Mixed time-criticality process interferences characterization on a multicore Linux system

Federico Reghenzani*, Giuseppe Massari*, William Fornaciari*
Dipartimento di Elettronica, Informazione e Bioingegneria - Politecnico di Milano
Via Ponzio 34/5, Milano, 20133, Italy
*name.surname@polimi.it

***Abstract*—The increasing interest in the integration of Mixed-Criticality Systems (MCS) in Commercial-Off-The-Shelf (COTS) platforms leads to an increasing number of challenges. The possibility of sharing computing resources among applications with different time criticalities is a key goal for COTS systems, but still hard to achieve. Classical approaches in real-time systems are not feasible when platform and operating system may introduce unpredictability in the task execution. Moreover, if the system must also meet non-functional requirements (e.g., thermal and power management), dynamic approaches of computing resources allocation are more effective than static ones. Unfortunately, this contributes to increasing the complexity of the scenario. In MCS, the overheads and the unpredictability caused by sharing resources like cache memories have been well studied. However, in some cases we could also consider the operating system itself as a potential source of unexpected and unpredictable latencies, if several running tasks perform system calls. This work aims at proposing a model for the intra-core and inter-core interferences and the analysis of the OS-induced latencies in a Linux real-time system, both essential for the creation of smart and effective run-time resource management policies.**

I. Introduction

In the last years, the end of Dennard's scaling [1] and the increasing strictness in non-functional requirements – such as power, energy, space, weight, cost and dependability constraints – induced a new trend in the embedded systems design, i.e. the integration of applications with different levels of criticality in the same system, possibly multi-cores [2]. Furthermore, the choice of Commercial-Off-The-Shelf (COTS) hardware is very attractive in order to reduce the design costs. In particular, we can observe this trend in avionics and automotive use-cases [3].

A **Mixed Criticality System (MCS)** is typically an embedded system, characterized by a workload including tasks with different criticality levels, that run – possibly interacting – on the same computational platform. Research on MCS includes several aspects of computer science and engineering. This paper focuses on so-called Mixed *Time*-Criticality Systems, ignoring other aspects, such as dependability. In this regard we may distinguish among the following criticality levels: *hard*, *firm*, *soft* real-time, and *best-effort*. The criticality level identification strictly depends on the type of actions performed and the context in which the given application runs. For this reason, in this paper we distinguish only between real-time (RT) and best effort (BE) tasks, without considering further real-time classes.

The most important requirement of a Mixed Time-Criticality System is to provide predictable and deterministic behaviours, in order to properly schedule real-time tasks. However, if the MCS relies on a multi-core COTS processor, this becomes a very challenging activity, along with the computation of the **Worst-Case Execution Time (WCET)** [4].

A. *Man-in-the-middle: the Run-time Resource Manager*

In 2016, the Linux Foundation started a project to advance the real-time Linux in order to provide a kernel able to compete with state of the art real-time operating systems (RTOSs). The use of Linux considerably simplifies the development process, in particular thanks to the reduced effort in the implementation of device drivers and the availability of libraries. However, Linux is extremely more complex than classical RTOS (free or commercial). Accordingly, it is practically impossible to use standard timing analyses, such as control-flow or path analysis.

Unfortunately, the increasing necessity of dealing with thermal, power and energy constraints – especially when time-varying – is an important challenge that cannot be effectively addressed by a RTOS. In this regard, an increasing trend is to use more complex operating systems and a large number of integrated applications, that require different solutions from formal methods and static analyses, which become infeasible in many use cases.

This scenario leads us to evaluate the necessity of integrating a *run-time resource manager* on top of the real-time Linux in order to mitigate the sources of unpredictability. The resource manager would be in charge of properly partitioning the computing resources of the platform, thus isolating tasks, by considering the criticality level, the performance requirements and the aforementioned non-functional constraints.

B. *Real-time and Linux*

Since 2000s, several real-time Linux systems based on a co-kernel schema have been developed and studied, e.g. RTAI [5] and Xenomai [6]. The co-kernel approaches usually consist of an additional real-time kernel in charge of handling the interrupts and deciding whether to run a real-time task or the Linux kernel, without the need of modifying the kernel to introduce real-time capabilities. However, co-kernel

2017 Euromicro Conference on Digital System Design (DSD), IEEE, doi: 10.1109/DSD.2017.18

approaches tend to demand a high development effort, since they often require the reimplementation of device drivers and libraries, in most cases, already available in Linux. Moreover, the mutual influences between the RT and the Linux kernel must be carefully evaluated, in order to avoid unexpected events and behaviours.

In this work we consider the PREEMPT_RT patch of the Linux kernel that is not currently merged in the main branch. Recently, PREEMPT_RT has gained interest in both industrial and scientific community. In 2016, Gosewehr et al. [7] evaluated the possibility to switch from RTAI to pure Linux with PREEMPT_RT for a network application, discovering that Linux is however a good candidate for distributed real-time systems, even if it does not guarantee the same level of determinism and latencies of RTAI.

Two articles in 2013 [8] [9] estimated the performance gap (always in terms of latencies and determinism) between Linux RT and co-kernel approaches. The outcome was that even if PREEMPT_RT substantially improved the real-time capability of Linux, it is not ready for hard real-time applications. However both works and the Kazan et al. study [10] shown a trend of improvements in the releases. Nevertheless, the real-time capabilities of recent versions of Linux kernel are not systematically studied yet. This kind of activity may lead to further improvements. Consequently, the state of the art in Linux PREEMPT_RT provides a scenario in which Linux – without a co-kernel approach – can be effectively used for soft real-time applications.

