Run-time Monitoring and Control for Temporal Fault Prevention in Mixed-criticality Systems

Daniel Loche[†*],
[*] Technocentre RENAULT
F-78280, Guyancourt, France
Email: daniel.loche@renault.com

Aléxis Génèrès[†], Michaël Lauer[†], Jean-Charles Fabre[†]
[†] LAAS-CNRS
31400, Toulouse, France
Email: firstName.lastName@laas.fr

***Abstract*—Multicore parallelism involve inter-tasks interferences leading to execution timing uncertainties. This is important for safety concerns in industrial applications (automotive for instance). Existing solutions limit significantly the use of available computing resources for low-criticality tasks to keep strong timing guarantees on high-criticality tasks. We propose a run-time monitoring and control approach for mixed-criticality systems to prevent temporal faults. Its objective is to guarantee a high-criticality function end-to-end deadline, even when interfered by low-criticality tasks parallel execution. Such approach aims at allowing better computing resource use with an additional guarantee for a given task chain end-to-end response time. We also propose an experimental protocol to characterise this task chain and calibrate the mechanism. Our approach has been implemented on a Linux-based experimental platform.**

***Index Terms*—multicore, real-time, mixed-criticality, Linux**

I. Introduction

Resource sharing in modern processors implies that concurrent software tasks are affected by execution interference (memory and caches, I/O, etc.). This phenomenon can lead to temporal faults when such interferences lead to huge execution time increases. Temporal faults concerns are typically avoided with time and space partitioning. Such temporal partitioning are dimensioned from Worst-Case Execution Time (WCET) estimations. However, in multicore environments, the amount of shared resources exacerbate the issue of timing faults due to more subtle interferences. Execution times estimations become either intractable or overly pessimistic in such context [1], leading WCET-based solutions to sub-use of computing resources.

However, dealing with Mixed-Criticality Systems (MCS) allows using degraded modes of operation as a safety mechanism to prevent temporal faults. The switch to a transient degraded mode is a viable option to prevent temporal faults and thus guaranteeing timing constrains on specific critical software tasks. In a dual-criticality system (with low (LO) and high (HI) criticality tasks), we define a LO-criticality mode (or "nominal" mode, i.e. all tasks running) and a HI-criticality mode with only HI-criticality tasks executions being guaranteed. Such degraded mode implies prioritising HI-criticality tasks in order to avoid interferences due to LO-criticality ones that could lead to temporal faults. It can be made on different aspects of the system (memory access, scheduling...) depending on the interferences source to mitigate. In our design, we do this with an "all-or-nothing" solution where the HI-criticality mode pauses LO-criticality tasks until the system comes back to LO-criticality mode. Our proposal relies on dynamic switching at run-time between LO and HI modes but in a way to keep the system as long as possible in LO-criticality mode.

The use of dynamic mode switches leads to the question of how and when. In this work we develop a mechanism based on the anticipation of temporal faults in order to avoid them.

Our main objective is to respect such constraints with the least possible performance compromises on non-critical software, to get the most benefit from multicore processors. In this work, we consider critical applications as implemented as task chains. A task chain is a sequence of tasks performing a complex action. Considering end-to-end deadline of task chains is more realistic in embedded application, as in the automotive domain. A multicore can aggregate mixed criticality applications like an infotainment system and an Advanced Driving Assistance System (ADAS). Such environment actually requires new solutions to avoid as much as possible interferences that leads to temporal faults on critical functionalities.

We focus on the remaining time until deadline of a functional critical task chain in the system to build our anticipation switch decision. We compute its remaining worst-case guaranteed time at run-time, to anticipate a risk of such estimation to exceed the critical function deadline as inspired by [2] but at a more macroscopic scale. In order to be as accurate as possible, and so avoid unnecessary tasks deactivation, we explicitly handle end-to-end deadlines. Indeed, those are usually handled task by task, meaning that the tasks own deadline are considered. While simpler to deal with, such approach is pessimistic [3].

In MCS, academic solutions exist to tackle both real-time guarantees and exploiting as much as possible computing resources. However they are not implemented in industrial systems as they seem too much complex, specific or intrusive [4]. We propose a non-intrusive framework (enabling Black Box/Legacy code use) and an experimental approach to reconcile those seemingly opposite goals. Our solution relies on the exploitation of task chain models with high level monitoring and anticipation of timing faults.

The novelty of our approach can be summed up as follows:

- use a Monitoring and Control Agent to check at run-time the execution of the HI-criticality task chains
- anticipate potential worst-case reaction time risks with respect to end-to-end deadlines

- in case of a potential deadline miss, degraded mode is triggered by temporarily deactivating LO-criticality tasks

In section II, we first present the Monitoring and Control Agent as a safety mechanism to prevent temporal faults of a critical task chain. In section III we propose an experimental protocol to configure and test it. Finally we present in section IV a first use case to illustrate and quantify the performance of our mechanism on a benchmark-based system.

II. Monitoring and Control Agent

In this section we describe our Monitoring and Control Agent (MCA) as a safety mechanism designed to avoid temporal faults in mixed-criticality systems. Its goal is to guarantee critical end-to-end task chain response times by avoiding interferences that could lead to such temporal fault.

The MCA role is first to monitor the state of a HI-criticality task chain to detect potential deadline miss. If such a potential fault is anticipated, then the MCA switches the system to HI-criticality mode, pausing all non essential workload (LO-criticality tasks), to prevent further interference on the HI-criticality tasks and allow a safe termination. To be efficient, the switch must be triggered only when necessary (as a "mode switch procrastination", as called in [5]). That is why we also focus on end-to-end deadline, rather than individual task deadlines, in order to avoid false-positive switching, meaning switching to HI-criticality mode although there is slack in the task chain. Indeed, with an end-to-end perspective, we can use the slack given by a task finishing early to compensate the lateness of an other task in the chain.

