On-line Scheduling in Multiprocessor Systems based on continuous control using Timed Continuous Petri Nets


Abstract—This work presents a fluid-time scheduler based on a sliding mode controller where the sliding surface is related to fluid task executions. The scheduler is applied to a model of tasks and CPUs designed with Timed Continuous Petri Nets (TCPN) under the infinite server semantics (ISS). Also, the paper proposes an implementation of this fluid scheduler as a feasible discrete scheduler where the number of task migrations and preemptions is bounded.

Keywords: TCPN, Modeling, Control, Scheduling.

I. INTRODUCTION

Modern embedded systems are increasingly more demanding, mostly due to the large number of tasks they must execute in a very short period, subject to real time and energy constraints. Real-time schedulers assure that all tasks meet their period and deadlines. They must be correct, but also feasible to implement, and optimal in terms of CPU usage.

A real-time scheduler ensures that a task’s instance activates according to a period and finishes before a deadline. With this purpose, some schedulers order tasks according to a fixed or dynamic priority. Rate Monotonic (RM) and Deadline Monotonic (DM) analysis ensure that a schedule is correct if the periodic instances (jobs) of each task are given a fixed priority according to their period (RM) or deadline (DM) [1]. Alternatively, dynamic priority schedulers such as EDF (Earliest Dead Line First) and LLF (Least Laxity First) dynamically compute a job’s priority according to the current system state. RM guarantees valid schedules on uniprocessor systems at the cost of wasting about 30% of the CPU time whereas EDF and LLF are optimal [1], [2], but this optimality does not hold on multiprocessors [3], [4]. Fair scheduling algorithms try to follow as closely as possible the fluid schedule, i.e., a schedule in which each task is executed at constant rate, to guarantee an optimal schedule in multiprocessor systems [5]. Fair schedulers allow a better control over the progress of each job while honoring periods and deadlines, but their implementation entails a large number of task preemptions (context switches), which hampers performance in practice. Proportionate Fairness (P-Fair) schedulers [6] are based on the rationale of fluid schedulers. In P-Fair all tasks’ jobs are executed at an approximately uniform rate by splitting them into sub-tasks which run for a time quantum. Other scheduling algorithms have emerged from this notion of proportionate fairness, like PD [7], PD² [8], DP−Fair [5] and BFair [9]. As all fair schedulers, they are optimal in multiprocessor systems but suffer from a large number of task preemptions and migrations. Hence, while these algorithms are theoretically optimal or high-performing, they are not feasible in real applications. Moreover, they cannot easily incorporate some continuous requirements demanded by modern embedded systems such as power consumption or thermal issues.

The main contribution of this paper is to propose an on-line fluid scheduler based on a sliding mode control technique [10], capable of integrating the CPU thermal models presented in [11]. We first propose a methodology to model the period, deadline and CPU cycles of each task, along with a set of CPUs, by means of a Timed Continuous Petri Net (TCPN). Based on this model, we build the on-line fluid scheduler. Through a Lyapunov stability analysis, the fluid scheduling is guaranteed, so it is optimal. We also introduce an algorithm to implement this fluid schedule as a discrete scheduler, where the number of task migrations and preemptions is bounded. We leverage the TCPN instantaneous marking to define priority firing rules and determine how tasks are allocated to processors. We prove that the size of the TCPN models increase linearly with the number of tasks and the complexity of the proposed algorithm is polynomial.

The paper is organized as follows. Section II provides basic definitions. Section III explains the model of periodic real-time tasks. Section IV introduces the CPU model, the allocation of tasks to CPUs and the global model. The fluid scheduler based on a sliding mode controller is presented in Section V. Section VI presents a discretization of the fluid scheduler. Section VII shows an illustrative example of a multiprocessor system. Finally, Section VIII concludes the paper.

II. FUNDAMENTALS

This section introduces basic definitions concerning Petri nets and continuous Petri nets. An interested reader may also consult [12], [13] to get a deeper insight in the field. In the sequel, given a matrix A and sets of indices (or nodes) I = {i₁, ..., iₙ} and J = {j₁, ..., jₘ}, it will be denoted as $A[I, J]$ the matrix built with the elements in the rows related to I and the columns related to J. The same notation will be used with vectors. Furthermore, $a_j$ will be used to denote the j-th element of vector $a$. 

---

*This work is partially supported by grants TIN2013-46957-C2-1-P and T48 RG (Aragon Gov. and ESF).