C. Paper goals and structure

In order to develop a resource manager for mixed criticality systems, by exploiting a Linux operating system with the PREEMPT_RT patch applied, we need to carry out a preliminary work on (1) modelling temporal partitioning of the tasks and (2) a quantitative assessment of latencies induced by the operating system during the execution of non-critical (non real-time) tasks. Therefore the goal of this study is two-fold:

- to provide a temporal-partition model for a run-time resource manager that considers non-deterministic behaviour of COTS platforms and the OS-induced interferences;
- to characterize the OS-induced latencies of a real-time Linux system under stress conditions in a mixed time-criticality environment.

Section II presents the state of the art in the analysis of multi-core effects on WCET and in real-time resource management approaches. Subsequently, the Section III describes the notations used in next Section IV to propose a model for the run-time resource management. Section V presents the experimental results, in particular the OS-induced latencies analysis. Finally, future works and conclusion are described in Section VI.

II. State of the Art

Excluding sporadic works in 80s, first articles about mixed criticality systems appeared in literature starting from 2007 and they were reviewed in 2013 by Burns et al. [2]. This well-received article is kept updated every year and it covers several aspects of MCS, including the resource sharing effects in multiprocessor environments. As highlighted by the review, inter-core and intra-core interferences are largely discussed and analyzed in literature. Since the hardware of embedded systems – in particular real-time systems – is traditionally designed for its specific purpose, literature proposes solutions that require the use of *Special Purpose Processors*. In general, they are very difficult to implement in COTS hardware and their analysis is outside the scope of this paper.

Any modern COTS multi-core processor is designed to be a *general-purpose processor (GPP)*, with performance in mind. This means that guaranteeing temporal determinism is not a design objective. Kotaba et al. [11] categorized the latencies in GPPs caused by resource sharing in six classes: system bus sharing, memory delays, cache effects, CPU internals (e.g. multi-threading and pipelines), addressable devices, and other effects (e.g. BIOS routines and microcode). They also proposed *resource limitation* as a solution to restrict the side effects of non real-time tasks, that was used in other works, like the *temporal isolation* proposed by Nowotsch et al. [12]. In this article, they used a run-time resource limitation based on a static analysis of the tasks, in order to run them concurrently in a multi-core based system, enforcing the calculated WCET by using temporal isolation.

A. Real-time and Resource Management

Several MCS scheduling strategies voluntarily ignore the effects of resource sharing. Giannopoulou et al. [13] proposed a design-time strategy to improve scheduling decisions, about task-core mapping. This approach reduces the shared memory interferences. However, as previously highlighted, in COTS systems and using complex operating systems, WCET cannot be deterministically calculated. Consequently, it becomes essential to have a run-time resource manager that, according to functional and non-functional constraints, provides a correct resource partitioning. A resource manager does not replace the scheduler, but it complements it, by taking coarse-grained decisions (usually on long periods). The scheduler instead provides the usual scheduling duties under the constraints set by the resource manager.

Kritikakou et al. [14] presented a hybrid design-time and run-time control for MCS. A WCET analysis is applied at run-time in order to check if a real-time task is able to meet the deadline, while other processes are concurrently running in the system. If the deadline cannot be met, the system switches to an *isolation scenario* in which only the real-time task is allowed to run.

Huber et al. [15] proposed a resource manager for MCS based on Network-on-Chip (NoC), in charge of properly allocate the interconnect bandwidth. The resource manager switches between different application configurations based on the system status, in order to maintain the required Quality-of-Service.

To the best of our knowledge, the resource management of MCS and COTS systems with complex OS – like Linux – has not been thoroughly studied. In this article we propose a model for the characterization of inter-core interferences affecting real-time tasks running in a MCS system.

B. Latency characterization

In 2002, Abeni et al. [16] analyzed the timer resolution of an x86 platform and the latencies of the non-preemptible sections of the operating system. Even if outdated, this is a good starting point to evaluate the WCET of real-time application running in isolation.

We are not aware of any previous work measuring the OS-induced latencies caused by non-critical tasks over real-time tasks in COTS platforms. The experimental evaluation (Section V) provides an estimation of the latencies occurring during the execution of system calls in user-space processes.

III. Resource Management Model

In literature, different models are used to describe resources and workload of a real-time system. Since a uniform model among scientific publications does not exist, this section describes the task and system model used in the subsequent sections.

A. Task model

Each task τ_i has a level of criticality decided at design time. In order to simplify the formal analysis we consider only two levels of criticality: RT real-time tasks and BE the best effort ones. Each task $\tau_i \in RT$ can be described by the tuple $(A_i, D_i, \overline{W}_i)$ where A_i represents the inter-arrival period, D_i the relative deadline and $\overline{W}_i$ the WCET required to finish the task. $\overline{W}_i$ is measured in ideal conditions, i.e. when the task executes in an isolated CPU without any interference due to resource contention.

The model of BE tasks is considered arbitrary in this work.

B. System model

Let $C = \{c_1, c_2, ..., c_n\}$ be the set of n homogeneous CPU cores available in the system.