In the following, we introduce the execution model considered in our work, then we describe the proposed MCA architecture to finally present in more details the principle of the anticipation mechanism.

A. Execution model

The model describes how HI-criticality tasks behave and are linked in a chain to produce a HI-criticality functionality. Note that this model is one among many usable with our approach. We choose this one for ease of presentation.

The system is composed of a set of tasks, partitioned into two sub-sets: HI, containing all HI-criticality tasks and LO containing all LO-criticality tasks. We make no particular assumption on LO except that we are able to stop and continue these task at run-time (typically with SIGSTOP and SIGCONT signals on a Unix-like system).

Each **task** $\tau_i \in HI$ is activated and executed with a period T_i. The **job** $\tau_{i,j}$ corresponds to a the j^{th} execution of τ_i. Its activation time is noted $a_{i,j}$, it starts at $s_{i,j}$, and ends at $e_{i,j}$. A job consumes all the data available to it when it starts, does some processing, and produces data when it ends.

Dependencies of tasks in HI are expressed with a **task chain** $\tau_1 \rightarrow \tau_2 \ldots \rightarrow \tau_n$, where τ_1 is the *entry* task and τ_n is the *exit* one. Note that the model can be extended to support tasks linked through a Directed Acyclic Graph (DAG) without difficulty. The behaviour of the dependency relation $\tau_i \rightarrow \tau_{i+1}$ is: to produce its output, a job $\tau_{i+1,k}$ consumes all pending data produced by τ_i since the last execution of τ_{i+1}, i.e. the start of $\tau_{i+1,j-1}$. More formally the job $\tau_{i,j}$ has an effect on $\tau_{i+1,k}$ iff it is the first job of τ_{i+1} starting after the end of $\tau_{i,j}$, i.e. it is such as $s_{i+1,k} \geq e_{i,j}$ and $s_{i+1,k-1} < e_{i,j}$. In that case, we call $\tau_{i+1,k}$ the successor of the job $\tau_{i,j}$. We note $succ()$ the function to find the successor of a job. The iterate function $succ^{n-1}()$ allows to find the job of the exit task depending on a job of the entry task. For instance, a 3-tasks chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ is depicted on Figure 1 where $succ^2(\tau_{1,1}) = succ(succ(\tau_{1,1})) = succ(\tau_{2,2}) = \tau_{3,2}$.

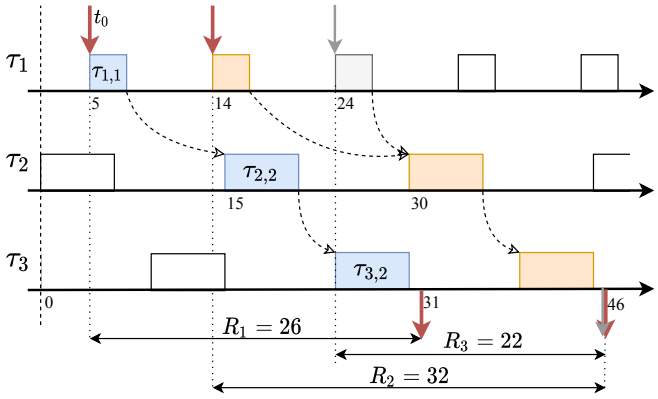


Fig. 1. Task chain run-time example with $\tau_1 \rightarrow \tau_2 \rightarrow \tau_3$

The notion of successor allows the definition of the response time of the chain: it is the elapsed time between the activation of an *entry* task job $\tau_{1,j}$ and the end of the *exit* task job $\tau_{n,k} = succ^{n-1}(\tau_{1,j})$. Noting R_j the response time of this activation, we have $R_j = e_{n,k} - s_{1,j}$. On Figure 1 the resulting response times R_1, R_2, R_3 of the first three entry task activation of the chain are represented. Note that with this definition, because tasks can have different periods, several jobs of τ_i can have an effect on a job of τ_{i+1}, as shown in Figure 1

Intuitively, an **end-to-end deadline** means that the time it takes for an input of the chain to have an effect on its output, i.e. its response time, must be bounded. Thus, given a deadline D, to be temporally safe our task chain must satisfy: $\max_{j \in \mathbb{N}}\{R_j\} \leq D$.

B. Anticipation mechanism principle

Our anticipation mechanism is based on the run-time monitoring of the task chain progress. To that end, we introduce the notions of **Task Chain State** and **Task Chain Execution Trace** (TCET). A TCET contains an entry task job and all the iterative successors of that job. At a time t a TCET can be *active*, if its entry task job has been activated and if its exit task job has not yet ended, or *inactive* otherwise. At time t, the **Task Chain State** is defined as $S(t) = \langle t_0, \tau_i \rangle$ with t_0 the oldest activation among active TCET, and τ_i the next task from this TCET to be executed. This way the task chain state indicates the remaining tasks to be executed on the chain and its current response time. Having an estimation of the *remaining Worst Case Response Time* ($rWCRT(\tau_i)$) at that moment for this TCET, the anticipation mechanism can figure out if we are

