ISSN 2007-9737

A Flexible Framework for Real-Time Thermal-Aware Schedulers using Timed Continuous Petri Nets

Gaddiel Desirena López[1], Laura Elena Rubio Anguiano[1], Antonio Ramírez Treviño[1], José Luis Briz Velasco[2]

[1] CINVESTAV-IPN Unidad Guadalajara,
Mexico

[2] Universidad de Zaragoza,
Spain

{gdesirena,lerubio,art}@gdl.cinvestav.mx, briz@unizar.es

Abstract. This work presents *TCPN-ThermalSim*, a software tool for testing Real-Time Thermal-Aware Schedulers[1]. This framework consists of four main modules. The first one helps the user to define the problem: task set with periods, deadlines and worst case execution times in CPU cycles, along with the CPU characteristics, temperature and energy consumption. The second module is the Kernel simulation, which builds up a global simulation model according to the configuration module. In the third module, the user selects the scheduler algorithm. Finally the last module allows the execution of the simulation and present the results. The framework encompasses two modes: manual and automatic. In manual mode the simulator uses the task set data provided in the first section. In automatic mode the task set is generated by parameterizing the integrated UUniFast algorithm.

Keywords. Scheduling, simulator, Petri nets.

1 Introduction

Many modern embedded real-time (RT) systems can benefit from today's powerful System-on-Chip multicores (MPSoCs). However, RT task scheduling on multiprocessors is far more challenging than traditional RT scheduling on uniprocessors. Although it is possible to find practical solutions for specific cases, considering thermal restrictions, optimizing energy or dealing with resource sharing are still open questions ([6]). We have been exploring the design of Thermal-Aware RT Schedulers in recent contributions, leveraging control techniques and combining fluid and combinatorial schedulers, taking the most of both approaches ([15]).

On the one hand, fluid schedulers avoid the NP-completeness of a solely combinatorial approach, and allow the use of continuous controllers which make easier to cope with disturbance rejection or to adapt to small parameter variations. On the other hand, the combinatorial approach better matches the nature of the problem, discretizing the fluid schedules to avoid large migrations and context switches.

Testing these kind of schedulers or experimenting with their variations by implementing them on a real system, even on specific evaluation platforms like Litmus-RT ([5]), can be overkilling if we are just performing preliminary explorations of the design space. For this reason, herein we are presenting a novel flexible simulation framework for real time thermal-aware schedulers: TCPN-ThermalSim. It is composed of four main modules, which depend on certain submodules to work.

The first module, which is referred as the configuration module, allows us to introduce the task set according to the three-parameter task model ([7]), or use the UUnifast submodule to generate automatically the task set. It also permits to describe the platform used to execute the task set, composed by the number of CPUs,

[1] Available at: https://www.gdl.cinvestav.mx/art/uploads/SchedulerFrameworkTCPN.zip

CPU frequencies and its thermal parameters. The second module is the Kernel simulation, which works over three submodules, where each submodule generates a Timed Continuous Petri Net (TCPN) model that will be merged in order to build a global simulation model. The third module focuses on the scheduler definition, it allows the addition of scheduler algorithms to its analysis. And the last module correspond to the execution of the simulation and present the results as graphs or data in the workspace.

This framework includes out-of-the-box the task, CPU usage and thermal general purpose models reported in [22]. We have included three schedulers: a global EDF scheduler from [1], and other two from our own authorship a RT fluid scheduler ([15]) and a thermal aware RT scheduler ([16]), however the user can define his own scheduler. Sec. 2 provides some background on multiprocessor RT sheduling and Timed Continuous Petri Nets (TCPN), since this formal tool will be used to model the task set, CPU usage and thermal aspects of task execution. Sec. 3 describes the simulation framework architecture and Sec. 4 the user interface. The schedulers included in the framework are described in Sec. 5. Sec. 6 shows a usage example with results, and Sec. 7 presents some conclusions and future work.

2 Background

There are three well-known avenues to leverage multiprocessors for RT scheduling. The *partitioned* approaches resort to statically allocating tasks to processors. This results into a simple schedulability analysis, since they can use results from RT uniprocessor scheduling. The downside is that this encompasses an NP-hard bin-packing problem and as a consequence, assuring schedulability imposes a maximum CPU utilization bound of 50% or less ([21]).

Global scheduling gets around the problem by dynamically allocating jobs (the periodic tasks' instances) to processors, achieving a 100% utilization bound. Thus, *Pfair*-based algorithms allocates a new task for execution every time quantum ([26]), which requires that all task parameters are multiples of such a time quantum. The principal inconvenience is that this approach triggers an unfeasible number of preemptions and migrations. By this reason, *Deadline partitioning* does only take scheduling decisions on the set of all deadlines of all tasks in the system, achieving optimality with fewer preemptions than *Pfair* ([18]).

A third approach mixes static and dynamic allocation. Thus, *clustered scheduling* statically allocate tasks to clusters of CPUs, but jobs can migrate within their cluster ([8]). Alternatively, *semipartitioned scheduling* preforms a preliminary static allocation of some tasks over the whole set of available CPUs, allowing the rest of them to migrate ([19]). Recently, [6] and [9] leverage this technique to lowering migrations. In order to test the performance of such schedulers and new ones in early stages, the herein proposed simulation framework is a powerful tool. It is capable to stress schedulers under very realistic conditions, avoiding the effort wasted in adapting the schedulers to actual operating systems and detect from early stages the consequences of a bad heating balance.

When a dynamic thermal balance is a requirement, in order to keep the maximum temperature under control, static allocation techniques are far more restrictive than global approaches. This is the reason why we consider global scheduling, with strategies to minimize preemption and migration like *deadline partitioning*.

Over the years different real time simulation tools have been proposed, including different features, for example, Cheddar ([25]) is a real time scheduler simulator written in Ada, it handles the multiprocessor case and provides implementations of scheduling, partitioning and analysis algorithms, but their interface is not very user friendly. Another tool is YARTISS ([10]), it evaluates scheduling algorithms by considering overheads or hardware effects, and its design focuses on energy consumption. On the other hand, SimSo ([11]) is a simulation tool that includes different scheduling policies, and takes into account multiple kinds of overheads. All of these tools represent a great aid when developing new algorithms, but none of them is capable of including a thermal model, for this reason we developed *TCPN-ThermalSim* as a framework

Table 1. Software simulation tools comparison

Framework	Programming language	Custom Schedulers	Energy considerations	Thermal considerations
Cheddar	Ada	✓		
SimSo	Python	✓		
YARTISS	Java	✓	✓	
TCPN ThermalSim	MATLAB	✓	✓	✓

capable of developing a thermal analysis of the scheduling algorithms.

Finally, since the proposed framework models tasks and CPUs using Timed Continuous Petri Nets (TCPN), this Section introduces basic definitions concerning Petri nets and continuous Petri nets. An interested reader may also consult [12], [13], [24] to get a deeper insight in the field.

2.1 Discrete Petri Nets

Definition **2.1** *A (discrete) Petri net is the 4-tuple* $N = (P, T, \boldsymbol{Pre}, \boldsymbol{Post})$ *where* P *and* T *are finite disjoint sets of places and transitions, respectively.* $\boldsymbol{Pre}$ *and* $\boldsymbol{Post}$ *are* $|P| \times |T|$ $\boldsymbol{Pre}-$ *and* $\boldsymbol{Post}-$ *incidence matrices, where* $\boldsymbol{Pre}(i,j) > 0$ *(resp.* $\boldsymbol{Post}(i,j) > 0$*) if there is an arc going from* t_j *to* p_i *(resp. going from* p_i *to* t_j*), otherwise* $\boldsymbol{Pre}(i,j) = 0$ *(resp.* $\boldsymbol{Post}(i,j) = 0$*).*

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 marking is also represented as a column vector* $\boldsymbol{M}$*, such that its* $i-th$ *element is equal to* $M(p_i)$*, named the tokens residing into* p_i*.* $\boldsymbol{M}_0$ *denotes the initial marking distribution.*

A transition $t \in T$ is said enabled at the marking $\boldsymbol{M} \in \mathbb{N}^{|P|}$ *iff* $\boldsymbol{M} \geq \boldsymbol{Pre}[p,t]$, the occurrence or firing of an enabled transition leads to a new marking distribution $\boldsymbol{M'} \in \mathbb{N}^{|P|}$ that can be computed by using $\boldsymbol{M'} = \boldsymbol{M} + \boldsymbol{C}[P,t] = \boldsymbol{M} + \boldsymbol{C} \cdot \boldsymbol{e_t}$, where $\boldsymbol{C} = \boldsymbol{Post} - \boldsymbol{Pre}$ is named the incidence matrix, and $\boldsymbol{e_t}$ denotes the $t-th$ elementary vector ($\boldsymbol{e_t}(k) = 1$ if $k = t$, otherwise $\boldsymbol{e_t}(k) = 0$).

