that if $p_1 \to p_2$ then the job(s) of p_1 execute earlier than the job(s) of p_2.

The real-time semantics is a relaxed version of the zero-delay one. It allows jobs to have any execution time and to start concurrently to each other at any time after their invocation. However, the FPPN execution should satisfy *timeliness* and *precedence*. Timeliness means completion within the relative deadline d_p. Precedence concerns the jobs of the same process and the jobs accessing the same channel. Each such subset of jobs should execute in a mutually exclusive way and respect the execution order of zero-delay semantics, sorted by invocation time and priority.

Proposition 2.1 (Deterministic Execution): The sequences of values written at all external and internal channels are functionally dependent on the time stamps of the event generators and on the data samples at the external inputs. [1]

III. Scheduling Models

A. Task Graph Derivation

FPPN is a model of computation designed to formalize the behavior of real-time tasks with deterministic communication, including those uniprocessor scheduling settings that exploit the schedule priority to ensure determinism. For the latter there exists a family of relevant scheduling techniques, such as [1], [2]. Such techniques can be seen as ready-to-use uniprocessor scheduling methods applicable to FPPN and related models, such as synchronous languages [2].

As a formal language, FPPN should show the same deterministic behavior no matter which platform it is implemented on. A correctly implemented formal language would ensure deterministic execution on multiple processors, but ensuring timeliness by *multiprocessor scheduling* would remain to be challenging. This problem gets even harder when sporadic tasks are involved. Therefore, to demonstrate scheduling for FPPNs, we consider a practically relevant subclass of FPPNs where the use of sporadic tasks is restricted.

From the subclass of FPPNs considered here one can statically derive a *task graph* which then serves as input to a scheduling algorithm. The algorithm generates a static schedule, where we model sporadic processes by periodic ones with worst-case demand of resources. To make it possible, we put a restriction that each sporadic process p be connected by a channel to exactly one 'user' process $u(p)$, which must be periodic and which must have at most the same period[2]: $T_{u(p)} \leq T_p$. This restriction is practically relevant, because a sporadic process often plays an utility role, 'configuring' some application parameters of a periodic process, *e.g.,* in Fig. 1 process CoefB configures the filter coefficients of user process FilterB.

The run-time sporadic jobs invoked inside the user period are modeled by 'periodic server' jobs that arrive at the boundaries of the user period intervals. As indicated in the task subgraph, the server jobs at time b must have precedence over the user job that also arrives at time b. This is so because for causality reasons the server jobs can only handle the real jobs that have been invoked in the past, *i.e.,* inside (a, b), whereas FPPN semantics requires that the earlier jobs have precedence

[1]All proofs can be found in extended version of this paper, [7].

[2]one could relax the restrictions on the number of user processes and their periods at the cost of somewhat more complex task graph construction

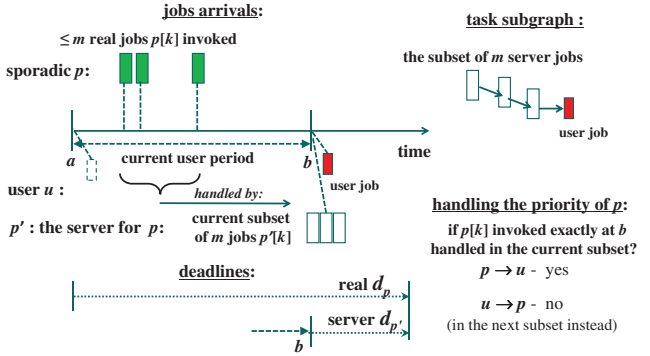

Fig. 2. Handling a Sporadic Process. (Handling the priority – see Sec. IV)

over the later ones. For convenience, we say that the server jobs for process p are generated by an imaginary m-periodic 'server process' p'. To ensure the precedence of the server jobs we set: $p' \rightarrow u(p)$. Note that this does not mean that sporadic processes must always have priority over their users, the higher priority is only required for their 'servers', which are imaginary processes introduced to define the scheduling algorithm.

The deadlines of the server jobs are corrected to compensate for worst-case one-period postponement of job arrival due to waiting until the job is handled by the server[3]: $d_{p'} = d_p - T_{u(p)}$. Thus, we effectively assume arrival at time b but count the deadline from time a, in order to be conservative.

