designs

MDPI

Article

A Lazy Bailout Approach for Dual-Criticality Systems on Uniprocessor Platforms

Saverio Iacovelli † **and Raimund Kirner** *,†

The School of Engineering and Computer Science, University of Hertfordshire, College Lane, Hatfield AL10 9AB, UK; savio.iacovelli@yahoo.com

* Correspondence: r.kirner@herts.ac.uk; Tel.: +44-1707-28-4125

† These authors contributed equally to this work.

Received: 20 October 2018; Accepted: 28 January 2019; Published: 1 February 2019

Abstract: A challenge in the design of cyber-physical systems is to integrate the scheduling of tasks of different criticality, while still providing service guarantees for the higher critical tasks in the case of resource-shortages caused by faults. While standard real-time scheduling is agnostic to the criticality of tasks, the scheduling of tasks with different criticalities is called mixed-criticality scheduling. In this paper, we present the *Lazy Bailout Protocol* (LBP), a mixed-criticality scheduling method where low-criticality jobs overrunning their time budget cannot threaten the timeliness of high-criticality jobs while at the same time the method tries to complete as many low-criticality jobs as possible. The key principle of LBP is instead of immediately abandoning low-criticality jobs when a high-criticality job overruns its optimistic WCET estimate, to put them in a low-priority queue for later execution. To compare mixed-criticality scheduling methods, we introduce a formal quality criterion for mixed-criticality scheduling, which, above all else, compares schedulability of high-criticality jobs and only afterwards the schedulability of low-criticality jobs. Based on this criterion, we prove that LBP behaves better than the original *Bailout Protocol* (BP). We show that LBP can be further improved by slack time exploitation and by gain time collection at runtime, resulting in LBPSG. We also show that these improvements of LBP perform better than the analogous improvements based on BP.

Keywords: real-time systems; Fixed-Priority Preemptive Scheduling (FPPS); mixed-criticality systems; cyber-physical systems

1. Introduction

Cyber-physical systems (CPS) typically require the integration of services of different criticality. At the same time, it is important that tasks of lower criticality have limited leverage to influence the schedulability of tasks with higher criticality in the case of resource shortages. Traditional real-time scheduling protocols, such as *rate-monotonic scheduling* (RMS) or *earliest deadline first* (EDF) [1], give priority to jobs with the most strict timing requirements. This approach works well as long as it can be assured that enough resources are available to schedule all tasks. However, in cases where availability of enough resources cannot be guaranteed, traditional real-time scheduling methods miss the flexibility to prioritise the resources to tasks of higher criticality.

Research on mixed-criticality scheduling protocols [2,3] has been started to overcome this limitation. The basic idea of mixed-criticality scheduling protocols is that, as long as enough resources are available, the scheduling priorities are defined by a real-time scheduling protocol. In the case of a resource shortage, e.g., a job overrunning its estimated worst-case execution time (WCET) [4], the tasks' criticalities are used as the primary criterion to allocate resources. A task's criticality can be derived from different aspects. One possibility is to express the relative importance or relative utility

of different services in a system as their criticality [5]. Another possibility is to express the relative level of assurance, for example, dictated by different development standards for safety critical or relevant systems, such as DO-178C [6] in the avionics domain, ISO26262 [7] in the automotive domain, or IEC 61508 [8] in the automation domain as different levels of criticality. However, the meaning of criticality is still sometimes subject of discussion, with Esper et al. assuming different execution modes [9] not originally described by Vestal [2]. In this paper, we do not mandate a specific procedure for defining criticality levels, as this is an orthogonal issue to the mixed-criticality scheduling discussed in this paper.

To apply mixed-criticality scheduling, at least two levels of criticality have to be defined, typically labelled as LO (low-criticality) and HI (high-criticality). A common approach is to assume for LO tasks only the knowledge of easy to derive optimistic WCET estimates while for HI jobs also a higher level of assurance based on safe upper WCET bounds is assumed. The active research challenge is to find ways to effectively combine the resource prioritisation based on criticalities with the scheduling priorities based on real-time constraints.

Recent mixed-criticality scheduling approaches are the *Bailout Protocol* (BP) by Bate et al. [10] and its extension that exploits the system slack time, named *Bailout Protocol-Slack* (BPS). The authors afterwards presented further extended versions of the BP, aiming at a higher utilisation of LO jobs. Such extensions use a dynamic approach to deploy gain times in order to reduce the duration and number of times the system switches to high-criticality execution mode and are denoted as *Bailout Protocol with Gain Time* (BPG) and *Bailout Protocol-Slack and Gain Time* (BPSG) [11].