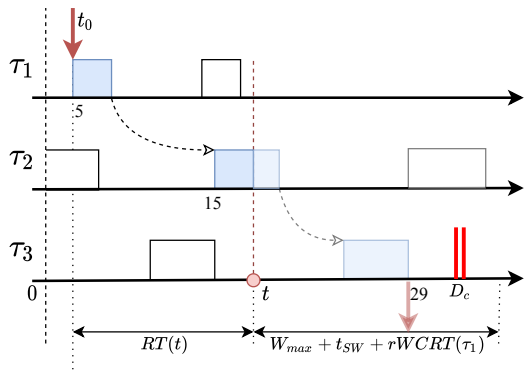


Fig. 2. active Task Chain Execution Trace with computed anticipation example

running into a potential deadline miss. For instance, at $t = 18$ on Figure 2, the task chain state would be $S(t) = \langle 5, \tau_2 \rangle$. With this $S(t)$ at a given time t, we have the current chain response time $RT(t) = t - t_0$ and the remaining response time $rWCRT(\tau_i)$ estimation (i.e. $\tau_2 \rightarrow \tau_3$ remaining) to check if the execution can finish in time.

Obviously, the estimation of $rWCRT(\tau_i)$ is an important element of the approach. It can be done either experimentally or analytically. We choose an experimental approach for our experiments as the analytical approach is intractable for complex application on a modern multi-core processor or would imply an overly pessimistic estimation. Details of the experimental protocol used for this estimation is given in section III. This estimation is made during system integration, without the LO-criticality tasks, thus $rWCRT(\tau_i)$ estimates the worst case time remaining before the end of the task chain if executed in HI-criticality mode.

To decide if it is safe to continue in LO-criticality mode, the anticipation mechanism periodically checks the task chain state. Each observation at a time t is considered temporally safe if the following inequality (adapted from [6]) holds:

$$RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW} \leq D \qquad (1)$$

where W_{max} is the worst time between each observations and t_{SW} the latency to switch to the HI-criticality mode. Let us assume that (1) holds, we show that it is safe to wait for the next observation to decide if there is a need to switch. Let t_{next} the time of the next observation. By definition, $t_{next} \leq t + W_{max}$ then necessarily $RT(t_{next}) \leq RT(t) + W_{max}$, thus $RT(t_{next}) + rWCRT(\tau_i) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$. Also, $rWCRT()$ can only decrease as time passes, so $rWCRT(t_{next}) \leq rWCRT(\tau_i)$ and $RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq RT(t) + rWCRT(\tau_i) + W_{max} + t_{SW}$. Since (1) holds, we have $RT(t_{next}) + rWCRT(t_{next}) + t_{SW} \leq D$.

Hence, it will be safe to switch to LO-criticality mode at the next observation. The setting of the W_{max} parameter is discussed in the next section.

C. Monitor & Control Agent Architecture

Most of our architectural choices have been made to facilitate portability and deployment of our solution. To that end, the MCA intervenes on the task at the highest possible abstraction level and does not require alteration of tasks code or binary.

To help with the estimation of $rWCRT$, we assume that the HI-criticality task chain execute on a single core. To avoid interference between the MCA and the task chain we prevent the MCA to use the same core. Lo-criticality tasks can execute on any core as depicted on Figure 3.

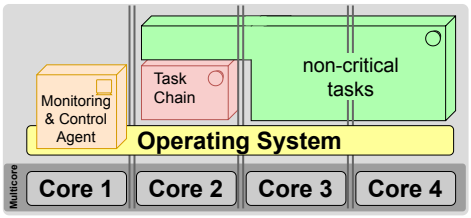


Fig. 3. Monitoring & Control Agent basic concept

The Monitoring and Control Agent is made of two components: a *Task Wrapper Component* and a *Core Control Component* as shown in Figure 4.

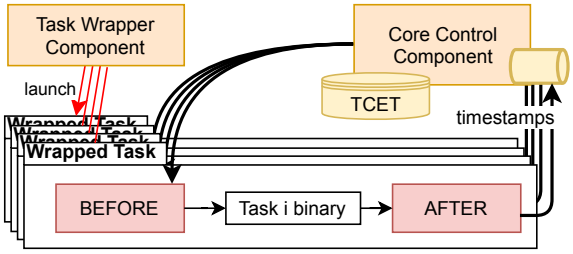


Fig. 4. Monitoring & Control Agent Architecture

1) Task Wrapper Component (TWC): It is responsible for encapsulating the system tasks between two software wrappers, "Before" and "After". Those wrappers have two roles:

- provide timestamps (start and end of HI tasks) to the Core Control Component.
- prevent LO tasks execution in HI-criticality mode.

The timestamps are queued to be processed by the Core Control Component to update the TCET. The "Before" wrapper is also used to prevent LO task execution in degraded mode. There is no need for an "After" wrapper for LO tasks.

2) Core Control Component (CCC): The Core Control Component executes with a period T_{ccc}. It updates each **active Task Chain Execution Trace** (TCET), taking into account timestamps received since its last execution and compute the task chain state $S(t)$, enabling the evaluation of $RT(t)$ and $rWCRT(\tau_i)$. Then CCC checks if inequality (1) is still true. If not, the CCC switches to degraded mode to guarantee the task chain deadline. The mode switch is realised through two actions: sending a Pause signal to every LO-criticality tasks, and signaling "Before" wrapper to prevent any new execution.

