Litmus-RT plugins for global static scheduling of mixed criticality systems

Laurent Pautet, Thomas Robert, Samuel Tardieu

Litmus-RT plugins for Global Static Scheduling of Mixed Criticality Systems

Laurent Pautet, Thomas Robert, Samuel Tardieu

LTCI, Télécom Paris, Institut Polytechnique de Paris, Paris, France

Abstract

Global static scheduling for Mixed Criticality (MC) systems demonstrates excellent results in terms of acceptance ratio and number of preemptions. But, no practical implementation and empirical evaluation have been presented yet for multi-processors systems. Moreover, the new kernel mechanisms it would require have not been studied.

In this paper, we present two contributions on the implementation of global static schedulers For MC systems: G-RES, a global table-driven reservations LITMUS^{RT} plugin, and G-MCRES, another LITMUS^{RT} plugin scheduling MC tasks with global table-driven reservations and enforcing safe criticality mode changes. These contributions aim to solve the problems of instantaneous migrations and simultaneous mode changes in the context of global static schedulers. We based our experiments on scheduling tables generated off-line by GMH-MC-DAG, a meta-heuristic to schedule multiprocessor systems composed of multi-periodic Directed Acyclic Graphs of Mixed Criticality tasks with multiple criticality levels. The performances are very good w.r.t those of LITMUS^{RT} and consistent with our temporal complexity evaluations.

Keywords: Mixed-Criticality Systems, Real-Time and Embedded Systems, High-Integrity Systems

1. Introduction

The adoption of multicore architectures in critical real-time systems leads to sharing more and more functionalities within a common execution platform. In traditional approaches such as the one proposed by the ARINC 653 architecture [7], in order to ensure safety constraints, only functionalities of the same criticality level share resources. This constraint limits the deployment of functionalities on multicore architectures to partitioned approaches, leading to a waste of resources [26]. To overcome this problem, the mixed criticality (MC) model proposes to execute tasks of different criticality levels on a common platform.

When a system runs in nominal mode, tasks are executed with an "optimistic" budget such as the worst-cast execution time (WCET) estimated by the system designer. When a task does not complete its execution within its optimistic budget, the system enters a higher criticality mode. Typically, only higher criticality tasks continue to run and use a "pessimistic" budget, such as the WCET determined by a certification authority.

Many contributions for scheduling such systems on multiprocessor architectures have been proposed in the literature [5]. Although, at the margin, few contributions demonstrate their survivability characteristics during runtime [2], most of these approaches have not demonstrated their usability in the critical systems industry. First, most of them partition the multicore system into multiple single-core subsystems at the cost of wasting resources. Second, most of them do not consider dependent tasks, although this model corresponds to the majority of industrial systems like those based on data flow graphs (SCADE, Simulink, ...). Third, the proposed scheduling algorithms may require significant modifications in an already certified real-time kernel. Finally, the approaches are often complex, not only because they integrate the complexity of multicore schedulers but also because they must ensure safe changes of execution mode.

As a consequence, few or no practical implementation and empirical evaluation have been presented yet. This is a major concern since schedulability performances also depend on scheduler implementation overheads. These overheads have many possible sources. In multicore platforms, preemptions and migrations are the number one source of issues. This paper aims at designing efficient execution platforms for global MC schedulers. To illustrate the discussion, we propose as a case study the implementation of GMH-MC-DAG[1] on top of LITMUS^{RT}. In other words, we use the scheduling tables generated off-line by GMH-MC-DAG as inputs to evaluate the proposed scheduler implementation.

GMH-MC-DAG is a good candidate for our study as it defines a meta-heuristic to schedule multicore systems composed of multi-periodic Directed Acyclic Graphs (DAG) of Mixed Criticality (MC) tasks. This approach produces offline scheduling tables supporting DAGs of real-time tasks with more than two criticality levels as often proposed in

[1] For *Generalized Meta Heuristic for Mixed-Criticality Directed Acyclic Graphs*

standards related to dependable system (*e.g.* Design Assurance Levels (DAL) in DO-178). GMH-MC-DAG also relies on time-triggered scheduling to improve the certifiability of the MC scheduler.

This paper is organized as follows. In section 2 we give an overview of related works. Section 3 describes the objectives of this contribution. In section 4, we give an overview of our case study which consists in the execution on top LITMUS^{RT} of scheduling tables generated off-line by GMH-MC-DAG algorithm. Section 5 presents our contribution to LITMUS^{RT} to support a global reservation-based scheduling through our G-RES plugin. From this first contribution, we describe in section 6 our G-MCRES LITMUS^{RT} plugin to enforce safe criticality mode changes, and we discuss the temporal complexity of the overerall solution. In section 7, we provide empirical overhead and latency results on the implementation of these two plugins. We conclude and give some perspectives in section 8.

2. Related Works

Supporting a very complete MC task model and providing an effective execution platform for it is an important topic for the real-time system community. Many theoretical works but little practical implementations have already been studied. First, we discuss the execution kernels that can support mixed criticality systems. Second, we present scheduling contributions that offer a rich MC task model at a reasonable cost in terms of kernel implementation or extension.

2.1. MC execution platforms

We choose to focus on kernel design issues for MC systems within the LITMUS^{RT} platform. Indeed, LITMUS^{RT} is an open real-time kernel well known by the real-time community, and is representative of the scheduling services that might be already present in an existing real-time kernel. Some prototypes of MC schedulers have already been implemented on top of LITMUS^{RT} such as MC2 [6].

An alternative to designing a complete MC kernel would be to take advantage of an existing one and to modify it as little as possible in order to limit the impact w.r.t. validation and certification. In particular, [3] highlights how a time-triggered kernel can provide enough support to integrate MC extensions by preserving certification properties. LITMUS^{RT} actually comes with the P-RES plugin which provides several budget reservation services for mono-processors and in particular a table-driven reservation one. Note that the time-triggered approach can also apply to a full time-predictable end-system with support for deterministic communication [15]. In addition, not only it is possible to benefit from a certified kernel but also from an entire environment for modeling, analysis and generation of high-integrity automatic code [9].

In section "System Issues" of [5], several solutions are proposed to enforce MC features, that is for instance to limit run-time overheads [8], to guarantee isolation [25] and to prevent interferences [12]. However, to ensure such properties, these platforms restrict the task model at the cost of poor scheduling performances. For instance, most of them target strongly partitioned MC systems.

LITMUS^{RT} already supports MCS through MC2 plugin [13, 6]. Yet, the MC2 platform considers partitioned scheduling only for the two most critical tasks, and global scheduling for soft real-time tasks (G-EDF). Hence, MC2 targets a trade-off that prioritize task isolation over scheduling performances. Indeed, it has average performances compared to state of the art MCS schedulers, [5]. For the same reason, virtual machine based approaches or ARINC partitioned systems introduce rather high overheads due to the strong memory isolation mechanisms [25].

A promising approach would be to target time-triggered (TT) kernels. [24] shows good results through an approach that generates scheduling tables. Yet, it is designed for mono-processor platforms and does not scale well to multi-core platforms. The same authors proposed an online mechanism to incorporate MC aspects into TT tables in [23]. This relies on unused time slots of high criticality level tasks assigned to low criticality level tasks.

2.2. MC task models

Our objective aims at targeting a rich MC task model to cover the kernel requirements needed to enforce MC features. More specifically, we want to demonstrate that multi-periodic DAGs of MC tasks with an arbitrary number of criticality levels can be executed by a global static scheduler thanks to the addition of simple and inexpensive mechanisms to these execution platforms.

Contributions considering MC tasks and multiple DAGs are [1, 16] and [19] which compute federated schedulers, a generalization of partitioned scheduling to DAGs. In [1], the author proposes a federated approach transforming the problem of scheduling multiple DAGs on a multi-core architecture, to schedule a single DAG on a cluster of cores of the multi-core architecture. However, we shall see these partitioned approaches can be outperformed in terms of schedulability performances. In [16] and [19], the task model is very restrictive as all tasks of the DAG have the same criticality HI or LO. At last, none of these approaches have been generalized to an arbitrary number of criticality levels.