Definition 3.1 (Task Graph): A task graph is a directed acyclic graph (DAG) $\mathcal{TG}(\mathcal{J}, \mathcal{E})$ whose nodes are jobs: $\mathcal{J} = \{J_i\}$. A job is characterized by a 6-tuple $J_i = (p_i, k_i, A_i, D_i, C_i)$, where: p_i is the process to which the job belongs, k_i is the invocation count of job, $A_i \in \mathbb{Q}_{\geq 0}$ is the arrival time, $D_i \in \mathbb{Q}_+$ is the required time (absolute deadline), $C_i \in \mathbb{Q}_+$ is the WCET (worst-case execution time). A job can be denoted $p[k]$, *i.e.,* k-th job of process p. The edges $\mathcal{E}$ are called precedence edges and represent constraints on job execution order.

The task graph for $\mathcal{PN}$ is derived as follows:

1) Obtain an imaginary process network $\mathcal{PN}'$ where each sporadic process p is replaced by m-periodic 'server' process p' with burst size $m_{p'} = m_p$, period: $T_{p'} = T_{u(p)}$, and priority relation: $\mathcal{FP}' : p' \rightarrow u(p)$.
2) Simulate the job invocation order in $\mathcal{PN}'$ for one hyperperiod, *i.e.,* time interval $[0, \mathcal{H})$, where $\mathcal{H}$ is the least common multiple[4] of T_p in $\mathcal{PN}'$. The simulation results in a sequence of jobs $J = (p_i[k_i])$. Sequence J defines a total order $<_J$
3) Construct graph $\mathcal{TG}(\mathcal{J}, \mathcal{E})$ where the nodes $\mathcal{J}$ are the elements of sequence J and the edges are defined for a pair of jobs $J_a = p_a[k_a]$ and $J_b = p_b[k_b]$ as follows:
   - $(J_a, J_b) \in \mathcal{E} \Leftrightarrow J_a <_J J_b \wedge (p_a \bowtie p_b \vee p_a = p_b)$, where:
   - $p_a \bowtie p_b \Leftrightarrow (p_a, p_b) \in \mathcal{FP}' \vee (p_b, p_a) \in \mathcal{FP}'$.

   and the job parameters for job $J_i = p[k]$ defined by:

[3]here we implicitly require that $d_p > T_{u(p)}$ but if it is not the case we can use server jobs with a period T' being a fraction of $T_{u(p)}$ instead, so that the server deadlines become positive

[4]$T_p \in \mathbb{Q}_+$, so the lcm is computed for rational numbers

- $A_i = T_p \cdot \lfloor (k-1)/m_p \rfloor$ and $D_i = A_i + d_p$ if p is m-periodic
- $A_i = T_{p'} \cdot \lfloor (k-1)/m_{p'} \rfloor$ and $D_i = A_i + d_p - T_{p'}$ if p is sporadic

4) Truncate all the required times D_i to the hyperperiod: $D_i := \min(\mathcal{H}, D_i)$. This is required by the algorithm described in Section III-B.
5) Remove redundant edges by transitive reduction.

Fig. 3 shows an example assuming $C_i = 25$ms.

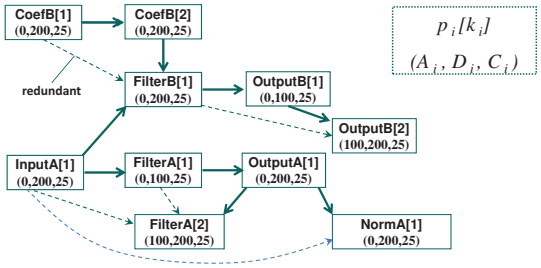

Fig. 3. Task Graph for the Process Network in Fig. 1

In this example, $\mathcal{H} = 200$. Every process is represented by $m_p \cdot \mathcal{H}/T_p$ vertices. Since CoefB is represented by its server process, its period 700 is replaced by the period of its user (FilterB), 200. Since $m_p = 2$, CoefB is represented by two jobs. InputA has priority over FilterA and NormA, and hence it is joined to both of them. However, in the latter case the edge is redundant due to a path from InputA to NormA.

B. Compile-time Scheduling Algorithm

To demonstrate availability of proper scheduling techniques applicable for FPPNs, in this paper the scheduling is defined by compile-time scheduling algorithm and online scheduling policy. The compile-time algorithm schedules the given task graph and prepares a configuration for the online policy. This algorithm must have a scalable complexity, as it may face large hyperperiods in multi-rate systems. We apply the precedence-constrained scheduling theory, which is commonly used in streaming languages [5] and which usually does not consider jobs with multiple different arrival times and deadlines, but they can be naturally incorporated, as shown here.