1G. Desirena-López, A. Ramírez-Treviño and D. Gómez-Gutiérrez are with the CINVESTAV-IPN Unidad Guadalajara, Av. del Bosque 1145, CP 45019, Zapopan, Jalisco, Mexico {gdesirena}@art, {dgomez}@gdl.cinvestav.mx

2C.R. Vázquez is with ITESM, Av. Ramón Corona 2514, CP 45201, Zapopan, Jalisco, Mexico cr.vazquez@itesm.mx

3J.L. Briz is with the DHH/I3A Univ. de Zaragoza, María de Luna 1 - 50018 Zaragoza, España briz@unizar.es
A. Timed continuous Petri nets

**Definition 2.1:** A (discrete) Petri net structure (PN) is a graph described by a 4-tuple \( N = (P,T,\text{Pre},\text{Post}) \) where \( P \) and \( T \) are finite disjoint sets of places and transitions, respectively. \( \text{Pre} \) and \( \text{Post} \) are \( |P| \times |T| \) \( \text{Pre} \)– and \( \text{Post} \)– incidence matrices, where \( \text{Pre}[i,j] > 0 \) (resp. \( \text{Post}[i,j] > 0 \)) if there is an arc going from \( p_i \) to \( t_j \) (resp. going from \( t_j \) to \( p_i \)), \( \text{Pre}[i,j] = 0 \) (resp. \( \text{Post}[i,j] = 0 \)) otherwise.

**Definition 2.2:** A (discrete) Petri net system is the pair \( Q = (N,M) \) where \( N \) is a Petri net and \( M : P \to \mathbb{N} \cup \{0\} \) is the marking function assigning zero or a natural number to each place. The value \( M[p_i] \) is named the tokens residing into \( p_i \). \( M_0 \) is named the initial marking.

**Definition 2.3:** A timed continuous Petri net (TCPN) is a time-driven continuous-state system described by the tuple \( (N,\lambda,m_0) \) where \( N \) is a PN structure and the vector \( \lambda \in (\mathbb{R}^+ \cup \{0\})^{|T|} \) represents the transitions rates determining the temporal evolution of the system. Transitions fire according to certain speed, which generally is a function of the transition rates and the current marking. Such function depends on the semantics associated to the transitions. Under infinite server semantics, the flow (the transitions firing speed, denoted as \( f(m) \)) through a transition \( t_i \) is defined as the product of the rate, \( \lambda_i \), and \( \text{enab}(t_i,m) \), the instantaneous enabling of the transition, i.e., \( f_i(m) = \lambda_i \text{enab}(t_i,m) = \lambda_i \min_{p_j \in \text{enab}(t_i,m)} f_{p_j} \). The firing rate matrix is defined by \( \Lambda = \text{diag}(\lambda_1,\ldots,\lambda_{|T|}) \).

For the flow to be well defined, every continuous transition must have at least one input place, hence in the following we will assume \( \forall t \in T, |t| \geq 1 \). The “\( \min \)” in the above definition leads to the concept of configuration. A configuration of a TCPN at \( m \) is a set of \((p,t)\) arcs describing the effective flow of each transition, in that case we say that \( p_i \) constrains \( t_j \) for each arc \((p_i,t_j)\) in the configuration. A configuration matrix is defined for each configuration as follows:

\[
\Pi(m) = \begin{cases} 
\frac{1}{\text{Pre}^{[i,j]}} & \text{if } p_i \text{ is constraining } t_j \\
0 & \text{otherwise}
\end{cases}
\] (1)

The flow through the transitions can be written in a vectorial form as \( f(m) = \Lambda \Pi(m) m \). We can apply a control action to the dynamical behavior of a PN system by adding a term \( u \) to every transition \( t_i \) such that \( 0 \leq u_i \leq f_i \), indicating that its flow can be reduced. Thus, the controlled flow of transition \( t_i \) becomes \( w_i = f_i - u_i \) and the forced state equation is

\[
\dot{m} = C[f - u] = Cw \\
0 \leq u_i \leq f_i
\] (2)

III. Modeling tasks

In this work we consider that a) Tasks are periodic, and their time periods are fixed and known; b) Tasks are independent, meaning that there is no precedence in their execution and that they do not share resources or interact to each other; c) Each task’s execution time, given in CPU cycles, is known and fixed; d) Each task has an associated deadline. Since tasks are periodic, this deadline is relative to the starting of the corresponding period. We formalize these assumptions by considering a set of independent periodic real-time tasks \( T = \{\tau_1,\ldots,\tau_n\} \), where each task \( \tau_i = (cc_i,d_i,\omega_i) \) consists of an infinite sequence of jobs that are released according to a period \( \omega_i \). The \( k \)-th job of \( \tau_i \), denoted as \( \tau_i^k \), runs during \( cc_i \) CPU cycles, becoming enabled at time \((k-1) \cdot \omega_i\). The job must be completed before its deadline \((k-1) \cdot \omega_i + d_i \) (\(k\)-th relative deadline).