In a previous work [17], we proposed GMH-MC-DAG which shows very good theoretical results in terms of schedulability performances and task model supported. It outperforms the partitioned approaches such as [1] in terms of acceptance ratio. Moreover, GMH-MC-DAG relies on time-triggered (TT) scheduling to improve the certifiability of the MC scheduler. Compared to other TT solutions such as [23], GMH-MC-DAG computes compatible scheduling tables, one per criticality mode. It has better schedulability performances as it does not rely on time slots that remained unused. Compared to other scheduling algorithms,

it supports an arbitrary number of criticality levels. Overall, GMH-MC-DAG is a good candidate to evaluate how to enrich an execution platform, in particular a TT one, to make it a MC execution platform.

2.3. Case study

While LITMUS^{RT} is a very complete real-time kernel, it does not support global static scheduling, that is table-driven reservations over multiple processors. Moreover, most of the MC plugins support partitioned scheduling. They do not support global mode change, that is a synchronized mode change over multiple processors. These two limitations are the motivations for our contributions. We use GMH-MC-DAG to illustrate them as it appears a good case study that enforces a complete MC task model without compromising the effectiveness of the execution platform. In our evaluation, the scheduling tables per processor and per mode generated off-line by GMH-MC-DAG will be used as an input.

3. Problem Statement

This section highlights two major issues when it comes to implement efficient mixed criticality schedulers.

From now, we use processor to designate the computing unit running a task let it be a core, a core thread or a processor in a multi-processor platform. Indeed, most global algorithms rely at some point on a global state transition. Such a transition often relies on scheduling event handlers that are atomic (and thus mutually exclusive to each other among processors). An alternative approach, the one we favor, requires to handle these events concurrently and thus support to some extent parallelism on scheduler state update. Such an implementation has to show that it preserves the consistency of the global scheduler state.

3.1. Overhead issues with global schedulers

[5] is the most up-to-date survey of MC models. Solutions inspired from global schedulers offer the best schedulability acceptance rate, *i.e.* the likelihood that the scheduler can ensure that deadlines are met for a given task set. Indeed, using a partitioned scheduling algorithm would introduce too many constraints to a system already strongly constrained as it first requires to assign tasks on processors.

Yet, the schedulability performance is not the only concern: schedulability also depends on overheads and behaviors at runtime of the scheduler implementation. These overheads have many sources but on multi-processors platforms, preemptions and migrations are the number one source of issues. Migrations in multi-processor schedulers are by nature distributed phenomena: a task is interrupted on a processor and resumes its execution on another one. The scheduler state has to maintain ready queues to identify which task could be ready to run if a processor is available. Conversely, each processor has to remember which task is currently running. Hence, its state is by nature distributed and state changes may happen concurrently, which represents a major risk of data races if not managed well.

Compared to partitioned schedulers, global schedulers may be valid theoretically but difficult to implement correctly. In a global scheduler with scheduling tables such as GMH-MC-DAG, a task may be suspended on one processor to be resumed on another processor at *the same time*. An inconsistency may arise if the second processor activates the next time slot of the task while the previous time slot has not been completed yet.

Research Objective RO1: As global schedulers incur many scheduling events that may occur simultaneously such as preemptions and migrations, would it be possible to implement the required state changes without mutually exclusive operations entailing costly synchronizations at the scale of the multi-processors system?

In particular, in the context of global table-driven schedulers, would it be possible to enforce an efficient and consistent migration of a task that suspends at the end of a time slot on a given processor to resume its execution instantaneously after at the start of the next time slot on another processor?

3.2. Synchronization issues during mode change

Theoretically, a mode change occurs simultaneously on all the processors in the MC system. In most schedulability analyses, the mode change cost is overlooked despite that it clearly incurs complex synchronization. First, this mode transition may induce an asynchronous rescheduling operation across all processors. Second, this rescheduling operation may happen on a processor which is already involved in a scheduling operation. At last, these rescheduling operations have to be managed simultaneously making it more complex to implement with non-atomic mutually exclusive primitives.

Research Objective RO2: Compared to partitioned MC schedulers, would it be possible to efficiently implement a mode transition that occurs simultaneously on all processors and concurrently to other scheduling operations without mutually exclusive operations entailing costly synchronizations at the scale of the multi-processors system?

In particular, in the context of global table-driven schedulers, would it be possible to enforce an efficient and consistent mode transition consisting in a synchronized change of scheduling tables? One problematic situation could occur when two simultaneous mode change requests are triggered by simultaneous budget overruns of two different tasks. While the system runs at criticality mode χ_ℓ, these two tasks can transition the system to criticality mode $\chi_{\ell+2}$ instead of $\chi_{\ell+1}$.

List of contributions

To answer the two research objectives described above, we propose two contributions:

- The first kernel mechanism is related to instantaneous task migrations and cyclic instantaneous migrations that have to be handled properly to avoid global deadlocks. It has been implemented in LITMUS^{RT} as the G-RES plugin.
- The second mechanism is related to a safe but efficient implementation of mode changes that can handle multiple simultaneous budget overruns. It has been implemented in LITMUS^{RT} as the G-MCRES plugin.

4. Background

In this section we introduce the task model we are targeting, then we present previous results related to GMH-MC-DAG scheduling on N-criticality systems. For more details, the reader should refer to [17] for the core approach and to [18] for the extensions (N criticality levels and optimizations of the offline part). Finally, we discuss the issues in supporting the online part of GMH-MC-DAG on a representative real-time kernel such as LITMUS^{RT}.

4.1. Task Model

A Mixed-Criticality System (MCS) is defined as a tuple $\mathcal{S} = (\Pi, \mathcal{CL}, \mathcal{G})$. Π is a homogeneous multi-processor architecture. $|\Pi| = m$ is the number of available processors on the platform. $\mathcal{CL} = \{\chi_1, \ldots, \chi_N\}$ is the ordered set of tasks criticality levels, each corresponding to a criticality mode as well. The system starts its execution in the lowest criticality mode χ_1. $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is a graph structured set of independent DAGs. A Mixed-Criticality Directed Acyclic Graph (MC-DAG), $G_g \in \mathcal{G}$, represents a program being executed in system $\mathcal{S}$. It is defined by a tuple: $G_g = (V_g, E_g, T_g, D_g)$ where $V_g \subseteq \mathcal{V}$, $E_g \subseteq \mathcal{E}$, and $\forall G_g, G_{g'} \in \mathcal{G}, V_g \cap V_{g'} = \emptyset, E_g \cap E_{g'} = \emptyset$. V_g is the set of vertices of the MC-DAG. Each vertex is a MC task executed by the program. $E_g \subseteq (V_g \times V_g)$ is the set of edges. Edges represent *precedence constraints* between two tasks. A task is ready to be executed as soon as all of its predecessors have completed their execution. T_g is the period of the graph, *i.e.* tasks without predecessors become active again once this period has been reached. The system is therefore *periodic*. D_g is the deadline (constrained or implicit) of the graph, *i.e.* all tasks of the graph need to be executed before this deadline.

Each vertex of the MC-DAG corresponds to a MC task, $\tau_i \in V$, defined as follows $\tau_i = (\chi_i, C_i(\chi_1), \ldots, C_i(\chi_i))$. $\chi_i \in \mathcal{CL}$ is the criticality level of the task. $C_i(\chi_1), ..., C_i(\chi_i)$ is the vector of WCETs of the task. $\{C_i(\chi_\ell) \in \mathbb{N} \mid \forall \chi_\ell \succ \chi_i, C_i(\chi_\ell) = 0\}$. We consider the discard MC model, the task is not executed on criticality levels that are higher than χ_i. We assume that $C_i(\chi_N)$ is monotonically increasing when N increases. In a multi-periodic system a task can have multiple activations, $j_{i,k}$ is the k-th job of task τ_i. $r_{i,k} = k \times T$ (resp. $d_{i,k} = k \times T + D$) is the release time (resp. deadline) of the k-th job of task τ_i.

When a job $j_{i,k}$ does not complete after having executed for $C_i(\chi_\ell)$ time units, a budget overrun is detected and the system makes a mode transition to mode $\chi_{\ell+1}$. All the tasks τ_j with $\chi_j \leq \chi_\ell$ are discarded.

4.2. GMH-MC-DAG Scheduler

In this section, we give a short overview of GMH-MC-DAG in order to detail the objectives of this contribution. GMH-MC-DAG is decomposed in an offline computation of scheduling tables and an online execution of these scheduling tables on a Time Triggered kernel [14].