Currently we assume non-preemptive scheduling on a set of M identical processors. We restrict ourselves to *non-pipelined* scheduling and thus truncate the deadlines to avoid overlap of subsequent task graph executions. We adopt this restriction because of too little previous research on scalable pipelined scheduling which would directly support periods, deadlines and bounded number of processors at the same time. [5]

The compile-time scheduling algorithm constructs a *static schedule*, where all start times are fixed. This schedule is repeated periodically and therefore is referred to as periodic frame. The frame period is the hyperperiod $\mathcal{H}$.

Let s_i be the starting time of job J_i *w.r.t.* the beginning of the frame. The execution interval for job J_i is interval $[s_i, e_i)$, where $e_i = s_i + C_i$. Let us define the following predicate: $\psi_{i,j} : e_i \leq s_j$, stating that job J_j executes after J_i.

[5]For this combination of constraints, pipelined streaming currently knows a rich set of scalable *timing analysis* but not *scheduling* techniques. One usually employs non deadline-aware retiming and unfolding [5] or employs less-scalable methods based on constraint solving and model checking.

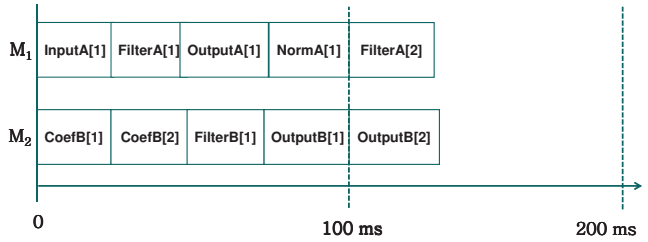

Fig. 4. A Static Schedule for the Task Graph in Fig. 3

Definition 3.2 (Static Schedule): A static schedule for a task graph $\mathcal{TG} = (\mathcal{J}, \mathcal{E})$ on a multiprocessor platform with a finite set M of processors consists of defining for each job J_i a mapping $\mu_i \in M$ and a start time $s_i \in \mathbb{Q}_{\geq 0}$. A schedule is called *feasible* if it satisfies the following constraints:

Arrival time:	$\forall i$	$s_i \geq A_i$	
Deadline:	$\forall i$	$e_i \leq D_i$	
Precedence:	$\forall i,j$	$(J_i, J_j) \in \mathcal{E}$	$\Rightarrow \psi_{i,j}$
Mutual exclusion:	$\forall i,j$	$\mu_i = \mu_j$	$\Rightarrow \psi_{i,j} \vee \psi_{j,i}$

The problem formulation can be seen as a generalization of the classical problem, where all jobs have a zero arrival and a common required time. The classical problem is NP-complete. Therefore a heuristic algorithm is generally required in practice. Like many precedence-constrained scheduling algorithms, at compile time we also use *list scheduling*, which assumes a heuristically computed *schedule priority* $\mathcal{SP}$, a total order where earlier jobs have higher priority. Note that $\mathcal{SP}$ should not be confused with functional priority, $\mathcal{FP}$, used to determine the precedences in $\mathcal{E}$. Normally, priority-based scheduling defines a job J_i to be *ready at time* t if at that time it has arrived and has not completed yet: $A_i \leq t < e_i$. The list scheduling extends this condition by also requiring that all predecessors should have completed: $\forall j \in Pred(i).e_j \leq t$, where $Pred(i) = \{j \mid (J_j, J_i) \in \mathcal{E}\}$. For a given $\mathcal{SP}$, **list scheduling** consists of a simple simulation of the fixed-priority policy using the updated definition of ready jobs.