We assume that there exists \( m \) identical processors and \( n \) tasks. We also assume that all task parameters, including task period and execution time are integers and that any task can be preempted at any time. We define the hyper-period as the period equal to the least common multiple of periods \( H = \text{lcm}(\omega_1,\omega_2,\ldots,\omega_n) \) of the \( n \) periodic tasks. A feasible schedule can be repeated every hyper-period [1].

The proposed modeling methodology models each task as the TCPN module of Fig. 1(a), which is explained below.

1) **Modeling task execution:** The period \( \omega_i \) of task \( \tau_i \) implies that, in average, \( \frac{1}{\omega_i} \) jobs arrive per second (i.e., arriving frequency). This is captured in the TCPN module of Fig. 1(a) as the firing rate \( \lambda^w_i = \frac{1}{\omega_i} \) of transition \( t_i^w \).

2) **Modeling task deadline:** The relative deadline \( d_i \) of task \( \tau_i \) is represented in the model of Fig. 1(a) by the marking \( d_i \) at place \( p_i^d \).

3) **Modeling duration of a task:** The duration of a task is represented in the TCPN model of Fig. 1(a) by the arc going from transition \( t_i^w \), representing the jobs arrival, to the place \( p_i^c \), representing the jobs that have arrived. The weight \( cc_i \) of the arc is included so the marking at \( p_i^c \) represents the jobs that have arrived (to be executed) in CPU cycle units.

IV. Modeling task allocation CPU and global model

We assume in this paper that all CPUs have the same capabilities, although this modeling methodology can be applied to heterogeneous cores. Thus, any task in the set \( T \) can be allocated to any CPU in the set \( \mathcal{P} = \{CPU_1,\ldots,CPU_m\} \). We also cling to the common assumption that task migration and preemption have no cost and every task’s job must execute sequentially on at most one processor at any given instant in time.
A. Modeling task allocation to CPU

A single CPU is modelled by the TCPN module of Fig. 1(b), consisting of places $p_{j}^{busy}$, $p_{j}^{idle}$ and transitions $t_{j}^{exec}$, $t_{i,j}^{alloc}$, ..., $t_{n,j}^{alloc}$. The marking at place $p_{j}^{idle}$ models the number of available CPU cycles per second (throughput capacity). The initial marking at $p_{j}^{idle}$ is the maximum throughput $\mu_{j}$ of CPU. Place $p_{j}^{busy}$ represents the busy state of the processor, the marking at this place represents the number of CPU cycles per second reserved for tasks execution (throughput being used). Transition $t_{j}^{exec}$ models the CPU execution rate. Transitions $t_{i,j}^{alloc}$ model the allocation of tasks $\tau_{1}$, ..., $\tau_{n}$, respectively, to processor CPU.

![Fig. 2. TCPN module allocation of n tasks to a single CPU.](image)

1) Building the global model: The global model is obtained by merging the tasks models (Fig. 1(a)) and the allocation CPU models (Fig. 1(b)). The global model for a single processor CPU is depicted in Fig. 2. The arcs from places $p_{i}^{cc}$ to transitions $t_{i,j}^{exec}$ represent that jobs of $\tau_{j}$ are being allocated to processor CPU. To merge the models, we add places $p_{i}^{alloc}$ and arcs going from $p_{i}^{alloc}$ to $p_{j}^{alloc}$. The marking of place $p_{i}^{alloc}$ stands for the total amount of jobs of $\tau_{j}$ that has been allocated to CPU, from the initial time.

The global model, encompassing $n$ tasks and $m$ CPUS, is depicted in Fig 3. Note that the resulting model is a deadlock-free Petri net.