2.2 Continuous and Timed Continuous Petri Nets

Definition **2.3** *A continuous Petri Net (ContPN) is a pair* $ContPN = (N, \boldsymbol{m}_0)$ *where* $N = (P, T, \boldsymbol{Pre}, \boldsymbol{Post})$ *is a Petri net (PN) and* $\boldsymbol{m}_0 \in \{\mathbb{R}^+ \cup 0\}^{|P|}$ *is the initial marking.*

The evolution rule is different from the discrete *PN* case. In continuous *PN*'s the firing is not restricted to be integer. A transition t_i in a *ContPN* is *enabled* at $\boldsymbol{m}$ if $\forall\ p_j \in^\bullet t_i, \boldsymbol{m}[p_i] > 0$; and its *enabling degree* is defined as $enab(t_i, \boldsymbol{m}) = min_{p_j \in^\bullet t_i} \frac{\boldsymbol{m}[p_j]}{\boldsymbol{Pre}[p_j, t_i]}$. The firing of t_i in a certain positive amount $\alpha \leq enab(t_i, \boldsymbol{m})$ leads to a new marking $\boldsymbol{m'} = \boldsymbol{m} + \alpha \boldsymbol{C}[P, t_i]$, where $\boldsymbol{C} = \boldsymbol{Post} - \boldsymbol{Pre}$ is computed as in the discrete case.

If $\boldsymbol{m}$ is reachable from $\boldsymbol{m}_0$ by firing the finite sequence $\boldsymbol{\sigma}$ of enabled transitions, then $\boldsymbol{m} = \boldsymbol{m_0} + \boldsymbol{C}\overrightarrow{\boldsymbol{\sigma}}$ is named the fundamental Eq. where $\overrightarrow{\boldsymbol{\sigma}} \in \{\mathbb{R}^+ \cup 0\}^{|T|}$ is the firing count vector, i.e $\overrightarrow{\boldsymbol{\sigma}}[t_j]$ is the cumulative amount of firings of t_j in the sequence σ.

Definition **2.4** *A timed continuous Petri net (TCPN) is a time-driven continuous-state system described by the tuple* $(N, \boldsymbol{\lambda}, \boldsymbol{m}_0)$ *where* $(N, \boldsymbol{m}_0)$ *is a continuous PN and the vector* $\boldsymbol{\lambda} \in \{\mathbb{R}^+ \cup 0\}^{|T|}$ *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 the infinite server semantics [23] the flow through a transition t_i (the transition firing speed) is defined as the product of its rate,

λ_i, and $enab(t_i, \boldsymbol{m})$, the instantaneous enabling of the transition, i.e., $f_i(\boldsymbol{m}) = \lambda_i enab(t_i, \boldsymbol{m}) = \boldsymbol{\lambda}_i \min_{p_j \in {}^\bullet t_i} \frac{\boldsymbol{m}[p_j]}{\boldsymbol{Pre}[p_j, t_i]}$ (through the rest of this paper, for the sake of simplicity the flow through a transition t_i is denoted as f_i).

The firing rate matrix is denoted by $\boldsymbol{\Lambda} = diag(\lambda_1, ..., \lambda_{|T|})$. For the flow to be well defined, every continuous transition must have at least one input place, hence in the following we will assume $\forall t \in T, |{}^\bullet t| \geq 1$. The "min" in the above definition leads to the concept of configuration.

A configuration of a *TCPN* at $\boldsymbol{m}$ is a set of arcs (p_i, t_j) such that p_i provides the minimum ratio $\boldsymbol{m}[p]/\boldsymbol{Pre}[p, t_i]$ among the places $p \in {}^\bullet t_i$ at the given marking $\boldsymbol{m}$. 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:

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

The flow through the transitions can be written in vectorial format as $\boldsymbol{f}(\boldsymbol{m}) = \boldsymbol{\Lambda\Pi}(\boldsymbol{m})\boldsymbol{m}$. The dynamical behaviour of a *PN* system is described by its fundamental equation:

$$\dot{\boldsymbol{m}} = \boldsymbol{C\Lambda\Pi}(\boldsymbol{m})\boldsymbol{m}. \quad (2)$$

In order to apply a control action to (2), a term u such that $0 \leq u_i \leq f_i(m)$ is added to every transition t_i to indicate that its flow can be reduced. Thus the *controlled flow* of transition t_i becomes $w_i = f_i - u_i$. Then, the forced state equation is:

$$\begin{gathered} \dot{\boldsymbol{m}} = \boldsymbol{C}[\boldsymbol{f}(\boldsymbol{m}) - \boldsymbol{u}] = \boldsymbol{Cw} \\ 0 \leq u_i \leq f_i(m). \end{gathered} \quad (3)$$

2.3 System Definition

The task model accepted by this framework follows the three-parameter task model ([7]), but is extended to include energy consumption parameters. Each periodic real-time task τ_i is described by a quadruplet $\tau_i : (cc_i, \omega_i, d_i, e_i)$, where cc_i is the worst-case execution time in cycles, d_i is the deadline, ω_i is the period, and e_i is the task energy consumption.

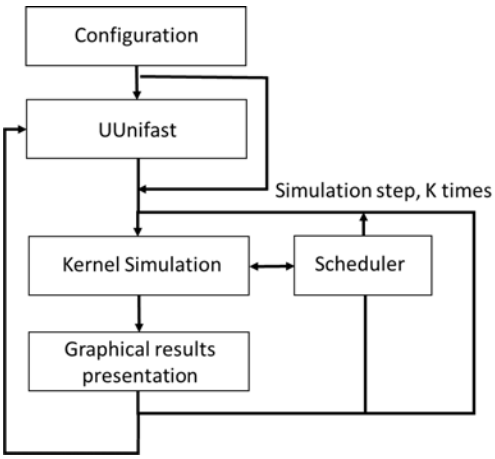


Fig. 1. Framework Architecture

The set of periodic tasks $\mathcal{T} = \{\tau_1, ..., \tau_n\}$ are executed on a set of identical processors $\mathcal{P} = \{CPU_1, \ldots, CPU_m\}$ with an homogeneous clock frequency $F \in \mathcal{F} = [F_1, ..., F_{max}]$. The hyper-period is defined as the period equal to the least common multiple of periods $H = lcm(\omega_1, \omega_2, \ldots, \omega_n)$ of the n periodic tasks.

A task τ_i executed on a processor at frequency F, requires $c_i = \frac{cc_i}{F}$ processor time at every ω_i interval. The system utilization is defined as the fraction of time during which the processor is busy running the task set i.e., $U = \sum_{i=1}^{n} \frac{c_i}{\omega_i}$. Herein we consider that the execution of real-time tasks in the system is preemptable.

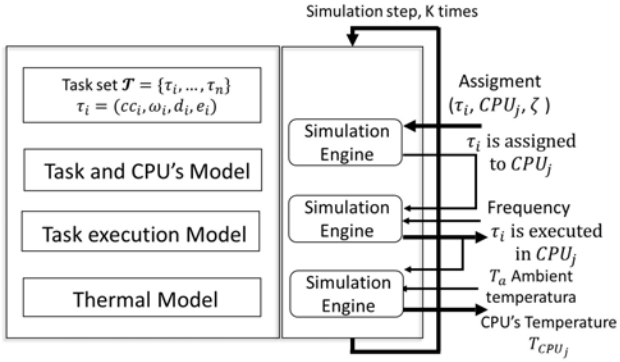


Fig. 2. Kernel of the simulator

3 Framework Architecture

The simulation framework has been programmed in MATLAB R2018a© ([20]) . It is distributed as open source software *as-is* ([14]). Its modular design provides flexibility to test a wide variety of schedulers and platforms. It includes a signal routing interface allowing switching among different user-defined scheduling algorithms.

This framework makes easier to evaluate a large number of different scenarios, where the platform (hardware), the set of tasks, and schedulers can be defined by the user through a Graphical User Interface (GUI).

Fig. 1 shows the main modules of the framework: Configuration, UUnifast, Kernel Simulation, Scheduler and Results. First, the user introduces the set of tasks, platform, and the scheduler in the *configuration* module. After the completion of the configuration stage, the simulation is executed. Later the results are presented to the user. The following subsections describe these modules.

3.1 Configuration Module

This module allows the introduction of all the information required by the framework. It is organized in four sections: a) Task definition, b) CPU definition, c) Thermal definition, and d) Scheduler definition. The order in which the information is introduced is irrelevant. The user can resort to default values or turn-off some sections, like the thermal definition.

a) The Task definition section allows two different ways to introduce the information. One way is to manually enter the number of tasks along with their parameters. Another way is to let the algorithm UUniFast ([4]) automatically generate a task set with the desired characteristics.

b) The CPU definition section requires two parameters: the number of CPUs and their frequency scale. The frequency scale is a set of normalized frequencies at which the platform could operate, where 1 indicates the highest frequency. The framework assumes homogeneous CPUs, a feature that will be relaxed in future releases.

c) The thermal definition section requires the Printed Circuit Board (PCB) and CPU dimensions as in Fig(3). Also requires the isotropic thermal properties: density, specif heat capacity, thermal conductivity coefficient, as well as the ambient temperature and the maximum operating temperature.

d) The Scheduler definition section is generic. The user either, can select one from a set of pre-programmed schedulers, or can define his/her own scheduler.

