Safe Scheduling on Multicores: an approach leveraging multi-criticality and end-to-end deadlines

Daniel Loche, Michaël Lauer, Matthieu Roy, Jean-Charles Fabre

Safe Scheduling on Multicores: an approach leveraging multi-criticality and end-to-end deadlines

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

Michaël Lauer[†], Matthieu Roy[†], Jean-Charles Fabre[†]
[†] LAAS-CNRS
31400, Toulouse, France
Email: first.lastName@laas.fr

***Abstract*—Memory access duration on multicore architectures are highly variable, since concurrent accesses to resources by different cores induce time interferences. Consequently, critical software tasks may be delayed by non-critical ones, leading to deadline misses and possible catastrophic failures. We present an approach to tackle the implementation of mixed criticality workloads on multicore chips, focusing on task chains, i.e., sequences of tasks with end-to-end deadlines. Our main contribution is a Monitoring & Control Agent able to stop non-critical software execution in order to prevent memory interference and guarantee that critical tasks deadlines are met. This paper describes our approach, and the associated experimental framework to conduct experiments to analyze attainable real-time guarantees on a multicore platform.**

***Index Terms*—multicore, real-time, deadline, task chains, mixed-criticality**

I. Introduction

The automotive industry evolution is more and more software-oriented. Recent cars require resource demanding software, at least for Advanced Driving Assistance Systems (ADAS) implementation. Moreover, system requirements on energy consumption, weight and space of embedded architectures are calling for a drastic reduction of computing units. Combining both trends leads to consider multicore processors as a platform to run mixed criticality workloads on a single multicore Electronic Computing Unit (ECU) that integrates features previously distributed on discrete monocore ECU.

Yet, multicore architectures impose that software has to cope with execution interferences due to memory, cache overwriting, I/O & resource sharing and possible tasks synchronization. Such interferences imply that execution times are hardly predictable. In this context, safe and accurate Worst-Case Execution Time (WCET) estimation is either intractable or overly pessimistic. WCET estimation on multicore is still an open question. Thus, it is not possible to use typical scheduling analysis to prove that the embedded software can correctly execute according to some deadline. Furthermore, integrating several features on a shared ECU means creating interferences between software tasks from different criticality level. In such case, interferences from non-critical tasks may impair the temporal behavior of critical ones.

Future automotive computer systems correspond to sets of functions implemented as chains of tasks. Real-time constraints of functions are expressed as end-to-end deadline on the chains of tasks. To handle real-time constraints, chains of tasks need to comply with end-to-end deadlines. A function has an input (request, sensor...) and generates an output (reply, actuator...). For instance, manual braking corresponds to a function where a driver braking input must activate the braking system. To fulfill this function, a chain of software tasks is executed (from the sensor to the actuator), passing through computing and decision components. To execute safely, the time elapsed between an input of a function and its corresponding output, i.e. its response time, must always be lesser or equal to its specified deadline. Our primary goal is to ensure that these constraints are met for every critical task chains—braking is a good example for this. We observe that on a multicore, an isolated chain executed on one core could suffer from execution interferences even from tasks executed on other cores. That's why current solutions give small place to non-critical tasks.

Consequently, in a multi-criticality multicore the main problem is to respect such constraints with the least possible compromises on non-critical software. This is essential to get the most benefits from multicore processors.

Our approach takes advantage of mixed criticality software to satisfy highly critical task chains timing constraints in a multicore environment and at the same time give as much computing resources as possible to non-critical tasks.

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

- we use a Monitoring and Control Agent to check at runtime the execution of the critical task chains
- we anticipate potential response time problems with respect to end-to-end deadlines
- in case of a potential deadline miss, we fall back to a safe state by temporarily deactivating non-critical tasks

The efficiency of our approach relies on the efficiency of our anticipation mechanism. In order to be as accurate as possible, and so avoid unnecessary tasks deactivation, we explicitly handle end-to-end deadlines. This is a part of our contribution. Indeed, end-to-end deadlines are usually handled task by task, meaning that the tasks own deadline are considered. While simpler to deal with, such approach is pessimistic [1].

In this paper, we first present the concept of our embedded Monitoring and Control Agent, from its definition to its generic architecture. Then we propose an evaluation framework and a tool to execute a task set on a real physical multicore platform and collect measurements.

Ongoing experiments are run to analyze the behavior of our system regarding tasks allocation, scheduling policies, task set characteristics and deadline satisfaction.

II. State of the Art

Current Operating Systems are dedicated either to guarantee real-time constraints with Real-Time Operating Systems (RTOS) [2], [3], [4] or to maximize resource use for a wide range of applications with General Purpose OS (GPOS) [5]. Few solutions try to mix both objectives with a more or less satisfactory result [6],[7],[8].

A. Real-time OS

In industrial applications, uncertainty factors are avoided as much as possible otherwise complexity is too high for debugging, to detect problems and avoid unexpected behavior that could lead to critical failures.

Generally, the management of real-time constraints relies on static scheduling with time slot reservation for each software partition (Time Division Multiple Access scheduling). Worst-case execution time (WCET) analysis are recommended by various safety standards like ARINC 653 in Avionics to ensure real-time constraints.