This article contains the following contributions:

1. *Lazy Bailout Protocol* (LBP), which is a mixed-criticality scheduling protocol inspired by the *Bailout Protocol* (BP) from Bate et al. [10,11], is introduced. Compared with BP and its derivatives, LBP does not abandon jobs immediately but rather keeps them for potential later execution during idle periods of the processor.
2. A formal criterion to compare different mixed-criticality scheduling protocols with priority given to high-criticality jobs is defined.
3. LBP is combined with the complimentary techniques used in BPG, BPS and BPSG, resulting in LBPG, LBPS and LBPSG, respectively, proving that LBP and its derivatives perform better than their corresponding BP-based protocols according to such a formal criterion.
4. The comparison and evaluation of BP, LBP and their derivatives protocols in a hard real-time setting is presented.

Section 2 presents an overview of the state of the art in mixed-criticality scheduling. A precise presentation of the scheduling problem is presented in Section 3. We present a new mixed-critcality approach named LBP in Section 4 that does not suddenly abandon LO task instances during resource shortages. In Section 5, we derive formal properties of LBP and its derivatives. Section 6 provides an experimental evaluation of the performance of the LBP-based approaches compared with other methods. Section 7 concludes the article.

2. Related Work

Most of the works about mixed-criticality systems that have been published by the real-time scheduling research community is based on a model proposed by Vestal [2]. The system model consists of a set of periodic tasks that perform functions having different criticalities and requiring different levels of assurance. Each task may have a set of alternative worst-case execution times, with each assured to a different level of confidence. The more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to become in practice. The final aim was to guarantee that safety-critical task instances do not miss their deadlines.

Crespo et al. reviewed the challenges of applying mixed-criticality in control systems and studied the possibility of using virtualisation as basis for building mixed-criticality partitioned

software architectures [12]. Their work reviews the challenges connected to systems with virtual partitions having different criticality that are executed in an independent way. Such systems are based on a hypervisor that provides temporal, spatial and fault isolation among partitions that contain components that have to be guaranteed at different assurance levels and on hierarchical scheduling as strategy to process jobs.

Ernst and Natale provided an explanation about the meaning of criticality and a review about the mixed-criticality model in current real-time research [13]. They highlighted how functional safety standards usually provide the basis to design industrial mixed-criticality systems. In fact, all industrial safety standards classify different levels of concern, called *Safety Integrity Levels* (SIL) in IEC 61508, *Automotive Safety Integrity Levels* (ASIL) in ISO 26262 or *Design Assurance Level* (DAL) in DO-178C. Each level involves a certain likelihood to perform successfully the required functions under certain conditions and within a stated period. In such standards, the definition of criticality levels is usually obtained as a result of a *Failure Modes, Effect and Criticality Analysis* (FMECA) process. However, these standards focus on the safety targets while engineers normally focus on metrics such as cost, performance and power consumption that are often in conflict with safety requirements. Such contrast grows with the autonomous driving and with the integration challenges derived from cyber-physical systems and Internet of Things.

Burns and Davis published a survey of research on mixed-criticality systems [14].The review contains an historical introduction of the topic and the challenges faced in developing better mixed-criticality on both single- and multi-processor systems. The key question emerging from their work is how to reconcile the conflicting requirements of partitioning for safety and sharing for efficient resource usage. Lastly, the review contains criticisms and limits of the current mixed-criticality approaches.

In 2011, Baruah et al. extended Vestal's model by proposing a refinement named *Adaptive Mixed-Criticality* (AMC) protocol together with related mixed-criticality response-time analysis techniques [15]. Such mixed-criticality schedulability tests have been recently extended for task sets containing tasks with arbitrary deadlines [16]. In 2013, Fleming et al. extended the response time bound techniques and the AMC protocol to work with multiple criticality levels [17]. The AMC protocol assumes two execution modes, a low-criticality mode (indicated as LO) and a high-criticality mode (indicated as HI). Once the system goes into the high-criticality mode, all LO task instances are abandoned and the system remains in that mode. However, to move mixed-criticality research into industrial practice, it is important to implement protocols whose runtime behaviour is acceptable for system engineers. Abandoning all LO tasks in high-criticality mode is not an acceptable behaviour and the system should return to the low-criticality mode, where all functionalities are provided, as soon as conditions are appropriate. Therefore, a simple but necessary extension to AMC is to allow a switch back to the starting mode when the system experiences an idle instant.