The CCC parameters t_{sw} and W_{max} are important to define. If those parameters are underestimated, then it is not safe to use inequality (1). We estimate them for our experimental platform in subsection IV-B. W_{max} is the maximum duration between two CCC checkpoints. It is directly dependent to the CCC period T_{ccc}. If Hi-criticality tasks are periodic, which is typical, it is simple to set this value, around the smallest task period. This way we have the guarantee of not overflowing the timestamps queue used by the CCC. A greater value is possible, but we must take care to process the $TCET$ updates faster than the arrival of timestamps. For other tasks activation models, we must identify the highest task timestamps arrival rate to avoid any queue overflow. It is also important to set T_{ccc} –and thus, W_{max}– as it will directly influence the sensitivity of our anticipation mechanism. With a higher CCC update frequency –and consequently a lower W_{max}– we switch to degraded mode later. Also, it will naturally use more computing resources. A higher value triggers sooner and may increase the number of unneeded switches to degraded mode (i.e. false positives).

3) Evaluation: We will evaluate the MCA on three criteria : efficiency, performance and quality.

Efficiency gives the ratio between the number of successful executions and the total number of executions (including those leading to a deadline miss). For a hard real-time task chain, efficiency needs to be 100%. For a soft real-time one, a lower value may be acceptable.

Performance is about the CPU time allowed for the LO-criticality tasks. It is given by $T_{nom}/T_{tot} = 1 - T_{deg}/T_{tot}$, with T_{tot} the run-time duration, T_{nom} the time spent in nominal mode and T_{deg} time in degraded mode. The more time spent in nominal mode, the better.

Quality, by the number of switches to degraded mode N_{sw} compared to the number of deadline misses N_{miss} if the mechanism was not active. N_{sw}/N_{miss} must be > 1 for the mechanism to be 100% effective but as close to 1 as possible to be qualitative, i.e. avoid the false positive cases. A score under 1 means the CCC was not perfectly calibrated for full temporal fault coverage and consequently, the mechanism does not anticipate all the temporal faults.

III. Framework and Experimental protocol

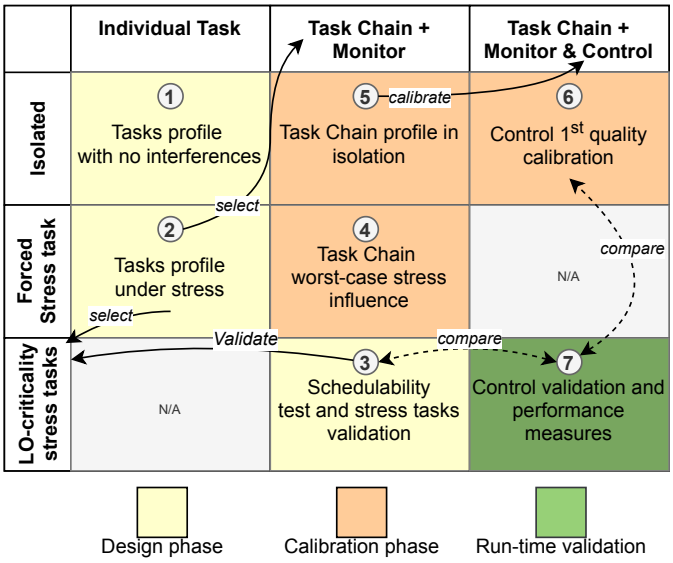
A. Objectives

This section describes the framework we developed to test such Monitoring & Control Mechanism presented in section II and the experimental protocol we follow to configure and test the mechanism. Each stage of the protocol has its own objective. Some of them can be passed depending on the system information already defined upstream.

The combination of such software framework & experimental protocol aims at several objectives:

- Quantitative analysis of **a)** end-to-end timing guarantees, **b)** available computing resources for LO tasks, **c)** computing resource overhead
- Sensitivity analysis to variation of the framework parameters and different task sets (different type of tasks, ratio of HI/LO tasks, average CPU load)

TABLE I
Experimental Flowchart



B. Experimental protocol

We present in this section the experimental protocol proposed to characterise the system tasks (the "workload") and calibrate the Monitoring and Control agent. The experimental protocol is divided in 7 steps separated in 3 phases : 1) Design phase, 2) Calibration phase, and 3) Run-time validation phase as summarized in Table I.

1) Design phase: This phase is needed if the workload involved does not come with a detailed specification, including their behavior and execution times. This is the case of our experiments as we select tasks from an already existing benchmark and we have no information about tasks execution times or even their compatibility with our real-time environment. Thus this phase is to characterise the available task set and define the workload specifications. It will be split into the HI-criticality task chain and LO-criticality tasks with their characteristics (min/avg/max execution time, periodicity...). This phase is defined by steps ①, ②, ③ in Table I.

a) Task profile with no interferences: First objective is to get a global idea of tasks execution time profiles. One experiment is made per task, the task being executed individually with the framework. The task is called periodically with a given input, and task response times are logged.

b) Task profile with forced stress: We add to the precedent step an artificial system load to cause high stress on cache, memory, I/O and computing use while the tasks are executed one by one. The output is a table with a profile for each task made of the min/average/max execution times and system metrics (system calls, context switches, scheduling interrupts, eventual period misses...). Such profile allows to categorise the tasks following their sensitivity to interferences compared to previous step ①. This allows to define which tasks can be used for the HI-criticality task chain or as stressing LO-criticality tasks but also discard any task that would not fit our needs.

c) Task Chain with realistic interferences: Previous step classified the task set between HI and LO-criticality tasks. We define on this step the specific task chain and LO tasks that will be studied next and verify the pertinence of such choice. We check the workload schedulability in the soft real-time sense (i.e. schedulable if deadlines tardiness are bounded by a reasonably small constant). We also measure the task chain response time profile under "realistic" conditions without the Control mechanism enabled. Expected result is a schedulable system with reference task chain response times with interferences.