Following only such logic, multicore-based solutions needs much more compromises on tasks execution. For instance, PikeOS hypervisor [2] obtained highest certification level for rail industry on a dual-core platform using temporal partitioning. Execution time is divided into slice resources and applicative software is separated into multiple partition resources, associated to external resources (I/O etc..). Associating time resources with partition resources, PikeOS ensures time and space isolation between critical and non-critical software, thus preventing failure consequences from one partition to the other. For instance, a critical resource partition can be affected alone to a temporal partition in order to avoid interferences with other resource partitions. This kind of method is effective but by design over-reserves execution time resources. It can answer well some needs as in avionics [4] or railways domains where it is possible to afford computing resource over-fit. Automotive industry uses AUTOSAR OS [3] where processors are not used at their full potential, in the favor of real-time constraints control. As they are trying to gather more and more software of different nature to reduced amount of computing units, such design is no longer appropriate without further enhancements as it does not take into account the risks from the coexistence of different software domains and possibly multiple OS. Consequently, there is a huge need to exploit more effectively the computing resources but still guaranteeing critical constraints.

All-in-all, for multiple tasks, even independent with each other, running concurrently on a multicore processor, a Worst-Case scenario includes maximum contention over memory access with full cache miss, but also maximum contention on the core's utilization and so on. This leads to an unrealistic WCET estimation that could never occur for most of the applications. Also, analytical approaches or formal computation of such values are complicated on a multicore as they are over-pessimistic and time-consuming. This leads to underused resources to have strict timing guarantees when such solutions are used. However, it is admitted that WCET are much easier to estimate on a monocore architecture. Real-time deadline can be ensured in this case. Highly critical systems are mainly based on monocore for this reason, when computing resource and cost are part of the design limitations.

B. General purpose OS

On the other hand, General Purpose Operating Systems bring the opposite pros and cons. Scheduling policies are designed to run lots of different kind of tasks, from highly interactive ones to background tasks. GPOS schedulers reached a complexity level high enough to get inexplicable behavior [9]. These systems are highly versatile in the variety of applications they can run at the expense of predictability. Linux is more and more used for embedded applications and it is possible to run both classic Linux processes and real-time ones. However, no real-time requirements can be strongly guaranteed with a vanilla kernel. Linux scheduler evolved a lot, even recently, and offers multiple scheduling strata to use, from the Completely Fair Scheduling [5] (CFS) for common process, to Round-Robin (RR) and Earliest Deadline First (EDF) for real-time tasks as described in [10]. Vanilla Linux has latencies at around hundred ms, but some patches reduce it down to micro-seconds. Comparisons can be found in [11] and [12]. In fact, Linux is even under studies to see to what extent it is possible to use it as a real-time OS [13], under certain constraints. The advantages of such perspective would be significant. This led us to choose Linux to implement our first proof of concept. This way we can take benefit from its versatility (software compliance, monitoring and configuration capacities) and we can add software tools following guidelines as those mentioned in [13] complemented by our mechanism to guarantee real-time constraints.

III. Monitoring & Control Agent

A. Concept Description

Our approach presents a software execution *Monitoring and Control Agent (MCA)* to guarantee end-to-end deadline constraints. We focus on the respect of end-to-end constraints of tasks chains, not individual tasks constraints. The idea behind this is to offer more "flexibility" on tasks scheduling for guaranteeing mandatory task chains constraints if we control only end-to-end constraints instead of every critical task timing constraint. By doing so, we gain "flexibility" as we allow some parts of the chain to be behind time as they can be compensated before the end of the chain without any external action. The MCA monitors at run-time the execution time of critical tasks and anticipate when the end-to-end deadlines may be compromised to stop non-critical tasks when needed in order to avoid such risk. The anticipation is based on the estimation of remaining WCET. Finally, when the critical task chain recovers from the potential risk, the non-critical tasks can resume their execution to get back to a nominal state.

We define a *degraded mode*, opposed to the *nominal mode* of execution. In nominal mode, critical and non-critical tasks are executed normally. In Degraded mode, non-critical tasks are not executed, to prevent further interferences on critical tasks. The degraded mode implies simpler WCET estimations because we eliminate the disturbances from non-critical tasks; such WCET will be lower than in a nominal mode. It is probably less pessimistic as we eliminate memory interferences, non-critical tasks scheduling and possible common resources (drivers for instance) usage. The main disturbances remaining will be only between the tasks from the chain. Consequently, our anticipation mechanism will be based on reduced estimation of WCET (compared to nominal mode), to activate degraded mode only as a last resort.

To reach degraded mode, MCA role is to pause/stop non-critical tasks execution. This control is triggered by an anticipation algorithm. To be efficient, this algorithm should trigger the control at the latest possible time while guaranteeing real-time end-to-end constraints.

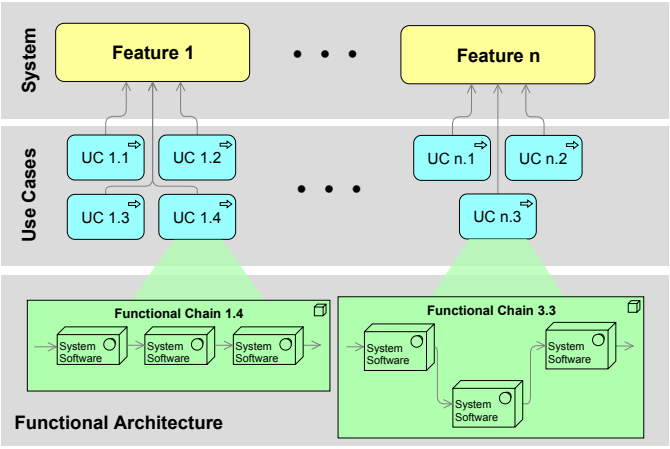

Fig. 1. Functional Architecture definition

1) Functional Specification: A critical task chain must describe the implementation of a system functionality from its triggering to its consequence. This would stick most of the time with a computing chain going from a sensor measure to an actuator command. First idea would be to stick with safety criticality levels (ASIL D to ASIL A and QM, for automotive applications), but we quickly notice that there is no direct link between this classification and critical tasks chains. A safety critical task is not necessarily defined from its timing constraints. The only possible conclusion here is that a critical task chain only includes non-QM tasks.