We assume that the task scheduling policy is periodically invoked. Generally, a task may require multiple periods to complete, as explained later in Section IV-B. The periods are identified by an index $z \in \mathbb{N}$. Moreover, the length of the period can be variable $P(z)$, but for the sake of simplicity we consider it constant in this paper ($P(z) = P$).

Using the notation of the model proposed by Mok et al. [17], a *resource partition* $\Pi(z)$ is a tuple (Γ, P) where Γ is an ordered set of N time pairs $\{(B_1, E_1), (B_2, E_2), ..., (B_N, E_N)\}$ that represents the sub-intervals starting at time B_i and ending at time E_i in which the contiguous time interval of period z with a maximum length equal to the period time P is divided.

We introduce then the *Supply Function* $S(t, z)$ defined as the function returning the amount of CPU time allocated by the scheduling policy in a range of time $[0; t]$ from a period z. To simplify the notation, let $S(z) = S(P, z)$ the amount of time assigned from the whole period. Accordingly, we can analytically express it as follows:

$$S(z) = S(P, z) = \sum_{\forall (B_i, E_i) \in \Gamma(z)} (E_i - B_i)$$

In order to deal with the allocation of CPU time in multi-processor systems, let's extend the used notation defining $\overline{\Pi}(z)$ as an array of resource partition, one for each processing unit $c_j \in C$:

$$\overline{\Pi}(z) = \{\Pi(z)_{c_1}, \Pi(z)_{c_2}, ..., \Pi(z)_{c_n}\}$$

The output of the resource manager for each time period z is a partition $\overline{\Pi}(z)$ for each task τ_i. Usually, it selects the tuple $(S(z), P)^{\tau_i}_{c_j}$ instead $(\Gamma(z), P)^{\tau_i}_{c_j}$, leaving to the scheduler the decision authority on how to split the enforcing period.

A correct partitioning requires that the total time in the period corresponds to the sum of time assigned to all partitions to each CPU in addition to the idle time. Thus, if the enforcing period is equal for all tasks, we can write:

$$\sum_{\forall \tau_i \in \{RT \cup BE\}} S^{\tau_i}_{c_j}(z) + S^I_{c_j}(z) = P \qquad \forall z, c_j \in C \tag{1}$$

where $S^I_{c_j}(z)$ is the idle time in which the CPU c_j is not allocated to any task. Additionally, consider the notation of total amount of time assigned to each criticality level:

$$S^{RT}_{c_j}(z) = \sum_{\forall \tau_i \in RT} S^{\tau_i}_{c_j}(z)$$

$$S^{BE}_{c_j}(z) = \sum_{\forall \tau_i \in BE} S^{\tau_i}_{c_j}(z)$$

that allows us to rewrite the Equation 1 as:

$$S^{RT}_{c_j}(z) + S^{BE}_{c_j}(z) + S^I_{c_j}(z) = P \qquad \forall z, c_j \in C \tag{2}$$

To summarize, the resource manager would be in charge of:

1) tuning the enforcing period length P;
2) selecting the CPU time allocated to each task for each core $S^{\tau_i}_{c_i}$;
3) implicitly selecting the enforced idle time $S^I_{c_i}$.

These parameters could be tuned at each enforcing period. However, in a real scenario changing the time allocation introduces a non-negligible overhead, that should be carefully evaluated, and usually leads to a limitation of the rate of possible partition changes.

The assignment of CPU time to the RT tasks cannot be blindly performed with respect to other BE tasks. As described in the next section, the interferences on RT tasks caused by BE tasks depend on $S^{BE}_{c_j}$ and may consequently require to adjust also $S^{RT}_{c_j}$. The resource manager must have a smart policy that, balancing the time assigned to the three partitions, could guarantee RT constraints and non-functional requirements, keeping the highest possible QoS for BE tasks.

Using this resource assignment model, it is possible to neglect the scheduling policy of the underlying operating system scheduler in first approximation. The RT tasks may have any scheduling policy (e.g. priority-based or deadline-based) independently from the resource management control. The scheduler is then free to decide the schedule in the boundaries of the resource partitions defined by the resource manager.

IV. Interferences assessment

In this section we propose a model for the intra-core and inter-core interferences to be used in high-level resource allocation policies.

In the next experimental section we will try to validate the model for the inter-core interferences.

A. Taxonomy

Following the classification of execution time interferences in a multi-core processor provided by Gracioli et al. [18], one could categorize the interferences in:

1) *Intra-task interference*: it is mainly the effects of having different tasks of the same application or of using large portions of memory that may cause several cache misses. The delays caused by this interference are self-inflicted, and consequently in this work they are assumed *a priori* analyzed and part of the normal WCET of the tasks.
2) *Intra-core interference*: it is caused by tasks co-running in the same core. It is mainly composed of the context switch time and the time needed to solve cache misses caused by the other applications. It strongly depends on the cache architecture and arbitration.
3) *Inter-core interference*: it is the delay caused by cache misses in shared cache levels among cores and shared resource access by processes running in other cores, including the operating system resources. These are the most difficult to analyze, since they cover several system components and unpredictable timing interactions between them. In particular, system calls may cause unexpected delays in multi-core systems due to the kernel-space concurrency.