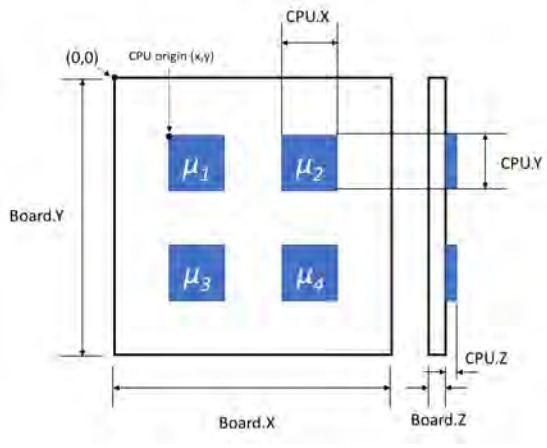


Fig. 3. PCB reference for thermal definition

The user should consider signals like CPU temperature, system utilization, energy consumption, (or a subset of them) to design her/his own scheduler. At every time step, the section generates a matrix of size *Number of tasks* $\times$ *Number of CPUs*, where the $ij-th$ entry represents the allocation of the $i-th$ task to the $j-th$ CPU.

3.2 UUniFast Submodule

The user can opt for the UUniFast algorithm ([4]) to generate the task set in the configuration stage, indicating the number of tasks to be generated, the system utilization U and a range for the task periods. The output is a feasible real-time task set with random task periods, WCETs, deadlines and consumed energy. UUniFast generates one set of

tasks at a time. It allows to stress the scheduler under analysis with different set of tasks.

3.3 Kernel Simulation Module

The *Kernel* module builds up a global simulation model according to the task set, CPUs, thermal and energy parameters and the selected scheduler, and runs the simulation.

The model represents task, CPU and thermal modules by a set of ordinary differential equations, and generates the signals to/from the scheduler. The scheduler can be represented either as a continuous or a discrete system. Accordingly, the scheduler can be modelled by the paradigm of differential equations or finite automata.

Next subsections describe how to build module's models. Later, we explain how the modules are merged into a global model.

3.3.1 Task Arrival and CPU's Submodule

The TCPN model representing the Task arrival and CPU's Module (Fig. 4) was first introduced in [17] and evaluated in [16]. Here we present a brief explanation of the TCPN model and the differential equations that represent the behaviour. The task module is composed by places p_i^w, p_i^{cc} and p_i^d and transitions t_i^w. Places p_i^w, p_i^{cc} and p_i^d represent that task τ_i belongs to the set of tasks, the CPU cycles of task τ_i that are arriving to the system, and the deadline of τ_i, respectively. $\lambda_i^w = \frac{1}{\omega_i}$ represents the arriving rate of task τ_i.

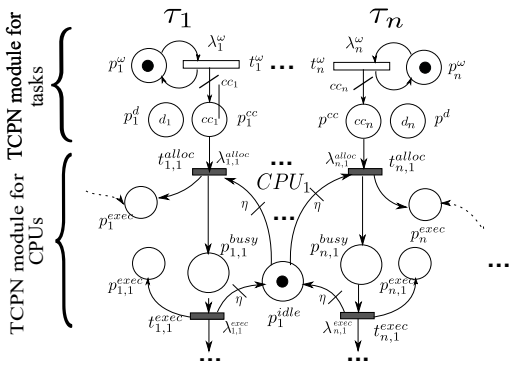


Fig. 4. Task and CPU TCPN module

The CPU module is composed of places $p_{i,j}^{busy}$, and p_j^{idle}, and transitions $t_{i,j}^{alloc}$, and $t_{i,j}^{exec}$. The marking in place $p_{i,j}^{busy}$ represents the amount of task τ_i that was allocated to CPU_j. The marking in place p_j^{idle} represents that CPU_j is idle. The firing of transition $t_{i,j}^{alloc}$ represents that task τ_i is being allocated to CPU_j, and the firing of transition $t_{i,j}^{exec}$ represents that task τ_i is being executed by CPU_j.

The differential Eqs. (4) and (5) representing the behaviour of the TCPN in Fig. (4) can be derived considering the following four vectors as the marking and transition vectors, respectively, of the task module, and the marking and transition vectors, respectively, of the CPU module:

$$\boldsymbol{m_\mathcal{T}} = [\boldsymbol{m_\mathcal{T}(p_1^w)},\ \boldsymbol{m_\mathcal{T}(p_1^{cc})},\ \boldsymbol{m_\mathcal{T}(p_1^d)}, ...,\ \boldsymbol{m_\mathcal{T}(p_n^w)},\ \boldsymbol{m_\mathcal{T}(p_n^{cc})},\ \boldsymbol{m_\mathcal{T}(p_n^d)}]^T,$$

$$\boldsymbol{T_\mathcal{T}} = [\boldsymbol{t_1^w},\ ...,\ \boldsymbol{t_n^w}]^T,$$

and

$$\boldsymbol{m_\mathcal{P}} = [\boldsymbol{m_\mathcal{P}(p_{1,1}^{busy})},\ \boldsymbol{m_\mathcal{P}(p_1^{idle})},\ ...,\ \boldsymbol{m_\mathcal{P}(p_{n,m}^{busy})},\ \boldsymbol{m_\mathcal{P}(p_m^{idle})}]^T,$$

$$\boldsymbol{T_\mathcal{T}} = [\boldsymbol{t_{1,1}^{alloc}}\ \boldsymbol{t_{1,1}^{exec}},\ ...,\ \boldsymbol{t_{n,m}^{alloc}},\ \boldsymbol{t_{n,m}^{exec}}]^T.$$

Eq. (4) models task arrival. $\boldsymbol{m_\mathcal{T}}$ describes how task are arriving to the system over time, and $\boldsymbol{w^{alloc}} = [{}_1w_1^{alloc}, \ldots, {}_1w_n^{alloc}, \ldots, {}_mw_n^{alloc}]$ represents the allocation of tasks to CPUs. And Eq. (5) describes how tasks are allocated to CPUs, where $\boldsymbol{m_\mathcal{P}}$ represents the reservation of CPUs to the allocated tasks. It is very important to realize that signal $\boldsymbol{w^{alloc}}$ must be computed by the scheduler. This signal is an input to the system and indicates when a task must be allocated to a CPU:

$$\dot{\boldsymbol{m}}_\mathcal{T} = \boldsymbol{C_\mathcal{T}\Lambda_\mathcal{T}\Pi_\mathcal{T}(m)m_\mathcal{T}} - \boldsymbol{C_\mathcal{T}^{alloc}w^{alloc}}, \quad (4)$$

$$\dot{\boldsymbol{m}}_\mathcal{P} = \boldsymbol{C_\mathcal{P}\Lambda_\mathcal{P}\Pi_\mathcal{P}(m)m_\mathcal{P}} + \boldsymbol{C_\mathcal{P}^{alloc}w^{alloc}}. \quad (5)$$

3.3.2 Task Execution Module

The task execution module is represented by places $p_{i,j}^{exec}$ depicted in Fig. 4. Considering the vector:

$$\boldsymbol{m_{exec}} = [m_{exec}(p_{1,1}^{exec}),\ ...,\ m_{exec}(p_{n,m}^{exec})]^T,$$

representing the marking of the places of the task execution module (for easily representation $m_{i,j}^{exec} = m_{exec}(p_{1,1}^{exec})$), then the task execution behavior is represented by Eq. (6):

$$\dot{\boldsymbol{m}}_{\boldsymbol{exec}} = \boldsymbol{A_{\mathcal{P}}} \boldsymbol{m_{A_{\mathcal{P}}}}, \tag{6}$$

where $\boldsymbol{A_{\mathcal{P}}}$ is built from $\boldsymbol{C_{\mathcal{P}}} \boldsymbol{\Lambda_{\mathcal{P}}} \boldsymbol{\Pi_{\mathcal{P}}}(\boldsymbol{m}) \boldsymbol{m_{\mathcal{P}}}$ considering only rows corresponding to places $p_{i,j}^{busy}$ and columns corresponding to transitions $t_{i,j}^{exec}$; other rows and columns are discarded. Marking $\boldsymbol{m_{A_{\mathcal{P}}}}$ considers the marking in places $p_{i,j}^{busy}$; other markings are discarded.

The marking of these places represent the amount of task τ_i that is executed by CPU_j. This signal is available for any scheduler.

3.3.3 Thermal Module

This work considers the thermal model presented in [17], the model was evaluated under comparison with simulations in ASYS ®. This model rewrites the thermal partial differential equation by a set of ordinary thermal differential equations. It is as precise as a Finite Element approach and has the advantage that a state model is derived from the analysis. It also avoids the calibration stages of RC thermal approaches, and only requires the isotropic thermal properties of the materials: density, specif heat capacity and thermal conductivity coefficient.

In this section we present a brief explanation, for a deeper insight please refer to [17] and [16]. The thermal module is composed of several thermal submodules, representing thermal conduction, convection and heat generation. The arc from transition $t_{1,1}^{exec}$ to place $p_1^{com_1}$ represents heat generation due to task execution.