We propose here a definition based around high-level specifications as represented in figure 1. The global system is defined as a set of features[1]. Every feature gathers a set of functionalities that are translated into Use Cases[2]. A Use Case defines a feature behavior for a given context and inputs (and the consequent outputs). Finally, those are translated into functional chains representing different functions and their interactions needed for the realization of the Use Case.

If we combine this information with a severity classification in case of failure of the use cases, it is possible to define critical chains as functional chains with a high severity risk. This is one possible criterion allowing an easy separation between a critical functional chain and the others. It could be adapted during the design phase, depending on the functional chains allocated to the processor.

Such information allows to define the software components involved in the critical task chain. All the software components used to realize a critical functional chain form a critical task chain at an OS point of view. At this point, it is possible to define the task chain end-to-end deadline, following the severity temporal risk in case of failure. Such deadline should be at minimum the sum of individual tasks deadline, but could probably be higher, depending on the global system and the task chain function. Our objective is to guarantee such critical task chain end-to-end execution time on the multicore.

2) Critical Task Chain specification: A task chain can be written as a set of n critical tasks τ_i, $i \in \{1..n\}$, with τ_1 being the *entry* task of the chain and τ_n the *exit* task. A task chain response time is the duration between the call to the starting task, to the end of execution of the chain ending task. The task chain end-to-end deadline then defines naturally the maximum allowed response time.

A critical task τ_i in a chain is defined by a set of precedence constraints represented by a list of tasks in the chain that should be executed before it. Only the starting task τ_1 has no precedence constraint, and this way will always be the first valid element of the chain. Note that we do not make any assumption about the task's activation and interaction model. It can be implemented by interrupts (triggers from precedent tasks) or simply periodic. We only consider a task from the chain as valid if its execution respects strictly its precedence constraints. This assumption supposes that when a task starts after the end of its precedent task in the chain, then the output data from it are ready for use. A change in the task model or in the way the precedence constraint is taken into account would require adjustments only in how we monitor the task chain state. Critical tasks have the following parameters:

- Priority level
- Core allocation - tasks are executed on specific cores, to avoid migration
- activation policy (e.g. periodic)
- Deadline

3) Non-critical Tasks specification: For remaining tasks, prerequisites are only a static Core allocation and an activation model that can be either on the same core as the critical task chain or another core. We need to be able to switch to a *degraded mode* by disabling all non-critical tasks and let only critical tasks running. The goal is to guarantee WCET for the critical task chain when executed in degraded mode. Such

[1]Features: all the services the system must provide. e.g: Lane Support System (LSS) is a feature.

[2]e.g: Lane Departure Warning & Lane Keeping Assist are part of the use cases of LSS feature.

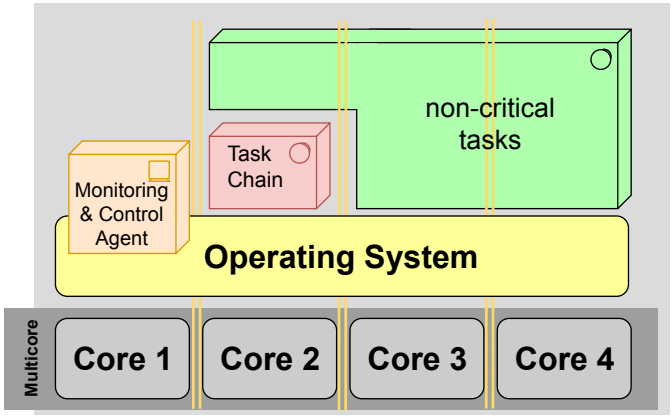

Fig. 2. Monitoring & Control Agent basic concept

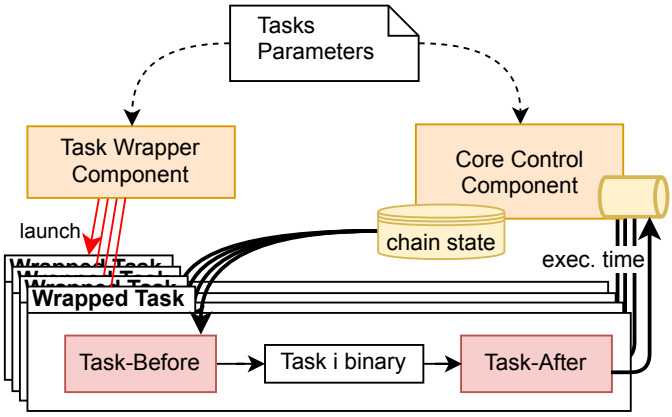

Fig. 3. Monitoring & Control Agent Architecture

analysis is to be done off-line and we describe a first method in section IV-C1- Framework Setup.