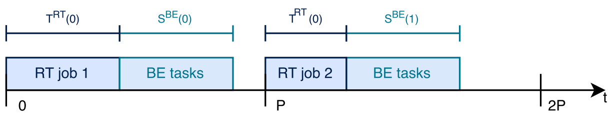
As already discussed, getting a strict upper bound of task WCET, even considering an isolation environment, it is difficult or even impossible in COTS platform. Computing the WCET of RT tasks is then necessary in order to apply a state-of-the-art measurement-based technique.

B. Resource management control

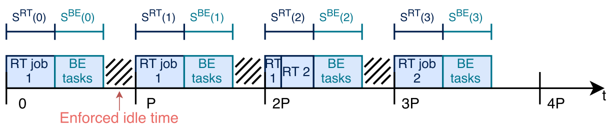
To simplify the subsequent analysis, let us make the following assumptions:

- only one single RT task τ_{RT} is present in the system, together with other BE tasks;
- the resource manager selects a constant enforcing period $P(z) = P$[1] and $\forall c_i \in C$ the time allocated to the RT task $S^{RT}_{c_i}$ and the time allocated to BE tasks $S^{BE}_{c_i}$. The fine-grained allocation of $S^{BE}_{c_i}$ to each BE task to maintain the best QoS is out of the scope of this paper;
- the RT task is divided in a sequence of inter-arrival jobs, considering the time-triggered model.

[1]in case of multiple RT tasks the selection of P can be made globally to the system or for single task. State-of-the-art schedulers are able to manage both conditions.



(a) Free-execution: the RT task is allowed to run until the end of the job. The resource manager has to set the correct enforcing period P.



(b) Constrained-execution: the RT task time is limited by $S^{RT}(z)$ and consequently the jobs are segmented over multiple periods.

Fig. 1. The two possible resource management controls.

These assumptions allow us to lighten the subsequently used notation, but they can be easily relaxed. In particular, the first assumption can be extended to have multiple tasks, considering the mutual interference of RT tasks in the WCET $\overline{W}^{RT}$.

A resource manager can enforce the CPU time assignment in MCS with two different policies: *Free Execution* and *Constrained Execution*.

1) Free Execution: The RT tasks are free to run until they have completed their jobs, then BE tasks can run during the remaining time until another RT task requires the CPU or S^{BE} is reached.

Selecting P as the inter-arrival period of the RT task allows the resource manager to select the fraction of remaining time to be assigned to BE tasks – and consequently the fraction of the remaining time in which the CPU remains idle. This scenario is depicted in Figure 1a.

Let be $c_{RT} \in C$ the CPU assigned to the RT task and $T^{RT}(z)$ the time actually used by the RT task in the period z. It follows that $T^{RT}(z) \leq P$ and:

$$T^{RT}(z) = T^{RT}_{\text{expected}}(z) + I(z) \tag{3}$$

where $T^{RT}_{\text{expected}}(z)$ is the execution time expected in case the real-time task runs in isolation. The assumption of an additive interference is generally valid [19] [20]. Thus, from the definition of WCET: $T^{RT}_{\text{expected}}(z) \leq \overline{W}^{RT}$.

$I(z)$ instead is the overhead caused by other tasks, in particular due to *intra-core* and *inter-core* interferences. It is easily noticeable that even if $\overline{W}^{RT} \leq D^{RT}$ the deadline can be missed if $I(z)$ is big enough to cause $T^{RT}(z) > D^{RT}$. Consequently, it is important to assess the magnitude of $I(z)$ in order to assign the correct time to RT and BE tasks.

Let be for any $c_i \in C$

$$\overline{S^{BE}_{c_i}}(z) = \{S^{BE}_{c_i}(0), S^{BE}_{c_i}(1), ..., S^{BE}_{c_i}(z-1)\}$$

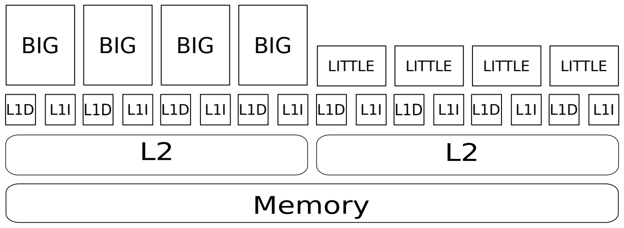


Fig. 2. The memory, cache and processor architecture of `big.LITTLE` Exynos5422.

the vector of previously assigned CPU c_i time to best effort tasks. Thus, one can write:

$$I(z) = d^{\text{curr}}(S^{BE}_{\forall c_i \neq c_{RT}}(z)) + d^{\text{prev}}(\overline{S^{BE}_{\forall c_i}}(z)) \qquad (4)$$

In this form $I(z)$ is the sum of two functions of the assigned execution time:

- d^{curr}: interferences generated in the current period, as a function of assigned time of BE tasks in other CPUs;
- d^{prev}: the remains from previous periods, as a function of previously assigned BE times.

Due to non-determinism and complexity of considered systems, analytical forms of d^{curr} and d^{prev} do not exist. However, Section V presents some experimental evaluation of the magnitude of them and it shows that d^{curr} exhibits a linear trend over the parameters. Under this assumption, d^{curr} can be written as:

$$d^{\text{curr}}(S^{BE}_{\forall c_i \neq c_{RT}}(z)) = \sum_{\forall c_i \in C, c_i \neq c_{RT}} d^{\text{curr}}_{c_i}(S^{BE}_{c_i}(z)) \qquad (5)$$

The effort required to the resource manager policy is mitigated since it can consider independently the inter-core interferences.