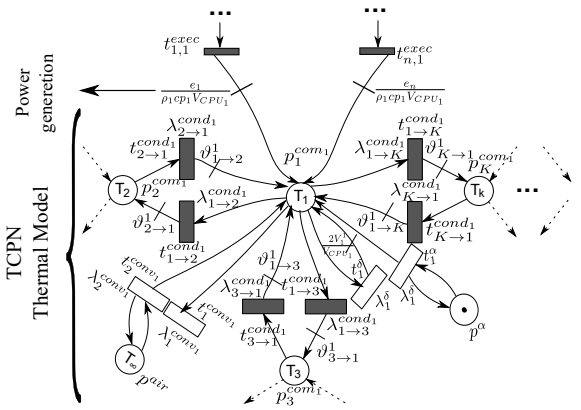


Fig. 5. Thermal module

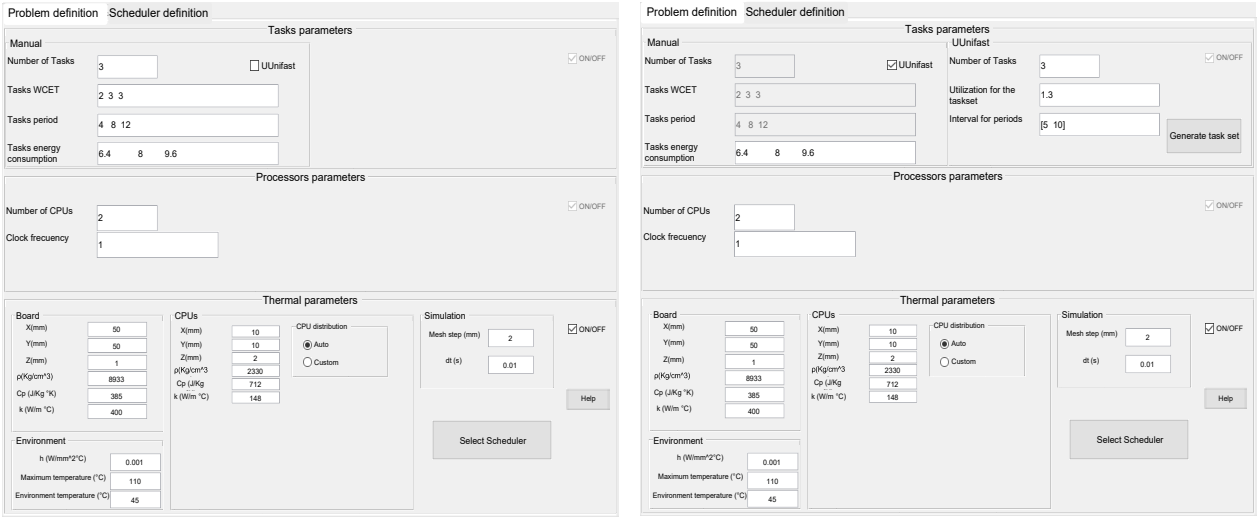
Fig. 5 depicts the thermal module. Following the numbering of places and transitions, the temperature behaviour of the system is represented by Eq. (7a):

$$\begin{aligned} \dot{\boldsymbol{m}}_{\boldsymbol{T}} =& \boldsymbol{C_T} \boldsymbol{\Lambda_T} \boldsymbol{\Pi_T}(\boldsymbol{m}) \boldsymbol{m_T} \\ &+ \boldsymbol{C_a} \boldsymbol{\Lambda_a} \boldsymbol{\Pi_a}(\boldsymbol{m}) \boldsymbol{m_a} + \boldsymbol{C_{\mathcal{P}}^{exec}} \boldsymbol{f^{exec}}, \end{aligned} \tag{7a}$$

$$\dot{\boldsymbol{m}}_{\boldsymbol{a}} = 0. \tag{7b}$$

$\boldsymbol{m_T}$ is the distribution of temperature over the system elements, $\boldsymbol{m_a}$ is the ambient temperature and it is considered constant, and $\boldsymbol{f^{exec}}$ is a variable depending on task allocation and the frequency at which CPUs are executing tasks. CPU temperature is available for scheduling purposes. Temperature depends on the task execution and frequency. In steady state, this is tantamount to say that temperature depends on task allocation ${}_j w_i^{alloc}$ and frequency. As mentioned above, these parameters are coded in $\boldsymbol{f^{exec}}$.

At each simulation step, the *Thermal model* reads the new system state and the ambient temperature to compute the new CPU temperature. The main output signals of the Kernel simulator are the task execution vector ($\boldsymbol{m_{exec}}$) and the CPUs temperature ($\boldsymbol{m_T}$) at each simulation step. The input (output) signals are received (sent) to the *Scheduler* module in order to obtain a feedback signal for the next step.



a) b)

Fig. 6. Main tab GUI. a) Manual mode, b) UUnifast mode

3.4 Scheduler Module

The scheduler module allows to select, at configuration time, one of the scheduling policies available in the framework, or any scheduler defined by the user. The TCPN The signals available to the scheduler from other modules are $m_{i,j}^{exec}$ representing the amount of τ_i executed by CPU_j, and m_{T_j} the CPU_j temperature. The signals that other modules require from the scheduler are the task allocations signals ${}_j w_i^{alloc}$.

The scheduling policies available in the framework are an RT Global Earliest Deadline First (*G-EDF*) ([1]), an RT fluid scheduler(*RT-TCPN*)([15]), and an RT thermal-aware fluid scheduler (*RT-TCPN* Thermal aware) for Dynamic Priority Systems (DPS) ([16])). The user can define a new scheduler as a continuous or discrete scheduler (Sec.5).

3.4.1 Custom Scheduler

This section describes how to implement a custom schedule. Algorithm 1 provides the executive cycle of the simulator. It includes the scheduler (user defined or pre-programmed one) in line 4.

The main input signals, from the simulation environment, that the user might use, are the CPU temperature, executed tasks, current CPU frequency and consumed energy, also the secondary signals $\boldsymbol{m}^{busy}$ (the CPU state), and all task and CPU parameters are available for scheduling purposes. The output signals that the scheduler module must deliver to the simulation engine are the $\boldsymbol{w}^{\boldsymbol{alloc}}$ and frequency. It is not needed that all the input/output signals be used by the scheduler; the only mandatory signal that the scheduler must deliver to the system is $\boldsymbol{w}^{\boldsymbol{alloc}}$. For presentation purposes, this subsection renames the variables $\boldsymbol{x_1} = \boldsymbol{Y_T}$ CPU temperature, $\boldsymbol{x_2} = \boldsymbol{m_{exec}}$ executed tasks, $\boldsymbol{x_3} = \boldsymbol{F}$ current CPU frequency and also the secondary signals $x_4 = \boldsymbol{m}^{busy}$.

If the user describes the scheduler as a continuous one, then he/she must write the signal walloc as:

$$\dot{\boldsymbol{w}}^{alloc} = \sum_i \boldsymbol{A_i x_i}.$$

If the user describes the scheduler as a discrete one, then he/she must write the signal $\boldsymbol{w}^{\boldsymbol{alloc}}$ as:

$$\boldsymbol{w}^{\boldsymbol{alloc}}(\zeta) = \Phi(\boldsymbol{x_1}, ..., \boldsymbol{x_4}).$$

In both cases, the functions represented by the matrices $\boldsymbol{A_i}$ or function Φ, are computed based on the scheduler objectives and task and CPU

parameters, as well as the current state of the system. It is the user responsibility the design of such a functions.

Notice that the simulator is time driven, thus, although in the discrete case the $\boldsymbol{w^{alloc}}$ signal depends on the event arrival, it is represented as a function of time the make it compatible with the whole simulation.

Finally, once the scheduler is defined, it is integrated to the simulation engine and used at each simulation step.

ALGORITHM 1: Simulation algorithm

Input: The TCPN model, the set of tasks $\mathcal{T}$, the hyperperiod H.
Output: The schedule $\boldsymbol{w^{alloc}} = [{}_1w_1^{alloc}, ..., {}_jw_i^{alloc}, ..., {}_mw_n^{alloc}]$

Initialize $\zeta = 0$, ${}_jw_i^{alloc} = 0$;
while $\zeta \leq H$ **do**
  Simulate the TCPN model from ζ *to* $\zeta + step$ *with* ${}_jw_i^{alloc} = 0$ *as an input;* `/* Solve Eqs. (6) to compute` m^{exec} `*/`
  Obtain the schedule ${}_jw_i^{alloc}$ *from the custom scheduling policy*
  $\zeta = \zeta + step$; `/* Update time */`
end

3.5 Building the Global Model

A full system in the simulation framework consists of a set of tasks, a set of CPUs on a platform, and a scheduler (as an input), represented by separated models. In order to simulate a full system, these models must be gathered into a global model. In the case of a continuous scheduler, the global model is simulated by solving the system:

$$\begin{aligned} \dot{\boldsymbol{M}} &= \boldsymbol{A}\boldsymbol{M} + \boldsymbol{B}\boldsymbol{w^{alloc}} + \boldsymbol{B'}\boldsymbol{m_a}, \\ \boldsymbol{Y} &= \boldsymbol{S}\boldsymbol{M}, \end{aligned} \tag{8}$$

where $\boldsymbol{M} = [\boldsymbol{m_\mathcal{T}}, \boldsymbol{m_\mathcal{P}}, \boldsymbol{m_T}]^T$, and the matrices are:

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{C_\mathcal{T}}\boldsymbol{\Lambda_\mathcal{T}}\boldsymbol{\Pi_\mathcal{T}} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{C_\mathcal{P}}\boldsymbol{\Lambda_\mathcal{P}}\boldsymbol{\Pi_\mathcal{P}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{C_\mathcal{P}^{exec}}\boldsymbol{\Lambda^{exec}}\boldsymbol{\Pi^{exec}} & \boldsymbol{C_T}\boldsymbol{\Lambda_T}\boldsymbol{\Pi_T} \end{bmatrix}, \tag{9}$$

$$\boldsymbol{B} = \begin{bmatrix} \boldsymbol{C_\mathcal{T}^{alloc}} \\ \boldsymbol{C_\mathcal{P}^{alloc}} \\ \boldsymbol{0} \end{bmatrix} \quad \boldsymbol{B'} = \begin{bmatrix} \boldsymbol{0} \\ \boldsymbol{0} \\ \boldsymbol{C_a}\boldsymbol{\Lambda_a}\boldsymbol{\Pi_a} \end{bmatrix}, \tag{10}$$

$$\boldsymbol{S} = \begin{bmatrix} \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{A_\mathcal{P}} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{S_T} \end{bmatrix}. \tag{11}$$

$\boldsymbol{A_\mathcal{P}}$ correspond to the output matrix for task execution and $\boldsymbol{S_T}$ represents the temperature output matrix. Thus the output vector $\boldsymbol{Y} = [\boldsymbol{m_{exec}}, \boldsymbol{m_T}]$ contains the task execution and the temperature of each processor. If the thermal module is not selected for simulation the global model is slightly different: vector $\boldsymbol{M}$ will only contain $\boldsymbol{M} = [\boldsymbol{m_\mathcal{T}}, \boldsymbol{m_\mathcal{P}}]$, every matrix (Eq.9 - 11) will lose its last row, and Eq.(9) will also lose its last column, and the output vector $\boldsymbol{Y} = [\boldsymbol{m_{exec}}]$ will only contain the task execution.

3.6 Results Module

After a simulation run, the Results module generates plots showing the allocation and execution of jobs to CPUs and the CPUs temperature evolution, as in Fig. 10 and Fig. 11, by using the tools and functions contained in MATLAB R2018a©(ie. Heat maps).

4 User Interface

Fig. 6 shows the main window of the simulator GUI. There are three areas, which contain information about *Tasks parameters*, *Processors parameters* and *Thermal parameters*.

The user can manually define the parameters of the tasks set in the area entitled *Tasks parameters*. The parameters to be defined are the number of tasks, and the value of the WCET, task period and task energy consumption per each task (see Fig. 6 a)). Alternatively, a random set of tasks can be configured by checking the UUniFast check box (Fig. 6 b)), setting the number of tasks, the utilization of the task set and the interval of periods, and then clicking the button *Generate task set*.

The area entitled *Processors parameters* allows to set the number of CPUS and the homogeneous

clock frequency. Last, the area *Thermal parameters* consists of three subareas. The first two correspond to the geometry (length X, height Y, and width Z measurements) and material properties (thermal conductivity k coefficients, density ρ, specific heat capacities C_p) of the board and CPUs.

The third subarea is for entering the mesh geometry (Mesh step) and the accuracy (dt) of the solutions obtained by the TCPN Thermal model ([17]). Once these parameters have been set, the next step is to select an scheduling policy by clicking the button *Select Scheduler* in the *Scheduler selection* tab (Fig. 7). The available scheduling policies appear in a selection list. User-defined scheduling policies will show up in this list too (Fig. 7 b)) by following the installation guidelines. A scheduler framework shows the structure of the scheduler for each selected scheduler policy.

The last step consists in clicking the button *Compute Schedule* to run the simulation. After a simulation run, the GUI allows to generates plots of the allocation and execution of task's jobs to CPUs and the CPUs temperature evolution. Fig. 7 b) shows the available buttons are: *Save data*, which save all the simulation data into a file, *Plot*, which plots task execution and CPU temperature evolution, and *Heat map*, which is only functional if the user configured the thermal parameters before the simulation.

5 Available Schedulers

This section is intended to show the available schedulers and their implementation in the framework. We only provide global multiprocessor schedulers by the reasons explained in Section 2. The scheduling policies available in the framework are an G-EDF ([1]), an RT fluid scheduler RT-TCPN ([15]), and an *RT-TCPN*Thermal aware scheduler that are describing below.

5.1 G-EDF

This scheduler implements a global EDF (G-EDF) algorithm. It is a global job-level fixed priority scheduling algorithm for sporadic task systems, which is optimal for implicit-deadline tasks with regard to soft RT constraints [2]. Jobs are allocated to CPUs from a single queue. The highest priority is assigned to the job with the earliest absolute deadline. The signal ${}_j w_i^{alloc}$ must be discrete. At each simulation step the scheduler can be written as: $\boldsymbol{w^{alloc}} = \Phi(\boldsymbol{m_{exec}})$. According to the EDF algorithm, the scheduling events are task activations and task completions, the only points at which task preemption can occur. The discrete function $\Phi(\boldsymbol{m_{exec}})$ is implemented by algorithm 2.

ALGORITHM 2: G-EDF

Input: $TCPN$ model, $\mathcal{T}$, the hyperperiod H.
Output: Schedule $\boldsymbol{w^{alloc}} = [{}_1 w_1^{alloc}, ..., {}_j w_i^{alloc}, ..., {}_m w_n^{alloc}]$

Initialize $i = 1$, $sd = sd_i$, $\zeta = 0$, $\forall \tau_i \in \mathcal{T}$ and $\forall CPU_j \in \mathcal{P}$;
while $\zeta \leq H$ **do**
  obtain the highest priority task τ_i using the EDF priority based on $m_{i,j}^{exec}(\zeta)$
  Set ${}_j w_i^{alloc} = 1$; `/* Assign the highest priority task` τ_i `to` CPU_j `*/`
  Simulate the $TCPN$ model from ζ to $\zeta + step$;
  `/* Solve Eqs. (6) to compute` $\boldsymbol{m_{exec}}$ `*/`
  $\zeta = \zeta + step$; `/* Update time */`
end

5.2 RT-TCPN

RT-TCPN is a scheduler based on the TCPN model presented in [15]. It is composed of a global fluid scheduler and the discretization of the fluid scheduler. The fluid schedule computation is limited to every deadline, whereas preemption and context switch occur at every quantum, at which we check the difference between the actual and the expected fluid execution. *RT-TCPN* obtains a discrete schedule that closely tracks the fluid one ($\boldsymbol{m_{exec}}$) by computing a schedule up to the hyperperiod ([3]).

The algorithm considers the ordered set of all tasks' jobs deadlines to define scheduling intervals, as in deadline partitioning ([18]). Each task τ_i

a) b)

Fig. 7. Scheduler selection tab GUI

must be executed $n_i = \frac{H}{\omega_i}$ times within the hyperperiod H. Thus every $q * \omega_i$, where $q = 1, ..., n_i$ is a deadline that must be considered in the analysis. These deadlines can be gathered and ordered in the set $SD_i = \{sd_i^1, ..., sd_i^{n_i}\}$. A general set of deadlines is defined as $SD = SD_0 \cup ... \cup SD_{|\mathcal{T}|}$ where $SD_0 = \{0\}$. The elements of SD can be arranged in ascendant order and renamed as $SD = \{sd_0, ..., sd_\alpha\}$, where α is the last deadline. We define the the quantum Q as in [15]. The fluid execution $\boldsymbol{FSC} = [{}_1FSC_{\tau_1}, ..., {}_jFSC_{\tau_i}, ..., {}_mFSC_{\tau_n}]$ is a vector with *number of tasks* $\times$ *number of CPUs* entries. The signal ${}_jFSC_{\tau_i}$ stands for the desired allocation of the $i-th$ task to the $j-th$ CPU at time ζ. This function can be computed as:

$$ {}_jFSC_{\tau_i}(\zeta) = \frac{{}_j\beta_i \times cc_i}{H}(\zeta), \tag{12}$$

where ${}_j\beta_i$ is an unknown parameter that represents the number of jobs of task τ_i which are assigned to CPU_j per time unit. This value is used to compute a distributed fluid schedule function that considers temporal constraints, according to an offline stage that solves the following linear programming problem:

$$\begin{aligned} &min \sum_{j=1}^{m} \sum_{\tau_a \in \mathcal{T}} {}_j\beta_a, \\ &s.t. \\ &\quad \left. \sum_{j=1}^{m} {}_j\beta_n = \frac{H}{\omega_n} \quad \forall i = 1, \ldots, n, \right\}, \\ &\quad \left. \sum_{\tau_a \in \mathcal{T}} \frac{cc_a \times {}_m\beta_a}{H} \leq 1 \quad \forall j = 1, \ldots, m \right\}. \end{aligned} \tag{13}$$

The actual execution $m_{i,j}^{exec}$ of τ_i in CPU_j must be equal to ${}_jFSC_{\tau_i}$. In the on-line stage a sliding mode controller yields an output signal ${}_jw_i^{alloc}$ that is proportional to the error (difference between ${}_jFSC_{\tau_i}(\zeta)$ and $m_{i,j}^{exec}$). The formal proof of this controller is given in [15]. The scheduler $({}_jw_i^{alloc})$ meets all deadlines assuming an infinitesimal share of the CPUs because its fluid nature, ans therefore must be discretized. The discrete scheduler must be written as: $\boldsymbol{W^{alloc}} = \Phi(\boldsymbol{m_{exec}}, \boldsymbol{FSC})$, where $\Phi(\boldsymbol{m_{exec}}, \boldsymbol{FSC})$ is described by Algorithm 3. The computed discrete schedule $\boldsymbol{W^{alloc}}$ matches the fluid schedule at every deadline $sd_k \in SD$. Since these time points are the deadlines of jobs, then the algorithm ensures that the discrete schedule meets all deadlines of all tasks.