B. Additional assumptions

As a first step, we study the case of a single task chain coexisting with other (non-critical) software. Further study will aim at managing more task chains on the same multicore. We suppose that every task from the task chain is allocated to the same core. From a safety point of view, gathering the whole task chain on the same core is to some extend pertinent as it isolates it from other less critical software. Also, as we consider mostly chains without internal parallel computing, the gain from spreading the critical tasks in different cores would not give much benefits for the chain response time. Finally, such grouping allows to consider a degraded sub-system with only the critical tasks running on one core, that could be considered as a single core for Worst-Case Execution Time estimations as we will see later in the paper.

Finally, we need an isolated entity able to monitor and control the system execution. It is important to ensure that our algorithm can be executed with a bounded periodicity, as the delay between two executions is taken into account in the anticipation. This can be made by simply giving a high priority to the monitoring and control task and ensuring it has a bounded execution time. The whole context of our first proposal is summed up in figure 2.

C. Monitoring & Control Agent Architecture

The Monitoring and Control Agent is made of two components: a *Core Control Component* and a *Task Wrapper Component* as shown in figure 3.

1) Task Wrapper Component (TWC): Tasks are wrapped with two software blocks: a "Before" and an "After" block. It has two roles:

- monitor the *tasks timing state* by sending timestamped start and end of the tasks messages for the Core Control Component.
- control correct switch to degraded mode by preventing eventual non-critical tasks execution in such mode.

The "Before" bloc is wrapped to every task. Before non-critical tasks execution, it prevents their execution if the Core Control Component switched to degraded mode due to the detection of a potential overloaded state. The switch to degraded mode can be made with different strategies:

- Pause non-critical tasks: the scheduler set them to paused state until task chain completion. Their context is saved and when risk is avoided (usually when the current chain reached its ending task), they are put back in the scheduling queue. Such solution is adapted if degraded mode duration is not too long and data freshness is not too much critical.
- Stop non-critical tasks: tasks are killed, and their context is not saved. During recovery, the tasks are started again.

For critical tasks, "Before" bloc is used for task chain monitoring, mainly to check the task chain beginning. It could be possible to use such data from the "Before" bloc to enhance the anticipation mechanism by taking into account what is the current task being executed (but not finished) in the remaining worst-case execution time estimation. The "After" bloc communicates execution time of every critical task to the Core Component for task chain monitoring too. Those monitoring messages are queued to be processed by the Core Control Component. There is no need of an "After" block for non-critical tasks.

2) Core Control Component (CCC): For each task chain, the Core Control Component stores a Task Chain Timing State (TCTS) with the current chain execution time and the list of executed/unexecuted tasks in the chain. A task is considered as executed only if it has a valid execution, as explained previously.

The Core Control Component updates the task chain timing state periodically to check if the deadline can still be guaranteed. The TCTS update frequency is fixed and not directly triggered by a monitoring message (i.e. at one TCTS update we could have multiple pending tasks ending to consider or no new tasks ending). Consequently, such period must be chosen regarding the tasks' execution times to avoid having too much monitoring messages to take into account. We trigger

the TCTS update periodically and not asynchronously (at each monitoring message) to avoid being dependent on when the "Before" and "After" blocs are executed. This way if a task takes too much time to end, we can still decide to switch to degraded mode only because we are getting closer to the deadline with no new monitoring message. We explain more in detail how to define this period with W_{max} choice bellow.

From a TCTS update, it is then possible to compute a Remaining Worst-Case Execution Time (RWCET) **in degraded mode** from the unexecuted critical tasks of the chain. If a potential end-to-end deadline miss is anticipated, non-critical tasks are stopped or paused (according to the chosen strategy) to switch to degraded mode and guarantee the task chain deadline.

In degraded mode, only the critical tasks are executed, on a single core. This makes Remaining Worst-Case Execution Time estimation easier, as we avoid interferences from external tasks and the possible execution variability is reduced. Such WCET analysis can be made following methods presented in [14] for instance, as only one core is activated in degraded mode. However, we want to see how experimental estimations of WCET could be used.

To sum up, CCC has at disposal at any TCTS update:

- task chain end-to-end deadline D_c
- individual tasks timestamped execution times
- RWCET in *degraded mode*

From this data, it is possible to compute the task chains current run-time WCET. We know **a)** the current task chain execution time, i.e. how much time has elapsed since the activation of the first task of the chain **b)** in the worst case, how long they could take to be executed in degraded mode. We compare this current run-time task chain WCET to the end-to-end deadline. This is made with the following formula adapted from [15], at a given time t:

$$ET(t) + RWCET(t) + W_{max} + t_{SW} \leq D_c \qquad (1)$$

Where $ET(t)$ stands for the current execution time, $RWCET(t)$ the task chain Remaining WCET when executed in isolation, W_{max} is the CCC updating period (*seen as a worst "observation delay"*) and t_{SW} the latency to switch to the monocore degraded mode (*seen as a "reaction time"*). When equation (1) becomes false, the CCC switches to degraded mode with only the critical task chain executed to guarantee its deadline. The mode switch is made first by sending a signal to every non-critical task from the CCC. Upon reception, they pause themselves. In addition, the "Before" wrapper blocks eventual non-critical tasks trying to be executed.