2) Constrained Execution: The RT task is allowed to execute in the CPU $c_{RT} \in C$ for a fraction of P, then it is suspended until the next period. In this case, the goal of the resource manager is even more critical, since a too small assignment of CPU time to RT task may lead to a deadline miss.

The period P may be smaller than inter-arrival time, in this way the computation of RT tasks can be split in several periods, according to the resource manager policy. This scenario is depicted in Figure 1b.

This is a typical case of continuous running processes – i.e. always in the *ready* state – that have to maintain a QoS. The execution time for the real-time task in period z is constrained by the allocation $S^{RT}_{c_{RT}}(z)$. Consequently, the allocated time must be:

$$\sum_{z=1}^{k} S^{RT}_{c_{RT}}(z) \geq \overline{W}^{RT} + \sum_{z=1}^{k} I(z) \qquad (6)$$

where $I(z)$ is the overhead due to interferences caused by other tasks and k is a sufficiently large number of periods. The previous considerations on $I(z)$ can be also applied for this case.

TABLE I
BENCHMARK MEASURED LATENCIES OF EXECUTION WITHOUT THE DISTURBANCES CAUSED BY STRESSER PROCESSES

Benchmark	**WCET** (μs)	**ACET** (μs)	**MCET** (μs)	**STDEV** (μs)	**STDEV** (%)
ctx	19.26	18.46	18.43	0.23	1.25
fcntl	19.39	16.98	17.85	1.75	10.29
mmap	228.71	223.63	223.60	1.17	0.52
proc	1222.60	1082.57	1084.08	37.83	3.49
signal	13.13	12.93	12.93	0.07	0.52
syscall	1.86	1.89	1.86	0.02	0.88
unix	55.43	56.22	55.40	0.26	0.47
fifo	64.96	64.65	64.64	0.14	0.22
fs	79.90	77.93	77.78	0.53	0.68
pipe	65.43	65.07	65.10	0.18	0.27
sem	2.14	1.81	1.80	0.10	5.35
udp	82.52	81.76	81.83	0.44	0.54

TABLE II
BENCHMARK MEASURED LATENCIES OF EXECUTION WITH THE PRESENCE OF STRESSER PROCESSES IN OTHER CORES

Benchmark	**WCET** (μs)	**ACET** (μs)	**MCET** (μs)	**STDEV** (μs)	**STDEV** (%)
ctx	19.79	19.26	19.31	0.25	1.28
fcntl	20.39	17.90	19.15	1.96	10.96
mmap	362.69	272.36	271.02	17.78	6.60
proc	1598.75	1424.74	1432.00	48.05	3.37
signal	13.85	13.41	13.23	0.32	2.39
syscall	1.99	1.95	1.96	0.01	0.68
unix	61.58	60.76	60.77	0.27	0.45
fifo	75.52	74.38	73.75	0.92	1.24
fs	91.38	87.76	88.49	2.96	3.38
pipe	72.62	72.35	72.35	0.10	0.14
sem	3.24	2.76	2.72	0.18	6.44
udp	86.37	85.24	85.19	0.36	0.42

V. Experimental Evaluation

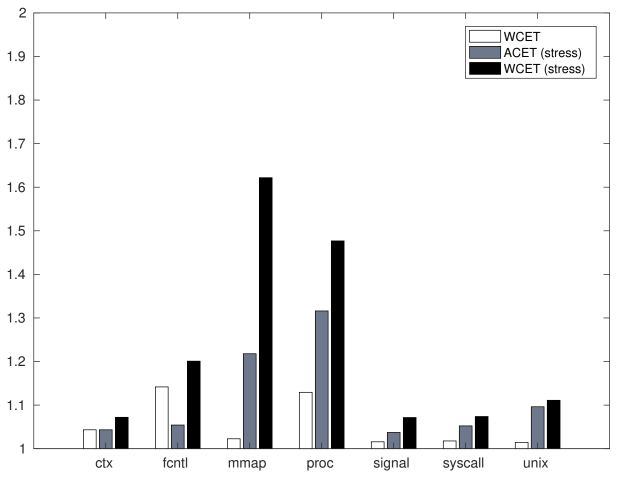
The effects of cache and memory sharing in MCS systems are well discussed in literature, thus the subsequent evaluation focuses on latencies induced by the operating system.

Specifically, in Linux user-space processes may affect the execution of other threads including OS system calls, even if the PREEMPT_RT patch is applied and the affected tasks have real-time priority. This is due to the contention of OS-level resources and synchronization mechanisms, that may lead to unpredictable latencies and thus unwanted non-deterministic behaviours.

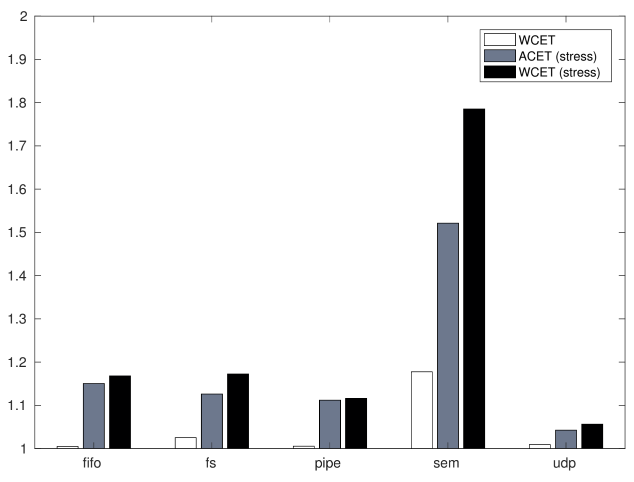
A. System setup

The system used in testing is an ODROID XU-3 powered by ARM `big.LITTLE` architecture as depicted in Figure 2. The Exynos5422 Cortex is composed of 4 Cortex A15 cores (`big`) and 4 Cortex A7 cores (`LITTLE`).