2) Calibration phase: This phase is mandatory to configure the Control mechanism to the software and hardware specificities and lower false-positive rate. It is made of steps ④, ⑤, ⑥ in orange boxes of Table I. Configuration includes task chain worst-case response time and intermediary response times in isolation. Performance optimisation consists in tweaking the switch time t_{sw} and anticipation execution frequency W_{max} constants, in the objective of lowering false-positive anticipation rates.

a) Task Chain with interferences: The task chain is then tested under a worst-case scenario. It is executed with the artificial system load, to stress as much as possible the task chain similarly to step ②. We get a baseline of the worst-case chain response time. This value is important because if the end-to-end deadline is always greater than the worst-case response time observed then the mechanism would be of no use (i.e. deadline never broken from temporal faults). This step gives a quantification of the task chain sensitivity to interferences and thus indicates the pertinence of using a Monitoring and Control Agent to manage them.

b) Task Chain profile with no interferences: The objective is to calibrate Control mechanism parameters : $rWCRT(\tau_i)$, Core Control Component period (T_{CCC}) and switch time (t_{sw}) to degraded mode. The task chain is executed alone with the MCA but with the Control mode switch disabled. We log every chain intermediary and end-to-end response times. The result gives the data of all the remaining response times obtained during the test. We set the $rWCRT(\tau_i)$ parameters as an upper limits of the remaining response times registered.

c) Task Chain with Control mechanism enabled: Finally, the Control mechanism is enabled, with the parameters set on previous step. As this step does not include the LO tasks that bring interferences to the task chain, the Core Control Component should not trigger any switch to degraded mode. This step is important for the final analysis as it already points out the base false positive rate obtained with chosen parameters. A qualitative MCA should have the least degraded mode switch possible. Otherwise it could mean that either the CCC parameters are not ideally set (typically W_{max}), or the expected timing delays caused from interferences are too close to the usual timing variation of the task chain execution even in isolation. In other words, the Control Component is not able to differentiate response time variations due to temporal faults from ones due to nominal execution time variations. Another possibility is the end-to-end deadline requirement is too close to the nominal end-to-end response time in isolation.

3) Run-time validation phase:

a) Task Chain with Control mechanism and realistic interferences: The validation phase implies a last step (⑦ in green box of Table I), which is with the whole final system being executed : HI task chain and LO tasks with the MCA enabled. The objective is to collect the concluding information on the Monitoring and Control Agent behavior to measure the 3 quantification criteria (efficiency, performance and quality) of the solution explained in subsubsection II-C3. We also use the data from steps ③ and ⑥ as a reference for the conclusions.

C. Implementation Framework

1) Hardware: The platform used for the experimentation is a barebone computer equipped with a processor Intel Core i5-8250U. This processor embeds 4 cores. It has 3 caches level, L1, L2 and L3 (shared), with respectively 32 KiB/core, 256 KiB/core and 8 Mib (shared). We fixed its frequency to 1400MHz and disabled hyper-threading for our tests.

2) Operating System: We used Linux (Linux Mint xfce 18.04, kernel 4.18.1) to mix general purpose and real-time applications with different scheduling policies ([7], [8]). Its versatility grants easier compatibility with benchmarking suites. Moreover, by adding Xenomai (v. 3.1) real-time co-kernel [9], it is possible to get closer to real-time applications with latencies lowered from milliseconds down to microseconds. It also grants an API for real-time application development, used for the MCA framework.

Such OS configuration allows us to specify a per-task core allocation and priority level. Linux scheduler as explained in [10] selects tasks first by priority level, (from 1 to 99 for real-time tasks domain). Then for a given priority level, multiple scheduling policies are possible: Global Earliest Deadline First, FIFO, Round-Robin, and other best-effort policies. To test a system using classic Round-Robin for instance, every task are launched at same priority level with Round-Robin policy. We use Rate-Monotonic scheduling policy for our tests this way.

3) Use Case Software: MiBench [11] plays the role of the task set to constitute our experimental workload. This benchmark suite gives source code for 30+ standalone binaries classified in six domains : automotive, security, network, telecommunication, office and consumer. Those tasks do different jobs similar to ones in these domains, with different levels of complexity that is of high interest for us.

To run an artificial system load as a "worst-case" cache, memory, CPU use and I/O stress, we use Linux *Stress-ng* tool presented in [12].

Thus we made an experimental platform with choices driven to both compensate the lack of a concrete industrial use-case and allow versatility in the experiments parameters in order to get as close as possible to an industrial automotive platform based on a multicore processor. This is summarised in Figure 5, the experimental platform has three input domains of which two are configurable. First input domain gathers the basic

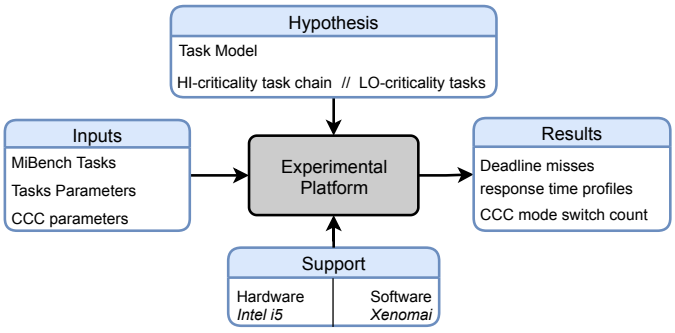


Fig. 5. Experimental Platform structure

hypothesis. The two others are the tasks inputs (task set, tasks parameters and MCA configuration) and the execution support (hardware and software presented above) for which different choices can be made. Based on this platform, we define an experimental protocol to get from the system specifications to the final implementation, with a calibrated MCA.