B. Fundamental equation of the global model

This equation dictates the dynamic evolution of the global TCPN model (Fig. 3), and is given by:

$$\dot{m} = C\Pi(m)m - \hat{C}u$$

$$0 \leq u \leq (\Lambda\Pi(m)m)[T^{alloc}]$$

where $\hat{C}$ has the columns of $C$ corresponding to transitions $t_{i,j}^{alloc}$ (denoted as $T^{alloc}$) in Fig. 3. The flow through these transitions controls the rate at which an scheduler allocates jobs to each CPU. The control vector $u$ represents the speed reductions (from the autonomous evolution) for the allocation of jobs to CPUs, i.e., if $u = 0$ then tasks are allocated according to the available jobs and CPUs (given by $(\Lambda\Pi(m)m)[T^{alloc}]$) but if $u = (\Lambda\Pi(m)m)[T^{alloc}]$ then no job is allocated.

V. FLUID SCHEDULING OF A SET OF REAL-TIME TASKS

Since the obtained global TCPN is modeled by differential state equations, different control techniques may be applied. This section proposes a fluid schedule based on a sliding mode control technique [10]. From now on, $\zeta$ represents the current time. The control approach herein reported starts by computing the task fluid-schedule function:

$$FSC_{\tau_{i}}(\zeta) = \frac{c_{i}}{\omega_{i}} \zeta$$

According to the fluid algorithms [6] [9], this function represents the optimal fluid execution of task $\tau_{i}$. Through this work, we will say that this function represents the execution percentage of task $\tau_{i}$ at time $\zeta$.

The execution error of task $\tau_{i}$ (denoted $E_{\tau_{i}}(\zeta)$) is the difference between the marking of places $p_{i}^{alloc}$ (total amount of jobs of $\tau_{i}$ allocated) in the global TCPN model and the optimal fluid execution.

$$E_{\tau_{i}}(\zeta) = m_{i}^{alloc}(\zeta) - FSC_{\tau_{i}}(\zeta)$$

Thus, if $E_{\tau_{i}}(\zeta) = 0, \forall \zeta > \zeta_{0}$, then the marking in places $p_{i}^{alloc}$ is equal to the optimal fluid schedule; hence the firing of transitions $t_{i,j}^{alloc}$ represents the optimal task allocation to CPU’s. In order to bring the error to zero, for each task $\tau_{i}$ and CPU’s, the following continuous sliding-mode control law is used [10]:

$$v_{i,j}^{alloc}(\zeta) = \left[\frac{1}{2} + \frac{1}{2}\text{sign}(E_{\tau_{i}}(\zeta))\right]f_{i,j}^{alloc}(\zeta)$$

where

$$\text{sign}(x) = \begin{cases} 
1 & \text{if } x \geq 0 \\
-1 & \text{if } x < 0
\end{cases}$$

and

$$f_{i,j}^{alloc}(\zeta) = \lambda_{i,j}^{alloc}\min(m_{i}^{cc}(\zeta), m_{j}^{idle}(\zeta))$$

When this control law is applied to the system (3), for each task $\tau_{i}$ and CPU’s, then each $E_{\tau_{i}}(\zeta)$ becomes zero and the fluid schedule meets the time constraints. This is formalized in the following proposition.
**Proposition 5.1:** Let $\mathcal{T}$ and $\mathcal{P}$ be the sets of $n$ tasks and $m$ processors, respectively, where the fluid execution tasks $FSC_{\tau_i}(\zeta)$ and execution errors $E_{\tau_i}(\zeta)$ are defined.

If the control law given by (6) is applied to the system (3), for each task $\tau_i$ and $CPU_j$, then the execution errors converge to zero asymptotically.

**Proof:** In order to prove that each $E_{\tau}$ converges to zero asymptotically, the following assumptions are made: first, since the fluid schedule can allocate all the jobs in such a way that they are executed in time by all the processors, the model assumes that there are enough $CPU$ cycles available in each place $p_{ij}^{dte}$, so these places do not restrict the flows $i_{i,j}^{alloc}$, consequently $\lambda_{i,j}^{alloc} = \lambda_{i,j}^{ alloc}m_{cc}^{ alloc}$ for each $\tau_i$ and $CPU_{j}$. Second, places $p_{ij}^{busy}$ and $p_{ij}^{dte}$ do not affect the control. Therefore, the global model will be analysed in the sequel without considering these places. The controlled flow of a transition $i_{i,j}^{ alloc}$ is $i_{i,j}^{ alloc} = f_{i,j}^{ alloc} - w_{i,j}^{ alloc}$, where $\lambda_{i,j}^{ alloc}$ is the control action as defined in (6). Note that when $i_{i,j}^{ alloc} = f_{i,j}^{ alloc} - w_{i,j}^{ alloc} > 0$, transition $i_{i,j}^{ alloc}$ is fired, i.e., jobs of $\tau_i$ are being allocated to $CPU_{j}$. Therefore, the evolution of task $\tau_i$ is described by the following state variables:

\[
\begin{align*}
\cdot m_i &= 0 \\
\cdot cc &= \frac{m_i}{\omega_i} - \sum_{j=1}^{m} w_{i,j}^{ alloc} \\
\cdot alloc &= \sum_{j=1}^{m} w_{i,j}^{ alloc} \\
\cdot alloc &= \sum_{j=1}^{m} w_{i,j}^{ alloc} \\
\end{align*}
\]