A Linux system able to achieve real-time constraints cannot trust only in the PREEMPT_RT patch. The system setup is a long and meticulous process that requires strong skills as also highlighted by previous work [8] [10]. The most important steps in order to obtain a Linux system with the lowest possible latencies/jitters and the highest degree of determinism are:



(a) First group



(b) Second group

Fig. 3. WCET and ACET overhead comparison of the first and second group of benchmarks. Data are normalized over the ACET in non-stressed condition.

- Properly configure the kernel at compile-time, by setting the preemption model, tuning the Read-Copy-Update (RCU) framework, selecting the appropriate timer tick policy, and many others small features to enable or disable.
- Select and configure the kernel modules to load. Each kernel module may introduce timers, kernel threads and IRQ routines, that have to be properly evaluated in terms of latencies and jitter. Usually, the non-essential kernel modules should not be loaded.
- Pin the IRQ threads and timers to CPUs where RT tasks are not expected to run, paying attention to the cache architecture.
- Selection and pinning of user-space system applications, based on the actual necessity of each of them.

After applying each configuration item, the effects on the system must be evaluated step-by-step. Linux provides several tools to check it, e.g. the analysis of interrupts and timers via the statistics provided by `/proc/*` files or the essential tool `ftrace` that allows one to analyze the kernel execution and its latencies [21]. The version of Linux used in next experiments is the 4.8 (October, 2016).

In addition to the system configuration, each RT application has to be modified to increase the determinism and avoid latencies [22]. First of all, the RT process has to set the RT scheduling policy (in our experiments SCHED_FIFO) and scheduling priority. Then, it should perform several steps to increase predictability, e.g. avoiding page-fault via stack pre-faulting and memory locking.

B. Methodology

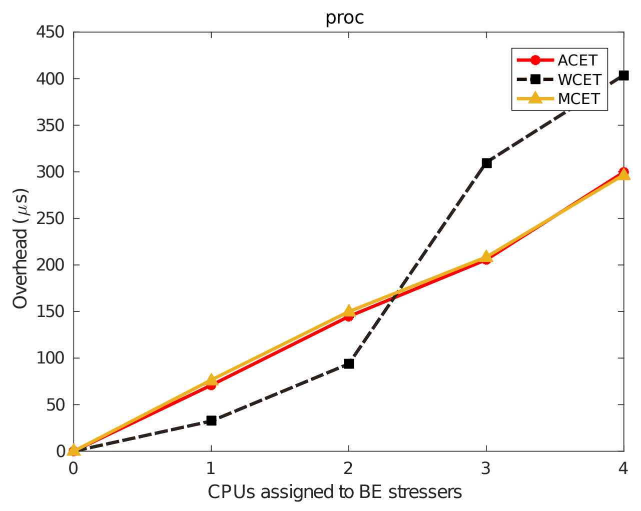
In the subsequent analyses the *lmbench* micro-benchmark [23] suite has been used. This suite is extensively used in literature and it provides several benchmarks to test the operating system performance. In particular, we focused on response latencies, i.e. most of the `lat_*` benchmarks:

- `ctx`: context switch time
- `fcntl`: locking/unlocking files latency
- `mmap`: time needed to map in memory 1MB of data
- `proc`: time needed to spawn a child and exit, it measure the performance of `fork` system call
- `signal`: latency to install and catch POSIX signals
- `syscall`: latency of a simple `getpid` system call
- `unix`: inter-process communication latency via UNIX sockets
- `fifo`: inter-process communication latency via FIFOs
- `fs`: time needed to create and delete 500 files
- `pipe`: inter-process communication latency via PIPEs
- `sem`: latency of `semop` system call
- `udp`: local communication latency via UDP sockets

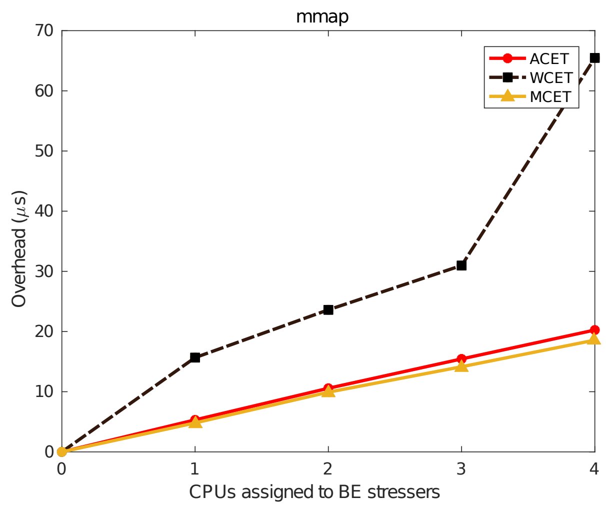
We slightly modified the suite in order to add the extra setup needed for RT applications and the measure of number of cycles (via the `PMCCNTR` register of the ARM CPU), besides the normal time measurement. This is performed in order to check during post-processing the data validity, since the pure time can be counterfeit by DVFS and clock instability.