The offline scheduler requires to compute scheduling tables respecting constraints that are described below. Integer linear or constraint programming [21] are some of the techniques used for this purpose. In order to avoid scalability issues, GMH-MC-DAG builds these tables by statically applying the G-EDZL online global scheduling algorithm [22]. This algorithm demonstrates a good acceptance ratio and a reasonable number of preemptions. This last criteria has a major performance impact on the memory size required to store the tables. As said above, the scheduling tables, one per criticality level, are built to enforce several constraints and optimizations that we describe in the next paragraphs.

For a criticality level χ_ℓ, each task τ_i of criticality level $\chi_i \geq \chi_\ell$ must be capable of executing within their $C_i(\chi_\ell)$ without missing its deadlines. This must also be guaranteed when the system performs a mode transition to criticality mode $\chi_{\ell+1}$. To achieve this safe mode transition, GMH-MC-DAG enforces a first constraint: at any time t while a job of task τ_i has not been fully allocated in mode χ_ℓ, the budget allocated to this job in mode χ_ℓ up to t must be greater than the one allocated to it in mode $\chi_{\ell+1}$ up to t. Intuitively this guarantees that whenever a budget overrun occurs, the final budget allocated to the job of τ_i is at least equal to its budget in mode $\chi_{\ell+1}$.

This requirement is illustrated in figures 1 and 2. On the scenario shown in figure 1 a criticality mode change affecting task τ_i can happen either at 50 ms if the task has not completed its job (budget overrun), or at any time before this date because of another task exhausting its own budget. A switch from mode χ_ℓ to mode $\chi_{\ell+1}$ guarantees that task τ_i will always be able to use a budget of $C_i(\chi_{\ell+1})$ (60 ms) if it is needed.

Similarly, on the scenario shown in figure 2 a criticality mode change affecting task τ_i can happen at 70 ms or earlier. However, if the switch from mode χ_ℓ to mode $\chi_{\ell+1}$ occurs between 10 ms and 30 ms or after 50 ms, the total budget available to task τ_i will be less than $C_i(\chi_{\ell+1})$ (60 ms). For this reason, the GMH-MC-DAG offline scheduling algorithm will never produce such a configuration.

A second constraint aims at ensuring that all precedence constraints are respected in any criticality mode and mode transition. To do so, the task deadlines are refined to ensure that data produced by the tasks of χ_ℓ-criticality (or lower) will be available when the $\chi_{\ell+1}$-criticality successors start their execution. With such constraints, a mode

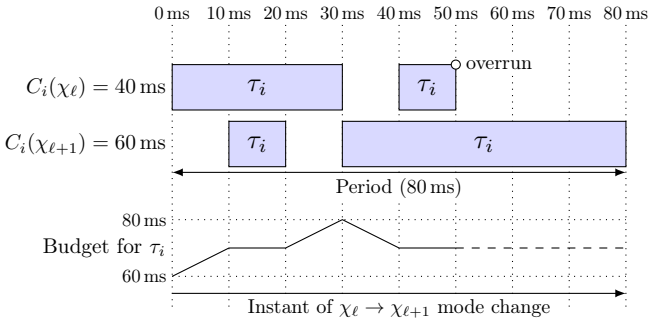

Figure 1: Budget guarantee after a mode change

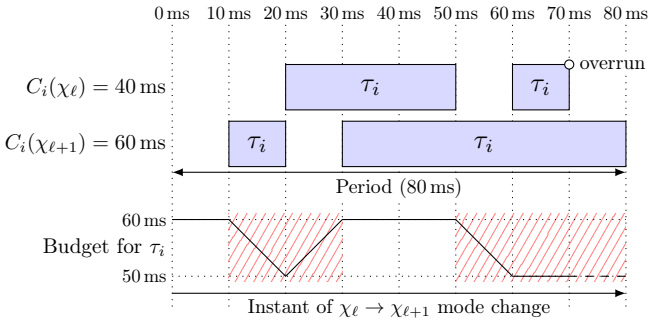

Figure 2: Insufficient budget after a mode change

transition consists only in a table transition if we disregard implementation issues, and precedence constraints are satisfied by construction.

Optimizations will mainly influence the acceptance rate, but they also have an impact on the number of preemptions and migrations and must be taken into account from an implementation point of view. For this reason, GMH-MC-DAG statically applies G-EDZL as global scheduling algorithm to build scheduling tables.

With this brief description of the offline building process of GMH-MC-DAG, we can now focus on the research objectives related to the required execution platform.

4.3. LITMUS^{RT} Environment

LITMUS$^{\text{RT}}$ is a soft real-time extension of the Linux kernel with focus on multiprocessor real-time scheduling. It provides numerous plugins implementing a large class of scheduling algorithms from the partitioned ones to the global or hybrid ones. It comes with a simple interface to design user-defined schedulers and a set of standard real-time components such as queues, timers or spinlocks. In particular, LITMUS$^{\text{RT}}$ P-RES plugin supports table-driven reservations, which can be used to implement time-triggered scheduling.

A vast collection of mixed criticality scheduling algorithms have also been developed on top of LITMUS$^{\text{RT}}$. It is an excellent platform to evaluate scheduler implementations and overheads but also to compare them with other alternatives. As we want to evaluate the design complexity and the overhead factors, we chose LITMUS$^{\text{RT}}$ as a base for our experimentation. We use the scheduling tables produced by GMH-MC-DAG as inputs to our contributions.

5. G-RES Plugin

As already said, we have to schedule table-driven periodic tasks on multiple processors. LITMUS$^{\text{RT}}$ does already support reservation based scheduling through the P-RES plugin and proposes three policies, periodic polling, sporadic polling and table-driven. This implementation enforces only partitioned schedules of tasks on multiprocessor architectures. Thus, it does not have to deal with issues raised when migrating threads or sharing reservations between processors (threads embody tasks in LITMUS$^{\text{RT}}$). We enriched the concepts provided in P-RES to enable global table-driven scheduling. This section explains how we extended this plugin and especially how we dealt with concurrency issues.

5.1. P-RES limitations

Table-driven schedules are usually described without loss of information over the hyper-period of the task set it handles (the least common multiple of task periods) as a sequence of intervals bound to tasks and processors. For each task, the sequence of intervals during which a task is scheduled in the table may potentially represent several jobs. P-RES uses the LITMUS$^{\text{RT}}$ task structure to which it attaches one reservation. Basically, the task structure defines the period at which jobs are activated. It is implemented in LITMUS$^{\text{RT}}$ with a thread that repeatedly executes the job body and then waits for its next activation. The task is in the suspended state meanwhile. The purpose of a reservation attached to a task is to store the schedule table description of the task. For example, on the scenario shown on figure 3, the reservation for the task τ_1 would contain the three intervals representing its two jobs $j_{1,1}$ and $j_{1,2}$ over the hyper-period.

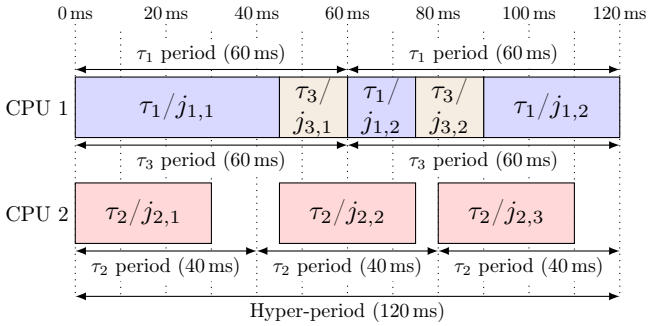

Figure 3: Relation between tasks (τ), jobs (j), periods and hyper-periods in P-RES

Definition 5.1. A P-RES reservation binds a sequence of disjoint time intervals, and a processing unit (CPU) identifier.

A reservation aims at scheduling a task *at an instant* t when it fulfills two conditions: that of being ready (not waiting on an event) and that of having an execution interval assigned to it (a reservation interval). In LITMUS^{RT}, *ready queues* are the main structures used to elect the tasks to be executed on a CPU. In P-RES, the election process is decomposed in two steps. Each CPU determines locally which reservation is active. Reservations bound to the same CPU are assumed to have mutually exclusive intervals. Hence, only one reservation is active at most. Once found, the reservation ready queue is used to determine which task bound to the reservation is active. While a reservation can be bound to many tasks, in order to implement an actual table-driven schedule, each reservation should initially be bound to only one task. This decomposition aims at reducing the scheduling overheads and the time spent in critical sections of the scheduler implementation. As reservations are bound to a single CPU and at most one reservation is bound to a task, no particular verification is done to check whether a task could be scheduled on two CPU simultaneously as it cannot happen by construction (partitioned scheduling). However, any global table-driven scheduling is required to prevent such a situation that might happen during an instantaneous migration.