We have to drive the variables $E_{\tau}$ to zero by means of the control actions $w_{i,j}^{ alloc}, \ldots, w_{i,j}^{ alloc}$. In order to prove the asymptotic stability of (5), a Lyapunov function (see, for instance, [14]) can be defined, satisfying $V(0) = 0, \forall x \neq 0$ and $V(x) > 0$ and $V(x) > 0 \forall x \neq 0$. The candidate Lyapunov function $V$ considered here is:

\[
V(E_{\tau_1}, \ldots, E_{\tau_n}) = \frac{1}{2} E_{\tau_1}^2 + \ldots + \frac{1}{2} E_{\tau_n}^2
\]

Note that $V = 0$ iff each $E_{\tau_i} = 0$. Furthermore, $V > 0$ if any $E_{\tau_i} \neq 0$. Thus, $V$ can be considered a Lyapunov function (and thus (5) is asymptotic stable) iff $V < 0$ for any $E_{\tau_i} \neq 0$. To prove this, the derivative of $V$ is computed as:

\[
\begin{align*}
\dot{V} &= \sum_{i=1}^{n} E_{\tau_i} \dot{E}_{\tau_i} \\
&= \sum_{i=1}^{n} \left( \sum_{j=1}^{m} \left[ \frac{1}{2} E_{\tau_i} f_{i,j}^{ alloc} - \frac{1}{2} E_{\tau_i} f_{i,j}^{ alloc} - E_{\tau_i} \frac{cc_i}{\omega_i} \right] \right) \\
&= \sum_{i=1}^{m} \left( \frac{1}{2} E_{\tau_i} f_{i,j}^{ alloc} - \frac{1}{2} E_{\tau_i} f_{i,j}^{ alloc} - E_{\tau_i} \frac{cc_i}{\omega_i} \right)
\end{align*}
\]

**Proof:**

- **a)** When $E_{\tau_i}$ is positive. The term (10) for the task $\tau_i$ becomes $-E_{\tau_i} \frac{cc_i}{\omega_i}$. Note that $0 < cc_i/\omega_i$, then the term is negative.

- **b)** When $E_{\tau_i}$ is negative. The term (10) for the task $\tau_i$ becomes $E_{\tau_i} \frac{cc_i}{\omega_i}$. Note that $0 < cc_i/\omega_i$, then the term is negative.

Now, the flow $f_{i,j}^{ alloc}$ through each transition $i_{i,j}^{ alloc}$ is computed as $f_{i,j}^{ alloc} = \lambda_{i,j}^{ alloc}m_{cc}^{ alloc}$. Thus, the term above is negative if $cc_i/\omega_i < \lambda_{i,j}^{ alloc}m_{cc}^{ alloc} < 0$, where $\lambda_{i,j}^{ alloc} = \lambda_{i,j}^{ alloc}$. In order to prove this inequality, it is required to compute a lower bound for $m_{cc}^{ alloc}$. For this, the dynamic of $m_{cc}^{ alloc}$ is represented as $m_{cc} = cc_i/\omega_i - \sum_{j=1}^{m} (f_{i,j}^{ alloc} - w_{i,j}^{ alloc})$. Since $E_{\tau_i} < 0$, then each control action $u_{i,j} = 0$. Thus, $m_{cc} = cc_i/\omega_i - \lambda_{i,j}^{ alloc}m_{cc}^{ alloc}$. The solution of this differential equation leads to:

\[
m_{cc}^{ alloc}(\zeta) = cc_i/\omega_i + \left( m_{cc}^{ alloc}(0) - cc_i/\omega_i \right) e^{-\lambda_{i,j}^{ alloc} \zeta}
\]