IV. Experimental Validation

A. Design phase

Using Mibench as a workload had advantages but also drawbacks. It allows to get specific tasks with a defined and already studied behavior but we are dependent on the way they are initially programmed. They might not completely fit our needs to simulate embedded applications or have incompatibilities with the chosen real-time environment. First step in using this benchmark is to check those criteria to select precisely the tasks from MiBench we use.

1) Task profile with no interferences: We need to establish the execution time profile of each task of the bench. As a result some tasks will be removed from the tests, either due to execution time magnitude differences or inconsistent behaviors between experiments. Accordingly, we measure on each experiment the min, max and median execution times, but also some system counters as the Xenomai mode switches and the amount of Linux system calls. Without interferences, the execution time characteristics should have low variations. We see in Table II a sample of the tasks characteristics collected, for 3 different profiles.

TABLE II
Tasks profiles in *Xenomai* environment

Task	execution times (ms)		System Counters	
	Median	Max	Mode Switch	Sys. Call
Patricia	0.026	0.099	10051	10338
FFT	7.36	7.39	58	2343
rijndaelE	140,11	141.81	158	446

With such data, we identified the majority execution time range in MiBench task set around 10ms (from 2-3ms to 20-30 ms) and the basic system calls and mode switch amounts due to initialisation phase (respectively 58 mode switches and ≈hundreds of system calls).

Consequently, we discard tasks out of the execution time magnitude like *adpcmCaudio_L* with an average execution time of 432 ms. By the end of step ①, we retained 34 tasks.

2) Task profile with forced stress: We add stress on cache level and communication bus from previous step experiments. The objective is to discriminate our tasks in two groups depending on their reaction under stress. If it increases execution time too significantly (more than x10 from average time in isolation) it means the tested task is not suited for the tested environment and suffers not only from interferences but also from LO-criticality tasks preemption. A significant increase in mode switches also indicates such behavior. The tasks that do not pass correctly this test will be either ignored or used LO-criticality stress tasks. Tasks without an exploding execution time or huge increase of mode switches will be used to generate the HI-criticality task chain. Execution time profiles of task used for this purpose are in Table III. We finally retained 22 tasks at the end of step ②.

TABLE III
Tasks profiles in *Xenomai* environment

Task	execution times isolated		execution times stressed	
	Median (ms)	Max (ms)	Median (ms)	Max (ms)
djpeg	1.97	2.28	19.91	211.53
rjindaelD	8.80	9.77	35.02	526.33
FFT	1.85	1.86	2.03	14.8
FFT^{-1}	3.56	3.57	4.05	19.74
bitcount	8.36	9.52	9.98	45.18

3) Task Chain with realistic interferences: At this point, we defined our task set, composed of the LO-criticality tasks used as "real" stress and the task chain made of 5 tasks : $FFT \rightarrow Bitcount \rightarrow Basicmath \rightarrow FFT^{-1} \rightarrow sha$. We need to verify the validity of our choice in term of schedulability and effectiveness of the LO-criticality tasks as interferences. Executing the whole task set together allow to verify both for this step ③.

The right part (red) of Figure 6 shows the task chain response time distribution profile with the full workload executed (i.e. LO-criticality tasks included). We see the perturbation due to the LO tasks on the critical task chain execution. Our workload is schedulable (no execution drops and deadline misses have reasonable overheads) and the task chain meets high response times compared to its average "nominal" response time for $\approx 10\%$ of the executions (above 200ms response time). We arbitrarily define the task chain deadline $D = 160$ms.

B. Calibration phase

This phase is dedicated to configure the Core Control Component parameters ($rWCRT_i(\tau_i)$, t_{sw} and W_{max}) and run the reference experiments of the task chain behavior on a worst-case stress context (step ④).

1) Task Chain with interferences: In this part we use *Stress-ng* to simulate a worst case stress condition. The task chain potential worst case response time in this context raises at 300ms. Such increase by 100% of the max chain response