Definition 5.2. An instantaneous task migration requires that a task resumes its execution on CPU 2 instantaneously after suspending its execution on CPU 1.

In a global table-driven schedule, such an instantaneous migration may occur when a reservation contains two intervals I and J on two different CPUs with a common upper and lower bound respectively ($Sup(I) = Inf(J)$). I and J are theoretically disjoint but may be adjacent. In this context, the actual start of the execution in interval J should only be allowed if the task has suspended its execution in interval I and its state is stored in memory to be restored on the other CPU. This is for example what happens in the scenario presented on figure 4 at the 90 ms instant: the task τ_1 migrates from CPU 2 to CPU 1.

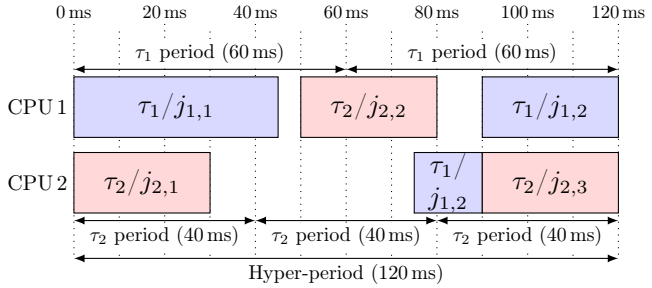

Figure 4: Migrating tasks in G-RES

An instantaneous migration can also be triggered as a side effect of the jitter observed when handling time-related events on different CPUs. We can have $Sup(I) < Inf(J)$ but $Inf(J) - Sup(I) \leq \delta t$ where δt represents a small mismatch in local clock states at the time the scheduling events corresponding to the end of the I interval and the beginning of the J interval are handled. In this case, an instantaneous migration will occur as well.

These synchronization issues during instantaneous migrations can have even worse consequences in the case of cyclic instantaneous task migrations.

Definition 5.3. A set of cyclic instantaneous task migrations is a set of instantaneous migrations that occur at the same theoretical time, and that result from a permutation of the tasks on the cores on which they were executing.

To give an example, such a situation arises when at time t, τ_1 migrates from $CPU1$ to $CPU2$ and τ_2 from $CPU2$ to $CPU1$. In LITMUS^{RT}, before descheduling its current task τ_i to execute another one τ_j, a CPU waits until τ_j is ready and possibly descheduled by another processor. If instantaneous migrations are not properly handled, cyclic instantaneous migrations would result in a deadlock detected but not recovered. Indeed, CPUs would be waiting for each others to deschedule the task they want to schedule. One may argue that such cyclic migrations should be prevented. But in practice, not supporting such migrations would lessen the performances of static scheduling such as GMH-MC-DAG. Thus we made it possible to support them.

P-RES does not provide any support to handle instantaneous migrations or cyclic instantaneous migrations, but LITMUS^{RT} offers reasonable primitives to handle such issues for usual task migrations (those not bound to reservations). This led us to develop a new plugin, called G-RES, implementing a global reservation-based scheduling. We first extend the definition of reservation.

Definition 5.4. A G-RES reservation is a sequence of ordered pairs of intervals and CPUs $(I_i, CPU_i)_{i \in \{1,\ldots,k\}}$, the intervals being disjoint from each other.

In order to schedule such reservations, we need to explain how we support instantaneous migrations and cyclic instantaneous migrations and what the main latency sources are.

5.2. Main latency sources

On real hardware with CPU cores operating autonomously from each other and communicating through asynchronous inter-processor interrupts, several latency sources make the migration non-instantaneous as illustrated in figure 5. In this scenario, a task τ_1 should migrate from CPU 1 to CPU 2 at instant T. The first source of latency is the delay t_h between the date at which the Linux kernel high-resolution timer has been set and the date at which the timer handler H_1 for task τ_1 is executed. The handler, which runs on an unspecified CPU ξ, realizes that τ_1 must migrate from CPU 1 to CPU 2. It asynchronously sends a rescheduling request to CPU 1 which is handled after a delay t_d as well as a rescheduling request to CPU 2 which

is handled after a delay t_s. CPU 2 must wait an extra time t_w until it is able to observe that CPU 1 has descheduled the task τ_1, as a task can never be running on two cores at the same time.

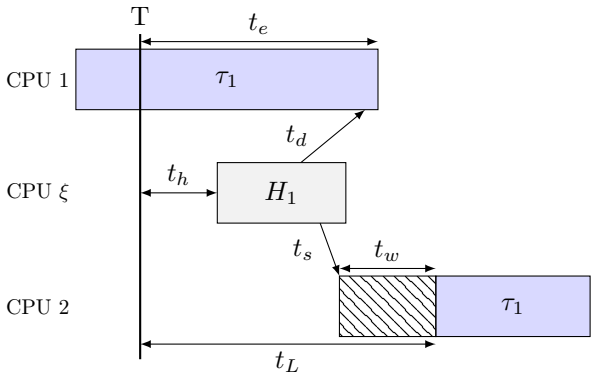

Figure 5: Main sources of latency during a CPU migration

After the migration process is complete, task τ_1 has run on CPU 1 until date $T + t_e$ and runs on CPU 2 since date $T + t_L$. Hence, $t_L - t_e$ is the overhead of our scheduler for task τ_1.

Section 7 contains experimental measurements of those latencies in our testing environment.

5.3. Support for cyclic instantaneous migrations

When using LITMUS$^{\mathrm{RT}}$ with any of its plugins as a kernel scheduler, the scheduling decision on a processor is made in 3 steps:

1. The Linux kernel informs the LITMUS$^{\mathrm{RT}}$ scheduler that a local scheduling decision must be made. The LITMUS$^{\mathrm{RT}}$ scheduler informs its currently active plugin, in our case G-RES, that a local scheduling decision must be made. The plugin selects the next task to run on the current processor, or none if no tasks are ready to execute, and informs the LITMUS$^{\mathrm{RT}}$ scheduler.
2. The LITMUS$^{\mathrm{RT}}$ scheduler checks if the requested task is already being scheduled on another processor. If it is the case, LITMUS$^{\mathrm{RT}}$ calls the plugin to let it decide whether it gives up its choice, *i.e.* schedule nothing right now, and requests a rescheduling event as soon as the current one is completed (*i.e.*, up to step 3). If it does not give it up, it waits for the requested task to be relinquished.
3. Once the requested task is available for scheduling, or if there is no task to schedule, the LITMUS$^{\mathrm{RT}}$ scheduler informs the Linux kernel of this decision. The Linux kernel deschedules the task currently running on the processor and schedules the requested task if one has been chosen, or a non-real-time task such as the "do-nothing" idle task otherwise.

When a processor tries to schedule a task that is still running on another processor, to enforce an instantaneous migration, at step 2, the G-RES plugin indicates that the processor wishes to temporarily give up on its current request and requests a new scheduling phase to be started as soon as possible. It then proceeds to step 3 with no task to schedule. When the request for the rescheduling event issued at step 2 is processed, the plugin asks for the exact same task that in step 1. This time, it will not give it up in step 2 as it cannot be responsible for the deadlock anymore: the task the plugin was previously executing has already been descheduled.

In the next section we show how the G-RES plugin is used as the foundation for the G-MCRES plugin to support a MC global static scheduler and especially mode transitions.

6. G-MCRES Plugin

This section describes the core features required to enforce mixed criticality global static scheduling. It explains why a budget overrun is easier to detect in the case of table-driven schedulers, and how we implement system-wide criticality mode changes in both directions in our G-MCRES plugin for LITMUS$^{\mathrm{RT}}$.

We also show how our approach guarantees the correctness of a mode change even in the presence of simultaneous or near-simultaneous scheduling events. Then we evaluate the temporal complexity of the worst-case mode change.

6.1. Per-mode scheduling tables and budget overrun detection

In our implementation, each task τ_i is described by a table containing the task reservations for each criticality mode $\{\chi_1, \chi_2, \ldots, \chi_i\}$ it can be scheduled in. Each reservation is bound to a task and a criticality mode. It describes the task behavior over the hyper-period of the task set computed by GMH-MC-DAG in the mode it is bound to (see section 5). When the task set is initialized, the scheduler goes through the list of tasks and registers the reservations bound to the lower criticality mode χ_1.