Therefore, $m_{cc}^{ alloc}(\zeta)$ is bounded, in fact $m_{cc}^{ alloc}(\zeta) > cc_i/\omega_i > \lambda_{i,j}^{ alloc}$. Assume that $cc_i/\omega_i > 1$, i.e., the total allocation rate $\lambda_{i,j}^{ alloc}$ (not execution rate) for task $\tau_i$ is larger than its arrival frequency $1/\omega_i$. Thus, since $m_{cc}^{ alloc}(0) = cc_i$ then $m_{cc}^{ alloc}(0) > cc_i/\omega_i \lambda_{i,j}^{ alloc}$. The bound for $m_{cc}^{ alloc}(\zeta)$ is then given by $m_{cc}^{ alloc}(\zeta) > cc_i/\omega_i \lambda_{i,j}^{ alloc}$, which implies $cc_i/\omega_i > \lambda_{i,j}^{ alloc} m_{cc}^{ alloc} < 0$, consequently (10) is negative.

Finally, since each term in the sum in (9) is negative then $V < 0$. 

A negative task error $E_{\tau_i}(\zeta)$ reveals the existence of unattended jobs, so the control (6) turns on transitions $i_{i,j}^{ alloc}$ in order to allocate jobs to the processors. Otherwise, if $E_{\tau_i}(\zeta)$ is positive or zero, it means that the processors are executing jobs on time, and the control turns off the transitions $i_{i,j}^{ alloc}$, i.e., stops allocating jobs. The complexity of the fluid scheduler depends on the numerical method used to solve the differential equation (3) (for instance, Runge-Kutta, Euler’s Method, etc). All these methods and (3) are solved in polynomial time at every integration step.

**VI. On-line discretization of a fluid schedule**

We have proved that the fluid scheduler proposed in the previous section is theoretically feasible (i.e, the execution error converges to zero and tasks meet the time constraints). As all fluid schedulers, it triggers an unfeasible number of task preemptions and migrations. To deal with this issue we provide a discrete implementation described by Algorithm 1.

Due to the periodicity of the schedule, we can limit the schedule of the tasks up to the hyper-period (from time 0 to time $H$) [6]. As mentioned before, a job $\tau_i^k$ must be completed before its $k-th$ deadline $sd_{i,k} = (k-1)\omega_i + d_i$, in absolute time. We define $SD_1 = \{sd_{1}, sd_{2}, \ldots\}$ as the set of all deadlines for all task’s jobs between zero and $H$. By defining $SD = SD_1 \cup \ldots \cup SD_{|\tau|}$, the elements of $SD$ are renamed in ascending order, according to their value, as $SD = \{sd_0, \ldots, sd_{r}\}$. Algorithm 1 requires a time period (quantum) $Q$, which is defined as the greatest common
divisor of the elements \( sd_i \in SD \) and the values of the function \( FSC_{\tau_i} \) evaluated at \( sd_i \).

Algorithm 1 On-line discretization of a fluid schedule

1. **Input** The TCPN and discrete PN of the set of tasks \( T \), the ordered set \( SD \) where any \( sd_k \in SD \) is lower or equal than \( H \). The quantum \( Q \). The task fluid-schedule functions \( FSC_{\tau_i} \).
2. **Output** A feasible discrete schedule.
3. **Initialize** \( i = 1, sd = sd_i, \zeta = 0 \)
4. while \( \zeta < H \) do
5. \( \text{All tasks are preempted from the processors} \)
6. Compute remaining jobs: \( RE_{\tau_i}(\zeta) = FSC_{\tau_i}(sd)-M_{\tau_i}(\zeta) \)
7. Compute the set of transitions \( \tau_{*} \) to be fired in the discrete model: \( ET(\zeta) = \{ \tau_{*} \mid RE_{\tau_i}(\zeta) > 0 \} \)
8. Compute the priority for every transition \( \tau_{*} \) in \( ET \):
   \[ PR_{\tau_i}(\zeta) = m_{\text{alloc}}(\zeta) - M_{\tau_i}(\zeta) \]
9. for \( j = 1 \) to \( m \) do
10. Select \( \tau_{*} \) with the highest priority value \( PR_{\tau_i} \) in \( ET \)
11. Fire \( \tau_{*} \) in the discrete PN, assign task \( \tau_{*} \) to processor \( j \) from \( \zeta \) to \( \zeta + Q \)
12. Remove \( \tau_{*} \) from \( ET \)
13. end for
14. **SIMULATE** the global TCPN model from \( \zeta \) to \( \zeta + Q \)
15. Update time: \( \zeta = \zeta + Q \)
16. if \( \zeta == sd \) then
17. \( i = i + 1, sd = sd_i \)
18. end if
19. end while