In order to create stress-conditions in the system we used the `stress-ng` tool [24] – a derivative work from the `stress` program created by Waterland [25]. The tool allows us to select the system function to stress and the number of threads to be used.

ODROID XU-3 implements a non-bypassable DVFS that may enter in action if the temperature raises beyond a warning point. This behaviour is enabled even if the operating system is set to do not perform any frequency scaling. Unexpected frequency changes were a problem in first measurements in laboratory environment temperature. Consequently, we decided to fix the frequency of `big` cores to 1.8 GHz and frequency of `LITTLE` cores to 1 GHz. After this downgrading, we have never experienced any other frequency change. Moreover, to be minimize the risks, we also added an extra cooling system to the board and a software that monitors for any frequency



(a) proc



(b) mmap

Fig. 4. Deterioration of ACET, WCET and MCET w.r.t. their values in isolation. The overhead is caused by stresser processes on `proc` and `mmap` benchmarks as the number of CPU cores available for BE tasks increase.

switching during our measurements.

All the tests have been repeated several times as specified in next sections.

C. OS-induced overhead

To quantify the overhead induced by the operating system calls, the `big` cores were divided in two groups, two cores dedicated to run the `stress-ng` and two cores to run the benchmark. `LITTLE` cores are used to run all the migratable IRQs, system and kernel threads.

We selected several benchmarks from the *lmbench* suite and divided them in two groups based on the execution time[2]: the first group for benchmarks able to run in less than a second, and the second group for the other benchmarks. The former group was executed for 20000 times while the second group for 500 times. The tests are repeated in stressing conditions, properly tuning the parameters of `stress-ng` command, i.e. the number of threads to use and the size of cache levels.

Then, the usual statistics calculated on the results were: Worst-Case Execution Time (WCET), Average-Case Execution Time (ACET), the Median-Case Execution Time (MCET) and standard deviation. The results are shown in Table I for the normal execution and in Table II under full stress conditions.

In Figure 3a and Figure 3b the WCET and ACET of two groups of benchmarks can be compared between normal and stressed conditions.

It is possible to notice a deterioration of WCET in the range $[3; 59]\%$ and of the ACET in the range $[4; 52]\%$. In the `ctx`, `unix`, and `fs` benchmarks MCET is higher than ACET only in stressed conditions, showing a worsening of the distribution of the samples. Benchmarks `pipe` and `udp` perform better during stressed conditions in terms of MCET w.r.t. ACET and in terms of standard deviation, but the absolute values of WCET, ACET and MCET are worse than normal non-stressed case.

The standard deviation is in general higher for most of the benchmarks during stressing conditions, suggesting an increase of variability of the kernel-space execution. Consequently, the execution time of RT tasks becomes less predictable. This is expected in a complex OS like Linux where RT tasks run concurrently with BE tasks and sharing system resources.

From the data we can state that inter-process communications – like `fifo`, `pipe`, `udp`, `unix` – are less affected by the BE workloads. The absolute values of standard deviations slightly decrease, indicating that the syscall execution does not introduce further variability. Consequently, this reduction can be explained by the increasing of the total execution time in the absence of other variability sources.

Conversely, the `mmap`, `proc`, `sem` significantly impact on the ACET and WCET. Also `mmap` and `proc` increase their standard deviations, in particular the `mmap` standard deviation increases of a factor of 15 in absolute terms. The `mmap` is actually a library call implemented by the C library (`libc`) that in turn calls the `mmap_pgoff()` system call. This kernel-space function – and nested function calls – uses several semaphores and mutexes that are probably the cause of the remarkable increase of variability.

It is possible to conclude that a RT task may experience unexpected delays if it uses any system call, due to the presence of other tasks in the system, with some calls potentially impacting more than others. Even if our tests were executed in extreme stress conditions, the introduced overheads are not negligible and must be taken in account in the WCET analysis or limited by the resource manager. Besides this, the different variability effects must be taken in account through an accurate profiling of the application. Depending on which system calls

[2]This *execution time* refers to the time of running the entire benchmark, including the application setup. It should not be confused by the measured latencies.

are used in the RT and BE tasks, the confidence interval width considered by the resource management policy has to be tuned according to the previous variability considerations.

D. Experimental verification of linearity

In order to check if the term d^{curr} in Equation 4 can be written as a sum of independent interferences of each core, we tested a linearity in the increasing of overhead when the number of CPUs dedicated to stresser processes increases. Also in this experiment `stress-ng` was used for stressing the cores dedicated to BE tasks, and we selected the `proc` and `mmap` to respectively test the process and the memory management of the Linux kernel.

As we can see in Figure 4a and Figure 4b, ACET and MCET maintain a very good linear trend, WCET presents an higher variability but consistent with the standard deviation of these benchmarks. Consequently, at least for 4 cores or less, it is possible to assume valid the Equation 5, i.e. d^{curr} is linearly independent with respect to the single d_c^{curr} of each core.