In section 4.1 we explained that a MC task τ_i has to be scheduled in all modes up to its criticality level χ_i. At any time t, a running task τ_i triggers a budget overrun detection in the current execution mode χ_c when its current job execution time reaches $C_i(\chi_c)$ before the job has been signaled as completed. Job completion states are closely monitored by LITMUS$^{\mathrm{RT}}$, and we only have to check them when budget overruns might occur.

Detecting a budget overrun when using a preemptive priority-based scheduler is costly as it requires to precisely keep track of the amount of processing power a task has used so far and permanently readjust the estimation of when the next budget overrun would happen. On the contrary, in table-driven approaches such as GMH-MC-DAG budget overruns can only occur at the end of the last interval allocated to each job of a task over its hyper-period. In figure 4, the overruns of task τ_1 may occur at instants

45 ms (end of job $j_{1,1}$) and 120 ms (end of job $j_{1,2}$) but not at instant 90 ms (job $j_{1,2}$ still has 30 ms of budget left).

In G-MCRES we identify and tag those intervals in a task reservation when the task first enters the system. G-RES triggers a timer event at the boundaries of every interval. While handling an end-of-interval event, we only have to check the status of the job. If the job is still active and the current interval is tagged as the last one available to a job, we have identified a budget overrun, and it might require a system criticality mode increase. However, care must be taken to account for the possibility of simultaneous (or near-simultaneous) budget overruns detections on different processors.

6.2. System criticality modes and mode changes

A shared variable `scm` (shared current mode) is used to store the value of the current mode of execution[2]. It is used to synchronize the update of the set of reservation tables that are active in the G-MCRES plugin. As `scm` tracks the current mode χ_c of the system, it is monotonically increasing during an hyper-period. In addition, each processor k has a processor-local variable `lcm[k]` (local current mode) that indicates the criticality mode in which the processor is currently working. Also, a shared integer variable `ac` (access counter) initialized at 0 represents the number of processors concurrently trying to increase the shared criticality mode in response to a budget overrun.

We update the local and shared variables when budget overruns occur. The purpose of adding a processor-local variable is to be able, for each processor that would detect an overrun, to associate this overrun with the criticality mode it occurred in.

As explained before, a budget overrun is always detected in a timer handler corresponding to an end-of-interval event. This timer handler, which is called from an interrupt service routine on processor k (which might or might not be the processor on which the job is currently executing), runs the code described in algorithm 1. This algorithm uses two kind of atomic operations to ensure synchronization between processors in case of multiple concurrent executions:

- CAS(a, b, c): the *compare-and-set* operation checks that a is equal to b, and if this is the case it sets a to c and returns *true*; otherwise it lets a untouched and returns *false*. This sequence of operations happen atomically with regard to multiprocessing.
- INC(j)/DEC(j): atomically increment/decrement an integer shared variable with regard to multiprocessing. If two processors execute any of those two operations simultaneously, they are automatically serialized.

This algorithm guarantees the following properties:

[2]We will see in section 6.3 this is a partial view of the real content of the `scm` variable. The extra piece of information stored within `scm` is not relevant to the algorithm presented here.

Algorithm 1: Mode change function in response to a budget overrun

```
1 INC (ac)
2 if CAS (scm, lcm[k], lcm[k] +1) then
3 |   deactivate reservation tables of mode lcm[k]
4 |   activate reservation tables for mode scm
5 |   request rescheduling of every processor
6 |   DEC (ac)
7 else
8 |   DEC (ac)
9 |   request local rescheduling
```

- If two processors j and k with `lcm[j]=lcm[k]=scm` call this function, only one of them will see a successful execution of the CAS operation at line 2 and will be in charge of implementing the mode change by updating the reservation tables. In other words, even multiple budget overruns occurring simultaneously or near-simultaneously at criticality mode C_k will cause only one increase of the criticality mode to $C_k + 1$.
- If any processor j with `lcm[j]<scm` calls this function, the CAS operation at line 2 will fail, and it will not change the reservation tables. In other words, if a processor is not yet aware of a recent criticality mode increase, it might trigger a budget overrun related to an obsolete reservation description. Yet, it will not modify the current mode as this mode change has already been accounted for.

The G-MCRES scheduling algorithm used to select the task to run on a processor k apart from mode changes may run concurrently to a mode change occurring at the initiative of another processor. This algorithm is described in algorithm 2 and is called by LITMUS^{RT} after having temporarily masked all interrupts on processor k. This means that a timing event occurring locally will not interrupt this algorithm and will be handled as soon as interrupts are enabled again and before executing any non-kernel code.

Algorithm 2: Scheduling function for processor k

```
   output: The elected reservation to run if any
1  lcm[k] ← scm
2  if ac ≠ 0 then
3  |   restart at line 1
4  selected ← gres_select_task ()
5  if lcm[k] ≠ scm or else ac ≠ 0 then
6  |   restart at line 1
7  return selected
```

This scheduling function starts by updating its local criticality mode to use the same value as the shared one

and waits for it to stabilize in case a mode change is in progress (checking the *ac* variable). At line 4, it uses the G-RES function that returns the next task to schedule on the current processor according to the active reservation tables. A side effect of function `gres_select_task` is to take care of setting up the timers used to detect the end of the current interval for the elected task. Remind those timers are also used to detect a budget overrun if the task does not complete before the last interval of its current job. No need to lock the access to scheduling table in this function because we check at the end that the timer configuration enforced is up to date. Indeed, before returning the selected task, we check that a mode change has not taken place or has not been started while using the G-RES selection algorithm. If such an event occurred, we restart the selection process to ensure that we have not worked with reservation tables in an inconsistent state.

The algorithms presented in this section do not deal with the possibility of lowering the system criticality mode. Indeed, we will show in the next section that restoring the regular behavior of the system is a much simpler operation that does not require any explicit synchronization between the processors.

6.3. Criticality mode reset

Once the criticality mode has been raised after the detection of one of several budget overruns, it needs to be reset to its lowest χ_1 value in order to return the system to its nominal behavior. In work conserving schedulers in which processor cores will be idle only if no task is ready to run, the transition to χ_1 usually happens when all cores are idle [20].

However, in table-driven schedulers this property does not necessary hold, as tasks are executed in their predetermined time windows rather than as fast as possible. The most appropriate time to reset the criticality mode to χ_1 is at the end of the current hyper-period, as the scheduling algorithm guarantees that all tasks whose criticality level is greater or equal than the current criticality mode have received their full execution budget.

In order to avoid a costly synchronization amongst all processors to reset the `scm` shared variable at the end of the current hyper-period, we have extended it with generational information: `scm` and the various `lcm[k]` contain in their most significant bits the hyper-period number (counting from 0 when the system starts) at which the latest change has been made, as shown in figure 6.

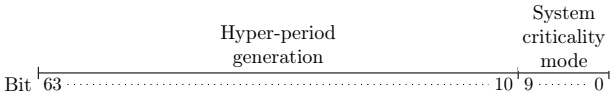

Figure 6: Encoding of the hyper-period generation

The current hyper-period generation can be easily computed at any time by dividing the system clock by the value of the hyper-period. By comparing the current hyper-period generation with the one encoded in `scm`, a processor can locally determine whether the system criticality mode is χ_1 or if the lowest bits of `scm` must be used. The value of `lcm[k]` is checked against the current hyper-period when entering the scheduling function on core k. If the stored `lcm[k]` corresponds to an earlier hyper-period, it gets reset to the value corresponding to the current hyper-period generation and criticality mode χ_1. This operation requires no synchronization with the other cores.

Algorithms 2 and 1 can be used as-is as the usual unsigned integer comparison function can be used to compare two extended values: a greater value either corresponds to a later hyper-period, or it corresponds to the same hyper-period with higher criticality mode. Since in a given hyper-period the system criticality mode can only be increasing, the greatest of two extended values represents the latest explicit criticality mode change in the system.

The choice of using 10 bits for the system criticality mode and 54 bits for the hyper-period when running on a 64 bit system is arbitrary, and it can be tuned at compile time. This particular setting allows using up to 1024 criticality modes for more than 5,000 years with an hyper-period duration is as short as 10 µs. Should our plugin be run on a 32 bit system with no 64 bit atomic access, using 4 criticality modes and 100 ms hyper-period would still allow the system to run for more than 3 years. If longer time spans, shorter hyper-periods or a greater number of criticality modes were necessary, the comparison operations could easily be enhanced to handle the wrap-around of the unsigned integer representing the hyper-period generation.