If the obtained static schedule satisfies the job deadlines then it is feasible, otherwise the selected schedule priority may be sub-optimal. Different heuristics exist for optimizing priority order $\mathcal{SP}$ [8]. For defining the heuristics as well as for facilitating the problem analysis in general it is useful to introduce the well-known notions of ASAP start time A'_i and ALAP[6] completion time D'_i (which stands for 'as soon' and 'as late' as possible). They provide a lower bound on s_i and an upper bound on e_i for any feasible schedule if one exists. These times can be defined by recursive formulas:

$$A'_i = \max(A_i, \max_{j \in Prec(i)} A'_j + C_j)$$

$$D'_i = \min(D_i, \min_{j \in Succ(i)} D'_j - C_j)$$

where $Pred(i)$ and $Succ(i)$ are predecessors and successors. A less commonly known fact about ASAP and ALAP is that they can serve to define an **utilization** metric that takes into account the precedence constraints. This metric was originally introduced for the case of no precedences and was called *load* [9]. We define the task graph load as:

$$Load(\mathcal{TG}) = \max_{0 \leq t_1 < t_2} \frac{\sum_{J_i:\ t_1 \leq A'_i \wedge D'_i \leq t_2} C_i}{t_2 - t_1}$$

where A'_i and D'_i are ASAP and ALAP times.

[6]in the literature ALAP often refers to arrival

Proposition 3.1 (Necessary condition for schedulability): A task graph $\mathcal{TG}$ can be scheduled on M processors only if $\forall i. A'_i + C_i \leq D'_i$ and $\lceil Load(\mathcal{TG}) \rceil \leq M$.

As for heuristics to compute $\mathcal{SP}$, apparently also in this case it is useful to consider the EDF (earliest-deadline first), often applied in preemptive online scheduling. For task graphs, the definition of EDF should be adjusted by using ALAP instead of the nominal job deadlines. Different variants of this heuristics exist, such as 'ALAP'-heuristic, b-level heuristic [8], modified deadline monotonic [1], *etc.*. For certain problem restrictions the EDF was proven optimal, see [8].

IV. Online Scheduling Policy

Our scheduling policy consists of repetition of the schedule frame with period $\mathcal{H}$. The jobs are mapped to the processors according to mapping μ_i. In *fully static scheduling*, we would also have used s_i for the start time of the jobs *w.r.t.* the start of the frame. However, the statically computed start times are not robust against inaccuracies in estimations of WCET, which can appear in measurement-based and probabilistic WCET estimations. Therefore instead we use a policy where the jobs synchronize with their predecessors instead of relying on s_i to ensure precedence constraint satisfaction. This policy is predictable and known as *static-order scheduling* [5]. We have adapted this policy to sporadic processes. After the start of the new frame, on each processor independently the scheduler picks the jobs in the order defined by the schedule s_i and executes a 'round' that consists of the following steps:

- **Synchronize Invocation**: Wait for the event invocation that corresponds to the current job. For periodic and m-periodic processes the event invocation occurs at time A_i. For sporadic ones the invocation occurs either at time A_i or earlier or does not occur at all. In the latter case at time A_i the job is marked 'false'.
- **Synchronize Precedence**: Wait until all task-graph predecessors that run on other processors have completed. For example for job FilterB[1] we would wait until InputA[1] has completed,
- **Execute the job**: unless it is marked 'false'.

Recall (see Fig. 2) that for each sporadic process p the variable number of jobs per frame is represented in the task graph by server jobs. In general, the latter can be split into $\mathcal{H}/T_{u(p)}$ subsets of the m_p jobs invoked in the same user period $T_{u(p)}$. The jobs in n-th subset arrive at the same time: $\mathcal{A}[n] = (n-1) \cdot T_{u(p)}$ and always have a direct precedence to the user job arriving at time $\mathcal{A}[n]$. For example, in Fig. 3, jobs CoefB[1] and CoefB[2] are in the same subset, they arrive at time 0 and have precedence edge to FilterB[1].

All the jobs in a subset are invoked in time interval between $a = \mathcal{A}[n] - T_{u(p)}$ and $b = \mathcal{A}[n]$, see Fig. 2. In our example, for the jobs of process CoefB, we have arrival: $\mathcal{A}[1] = 0$ and $T_{u(p)} = 200$, so the jobs can be invoked in the interval from $a = -200$ to $b = 0$. The negative time values should not be surprising, as they are relative to the start of the current frame.

Consider the t-th job inside the subset. This job represents the t-th real job invoked between a and b. If at run time less than t jobs were invoked our scheduling policy marks the t-th job and later jobs in the subset as 'false' and skips them.

If the real job of process p is invoked between a and b then the online policy should see them as part of the subset, see Fig. 2. However, what if the job arrives exactly on the boundary? By periodicity, it is enough to consider only boundary b. If the process has a higher priority than its user then it should be executed before the user and therefore included in the subset. Otherwise it should be executed after the user and thus postponed to the next subset (or frame).