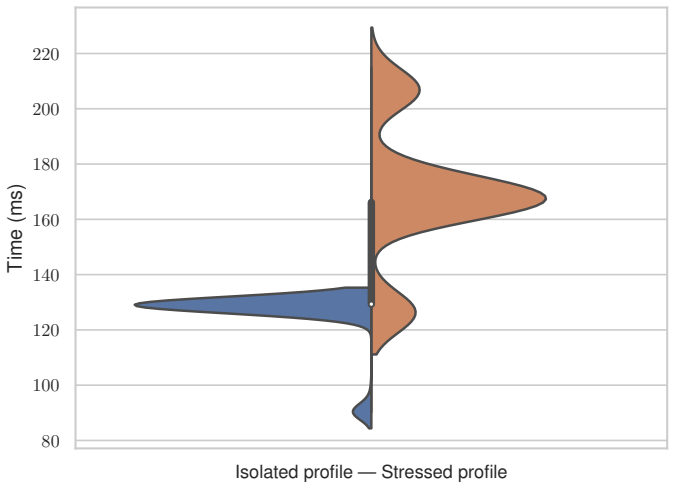


Fig. 6. Task chain end-to-end rWCRT distributions

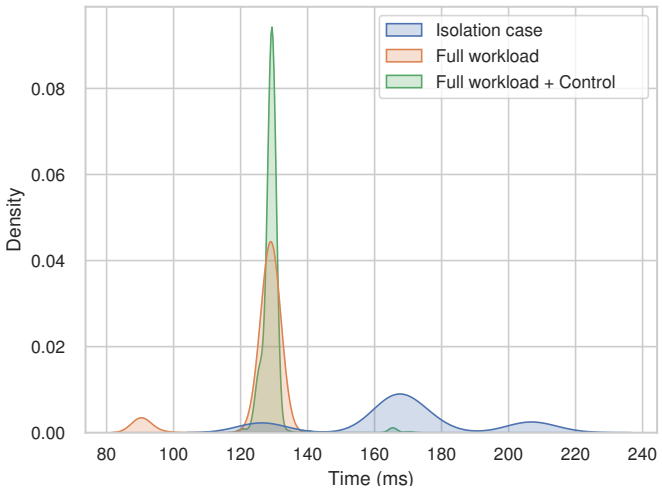


Fig. 7. Task Chain response time profile from steps ③, ⑥, ⑦

time under this scenario indicates the pertinence of using a MCA. Regarding such result, our workload stresses the task chain in a significant magnitude.

2) Task Chain profile with no interferences: For step ⑤, we execute the task chain in isolation (i.e. degraded mode). Execution time profile is on the left part (blue) of Figure 6. We calibrate the Monitor & Control mechanism parameters. We need the different $rWCRT$s for each value of τ_i as defined in subsection II-B. For such linear 5-task chain we logically have $i \in \{1, 5\}$. At run-time, the remaining response times are logged in degraded mode, i.e. the task chain in isolation, and we keep an upper value of the worst measured remaining response time for each τ_i as its $rWCRT(\tau_i)$ in Table IV. Finally, regarding previous results from step ③, we set $W_{max} = 1$ms, and $t_{sw} = 500\mu$s for our platform.

TABLE IV
TASK CHAIN $rWCRT(\tau_i)$ VALUES IN DEGRADED MODE

$rWCRT$	τ_0	τ_1	τ_2	τ_3	τ_4
time (ms)	129	93	68	49.5	25

3) Task Chain with Control mechanism enabled: With the previous calibration, we can execute the task chain alone with the Control mechanism enabled. In this isolation case, we should see almost no switch to degraded mode (and on a perfect case, no switches at all) as they must be false-positive. This experiment allows to validate the parameters set on the previous step. On our tests, we measured 0.3% of false positive triggers to degraded mode. The task chain in degraded mode response time distribution profile is illustrated in Figure 7.

C. Run-time validation phase

As a final experiment, we test the complete workload (HI and LO tasks) with the Monitoring & Control Agent enabled and configured from previous step. First we observe the MCA CPU use, that is inferior to 1%. For a 120s long experiment, it ran for 1.3s overall (including setup time). We were not able to find any difference regarding CPU percentage use with and without our mechanism, either with a big task sets (small tasks only, CPU usage around 80% displayed) and with smaller task sets (e.g. only the task chain described above). Such footprint is low enough to include easily such mechanism.

In term of **efficiency**, our MCA prevented every task chain execution over a 170ms response time. Only 6 occurrences (0.1%) missed the deadline set at 160ms. The MCA brought down the average response time of the chain from 168ms (no Control enabled) to 129ms. Such value is way closer to the average task response time profile in isolation (125ms). The few missed deadlines can be explained by the implementation framework we used, with a workload (MiBench tasks) not fully compliant with real-time programming constructs recommendations that causes uncontrolled Linux system calls for instance. In conjunction with the exacting deadline we arbitrarily set at 160ms while the general workload is demanding (generating 84% deadline misses without the MCA in step ③), this explains this non-perfect result. We could use more pessimistic $rWCRT(\tau_i)$ values to achieve no deadline misses, at the expense of a worse result on the quality criteria. By the end it is a question of compromise, depending on the specific needs.

The **quality** of our calibration seems promising as there were less switches to degraded mode with the Control enabled than the number of deadline misses with no Control at all. This implies that preventing a deadline miss had a more general impact reducing the overall number of timing faults.

In term of **performance**, the system maintained LO-criticality mode for 82s / 120s total, i.e. a performance factor of 0.69 for a loss of 31% of the time in degraded mode.

All those metrics are promising for the use of a Monitoring and Control Agent in order to change a chain response time at an optimum value to avoid the great majority of the deadline misses and on the same time still take few compromises on the LO-criticality tasks execution.

V. Related Work

Current industrial applications usually separate critical and noncritical software to avoid the complexity inherent to mixed-criticality systems. But with the need for more computing resources and the systems complexity raising, MCS is necessary. Consequently, industrial tend to skirt temporal faults inherent to MCS by completely isolating (in time and space) HI and LO-criticality executions. This is done for instance with PikeOS hypervisor [13] that allows MCS certification for rail industry. Other solutions relies on WCET estimations that are function of the criticality level with larger values for higher criticality tasks. But in a multicore context, those WCET are overly pessimistic. Domains like avionic [14] and automotive [15] uses such method with static scheduling to prevent any consequences from interferences. The overall consequences of those solutions are an under-use of computing resources.

Other solutions exist to manage real-time constrains on multicore mixed-criticality system in more flexible ways. We can distinguish approaches based on task allocation and scheduling policies [16],[17], [18],[2], resource management policies [19],[20],[21],[22]. Some solutions also convene multiple vectors of action at the same time, typically [23],[24].

Overall, those solutions tackle mainly one side of the problem to either guarantee real-time constrains or optimise cores utilization. Otherwise, solutions are focused on a more specific sub-part of the problem, typically with different or more precise hypothesis on the scheduling policy or the task model for instance. Our solution is the first one trying to take advantage of slack time to allow some execution times overheads to stay in Lo-criticality mode with a guaranty on HI-criticality tasks. Xu & al.[5] had such approach in a uni-processor basis only.