The reservations mentioned in section 6.2 are also structured in a way that the latest hyper-period interval in a task reservation at any criticality mode branches to the first interval at criticality mode χ_1. It ensures that unless another budget overrun increases the system criticality mode every task will be back to the lowest criticality mode after the end of the current hyper-period. Also, the timer associated with a task whose criticality level is lower than the current criticality mode will be configured to trigger at the time of the beginning of the first interval at criticality mode χ_1 in the next hyper-period.

After having shown that resetting the criticality mode to χ_1 after a criticality mode increase does not require any synchronization operation between the different processors at the end of the hyper-period, we will now present a breakdown of the latency of a criticality mode increase.

6.4. Temporal complexity of criticality mode changes

Our approach introduces overheads during the calls to the scheduling and mode change functions that are executed at the end of each interval of task reservations. Yet, these events can occur simultaneously and represent global changes to the scheduler state. We make a distinction between the latency of these transitions and their overhead.

On the one hand, the overhead is a meaningful parameter when assessing the system performances in the absence of mode changes. On the other hand, the latency, which represents the total delay between the start of the mode change and its completion on all the processors, has a significant impact because it delays the time at which all cores are running the tasks corresponding to the new criticality mode. During this delay, all the other processors are potentially wasting their time and energy, either by still executing prior tasks that may be discarded in the next mode, or by actively waiting for the reservation tables to be updated.

Criticality mode changes interact with the scheduling functions as in the worst case they can trigger the rescheduling of all the cores. We will first discuss more in detail the latency of migrations without mode changes. The worst case situation is caused by a cyclic instantaneous migration of all tasks. In these conditions, the latency is the maximum latency for all the tasks involved in the migration. In order to capture the total latency, we need to account for:

- The scheduling of all the timer handlers: in the worst case, each core is executing a task, and those tasks all simultaneously reach the end of their time interval, triggering the execution of a timer handler. In the worst case, all these timer handlers would all be scheduled on the same core as a single event. It means that this core would incur an overhead proportional to the number of cores (times the overhead of a single timer handler) but also that the last handler sees its execution delayed by this overhead.
- The re-execution of `gres_select_task()`: this function is in charge of telling the Linux kernel which task should run now on the current core so that the kernel can perform a context switch if needed. But this function is also in charge of preventing deadlocks that may occur during cyclic instantaneous migrations as explained in section 5.3. Roughly speaking, handling a migration in the worst case scenario requires executing the three steps described beforehand twice. We denote R_a the first execution of those three steps, and R_b the second one. R_a only enforces the descheduling of the previous task and runs in constant time in the absence of a mode change. R_b corresponds to the case where our plugin requests the scheduling of a task which is still scheduled by the Linux kernel on another core, in which case it is necessary to wait until the task is no longer scheduled somewhere else. This extra delay is capped by the end of the execution of R_a on the other cores, since at this stage all tasks have been descheduled from the processor they were running on before the migration.

In figure 5 we have shown that the total latency for a single task can be roughly decomposed into $t_h + t_s + t_w$. However, those parameters are not constant and vary with the global state of the scheduler: t_h depends on the number of tasks involved in the migration, and t_w depends on the order in which the handlers of those task timer handlers are executed.

The different value taken by those parameters for a three tasks cyclic instantaneous migration are shown in figure 7. Using t_x^i to denote the delay t_x specific to task τ_i, the total latency is the maximum of $\{t_L^1, t_L^2, t_L^3\}$. In this example, all task timers have been started on CPU 1 and their handlers execute there sequentially from the timer interrupt service routine in the order H_1, H_2, and H_3. A time dependency chain can be identified between H_1, H_2 then H_3 running on CPU 1, R_a for task τ_3 running on CPU 3, followed by R_b for task τ_3 running on CPU 1. We introduce worst case execution times, waiting time excluded, for the migration process elementary steps, noted γ_x . Hence, each handler is executed in at worst γ_h. R_a is executed in at worst γ_a, and R_b takes no longer that γ_b. t_s being the worst case delay to start a scheduling request on a remote core. Given those parameters, the worst case latency is bounded by the following expression $(m \cdot \gamma_h) + (t_s + \gamma_a) + \gamma_b$, where m is the number of cores. In this context, we can safely conclude that there are no real concerns for scalability as long as γ_h remains low.

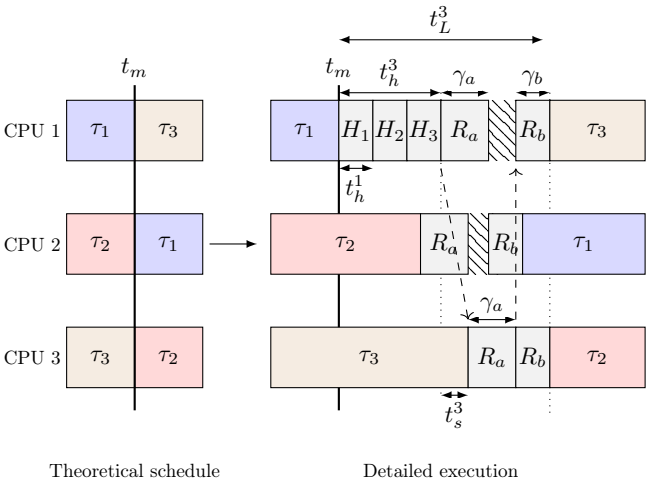

Figure 7: Total latency decomposition for migrations

A similar reasoning can be followed to study criticality mode changes. The first concern would be the interaction of algorithm 1 with itself. Each core detecting a budget overrun will execute this algorithm from its own timer handler, but only one core will be able to execute lines 3 to 5 and be responsible for a significant part of the worst case latency.

Indeed, algorithm 1 has two distinct behaviors depending on the value of `scm` compared to $|lcm[k]|$ for processor k. Either it goes through a short path and simply requests a local rescheduling, or it executes the actual update of the scheduling tables. The execution time associated with the tables update is proportional to the number of tasks (m)

and the number of time intervals per reservation. With at most W intervals per reservation, the time needed to execute lines 3 to 5 is at most proportional to $m \cdot W$, as the algorithm has to select the next interval of interest for every task and update task-specific timers.

Once the reservation tables update has been done, the mode change is complete as soon as the rescheduling requested on line 5 of algorithm 1 has been performed by all the cores. We can reuse the previous analysis to estimate the complexity of this global rescheduling. In the worst case, even if other cores require their rescheduling before, we can show that the actual rescheduling is delayed to wait for the completion of algorithm1 on the core updating the reservation tables. Algorithm 2 uses the value of `ac` to guarantee the correctness of our approach. We re-execute the scheduling functions as long as the `ac` counter is not equal to 0. It means that in the worst case the execution time of algorithm 1 is simply added to the worst case latency due to migrations.

The next section presents empirical assessments of some of those latencies. In particular, it shows that the worst case latency (t_L^3 in the exemple of figure 7) stays within reasonable bounds even if a large number of tasks are parts of a cyclic instantaneous migration.

7. Experiments

Unless specified otherwise, the tests have been performed on a Hewlett-Packard EliteBook 840 G2 laptop running LITMUS^{RT} with our G-MCRES plugin. The processor is an Intel Core i7-5600U CPU running at 2.6 GHz with 2 physical cores and 4 threads. In this section, we will use the term "core" to designate an autonomous execution unit, which can be either a physical core or a thread within a physical core.

7.1. Decomposition of a criticality mode change

The first experiment shown in figure 8 confirms that the delay between a budget overrun causing a criticality mode change and the end of the system reconfiguration in the upper mode depends only on the number of tasks. 4, 8 or 12 tasks are running on up to 4 cores, and one of the tasks triggers a criticality mode change. As expected, we note that the time to perform the complete criticality mode change grows linearly with the number of tasks, and that it does not depend on the number of cores. As explained in section 6.2, updating the scheduling tables may trigger a rescheduling operation on arbitrary cores. Those operations are asynchronous and start in parallel with each other and progress once the table update is over.

The whole mode change can be further decomposed into successive steps. Figures are given for the 8 tasks and 4 cores case (43.6 µs):

- The delay t_h (see figure 5) between the theoretical budget overrun detection and the beginning of the execution of the timer handler. This delay is around 330 ns in our case and accounts for less than 1% of the total mode change time.
- The time necessary to update the scheduling tables (32.5 µs, or 74.5% of our mode change time).
- The delay until all concerned cores have been rescheduled make up for the rest of the time. This process (21.7 µs) starts during the scheduling tables update and overlaps with the end of the tables update itself.