Fig. 8 depicts the scheme of the algorithm implemented in this framework. The dotted box A contains a set of signals that represent the normal behavior of the system. Signal A.1 describes the function ${}_jFSC_{\tau_i}(\zeta)$ obtained in the off-line stage, and is used as the reference for the on-line stage. In the on-line stage the controller yields an output signal ($ {}_jw_i^{alloc}$, named A.2 in Figure 8). The controller output is integrated by the TCPN model. Signal A.3 describes the output of this model. Finally, the algorithm *RT-TCPN* uses the difference between $m_{i,j}^{exec}$ (the expected executed amount of τ_i in CPU_j) and $M_{i,j}^{exec}$ (the actual executed amount of τ_i in CPU_j) to compute ${}_jW_i^{alloc}$, i.e. the on-line allocation of τ_i to CPU_j at every quantum Q.

5.3 RT-TCPN Thermal Aware

The schedule computation in this algorithm encompasses temporal and thermal constraints. The temperature of the chip must be kept under

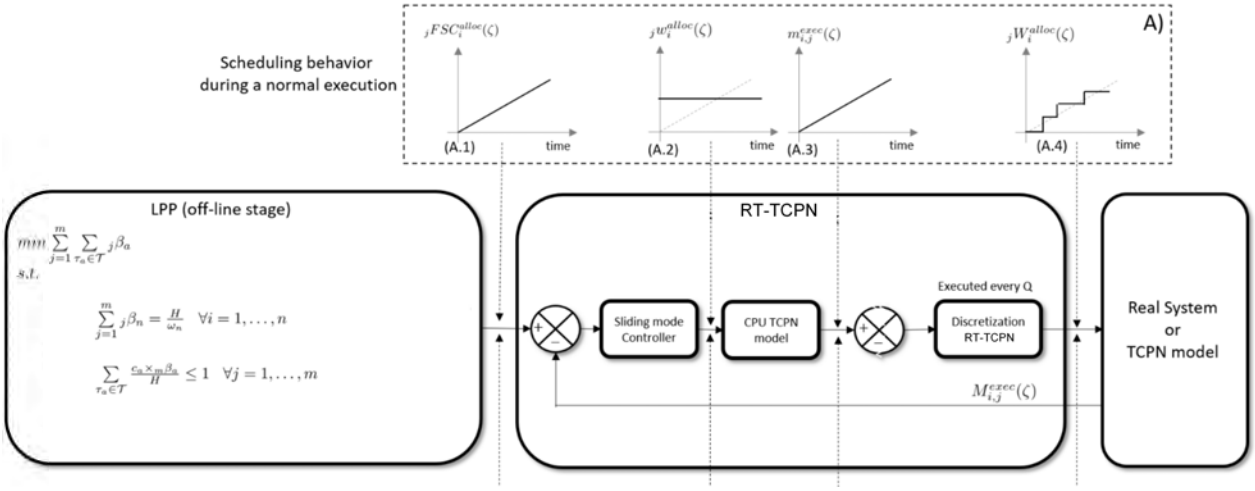


Fig. 8. RT-TCPN Scheme

ALGORITHM 3: RT-TCPN

Input: $\mathcal{T}$, the ordered set SD. The quantum Q. The fluid schedule ${}_j\boldsymbol{FSC_{\tau_i}}$.
Output: The discrete schedule $\boldsymbol{W^{alloc}} = [{}_1W_1^{alloc}, ..., {}_jW_i^{alloc}, ..., {}_mW_n^{alloc}]$

Initialize $i = 1$, $sd = sd_i$, $\zeta = 0$, $M_{i,j}^{exec}(\zeta) = 0\ \forall \tau_i \in \mathcal{T}$ and $\forall CPU_j \in \mathcal{P}$;
for $\zeta \leq H$ **do**
  All tasks are preempted from the processors;
  $RE_{i,j}(\zeta) = {}_jFSC_{\tau_i}(sd) - M_{i,j}^{exec}(\zeta)$; `/* Compute remaining jobs */`
  $ET_j(\zeta) = \{\tau_i | RE_{i,j}(\zeta) > 0 \forall CPU_j \in \mathcal{P}\}$;
  `/* Compute the set of tasks to be executed */`
  $PR_{i,j}(\zeta) = m_{i,j}^{exec}(\zeta) - M_{i,j}^{exec}(\zeta)$; `/* Compute the priority for every task` τ_i `in` $ET_j(\zeta)$ `*/`
  for $j = 1$ *to* m **do**
    ${}_j\omega_a\ 0, 1 \leq j \leq m,\ 1 \leq i \leq a$;
    Select the task τ_a for CPU_j with the highest priority value in $ET_j(\zeta)$;
    ${}_jW_a^{alloc} = 1$;
    Remove the task τ_a from $ET_k(\zeta)$ for all $1 \leq k \leq m$ and $k \neq j$;
    $M_{a,j}^{exec}(\zeta + Q) = M_{a,j}^{exec}(\zeta) + Q \times {}_j\omega_a$ Remove τ_a from ET_j ;
  end
  ; `/* Solve Eqs. (6) to compute` m^{exec} `*/`
  $\zeta = \zeta + Q$; `/* Update time */`
  if $\zeta == sd$ **then**
    $i = i + 1, sd = sd_i$
  end
end

a temperature threshold $\boldsymbol{T_{max}}$ that depends on system design requirements. The fluid schedule function introduced in [3] is extended to include thermal restrictions. This is achieved by considering the steady state of Eq. 7a, i.e. if the schedule is periodic, fluid and evenly distributed over the hyperperiod, then the temperature must reach a steady state. Thermal Eqs. (7a and 7b) can be rewritten as:

$$\begin{aligned} \dot{\boldsymbol{M}}_{\boldsymbol{T}} &= \boldsymbol{A_T M_T} + \boldsymbol{B_T w^{alloc}} + \boldsymbol{B'_T m_a}, \\ \boldsymbol{Y_T} &= \boldsymbol{S' M_T}, \end{aligned} \tag{14}$$

$\boldsymbol{A_T}$ corresponds to the system matrix, $\boldsymbol{B_T}$ is the input matrix, and $\boldsymbol{B'_T}$ conforms the matrix associated to the ambient temperature (m_a which is considered constant). These matrices are:

$$\boldsymbol{A_T} = \begin{bmatrix} \boldsymbol{C_T \Lambda_T \Pi_T} & \boldsymbol{C_{\mathcal{P}}^{exec} \Lambda^{exec} \Pi^{exec}} \\ \mathbf{0} & \boldsymbol{C_{\mathcal{P}} \Lambda_{\mathcal{P}} \Pi_{\mathcal{P}}}, \end{bmatrix} \tag{15}$$

$$\boldsymbol{B'_T} = \begin{bmatrix} \boldsymbol{C_a \Lambda_a \Pi_a} \\ \mathbf{0}, \end{bmatrix} \quad \boldsymbol{B_T} = \begin{bmatrix} \mathbf{0} \\ \boldsymbol{C_{\mathcal{P}}^{alloc}}. \end{bmatrix} \tag{16}$$

Thus, in a thermal steady state $\dot{\boldsymbol{M}}_{\boldsymbol{T}} = \mathbf{0}$, $\boldsymbol{M_T}$ and $\boldsymbol{Y_T}$ are respectively renamed as $\boldsymbol{M_{T_{ss}}}$ and $\boldsymbol{Y_{T_{ss}}}$. indicating a system steady state temperature, which can be computed as follows:

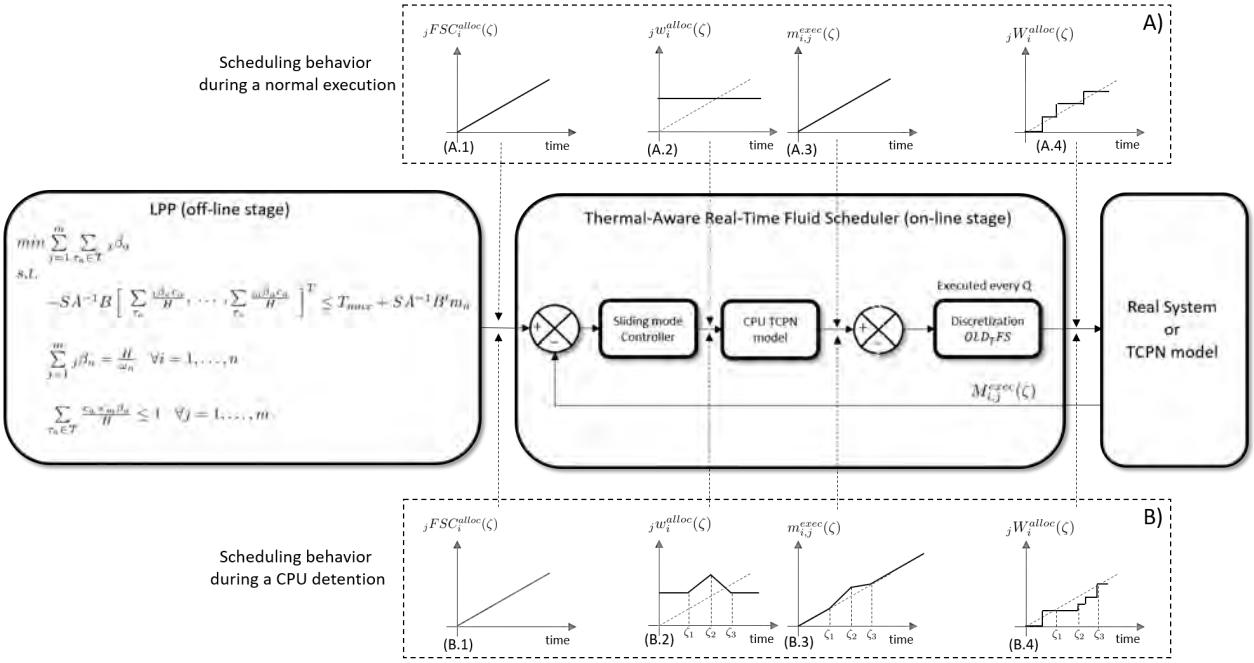


Fig. 9. RT-TCPN Thermal-Aware Scheme

$$\begin{aligned} \boldsymbol{M_{T_{ss}}} &= -\boldsymbol{A_T^{-1}}(\boldsymbol{B_T w^{alloc}} + \boldsymbol{B'_T m_a}), \\ \boldsymbol{Y_{T_{ss}}} &= \boldsymbol{S' M_{T_{ss}}}. \end{aligned} \tag{17}$$

The steady state temperature $Y_{T_{ss}}[k]$ of CPU_k must be less than or equal to its maximum temperature level i.e., $Y_{T_{ss}}[k] \leq T_{max_k}$ so as not to violate the thermal constraint. In a vectorial form: $\boldsymbol{S' M_{T_{ss}} \leq T_{max}}$.

The thermal constraint is derived by combining the last expression and Eq. (17).

$$-\boldsymbol{S' A_T^{-1} B_T w^{alloc}} \leq \boldsymbol{T_{max}} + \boldsymbol{S' A_T^{-1} B'_T m_a}. \tag{18}$$

The task allocation vector $\boldsymbol{w^{alloc}}$ depends on the unknown parameter ${}_j\beta_i$. These parameters are used to compute the distributed fluid schedule function that considers thermal and temporal constraints, according to the following linear programming problem:

$$\begin{aligned} &min \sum_{j=1}^{m} \sum_{\tau_a \in \mathcal{T}} {}_j\beta_a \\ &s.t., \\ &\left. \begin{aligned} &-\boldsymbol{S A_T^{-1} B_T} \left[\sum_{\tau_a} \tfrac{1\beta_a cc_a}{H}, \cdots, \sum_{\tau_a} \tfrac{m\beta_a cc_a}{H} \right]^T \leq \\ &\boldsymbol{T_{max}} + \boldsymbol{S A_T^{-1} B'_T m_a} \end{aligned} \right\}, \\ &\left. \sum_{j=1}^{m} {}_j\beta_n = \tfrac{H}{\omega_n} \quad \forall i = 1, \ldots, n \right\}, \\ &\left. \sum_{\tau_a \in \mathcal{T}} \tfrac{cc_a \times m\beta_a}{H} \leq 1 \quad \forall j = 1, \ldots, m \right\}. \end{aligned} \tag{19}$$

The first constraint is the thermal constraint. The other two constraints are a straightforward extension of the time and CPU utilization used in the *RT-TCPN* algorithm for the multiprocessor case. The required fluid schedule function ${}_jFSC_{\tau_i}(\zeta)$ is defined similarly as Eq. (12). The temporal and thermal requirements are accomplished as long as the tasks are executed according to this function. Task execution is represented by variable $\boldsymbol{m_{i,j}^{exec}}(\zeta)$, hence the *task execution error* is defined as $e_{i,j}(\zeta) = {}_jFSC_{\tau_i}(\zeta) -$

$\boldsymbol{m}_{i,j}^{exec}(\zeta)$. If $e_{i,j}(\zeta) = 0$, then tasks are executed at the adequate rate, and the time and thermal constraints are met. Therefore, this error can be kept equal to zero by appropriately selecting ${}_jw_i^{alloc}(\zeta)$. We propose a sliding mode controller for this purpose. The formal proof for this controller is beyond the scope of this paper. We provide the following key hints, nonetheless.

Considered the sliding surface:

$$\mathcal{S}_{i,j}(\zeta) = \frac{K_1}{\lambda_{i,j}^{exec}} e_{i,j}(\zeta) + \frac{{}_j\beta_i \times cc_i}{H\lambda_{i,j}^{exec}} - \boldsymbol{m}_{i,j}^{busy}, \quad (20)$$

$${}_jw_i^{alloc}(\zeta) = {}_j\hat{w}_i^{alloc}(\zeta) + \frac{K_1}{\lambda_{i,j}^{exec}} \frac{{}_j\beta_i \times cc_i}{H}, \quad (21)$$

where ${}_j\hat{w}_i^{alloc}(\zeta) = K_2 sign(\mathcal{S}_{i,j}(\zeta))$ and $sign(x) = 1$ if $x \geq 0$; 0 otherwise.

Figure 9 shows how the implemented algorithm works. It is composed of the off-line and online stages. Based on an LPP, the off-line stage computes the functions ${}_jFSC_{\tau_i}(\zeta)$ to meet the temporal and thermal constraints. The on-line stage use these functions as the target for task execution. The sliding-mode controller continuously allocates tasks to CPUs to ensure that the task execution tracks the functions ${}_jFSC_{\tau_i}(\zeta)$. The dotted box A in figure 9 shows the set of signals that represent the normal behavior of the system. Box B depicts a disturbance, an unexpected behavior of the system such as a CPU detention. Signal B.1 describes the ${}_jFSC_{\tau_i}(\zeta)$, as in the normal case. If a CPU detention occurs at time ζ_1, then the difference between ${}_jFSC_{\tau_i}(\zeta)$ and the actual execution of τ_i in CPU_j starts to increase at this time. Thus the controller output signal (${}_jw_i^{alloc}$, named B.2) also starts to increase at time ζ_1. The controller output is integrated by the TCPN model (Eq. 8) producing an output $m_{i,j}^{exec}$ (signal B.3) that is greater than ${}_jFSC_{\tau_i}(\zeta)$. Finally, the algorithm *RT-TCPN* Thermal aware computes ${}_jW_i^{alloc}$, which also increases at time ζ_1. When the CPU resumes at time ζ_2, the controller allocates tasks to CPUs more often than in the normal case, using the CPU idle periods until normal operation is reached at time ζ_3. Approaches that do not include a continuous controller are not able to recover from CPU detentions or other disturbances that can temporarily stop task execution.

6 Example

We illustrate the usage of the simulation framework with the following experiments. First, we compare the temperature variations of the simulated system as generated by the available schedulers.

In the experiments we assume a platform composed of two homogeneous $1cm \times 1cm$ silicon microprocessors mounted over a $5cm \times 5cm$ copper heat spreader as in [17]. The thickness of the silicon microprocessors and the copper heat spreader are $0.5mm$ and $1mm$ respectively.

The experiment considers the task set $\mathcal{T} = \{\tau_1, \tau_2, \tau_3\}$. The maximum operating temperature of the cores is set at $T_{max_{1,2}} = 80^oC$. The temporal parameters of each task are its $WCET$ (in CPU cycles), period ω and deadline d (with $\omega = d$ in this case), resulting in $\tau_1 = (2 \times 10^9, 4, 4, 35)$, $\tau_2 = (3 \times 10^9, 8, 8, 40)$, $\tau_3 = (3 \times 10^9, 12, 12, 45)$, and the consumed energy e.

Fig. 10 depicts the temperature obtained by the schedulers for both CPUs. *G-EDF* and *RT-TCPN* obtain a feasible schedule, however the resulting temperatures violate the thermal constraint. In contrast, the *RT-TCPN* Thermal scheduler meets both thermal and temporal requirements. Fig. 11 presents the heat map obtained by the simulator.

7 Conclusion and Future Work

Designing, testing and comparing RT scheduling methods on multiprocessors is a gruesome and time consuming chore, all the more when thermal restrictions are considered. Resorting to a real system implementation is overkilling during the early design stages. We have developed a simulation framework which encompasses modules for defining and modelling tasks, CPUs, thermal properties and three global RT schedulers out-of-the box. It is available at https://www.gdl.cinvestav.mx/art/uploads/SchedulerFrameworkTCPN.zip and it is distributed as open source software *as-is*.