VI. Conclusion

As a conclusion, we opened the way to leverage task-chain based monitoring as a new solution to guarantee real-time constraint while taking more benefits from the multicore computing resources. We presented a mechanism to monitor HI-criticality tasks and control LO-criticality tasks execution in a MCS to find a compromise between high computing resource use and real-time constraint guarantees. Our high-level approach allows non-intrusive integration for industrial application, based on experimental calibration.

Our Monitoring and Control Agent was applied to a case study. It followed an experimental protocol describing the requirements from Design to Implementation phase for its calibration. We quantified our solution capabilities based on the 3 factors of **a)** Efficiency to guarantee real-time constraints, **b)** Performance of giving CPU time for the LO-criticality tasks, **c)** Quality of the anticipation mechanism to limit the false-positive degraded mode switch rate. Our MCA allowed a drastic reduction of deadline misses despite the unfavorable soft real-time environment but still allowing significant execution of LO-criticality tasks with a low rate of false-positive switches to degraded mode.

Our objective today is to extend this work to multiple HI-criticality task chains simultaneously. Our current implementation already manages it, the challenge relies on the anticipation computing to take into account a more complex degraded mode with more concurrent tasks still active. Addressing this problem could lead to extend the solution to more than a dual-criticality system, with a task chain per criticality level and its associated degraded mode. Our on-going work is investigating an intermediate degraded mode that does not relies on task pausing, but based on hardware cache partitioning for instance.

References

[1] S. Baruah, V. Bonifaci *et al.*, "Scheduling real-time mixed-criticality jobs," in *IEEE Transactions on Computers*, vol. 61, 2012, pp. 1140–1152.

[2] A. Kritikakou, T. Marty *et al.*, "DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems," *ACM Tran. on Design Automation of Electronic Systems*, vol. 23, 2017.

[3] J. C. Palencia and M. G. Harbour, "Schedulability analysis for tasks with static and dynamic offsets," in *19th Real-Time Systems Symposium*. IEEE, 1998, pp. 26–37.

[4] J. Zhe, "How to Build a Mixed-Criticality System in Industry," in *7th International Workshop on Mixed Criticality Systems*, Hong-Kong, 2019.

[5] B. Hu, L. Thiele *et al.*, "FFOB: efficient online mode-switch procrastination in mixed-criticality systems," *Real-Time Systems*, vol. 55, no. 3, pp. 471–513, 2019.

[6] A. Kritikakou, C. Pagetti *et al.*, "Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems," in *26th Euromicro Conference on Real-Time Systems (ECRTS14)*. IEEE, 2014, pp. 119–128.

[7] C. S. Wong, I. Tan *et al.*, "Towards Achieving Fairness in the Linux Scheduler," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 34–43, 2008.

[8] J. Lelli, G. Lipari *et al.*, "An efficient and scalable implementation of global EDF in Linux," *7th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT'11)*, 2011.

[9] P. Gerum, "Xenomai - Implementing a RTOS emulation framework on GNU/Linux," Xenomai, Tech. Rep., 2004.

[10] N. Ishkov, "A complete guide to Linux process scheduling," 2015.

[11] M. R. Guthaus, J. S. Ringenberg *et al.*, "MiBench: A free, commercially representative embedded benchmark suite," in *4th International Workshop on Workload Characterization*. IEEE, 2001.

[12] C. I. King, "[Online] https://kernel.ubuntu.com/~cking/stress-ng/."

[13] S. Fisher, "Certifying Applications in a Multi-Core Env.: The World's First Multi-Core Certification to SIL 4." SYSGO AG, White Paper, 2013.

[14] P. J. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)," in *IEEE/AIAA 27th Digital Avionics Systems Conference*, 2008.

[15] AUTOSAR, "Timing Analysis," *Standard Release 4.3.0*, 2016.

[16] D. Tamas-Selicean and P. Pop, "Design Optimization of Mixed-Criticality Real-Time Applications on Cost-Constrained Partitioned Architectures," in *32nd IEEE Real-Time Systems Symposium*, 2011, pp. 24–33.

[17] H. Xu and A. Burns, "A semi-partitioned model for mixed criticality systems," *Journal of Systems and Software*, vol. 150, pp. 51–63, 2019.

[18] J. L. Herman, C. J. Kenna *et al.*, "RTOS Support for Multicore Mixed-Criticality Systems," in *18th Real Time and Embedded Technology and Applications Symposium*. IEEE, 2012, pp. 197–208.

[19] B. C. Ward, J. L. Herman *et al.*, "Making Shared Caches More Predictable on Multicore Platforms," in *25th Euromicro Conference on Real-Time Systems*. IEEE, 2013, pp. 157–167.

[20] V. Izosimov and E. Levholt, "Mixed Criticality Metric for Safety-Critical Cyber- Physical Systems on Multi-Core Architectures," in *4th MEDIAN Workshop*, 2015.

[21] A. Blin, C. Courtaud *et al.*, "Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System," in *28th Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2016, pp. 109–119.

[22] G. Fohler, G. Gala *et al.*, "Evaluation of DREAMS resource management solutions on a mixed-critical demonstrator," in *ERTS*, Toulouse, 2018.

[23] M. Chisholm, N. Kim *et al.*, "Supporting mode changes while providing hardware isolation in mixed-criticality multicore systems," in *25th Int. Conference on Real-Time Networks and Systems*, 2017, pp. 58–67.

[24] J. H. Anderson, S. K. Baruah *et al.*, "Multicore Operating-System Support for Mixed Criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, vol. 4. Citeseer, 2009.