Algorithm 1 computes a discrete schedule from the fluid schedule introduced in Section V. It requires of a new discrete Petri net where this PN has one source transition \( t_{\tau_i} \) and one sink place \( p_{\tau_i} \) per task \( \tau_i \). The firing of a transition \( t_{\tau_i} \) determined by the algorithm, means that \( Q \) CPU cycles of task \( \tau_i \) are allocated. The marking of a place \( p_{\tau_i} \), denoted \( M_{\tau_i} \), represents the total amount of CPU cycles of task \( \tau_i \) that has been allocated from the initial time, and it constitutes the analogue of place \( p_i^{\text{alloc}} \) in the fluid model of Fig. 3. The discrete schedule resulting from Algorithm 1 equals the fluid schedule at every deadline time \( sd_k \in SD \), i.e. it ensures that the discrete schedule meets all deadlines of all tasks.

If at any time \( \zeta, sd_i < \zeta < sd_{i+1} \), it holds that \( FSC(sd_k) > M_{\tau_i}(\zeta) \) (the required fluid schedule at the end of the interval is bigger than the current discrete schedule), then \( \tau_i \) must be allocated in a CPU. Thus the \( m \) tasks with the current positive greatest remaining jobs execution \( (RE_{\tau_i}(\zeta) = FSC(sd_k) - M_{\tau_i}(\zeta)) \) and task priority function \( PR_{\tau_i}(\zeta) = m_{\text{alloc}}(\zeta) - M_{\tau_i}(\zeta) \) must be allocated to a CPU.

In this paper, the value of \( m_{\text{alloc}}(\zeta) \) can be replaced by \( FSC(\zeta) \) since they have the same value. However, we use it here because in our target systems we will include thermal characteristics, making these values no longer be equal because we will have to balance thermal and temporal trade-offs. Note that the algorithm executes \( I = \frac{H}{Q} \) times, where \( H \) is the hyper-period, thus the loop in step 4 of the algorithm runs \( I \) times. The instruction inside this loop runs in polynomial time in the size of the transitions of the TCPN. As mentioned in previous section, the execution on the TCPN is polynomial and therefore the algorithm is polynomial too.

**Proposition 6.1:** If a feasible fluid schedule is given as input to Algorithm 1, then the resulting discrete schedule has the following properties.

1) It meets task time constraints at every scheduling point \( sd_k \in SD \).
2) The number of task migrations and preemptions is bounded.

**Proof:** Part 1) Sentence 1.

From the definition of quantum, we know that the time interval \( [sd_k, sd_{k+1}] \) is divided by the quantum into \( D_{k+1} = (sd_{k+1} - sd_k)/Q \) time sub-intervals. From [6] we know that the fluid schedule meets the task time constraints at every time, which is specially true at the \( sd_k \) points. Moreover, since the fluid schedule is feasible then, at time \( sd_k \), the CPU’s are capable to execute the required percentage \( FSC_{\tau_i}(sd_k) \) of any task \( \tau_a \). Assuming that there exist \( m \) processors and \( n \) tasks, and since the \( m \) processors are capable to execute the fluid schedule, then:

\[ m \cdot D_{k+1} \geq \sum_{i=1}^{n} (FSC_{\tau_i}(sd_{k+1}) - FSC_{\tau_i}(sd_k)) \]  

At time zero, both the fluid schedule and discrete schedule have executed zero percentage of each task, thus the discrete schedule meets task time constraints at \( sd_0 \).

Now, we will show that if the discrete schedule meets the fluid schedule at any \( sd_k \) then it meets the fluid schedule at any \( sd_{k+1} \) as well.

Proceeding by contradiction, assume that the discrete schedule does not meet the fluid schedule at \( sd_{k+1} \). Then the remaining jobs functions are positive for some tasks (i.e. the discrete PN has executed these tasks in a lower percentage than the fluid one). For the sake of explanation, suppose that there exists only one task \( \tau_a \) such that \( RE_{\tau_a}(sd_{k+1}) = N_a > 0 \). Since (11) holds, then the processors have the capability to execute the required percentage of tasks at time \( sd_{k+1} \). However, since \( \tau_a \) was not allocated in the required percentage \( (FSC_{\tau_a}(sd_{k+1})) \), then \( \tau_a \) had some remaining jobs at time \( \zeta = sd_{k+1} - Q \). Therefore, two cases are possible:

Case 1: If \( N_a = \alpha Q \), where \( \alpha = 1 \) (i.e., \( PR_{\tau_a} = Q \))

In this case two possibilities arise:

a) \( \tau_a \) was fired at time \( \zeta \), then \( \tau_a \) finishes the required percentage of execution, i.e., \( RE_{\tau_a}(sd_{k+1}) = 0 \), which is a contradiction.

b) \( \tau_a \) was not allocated at time \( \zeta \). Thus, according to step 6 of the algorithm, \( m \) tasks, different from \( \tau_a \), were found having priority larger or equal than that of \( \tau_a \), thus they were allocated.