Therefore, if $p \rightarrow u(p)$ then the server jobs arriving at b handle the real jobs invoked in the right-closed interval $(a, b]$, and in the opposite case the interval is left-closed.

Proposition 4.1 (Schedule Correctness): When based on a feasible static-schedule input, the static-order policy always meets the deadlines and correctly implements the real-time semantics of FPPN.

V. Experiments

In the context of CERTAINTY EU project an FPPN-related programming language was defined. For that language we developed scheduling and code generation tools as well as a runtime environment for shared-memory multiprocessors [10]. The execution times for scheduling are obtained from profiling, which is suitable for soft real-time applications. The runtime was deployed to Linux multi-thread as well as MPPA many-core [11] platforms. The tools are based on automatic translation of the FPPN network and the schedule to a network of timed automata. We support both the zero-delay semantics for simulation and real-time semantics for concurrent real-time execution, where multiple process automata can be mapped to the same thread according to static mapping μ_i.

A. Streaming Application: FFT Transform

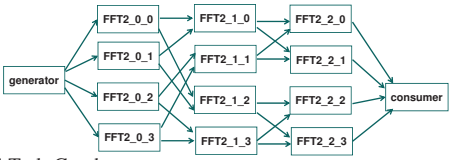

Fig. 5. FFT Task Graph

In this use case we have programmed a classical streaming application: the FFT (Fast Fourier Transform) of four floating-point numbers, shown in Fig. 5. We use our design framework to compile and run this application on the Kalray MPPA platform. All processes had the same period and deadline $T_p = d_p = 200$ ms, and the direction of data flow in FIFO channels coincided with functional priority relation, and hence the task graph maps one-to-one to the process-network graph.

The execution times of all processes were roughly 14ms, which resulted in a load 0.93. However, single-processor mapping did not meet deadlines, due to the runtime overhead. The application was thus mapped on two processors, where no deadline misses were observed. The Gantt chart of the execution traces is shown in Figure 6. The first two rows show the application jobs on two processors, while the third one shows the execution of the runtime, on a separate processor.

As it can be seen the runtime causes an overhead at the beginning of each frame, which is 41 ms for the first frame (probably due to initial cache misses) and 20 ms for all subsequent frames, required to manage the arrival of 14 jobs. Also inside the frame the runtime serves read/write synchronisation

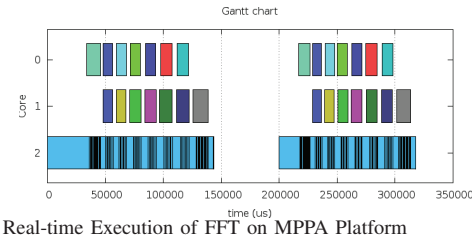

Fig. 6. Real-time Execution of FFT on MPPA Platform

requests from the processes. While read/write overhead is included in WCET estimations, the arrival overhead is not, and taking it into account in schedulability analysis is future work. For now we modeled it by an extra 41 ms job with a precedence edge directed to the generator. This yielded a load of ≈ 1.2, which explains the deadline misses in single-processor mapping. We also observe that this application is very fine grain (processing just one number per job), whereas more coarse grain implementation would make the relative impact of overhead small compared to the computation times.

B. Reactive-Control Application: FMS

In this experiment we consider a subsystem of avionics Flight Management System (FMS) [4]. Because of current limitations of our tools, we could only run it on a Linux platform. We used one with an Intel i7 processor at 3.6GHz. Figure 7 shows the application process network. This FMS subsystem is responsible for calculating the best computed position (BCP) and predicting the performance (*e.g.,* fuel usage) of the airplane based on the sensor data and sporadic configuration commands from the pilot.

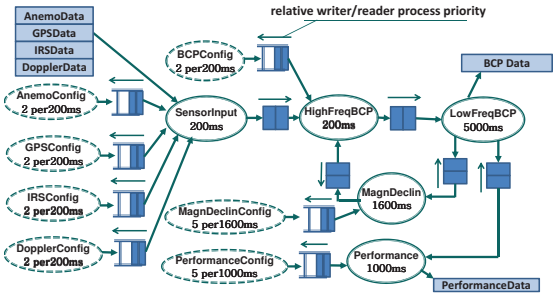

Fig. 7. FMS Process Network

The sporadic processes had less functional priority than their periodic users. The relative functional priority of the periodic processes is rate-monotonic, which was in line with the scheduling priority of the original uniprocessor prototype, making the two implementations functionally equivalent, which we verified by testing.