W_{max} is the maximum duration between two CCC checkpoints. It is directly dependent to the Core Control Component periodicity T_{ccc} as we are anticipating a risk of being in a state where even in isolation the task chain could not reach its deadline. We add W_{max} as it represents directly the fact that we *anticipate* a failure that could happen at the next CCC checkpoint. This way, when inequality (1) becomes false, it means that at next checkpoint, we could potentially be in a state where an unavoidable failure will occur. Consequently, we have $W_{max} = T_{ccc}$ as we have a strict (guaranteed) periodic CCC. With a periodic task activation model, it is simple to set this value, around the smallest task execution time. This way we have the guarantee of not overflowing the monitoring message queue used by the Task Wrapper Component. A greater value could be possible, but we must take care to process the TCTS updates faster than the arrival of monitoring messages, this should probably be made experimentally if we have no data on the maximal task's frequency. For other tasks activation models, we must identify identically the highest task monitoring messages upcoming rate to ensure being able to compute them all.

It is also important to set W_{max} correctly –and thus, the Core Control Component period– 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, but we are also more dependent on the quality of the RWCET estimation. Also, it will naturally use more computing resources. On the contrary, a higher value gets more secure anticipations that triggers sooner, but we also increase the number of switches to degraded mode that were not needed (false positives). Future work could aim at measuring more precisely W_{max} influence.

One should note that what makes such approach possible is the evolution of the RWCET at run-time and as the TCTS evolves. It would not be possible to apply such approach when it comes to monitor & control individual tasks to guarantee their individual deadlines. For individual deadlines, our method would fit only if we are able to monitor tasks timing state "inside" the tasks execution, i.e. instrumenting the tasks source code to add internal checkpoints. Such approach on individual tasks would discard by definition the use of black box software assumption for instance, and otherwise would need much higher refresh rate frequencies in order to follow individual tasks execution timing state. Such solution is presented for individual tasks in [16].

IV. Implementation Framework

A. Objectives

We describe in this section how we decided to implement the Monitoring & Control Agent principle on a first proof-of-concept platform according to III- Monitoring & Control Agent guidelines. Such software & hardware platform aims several objectives:

- Validate qualitatively our approach for **a)** end-to-end timing guarantees, **b)** available computing resources for non-critical tasks, **c)** scalability to task set size,
- Compare our approach in term of average CPU usage and check solution weight on overall execution
- Perform a sensitivity analysis to identify how the system behaves according to, for instance, the ratio of critical/non-critical tasks, the time period of tasks and their execution time. Such analysis can open the way

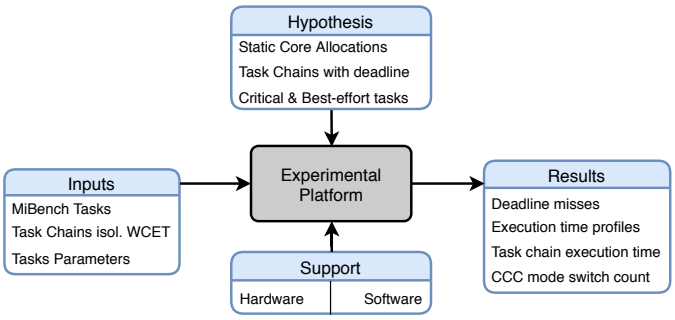

Fig. 4. Experimental Platform

to conclusions on task allocation policy and extend the system to more tasks chains or cores.

As presented in figure 4, the experimental platform has three input domains of which two are configurable. First input domain gathers the basic hypothesis. The two others are the tasks inputs (task set, tasks parameters and task chains WCET method) and the execution support (hardware and software presented above) for which choices can be made. We present here the choices made in our case.

B. Support - Operating system and Run-time environment

1) Linux OS: We decided to use Linux (latest Linux Mint xfce distribution) to take benefit of its possibility to run both classic Linux processes and Real-time processes with different scheduling policies (see II-B). Its versatility grants easier compatibility with benchmarking suites and as presented at State of the Art, we can use some mechanisms to get closer to a real-time embedded system.

Notably, POSIX enables to force tasks execution to dedicated cores and change both priority and scheduling policy. As we are in a controlled context that suppose no malicious behavior, we do not implement mechanisms like memory protection or strong space isolation policies. As stated before, vanilla Linux Kernel is not made for hard real-time application. That is mainly because kernel is not preemptive on most parts of it, this can cause high latency for real-time interrupts, from kernel code execution that could be linked to non-critical applications. Therefore, we add a Xenomai co-kernel to improve latency down to micro-seconds and run our MCA to respect desired real-time constraints. Please note that from Linux point of view, “threads” and “processes” are equivalent and correspond to “tasks” for us.

2) Xenomai co-kernel patch: Xenomai is a real-time kernel that can be installed as a co-kernel to a classic Linux distribution as presented in deep by [17]. Our framework and experiments are implemented on the real-time APIs proposed by Xenomai 3.0.5. In such configuration, it adds an interruption pipeline (ADEOS) directly between the hardware and OS low-level software (i.e. Hardware Abstraction Layer, OS Kernel and drivers). This enables to catch all the interrupts and distribute them in priority to Xenomai real-time kernel.

Such operating system allows us to specify the tasks allocation to cores with a core affinity parameter. It is also possible to set a priority level for every task. Linux scheduler selects tasks first by priority level, (from 1 to 99 for real-time tasks). 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, we need to launch every task with same Linux priority level, and with Round-Robin policy. We can recreate a Rate-Monotonic policy the same way, by using tasks priorities depending on their period, and just FIFO scheduling policy at equal priority levels (keep in mind that scheduler considers priority *before* policy). More about Linux scheduling policies can be found in [18].