The main contribution compared with different real time simulation tools relies on its capability of handling temperature analysis over different scheduling policies, which can be very useful in order to detect thermal management problems

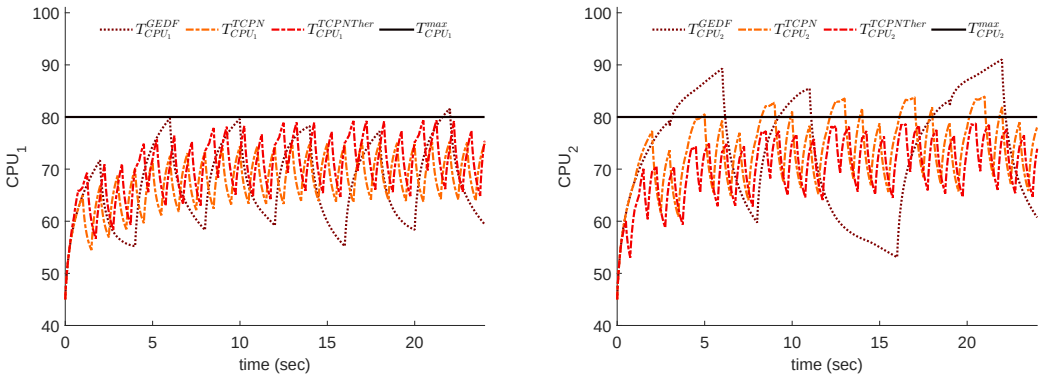


Fig. 10. Temperatures comparison obtained by the implemented schedulers

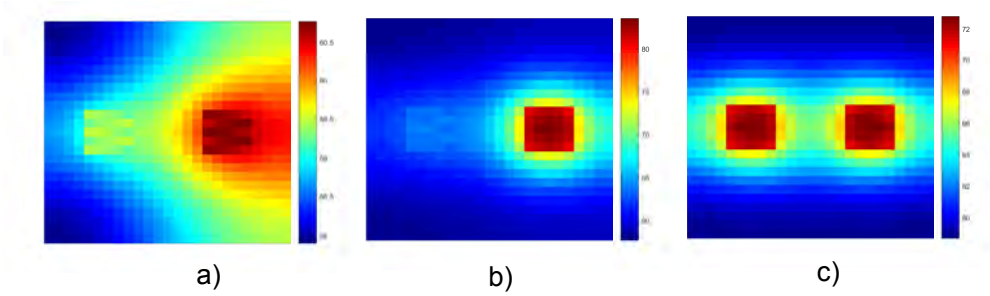


Fig. 11. Heat map obtained by a) *G-EDF*, b) *RT-TCPN* c) *RT-TCPN* Thermal Aware

that can be solved trough a correction in the scheduling algorithms.

We are working to include a number of improvements such as simpler procedures to replace or customize the thermal, task and CPUs models, additional algorithms to generate task sets and thermal-energy aware schedulers. Following the trend of some current multiprocessors, the framework will also include models for heterogeneous CPUs with per-CPU frequency adjustments.

Acknowledgements

This work was partially supported by grants TIN2016-76635-C2-1-R (AEI/FEDER, UE), gaZ: T48 research group (Aragón Gov. and European ESF), and HiPEAC4 (European H2020/687698).

References

1. **Baker, T. P. (2005).** A comparison of global and partitioned EDF schedulability tests for multiprocessors. *International Conf. on Real-Time and Network Systems*.
2. **Baruah, S., Bertogna, M., & Butazzo, G. (2015).** *Multiprocessor Scheduling for Real-Time Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.
3. **Baruah, S. K., Cohen, N. K., Plaxton, C. G., & Varvel, D. A. (1996).** Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, Vol. 15, No. 6, pp. 600–625.
4. **Bini, E. & Buttazzo, G. C. (2005).** Measuring the performance of schedulability tests. *Real-Time Systems*, Vol. 30, No. 1-2, pp. 129–154.
5. **Brandenburg, B., Block, A., Calandrino, J., Devi, U., Leontyev, H., & Anderson, J. (2007).** LITMUSRT: A status report. *Proceedings of the 9th real-time Linux workshop*, pp. 107–123.

6. **Brandenburg, B. B. & Gül, M. (2016).** Global scheduling not required: Simple, near-optimal multiprocessor real-time scheduling with semi-partitioned reservation. *IEEE Real-Time Systems Symposium (RTSS 2016)*, pp. 99–110.
7. **Buttazzo, G. (2011).** *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media.
8. **Calandrino, J. M., Anderson, J. H., & Baumberger, D. P. (2007).** A hybrid real-time scheduling approach for large-scale multicore platforms. *Proceedings of the 19th Euromicro Conference on Real-Time Systems*, ECRTS '07, IEEE Computer Society, Washington, DC, USA, pp. 247–258.
9. **Casini, D., Biondi, A., & Buttazzo, G. (2017).** Semi-Partitioned Scheduling of Dynamic Real-Time Workload: A Practical Approach Based on Analysis-Driven Load Balancing. **Bertogna, M.**, editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 13:1–13:23.
10. **Chandarli, Y., Fauberteau, F., Masson, D., Midonnet, S., & Qamhieh, M. (2012).** Yartiss: A tool to visualize, test, compare and evaluate real-time scheduling algorithms. *WATERS 2012*, UPE LIGM ESIEE, pp. 21–26.
11. **Chéramy, M., Hladik, P.-E., & Déplanche, A.-M. (2014).** Simso: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. *5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, pp. 6–p.
12. **David, R. & Alla, H. (2008).** Discrete, continuous and hybrid Petri nets. *Control Systems, IEEE*, Vol. 28, No. 3, pp. 81–84.
13. **Desel, J. & Esparza, J. (1995).** *Free Choice Petri Nets*. Cambridge Tracts in Theoretical Computer Science 40.
14. **Desirena, G., Rubio, L., Ramirez, A., & Briz, J. (2019).** Thermal-aware hrt scheduling simulation framework.
15. **Desirena-Lopez, G., Briz, J. L., Vázquez, C. R., Ramírez-Treviño, A., & Gómez-Gutiérrez, D. (2016).** On-line scheduling in multiprocessor systems based on continuous control using timed continuous petri nets. *13th International Workshop on Discrete Event Systems*, pp. 278–283.
16. **Desirena-Lopez, G., Ramírez-Treviño, A., Briz, J. L., Vázquez, C. R., & Gómez-Gutiérrez, D. (2019).** Thermal-aware real-time scheduling using timed continuous petri nets. *ACM Transactions on Embedded Computing systems. To appear, accepted Apr. 2019*.
17. **Desirena-Lopez, G., Vázquez, C. R., Ramírez-Treviño, A., & Gómez-Gutiérrez, D. (2014).** Thermal modelling for temperature control in MPSoC's using fluid Petri nets. *IEEE Conference on Control Applications part of Multi-conference on Systems and Control*.
18. **Funk, S., Levin, G., Sadowski, C., Pye, I., & Brandt, S. (2011).** Dp-fair: a unifying theory for optimal hard real-time multiprocessor scheduling. *Real-Time Systems*, Vol. 47, No. 5, pp. 389–429.
19. **Kato, S. & Yamasaki, N. (2007).** Real-time scheduling with task splitting on multiprocessors. *Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '07, IEEE Computer Society, Washington, DC, USA, pp. 441–450.
20. **MATLAB (2018).** *version 9.4 (R2018a)*. The MathWorks Inc., Natick, Massachusetts.
21. **Oh, D.-I. & Bakker, T. (1998).** Utilization bounds for n-processor rate monotone scheduling with static processor assignment. *Real-Time Systems*, Vol. 15, No. 2, pp. 183–192.
22. **Rubio-Anguiano, L., Desirena-López, G., Ramírez-Treviño, A., & Briz, J. (2018).** Energy-efficient thermal-aware scheduling for rt tasks using tcpn. *IFAC-PapersOnLine*, Vol. 51, No. 7, pp. 236–242.
23. **Silva, M., Júlvez, J., Mahulea, C., & Vázquez, C. R. (2011).** On fluidization of discrete event models: observation and control of continuous Petri nets. *Discrete Event Dynamic Systems*, Vol. 21, No. 4, pp. 427–497.
24. **Silva, M. & Recalde, L. (2007).** Redes de Petri continuas: Expresividad, análisis y control de una clase de sistemas lineales conmutados. *Revista Iberoamericana de Automática e informática Industrial*, Vol. 4, No. 3, pp. 5–33.
25. **Singhoff, F., Legrand, J., Nana, L., & Marcé, L. (2004).** Cheddar: a flexible real time scheduling framework. *ACM SIGAda Ada Letters*, volume 24(4), ACM, pp. 1–8.
26. **Srinivasan, J., Adve, S. V., Bose, P., & Rivers, J. A. (2005).** Exploiting structural duplication

for lifetime reliability enhancement. *Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on*, IEEE, pp. 520–531.

Article received on 24/10/2018; accepted on 16/02/2019.
Corresponding author is Gaddiel Desirena López.