If the behavior of the algorithm is analyzed at time \( \zeta - Q \) (a previous time step), it will result that \( m \) tasks were found, different than \( \tau_a \), having larger or equal priorities than that of \( \tau_a \) (otherwise, task \( \tau_a \) would be allocated and thus finished its execution). By repeating this analysis, going back in time until \( sd_k \), it will be obtained that \( RE_{\tau_a}(sd_k) = N_a > 0 \), i.e., the discrete schedule does not meet the fluid schedule at \( sd_k \), a contradiction.

Case 2: If \( N_a = \alpha Q \), \( \alpha > 1 \), then at time \( \zeta \), \( PR_{\tau_a}(\zeta) > PR_{\tau_a}(\zeta) \), for some task \( \tau_i \neq \tau_a \). Thus \( \tau_a \) was allocated at time \( \zeta \) and \( \alpha = \alpha - 1 \). Let \( \zeta = \zeta - Q \). If \( sd_k \) is reached then
\( RE_{\alpha}(sd_k) = N_a > 0 \), i.e., the discrete schedule does not meet the fluid schedule at \( sd_k \), a contradiction, otherwise if \( \alpha = 1 \Rightarrow 1 \) then go to Case 1.

Part 2) Sentence 2.

Since the hyper-period \( H \) is divided into \( I = \frac{H}{Q} \) subintervals and task migrations and preemptions occur at the end of this subintervals then the number of task migrations and preemptions is bounded by \( I \).

VII. Example

There exists a set of tasks: \( \mathcal{T} = \{ \tau_1, \tau_2, \tau_3 \} \), where \( \tau_1 = (9, 10, 10) \), \( \tau_2 = (9, 10, 10) \), \( \tau_3 = (8, 40, 40) \) running on two processors. The hyper-period is \( H = 40 \). Applying the fluid controller, the fluid schedule, marking in \( m_{\text{alloc}}(\zeta) \), is obtained. Fig. 4 shows the fluid schedule and that the errors \( E_{\tau} \) converge to zero, therefore the fluid schedule based on the sliding-mode control meets the time constraints.

![Feasible fluid schedule](image)

**Fig. 4.** A feasible fluid schedule. The execution error \( E_{\tau} \) converges to zero due the control action.

In Algorithm 1 \( SD = \{10, 20, 30, 40\} \) and \( Q = 1 \). The computed discrete schedule is depicted in Fig. 5. Note that at every \( sd \in SD \) the accumulated tokens in \( m_{\text{alloc}}(sd) \) meets the fluid schedule requirements. For instance at \( sd = 10 \), the fluid schedule indicates that task \( \tau_1 \) requires 9 time units. Analyzing Fig. 5, \( \tau_1 \) is executed 1 time units in \( CPU_1 \) from \( \zeta = 0 \) to \( \zeta = 1 \) and 3 time units in \( CPU_2 \) from \( \zeta = 1 \) to \( \zeta = 4 \) and finally 5 time units in \( CPU_1 \) from \( \zeta = 5 \) to \( \zeta = 10 \), thus \( \tau_1 \) is executed 9 time units at \( sd = 10 \).

VIII. CONCLUSIONS

We propose a fluid scheduler based on a sliding mode control technique, designed to easily integrate thermal restrictions in the near future. The scheduler is derived from a TCPN model of tasks and CPUs, which constitutes itself a modeling methodology for fluid schedulers in real-time systems. Moreover, we present a discrete implementation of the fluid scheduler, which allows reducing task switching and migration. The scheduler here presented shares with P-fair algorithms their ability to exploit CPU time. Furthermore, the scheduler keeps the number of context switches and migrations reasonably low, by solving the control equation only upon the jobs’ deadlines, profiting from the qualities of deadline partitioning algorithms. Future work will address the design of fine thermal-aware schedulers using this model.

REFERENCES