3) Hardware: The platform used for the experimentation is a bare-bone computer equipped with an Intel Core i5-8250U. This processor embeds 4 cores, with possible multi-threading (8 threads, disabled for our tests), from 1.60 GHz to 3.40 GHz. It has 3 caches level, L1, L2 and L3 (shared), with respectively 32 KiB/core, 256 KiB/core and 8 Mib.

C. Software Setup

On this Platform Support, we need to implement the framework for the Monitoring and Control Agent and the task set configuration.

1) Framework Setup: The Core Control Component needs to be implemented in a safe way from the rest of the system. The constraint is to ensure the CCC execution without exceeding its execution period, in order to get a reliable $Wmax$ value equal to the execution period. We execute it with highest priority, on an isolated core, with dedicated memory reserved. This way we ensure no cache replacement by other tasks, and it is executed prior to any other task. Those are drastic solutions, that could be soften, but they are simple to implement in our platform support. Moreover, we are going to stress the system with various workloads, such solutions avoid verifying the timing guarantee on the CCC again for every experiment.

Both the Core Control and the Task Wrapper Components have a configuration file grouping several input parameters. We consider a system with periodic tasks, defined also by a deadline, a core allocation, a priority level and a group ID to identify the belonging to a task chain, which leads to 2 additional parameters: the tasks precedence constraint list and their off-line defined WCET in degraded mode. We choose a periodic task activation model because our tasks will be from a benchmark that includes no specific communication or data-sharing between tasks.

The TWC encapsulates all the tasks and launch them into the real-time environment. Encapsulation only adds a few lines of code, to get real-time clock timestamps and queue them to the Core Control Component.

All the tasks are launched according to their core allocation and scheduling policy. Consequently, the scheduling is done by core with no migrations (partitioned scheduling). Further experiments plan on using semi-partitioned scheduling, or more complex mixed-critical scheduling on multicore such as

TABLE I
MIBENCH SELECTED TASKS

Automotive	basicmath, bitcount, qsort, susan (smooth, edges, corners)
Network	dijkstra, patricia
Consumer	jpeg (code & decode), typeset
Office	stringsearch
Security	blowfish, rijndael, sha
Telecom	adpcm (coding & decoding), CRC32, FFT, gsm

MC^2 [19] for instance. Such policy, combined with our mechanism, could bring good performance with strong guaranties on such system. With such inputs the CCC knows task chain constraints and process the monitoring messages at run-time. To finish configuration, we need to define the work load task set and its off-line characterization in order to be used by the CCC and reduce computing needs at run-time.

2) Task Set definition: As we do not have yet real industrial application for testing, for now the MiBench Benchmark suite [20] has been used for our experiments. The objective is to use applications similar as much as possible to computation profiles that could be found in real applications, in order to reproduce memory containment and resource usage close to real cases.

MiBench consists of a large panel of tasks with different memory needs and execution profiles to mimic existing applications. We have at disposal applications from 5 different domains, as presented in the table I. It is used here to validate the framework and put into practice our experiments.

We selected a set of 16 applications from MiBench for our experiments. Most of them exists in "small" and "large" version that allows to change proportionally their execution time and resource needs. Also, some of these tasks may have several variants according to setup parameters. For instance, $Sunsan$ has 6 different variants: edge detection, corner detection and smoothing, all 3 existing in both "small" and "large" version which works with a bigger image for processing. This way, those 16 applications leads to 45 different possible tasks for our experiments. It enables to test different combination following the "size" and number of tasks but also the kind of tasks we use. Tasks profile classification were already made by Guthaus & al. in [20] and detailed work about their memory consumption can be found in [21].

From these tasks, we define several task chains with different profiles. We define a base task chain composed of both "small" (S) and "large" (L) tasks, from Automotive, Network and Security domains as represented in figure 5. The chain starts with $bitcount$ "small" task and ends with FFT "small". Arrows represent the imposed precedence constraint. Here, FFT_inv and $CRC32$ can be executed in any order, but only after $bitcount$ execution for instance. This way we define 5 different task chains, with a number of tasks going from 3 (large tasks) to 15 (small tasks) per chain. Every experiment is made with only one task chain from the 5 defined, with other tasks from the bench only used as non-critical workload. Changing the task chain will be made to check its impact on the Monitoring and Control Agent efficiency.

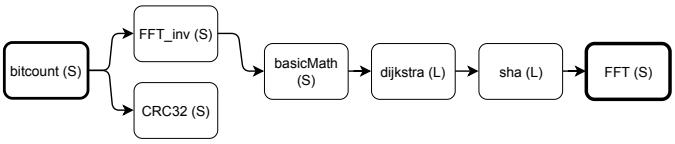

Fig. 5. "Compute Trajectory" task chain example, with MiBench tasks

We define the end-to-end deadline of these chains simply as the sum of the deadlines from each task they are made of. In a real system it would (probably) be a higher value, but it could not be less and thus it gives a good baseline for testing. Question is, what are individual tasks deadlines? Here comes first step of off-line characterization.

D. Off-line characterization

1) Individual Tasks: The first step is to characterize the task set on our platform to assign task deadlines and task chains end-to-end deadlines, as MiBench does not include such information. We execute every task individually, wrapped in our framework, to estimate their nominal execution time profile. Such value gives us a reference to set individual deadlines, proportional to such values. It is also possible to determine tasks period, equal to deadline. End-to-end deadlines are consequently defined.

2) Task chains: We need 3 elements to get a fully functional Core Control Component: **a)** Degraded mode WCET, **b)** t_{SW} time to switch to degraded mode, **c)** W_{max} CCC execution period, .