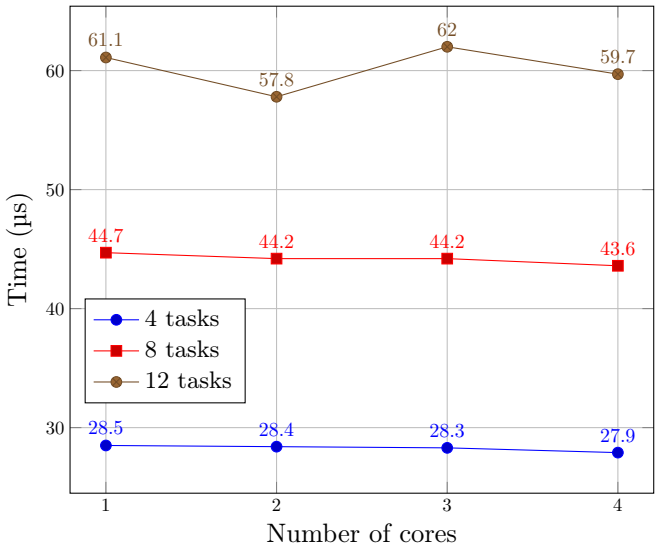

Figure 8: Measured mode change times

7.2. Inter-processor interrupts latency

The Linux kernel uses inter-processor interrupts to communicate between cores, *e.g.* to request that another core reschedules a new task. In this test, we make a single task continuously migrate from one core to the next and measure the delay between the rescheduling request sent by the timer handler when the task must start executing and the effective rescheduling. The time windows are chosen small enough as to allow a dead time before a migration; those are not instantaneous migrations as defined in section 5.3.

A Linux high-resolution timer is created for every task when the task is first loaded into the system. This timer is used throughout the task lifetime, and is rearmed to trigger at this task next interval boundary. As a consequence, successive timer handler calls will always take place on the same logical core. In our test, the timer triggering the scheduling requests is created and runs on logical core 0.

Our computer contains two physical cores, each containing two threads, for a total of four logical cores. In figure 9, we can distinguish distinct latencies for the various logical cores:

- Logical core 0 is rescheduled by pending a local interrupt, which will execute as soon as interrupts are locally enabled again, as the currently executing timer handler already executes in an interrupt context.

- Logical core 1 is rescheduled using an inter-processor interrupt sent by logical core 0. However, as both logical cores are located in the same physical core, the latency is smaller than when addressing another physical core.
- Logical cores 2 and 3 are rescheduled using an inter-processor interrupt which is routed to the physical core they both share. As a result, the latency is three times higher than it is to contact logical core 1.

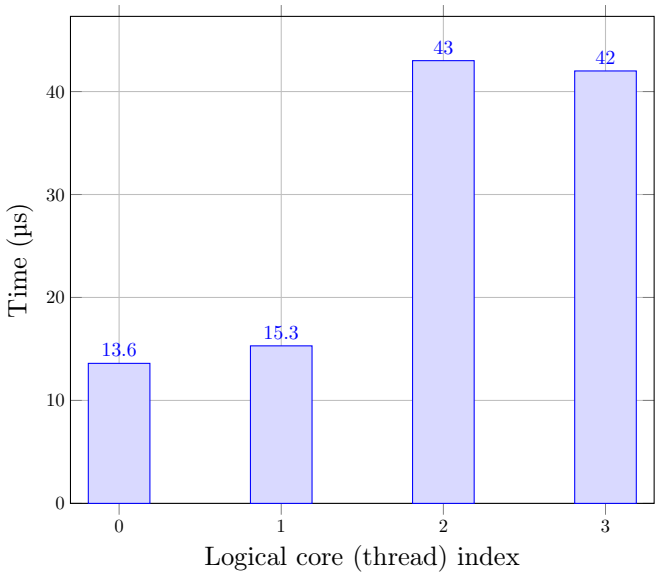

Figure 9: Rescheduling delays (timer handler runs on core 0)

7.3. Worst-case scenario

In this scenario, we run N tasks on N cores. Every 10 ms, each task requests a migration to the next core in a cyclic way. This is a worst-case scenario, because:

- Task-specific timers trigger at the same time for all the tasks.
- Each core is busy with a task at any time.
- Each task migrates to the next CPU at every 10 ms time window, forming a cyclic instantaneous migration set as described in section 5.3 and illustrated on figure 7.

In order to be able to study the results for a greater number of cores, we have used a Dell Precision Tower 5810 computer running Arch Linux on an Intel Xeon E5-1660 CPU running at 3.5 GHz with 8 physical cores and 16 threads. The LITMUS$^{\mathrm{RT}}$ kernel running our plugins was installed on a Ubuntu 18.04 system running inside a QEMU virtual machine [4]. The virtual machine uses hardware virtualization to gain direct access to the processor core and execute computationally intensive tasks natively. The processor cores were configured at full speed to prevent frequency scaling that could skew the measures.

Figure 10 shows two sets of data. The first set is the delay between the data at which a task decides to migrate and at which it is scheduled on the destination CPU (t_s+t_w in figure 5). The second set is the total delay between the first migration request at a given time window and the last successfully scheduled task for this time window. If we look at the graph, it looks like the migration time, be it individual or total, grows roughly linearly with the number of tasks and cores.

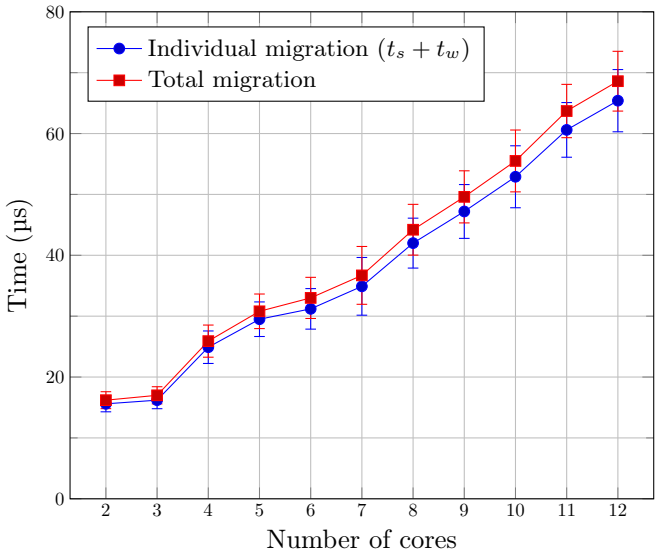

Figure 10: Measured individual and total core migration times (mean and standard deviation)

We can further decompose the individual migration time in two parts, as shown in figure 11. It is interesting to note that the main component of the migration time in our scenario is what is labeled "Busy waiting" on the graph. This delay corresponds to the busy loop performed by LITMUS$^{\mathrm{RT}}$ to wait until the requested task has been descheduled from the core it was previously executing on. In our worst-case scenario, every task is always executing somewhere. Another smaller factor increasing with the number of cores involved in the migration time is the time needed to transfer every task context at the same time from one core to another using the processor internal bus.

In real-world applications whose time windows are computed offline, cyclic instantaneous migrations are less likely to happen. However, as we noted in section 5.1, some particular configurations with very constrained resources might be schedulable if we allow those cyclic instantaneous migrations to take place, notably at the time of a criticality mode change. Even though we forged the worst possible scenario for the purpose of the experiments, we are satisfied with the overall performances.

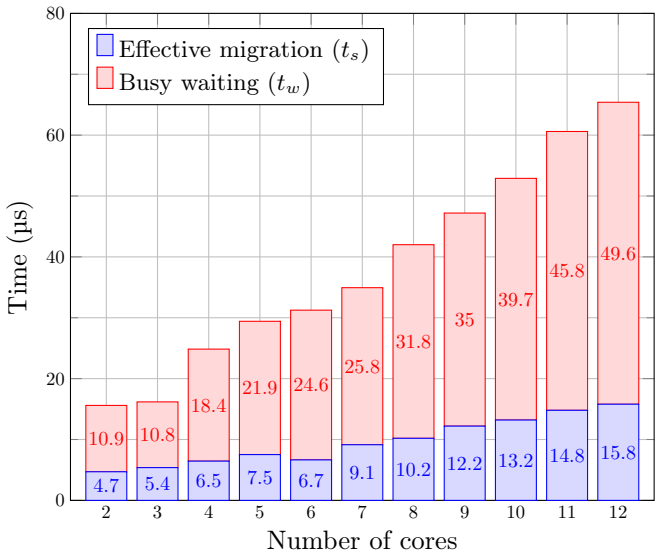

Figure 11: Decomposition of individual core migration time

8. Conclusion

This paper presents the design of run-time mechanisms that enable the deployment of global table driven schedules to support the GMH-MC-DAG task model and scheduling approach. We pointed out that this model is leveraging important limitations of MC task models as it allows expressing precedence constraints and several criticality levels. In addition to this, we selected this scheduling approach because it targets time triggered architectures that are particularly easy to verify and certify. We decomposed this implementation in two LITMUS^{RT} plugins to separate concerns: global table driven reservations, and criticality mode change support.