VI. Conclusions and Future Works

The interest in Linux real-time systems is increasing and with that the challenges to be addressed. As highlighted in the state of the art, a systematic analysis of real-time capability of recent Linux kernel is required. Specifically, the impact of PREEMPT_RT patch and the subsequently comparison with co-kernel approaches have to be assessed. In our opinion, the lack of this analysis is one of the most urgent topic, especially for the use of Linux in industrial applications.

This article proposed a model to be used by a resource manager in order to properly allocate tasks over available CPUs. In the experimental part, we performed a quantitative analysis on the OS-induced overhead when real-time tasks are executed in MCS context. In this regard, we discovered that a non real-time task may increase the WCET and ACET of a critical task of over 50% only using Linux system calls. To this overhead, the well known interferences caused by caches have to be added. At the end, we verified that the interferences caused by multiple tasks on different CPUs can be considered independent among them, simplifying the proposed model.

This work wants to be the first but essential step to extend the *Barbeque Run-Time Resource Manager* [26] as a possible solution to mitigate such interferences enabling an efficient execution of mixed-criticality tasks on a Linux platform.

Acknowledgments

This work has been partially funded by EU H2020 projects M2DC (id: 688201) and MANGO (id: 671668).

References

[1] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[2] A. Burns and R. Davis, "Mixed criticality systems-a review," *Department of Computer Science, University of York, Tech. Rep*, 2013.

[3] J. Nowotsch and M. Paulitsch, "Leveraging multi-core computing architectures in avionics," in *2012 Ninth European Dependable Computing Conference*, May 2012, pp. 132–143.

[4] H. Shah, A. Raabe, and A. Knoll, "Challenges of wcet analysis in cots multi-core due to different levels of abstraction," in *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, vol. 85, 2013, p. 95.

[5] P. Mantegazza, E. Dozio, and S. Papacharalambous, "Rtai: Real time application interface," *Linux Journal*, vol. 2000, no. 72es, p. 10, 2000.

[6] P. Gerum, "Xenomai-implementing a rtos emulation framework on gnu/linux," *White Paper, Xenomai*, p. 81, 2004.

[7] F. Gosewehr, M. Wermann, and A. W. Colombo, "From rtai to rt-preempt a quantative approach in replacing linux based dual kernel real-time operating systems with linux rt-preempt in distributed real-time networks for educational ict systems," in *Industrial Electronics Society, IECON 2016-42nd Annual Conference of the IEEE*. IEEE, 2016, pp. 6596–6601.

[8] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Linux preempt-rt vs. commercial rtoss: how big is the performance gap?" *GSTF Journal on Computing (JoC)*, vol. 3, no. 1, p. 136, 2013.

[9] F. Cerqueira and B. Brandenburg, "A comparison of scheduling latency in linux, preempt-rt, and litmus rt," in *9th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. SYSGO AG, 2013, pp. 19–29.

[10] H. Fayyad-Kazan, L. Perneel, and M. Timmerman, "Linux preempt-rt v2. 6.33 versus v3. 6.6: better or worse for real-time applications?" *ACM SIGBED Review*, vol. 11, no. 1, pp. 26–31, 2014.

[11] O. Kotaba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling, "Multicore in real-time systems–temporal isolation challenges due to shared resources," in *Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems*, 2014.

[12] J. Nowotsch, M. Paulitsch, D. Bühler, H. Theiling, S. Wegener, and M. Schmidt, "Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement," in *2014 26th Euromicro Conference on Real-Time Systems*. IEEE, 2014, pp. 109–118.

[13] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele, "Mapping mixed-criticality applications on multi-core architectures," in *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*. IEEE, 2014, pp. 1–6.

[14] A. Kritikakou, C. Pagetti, O. Baldellon, M. Roy, and C. Rochange, "Run-time control to increase task parallelism in mixed-critical systems," in *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*. IEEE, 2014, pp. 119–128.

[15] B. Huber, C. El Salloum, and R. Obermaisser, "A resource management framework for mixed-criticality embedded systems," in *Industrial Electronics, 2008. IECON 2008. 34th Annual Conference of IEEE*. IEEE, 2008, pp. 2425–2431.

[16] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole, "A measurement-based analysis of the real-time performance of linux," in *Proceedings. Eighth IEEE Real-Time and Embedded Technology and Applications Symposium*, 2002, pp. 133–142.

[17] A. K. Mok, X. Feng, and D. Chen, "Resource partition for real-time systems," in *Proceedings Seventh IEEE Real-Time Technology and Applications Symposium*, 2001, pp. 75–84.

[18] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, "A survey on cache management mechanisms for real-time embedded systems," *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 32, 2015.

[19] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory access control in multiprocessor for real-time systems with mixed criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*. IEEE, 2012, pp. 299–308.

[20] D. B. Oliveira and R. S. Oliveira, "Timing analysis of the preempt rt linux kernel," *Software: Practice and Experience*, 2015.

[21] S. Rostedt, "Finding origins of latencies using ftrace," *Proc. RT Linux WS*, 2009.

[22] D. Duval, "From fast to predictably fast." Citeseer, 2009.

[23] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[24] stress-ng. [Online]. Available: http://kernel.ubuntu.com/ cking/stress-ng/

[25] A. Waterland, "stress posix workload generator," 2013.

[26] P. Bellasi, G. Massari, and W. Fornaciari, "A rtrm proposal for multi/many-core platforms and reconfigurable applications," in *Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC)*. IEEE, 2012, pp. 1–8.