Degraded mode analysis is made by executing only the selected task chain in isolation on one core several times and monitor execution time. Then, we extract for each task τ_i of the chain their maximum observed $WCET(\tau_i)$ in such condition. This way, we define that at time t, if it remains N tasks in the chain to execute, we have:

$$RWCET(t) = \sum_{i=1}^{N} WCET(\tau_i) \tag{2}$$

Such WCET estimation for the task chain presents some limits. On one hand we have no strong guarantee that we measured the real WCET for each task. However, such approximation may even be over-estimated because we sum every tasks WCET. In fact, we know that such configuration should not occur as not every task takes its WCET value at the same time. This method advantages are its quite easy implementation and run-time computing is straightforward and lightweight. We have to measure such estimation quality and its impact over the anticipation mechanism. To do this, we will compare the remaining time between its real termination and its current deadline, every time the CCC switched to degraded mode. The closer to zero this value the better, but without exceeding the deadline of course.

TABLE II
MONITORING & CONTROL AGENT OVERHEADS SAMPLE

	Monitoring & Control ON		Monitoring & Control OFF	
Task	Median (ms)	Max (ms)	Median (ms)	Max (ms)
FFT (S)	33.80	35.63	32.72	34.01
Sha (L)	31.86	98.69	32.88	129.46
CRC32 (S)	34.70	36.56	28.63	29.80
rev FFT (S)	36.89	98.50	33.55	37.20
Bitcount (S)	13.61	31.41	12.02	13.00
dijktra (L)	51.12	63.38	38.13	67.55
Basicmath (S)	27.63	98.50	10.75	11.20

W_{max}, is related to the execution frequency of the Core Control Component, to check TCTS updates and verify equation (1). We set such frequency from the lowest critical task period value, in order to get at most one evolution on the task chain state per period (either a task start or a task ending). There is no need to go faster as we will get no state evolution. We could check if a slower execution rate could contribute in letting the system balance itself during the execution. However, with a higher W_{max} we notice from equation (1) that the mechanism will anticipate sooner and thus possibly switch to degraded more often. For our experiments and considering our task set, the minimal task period is 10ms. We set $W_{max} = 5ms$ lower than the minimal period.

t_{SW}, is the time duration to switch to degraded mode. As it is dependent on the framework implementation, we just run a few tests with arbitrary switches to degraded mode and measure its maximum duration. On our platform and without framework, we estimated $t_{SW} = 2ms$.

V. FIRST EXPERIMENTS

A. Monitoring and Control Agent Overhead

First objective is to quantify the overhead from our framework on the task's execution time and CPU usage. We executed the same task set during a fixed duration (5 minutes) with Round-Robin scheduling. The experiment is made twice: one time with the Monitoring & Control Agent enabled and a second time without it. We then look at the execution profile of every task on the set and compare them on both configurations. Experiments of 5 minutes represents from 2200 execution measures for the task with the highest execution period to 12500 executions for tasks with highest frequency. We made such experiments multiple time to check consistency.

It revealed some outliers execution times: for instance $dikstra$ (large version) taking 16 seconds instead of usual $\simeq$50 ms. Such values represents less than 0.001% of the tasks executions. Such aberrant values may be due to some side effects within the platform or the benchmark and are removed from our analysis.

We found out that the Monitoring & Control Agent represents less than 1% of CPU usage. 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). Also, CPU temperature showed no significant change during execution.

However, regarding the tasks' execution time, we got mitigated values. For most of the tasks, execution time had no significant changes (less than a millisecond). But in some cases, our mechanism appeared to bring some variability, and consequently tasks median execution time is not changed, but the task execution profile spreads more, showing a higher maximum value. This is the case with $reverse\ FFT$ and $Basicmath$ for instance. Also, in fewer cases, the task presented more than a 50% increase in median execution time, such as $Dijktra$ algorithm. This shows that some tasks from MiBench do not seems adapted as real-time tasks due a lack of determinism in their execution with our framework. Without any correction, it implies using high WCET values for such tasks if they are used in a critical task chain, and consequently the RWCET(t) may be overly pessimistic.

B. Future Work

1) Sensitivity Analysis: Our experimental platform is summarized in figure 4. Such setup allows sensitivity analysis based around various parameters from the hardware (any running Linux with Xenomai) to task chain and the non-critical task set choice, through different scheduling policies and CPU load changes.

We described how to generate a task set to work with, made of 1 task chain (with associated off-line analysis) and a certain number of non-critical tasks selected from MiBench suite. The amount and the profile of those tasks is the first parameter we will be able to change in order to see its influence on our experiments (see Input domain in figure 4: MiBench tasks). We can select tasks following their memory use profile, change the number of "small" and "large" tasks... Tasks parameters allows to set processor charge, by changing their execution period.

First in-deep experiments will try to see the influence of the scheduling policy on our approach. The idea is to see for different policies (typically round-robin, rate-monotonic or more complex policies like MC^2) its consequence on the task chain execution time profiles: average and maximum chain response time. We can also see if the Core Control Component triggers degraded mode more or less often and consequently gives different execution time for non-critical tasks.

Second experiments lead to the influence of $RWCET(t)$ and W_{max} estimation, in order to see to what extent they change the performance of our anticipation mechanism. The $RWCET(t)$ computation method could possibly be changed for possible enhancements.