The main problems tackled by this contribution are concurrency issues in the deployment on multi-core or processor of such global schedulers. For these architectures, the scheduler is a distributed algorithm with a mix of event and time triggered behaviors. The first concurrency issue is related to instantaneous task migrations and cyclic instantaneous migrations that have to be handled properly to avoid global deadlocks. The second issue is related to a safe but efficient implementation of mode changes that can handle multiple simultaneous budget overruns. The design has been integrated to LITMUS^{RT} to check it has no side effect and can be integrated into regular services of real-time kernels.

The unified code for the G-RES and G-MCRES plugins has been publicly released[3] and will be submitted for inclusion into LITMUS^{RT}. We also aim at integrating the mechanisms implemented in our plugins in a more time predictable platform like an ARINC-653 compliant one such as the POK kernel [10, 11].

[3]The source code for the plugins is available from `https://aces.wp.imt.fr/g-mcres/` under the terms of the GNU General Public License version 2.

Acknowledgments: the authors would like to thanks Louise Flick for having implemented the first draft of G-RES plugin.

References

[1] Sanjoy Baruah. The federated scheduling of systems of mixed-criticality sporadic DAG tasks. In *Real-Time Systems Symposium (RTSS), 2016 IEEE*, pages 227–236. IEEE, 2016. 2.2

[2] Sanjoy Baruah and Alan Burns. Expressing survivability considerations in mixed-criticality scheduling theory. *Journal of Systems Architecture*, 109:101755, 2020. 1

[3] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *Proc. IEEE Real-Time Syst. Symp.*, pages 3–12, 2011. 2.1

[4] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, pages 41–46, Anaheim, California, USA, 2005. 7.3

[5] Alan Burns and Robert Davis. Mixed criticality systems-a review. *Department of Computer Science, University of York, Tech. Rep*, pages 1–81, 2019. 1, 2.1, 3.1

[6] Micaiah Chisholm, Namhoon Kim, Bryan C Ward, Nathan Otterness, James H Anderson, and F Donelson Smith. Reconciling the tension between hardware isolation and data sharing in mixed-criticality, multicore systems. In *2016 IEEE Real-Time Systems Symposium (RTSS)*, pages 57–68, 2016. 2.1

[7] Airlines Electronic Engineering Committee et al. ARINC 653 — avionics application software standard interface. 2003. 1

[8] Robert I Davis, Sebastian Altmeyer, and Alan Burns. Mixed criticality systems with varying context switch costs. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 140–151, 2018. 2.1

[9] Julien Delange, Jérôme Hugues, Laurent Pautet, and Bechir Zalila. Code Generation Strategies from AADL Architectural Descriptions Targeting the High Integrity Domain. In *Embedded Real Time Software and Systems (ERTS2008)*, toulouse, France, January 2008. 2.1

[10] Julien Delange and Laurent Lec. POK, an ARINC 653-compliant operating system released under the bsd license. In *13th Real-Time Linux Workshop*, volume 10, 2011. 8

[11] Julien Delange, Laurent Pautet, Alain Plantec, Mickael Kerboeuf, Frank Singhoff, and Fabrice Kordon. Validate, simulate, and implement arinc653 systems using the aadl. *Ada Letters*, 29(3):31–44, November 2009. 8

[12] Xavier Jean, David Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. Ensuring robust partitioning in multicore platforms for ima systems. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 7A4–1–7A4–9, 2012. 2.1

[13] Namhoon Kim, Jeremy Erickson, and James H Anderson. Mixed-criticality on multicore (MC2): A status report. In *Proceedings of the 10th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 45–50, 2014. 2.1

[14] Hermann Kopetz. The time-triggered model of computation. In *Real-Time Systems Symposium*, 1998. 4.2

[15] Eleftherios Kyriakakis, Maja Lund, Luca Pezzarossa, Jens SparsÃ¸, and Martin Schoeberl. A time-predictable open-source ttethernet end-system. *Journal of Systems Architecture*, 108:101744, 2020. 2.1

[16] Jing Li, David Ferry, Shaurya Ahuja, Kunal Agrawal, Christopher Gill, and Chenyang Lu. Mixed-criticality federated scheduling for parallel real-time tasks. *Real-Time Systems*, 53(5), 2017. 2.2

[17] Roberto Medina, Etienne Borde, and Laurent Pautet. Scheduling multi-periodic mixed-criticality DAGs on multi-core architectures. In *Proc. IEEE Real-Time Syst. Symp.*, pages 254–264, 2018. 2.2, 4

[18] Roberto Medina, Etienne Borde, and Laurent Pautet. Generalized mixed-criticality static scheduling for periodic directed

acyclic graphs on multi-core processors. *IEEE Transactions on Computers*, pages 1–1, 2020. 4

[19] Risat Mahmud Pathan. Improving the schedulability and quality of service for federated scheduling of parallel mixed-criticality tasks on multiprocessors. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 106. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018. 2.2

[20] François Santy, Laurent George, Philippe Thierry, and Joël Goossens. Relaxing mixed-criticality scheduling strictness for task sets scheduled with fp. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 155–165, 2012. 6.3

[21] Klaus Schild and Jörg Würtz. Scheduling of time-triggered real-time systems. *Operating Systems of the 90s and Beyond*, 5(4):335–357, 2000. 4.2

[22] Suk Kyoon Lee. On-line multiprocessor scheduling algorithms for real-time tasks. In *Proceedings of TENCON'94 - 1994 IEEE Region 10's 9th Annual International Conference on: 'Frontiers of Computer Technology'*, pages 607–611 vol.2, 1994. 4.2

[23] Jens Theis and Gerhard Fohler. Mixed criticality scheduling in time-triggered legacy systems. *Proc. WMC, RTSS*, pages 73–78, 2013. 2.1, 2.2

[24] Jens Theis, Gerhard Fohler, and Sanjoy Baruah. Schedule table generation for time-triggered mixed criticality systems. *Proc. WMC, RTSS*, pages 79–84, 2013. 2.1

[25] Salvador Trujillo, Alfons Crespo, and Alejandro Alonso. MultiPARTES: Multicore virtualization for mixed-criticality systems. In *2013 Euromicro Conference on Digital System Design*, pages 260–265, 2013. 2.1

[26] Yuanbin Zhou, Soheil Samii, Petru Eles, and Zebo Peng. Scheduling optimization with partitioning for mixed-criticality systems. *Journal of Systems Architecture*, 98:191–200, 2019. 1

Laurent Pautet is a Professor at Telecom Paris. His research activities focus on design and validation of critical embedded systems (non-functional properties verification, real-time scheduling, real-time kernels). He contributes technically and scientifically to several international free software projects such as GNAT/GCC. He is also a member of standard committees such as AADL's one. He is the author or co-author of more than 25 papers in international journals, more than 100 papers in international conferences. He is an editor and a contributor to several books in the area of distributed real-time embedded systems.

Thomas Robert received his Ph.D. degree from Institut Polytechnique de Toulouse. This work tackled run-time mechanisms to monitor real-time system behavior. Since 2009, he is an Associate Professor at Telecom Paris, a member of the Institut Polytechnique de Paris. He is doing his research in the LTCI lab. His research is organized on two axes: models to design and monitor scheduling algorithms of real-time embedded systems, and quantitative security risk analysis for industrial networked systems.

Samuel Tardieu is an Associate Professor at Telecom Paris, and a member of the LTCI lab and Institut Polytechnique de Paris. His research areas encompass real-time embedded operating systems operating under safety critical constraints. Also a free software movement activist for 25 years, he developed or contributed to many free software projects including the first implementation of Ada distributed system annex for the GNAT compiler.