For this process network we encountered a too high code generation overhead due to a long hyperperiod (40 s) (an online policy subroutine handling a few thousands jobs explicitly). Therefore, we reduced it to 10 s by reducing the period of MagnDeclin from 1600ms to 400ms and executing the main body of the job once per four invocations. The derived task graph contained 812 jobs and 1977 edges. The load of this task graph was low ≈ 0.23 and, consistently, a single-processor mapping encountered no deadline misses. To test multi-processor execution, we still generated schedules for different number of processors and reached similar conclusions as for FFT concerning the runtime overhead and the job granularity. For the Gantt chart and more results see [7].

VI. Related Work and Conclusions

In this work we proposed FPPN, a new model of computation that generalizes the determinism of real-time fixed-priority systems [1], [2], [3] from single- to multiprocessor platforms by reproducing the deterministic behavior of such systems using precedence constraints in semantics. From a quite general practically-relevant subclass of FPPN, we derive static task graphs and adapt the corresponding scheduling methods to support deterministic and predictable communication between deadline-constrained periodic and sporadic processes.

FPPN combines certain concepts of synchronous and streaming languages. While practiced for a long time in streaming, derivation of multi-tasking models from synchronous languages has received attention only recently. [2] proposes a task graph derivation and scheduling algorithm, not supporting, however, sporadic events and multiple processors. [1] also propose a task-graph priority assignment algorithm for uniprocessors. From the streaming domain, the novelty of our approach is the support of (a)periodic deadline-constrained events. [12] is one of the few streaming languages supporting external events with deadlines, though only periodic ones. In [6] reactive process networks (RPNs) are proposed. FPPNs represent a restriction of RPNs adapted to real-time tasks by introducing the priority and explicit timing of events.

We provide online prototype tools [10] and present promising evaluation results on two use cases, including one from industry. In future work we plan to support buffering and pipelining, as well as mixed-critical scheduling.

References

[1] J. Forget, F. Boniol, E. Grolleau, D. Lesens, and C. Pagetti, "Scheduling dependent periodic tasks without synchronization mechanisms," in *RTAS'10*, pp. 301–310.

[2] S. Baruah, "Semantics-preserving implementation of multirate mixed-criticality synchronous programs," in *RTNS'12*, pp. 11–19, ACM, 2012.

[3] D. Claraz, F. Grimal, T. Laydier, R. Mader, and G. Wirrer, "Introducing multi-core at automotive engine systems," in *ERTSS'14, Embedded Real-time Software and Systems*, 2014.

[4] G. Durrieu, M. Faugère, S. Girbal, D. G. Pérez, C. Pagetti, and W. Puffitsch, "Predictable flight management system implementation on a multicore processor," in *ERTSS'14*, 2014.

[5] S. Sriram and S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization, Second Edition*. Signal Processing and Communications, Taylor & Francis, 2009.

[6] M. Geilen and T. Basten, "Reactive process networks," in *EMSOFT'04*, (New York, NY, USA), pp. 137–146, ACM, 2004.

[7] P. Poplavko, D. Socci, P. Bourgos, S. Bensalem, and M. Bozga, "Models for deterministic execution of real-time multiprocessor applications (extended version)," technical report TR-2014-12, Verimag, 2014.

[8] Y.-K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, pp. 406–471, Dec. 1999.

[9] J. W. S. Liu, *Real-Time Systems*. Prentice-Hall, Inc., 2000.

[10] P. Poplavko, P. Bourgos, D. Socci, S. Bensalem, and M. Bozga, "Multicore code generation for time-critical applications, http://www-verimag.imag.fr/multicore-time-critical-code,470.html."

[11] B. D. de Dinechin, D. van Amstel, M. Poulhiès, and G. Lager, "Time-critical computing on a single-chip massively parallel processor," in *DATE'14*, EDAA, 2014.

[12] S. J. Geuns, J. P. H. M. Hausmans, and M. J. G. Bekooij, "Sequential specification of time-aware stream processing applications," *ACM Trans. Embed. Comput. Syst.*, vol. 12, pp. 35:1–35:19, Mar. 2013.