2) Prospectives: We see many optimizations and enhancements of our framework. We started with a simplified but realistic context to guarantee one critical chain end-to-end deadline on a multicore. The long-term objective is trying to extend such mechanism to multiple task chains on the same multicore.

Also, we plan on enhancing the degraded mode switch in order to avoid a brutal "all to nothing" switch of non-critical tasks. The hypothesis (see Hypothesis domain in figure 4) can evolve to enhance our first version of the Framework, adding criticality layers to avoid stopping every non-critical tasks at once, or changing the tasks core allocation. We could try to disable only a sub-part of non-critical tasks, but possibly sooner, to avoid disabling them all and still having guarantees on end-to-end deadlines. This would be translated into different levels of degraded mode.

Finally, a good possible enhancement would be to implement the Core Control Component on a separated processor or even a dedicated FPGA, to keep use of all the multicore computing resource for the application. Some ongoing work is already studying such perspective with [22] for example. The use of an external dedicated hardware for monitoring raises new questions. An analysis is needed to measure and take into account the possible communication delays in the equation. On one hand it would probably affect the switch time t_{SW}. On the other hand, the whole multicore become available for our task set and consequently we do not have to check for potential interferences from the task set to the Core Control Component performances. It is a clear perspective for future work.

C. Comparison with other approaches

This task chain-based approach compares with other individual task timing constraint-based approaches. The first comparison point concerns computing resource use for non-critical tasks: the higher the better. This point can be quantified in our approach by comparing the number of non-critical tasks execution during a fixed amount of time, either with our Monitoring & Control Agent and with other approaches like a bare G-EDF or simply by disabling our Agent.

The second comparison point is to see on the opposite point of view the capacity of our mechanism to respect end-to-end deadlines, compared to other policies that are not dedicated to such goal. We want to see for different task sets if our system still manages to guarantee end-to-end deadlines without going to degraded mode 100% of the time, compared to end-to-end response time measured with other solutions.

We expect from on-going experiments to see how the Monitoring & Control Agent allows a gain in maximum computing resource used (average CPU %), still being able to respect end-to-end deadlines. We will finally check its influence on the tasks' execution time profiles (average, median and maximum execution times for individual tasks and the task chain) and the influence of said parameters at run-time.

VI. Conclusion

We defined a complete process to instrument mixed-critical tasks on a multicore real-time Linux platform with a Monitoring and Control Agent to guarantee end-to-end constraints for critical task chains. Our ongoing work aims to **a)** validate the approach compared to other strategies handling mixed criticality application through on-going experiments and **b)** analyze the effect of different factors on the execution of critical software. Finally, we will apply our approach to a real automotive applications case studies with Renault.

References

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

[2] S. Fisher and S. AG, "Certifying Applications in a Multi-Core Environment: The World's First Multi-Core Certification to SIL 4." 2013.

[3] AUTOSAR, "Timing Analysis," *Standard Release 4.3.0*, p. 118, 2016.

[4] P. J. Prisaznuk, "ARINC 653 role in Integrated Modular Avionics (IMA)," in *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, Oct. 2008, pp. 1.E.5–1–1.E.5–10.

[5] 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.

[6] G. Giannopoulou, N. Stoimenov *et al.*, "Scheduling of mixed-criticality applications on resource-sharing multicore systems," in *ACM International Conference on Embedded Software*, 2013, p. 17.

[7] B. C. Ward, J. L. Herman *et al.*, "Making Shared Caches More Predictable on Multicore Platforms." IEEE, Jul. 2013, pp. 157–167.

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

[9] J.-P. Lozi, F. Gaud *et al.*, "The Linux Scheduler: a Decade of Wasted Cores," p. 16, 2016.

[10] 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.

[11] F. Cerqueira and B. B. Brandenburg, "A Comparison of Scheduling Latency in Linux, PREEMPT RT, and LITMUSRT." SYSGO AG, 2013, pp. 19–29.

[12] D. J. H. Brown and B. Martin, "How fast is fast enough? Choosing between Xenomai and Linux for real-time applications," Tech. Rep., 2010.

[13] I. Allende, I. T. R. Centre *et al.*, "Towards Linux for the Development of Mixed-Criticality Embedded Systems Based on Multi-Core Devices." IEEE, 2019, p. 8.

[14] R. Wilhelm, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, Apr. 2008.

[15] A. Kritikakou, C. Pagetti *et al.*, "Run-Time Control to Increase Task Parallelism In Mixed-Critical Systems." IEEE, Jul. 2014, pp. 119–128.

[16] A. Kritikakou, T. Marty, and M. Roy, "DYNASCORE: DYNAmic Software COntroller to Increase REsource Utilization in Mixed-Critical Systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 23, no. 2, pp. 1–26, 2017.

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

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

[19] J. L. Herman, C. J. Kenna *et al.*, "RTOS Support for Multicore Mixed-Criticality Systems." IEEE, Apr. 2012, pp. 197–208.

[20] M. R. Guthaus, J. S. Ringenberg *et al.*, "MiBench: A free, commercially representative embedded benchmark suite." Austin, TX, USA: IEEE, Dec. 2001, p. 12.

[21] A. Blin, C. Courtaud *et al.*, "Understanding the Memory Consumption of the MiBench Embedded Benchmark." Marakech, Morocco: Netys, 2016, p. 16.

[22] D. Solet, S. Pillement *et al.*, "HW-based Architecture for Runtime Verification of Embedded Software on SoPC systems," in *2018 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. Edinburgh: IEEE, Aug. 2018, pp. 249–256.