However, going back to the low-criticality starting mode only in case of idle instants leads to a high amount of LO tasks interrupted or abandoned and this is still not satisfactory. Different complementary ways of guaranteeing a higher level of service for LO tasks have been proposed, e.g., extending their periods and/or deadlines such as in the elastic task model [18] or reducing their execution times by switching to a simpler version of the software [19].

The *Priority May Change* (PMC) strategy has been proposed to better manage the overload situations in which higher priority LO tasks could preempt lower priority HI tasks [20]. The AMC algorithm assigns a single priority to each task by considering together both low- and high-criticality modes, whereas PMC computes priorities in two steps. Firstly, priorities are assigned to tasks according to some predefined policy such as deadline monotonic [21] and such priorities are used while the system is in low-criticality execution mode. Once the system switches to high-criticality mode, HI task priorities are re-assigned according to a priority ordering policy that is optimal for tasks with release jitter [22]. However, PMC does not dominate the standard AMC but it has performances similar to it.

In 2014, Fleming and Baruah proposed a scheme in which system designers can assign to lower critical functionalities a utility that is used to decide in which order their instances have to be suspended during an overload occurrence [23]. Such method allows the system designer to control how non-critical functionalities degrade after the most critical ones overrun their optimistic time threshold. The utility value is assigned as an ordinal scale [24] to provide a predefined order in which LO task instances are abandoned, with least important task instances being abandoned first. The authors adapted the Audsley priority assignment technique [25] to assign lower priority to lower utility LO tasks. Such protocol allows increasing performances for LO tasks and processing them for a significantly increased amount of time.

Somehow, the former methods considered thus far allow for LO task invocations to execute after a criticality mode change but they are mainly best effort and do not have a predefined minimum threshold guaranteed for lower critical tasks. Since most hard real-time systems could miss some deadlines provided that it happens in a known and predictable way, the *Adaptive Mixed Criticality with Weakly-Hard constraints* (AMC-WH) was introduced in 2015 [26] and represents an extension of AMC [15] that integrates the notion of weakly-hard constraints. The definition of weakly-hard real-time system was given in 2001 [27] to indicate systems in which hard real-time tasks are permitted to miss some deadlines as long as the number of missed deadlines is strictly bounded. The AMC-WH is a scheduling policy that allows a number of consecutive instances per LO task to be skipped during the high-criticality execution mode. This reduces the overall system load, frees more resources for highly critical tasks and provides a degraded but guaranteed minimum quality of service for LO tasks upon a criticality mode change. The number of skips permitted and the number of subsequent deadlines that must be met could be a requirement deduced either from the design of a control algorithm [28] or from physical properties of the system. Even if AMC-WH allows scheduling more LO task instances if compared with previous policies, it does not provide a fast recovery from the high-criticality execution mode since it is still necessary to wait for idle instants to go back to the starting mode. This leads to unnecessary abandonments of LO instances.

Such problem was considered with the *Bailout Protocol* (BP) [10]. The BP still represents an AMC refinement and hence exhibits both low- and high-criticality execution modes. The low-criticality mode is named *Normal* mode while the high-criticality execution mode is represented by both the *Bailout* and *Recovery* modes. Similar to AMC, the system starts its execution in the LO criticality mode, *Normal* mode, and whenever a HI job exceeds its optimistic WCET, then it switches to the Bailout mode. The protocol aims to restore the normal execution mode as soon as possible to minimize the number of LO instances that miss their deadlines or are not executed at all. LO tasks are still abandoned during the high-criticality execution but they contribute to make the switch back to the starting execution mode faster by means of a *Bailout Fund* (BF). In fact, if BF becomes not strictly positive during the Bailout mode, the system enters the Recovery mode to allow the lowest priority pending HI job to complete its execution before going back to the *Normal* mode without waiting for an idle instant. Once the system is back to the starting mode, all lower critical functionalities start again to be processed with their full timely behaviour. The strength of this protocol is that to provide an effective control mechanism to go back to the low-criticality mode, where all jobs can start and being processed. However, the main weakness of BP is that to immediately drop low-critical instances during the high-criticality modes. Because of this, the percentage of LO jobs that miss their deadlines is still high.

An orthogonal approach to improve the overall service for LO tasks is based on a method introduced by Santy et al. [29]. This approach was subsequently refined by Burns and Baruah [19]. They scaled up the optimistic WCETs of HI tasks using sensitivity analysis until the system is schedulable. If used together with the BP the resulting protocol is named *Bailout Protocol-Slack* (BPS). More recently, Bate et al. further refined BP with a second complementary technique [11]. Such approach consists of an update of the optimistic time budget made at runtime by collecting the so-called gain time, i.e., the spare CPU time not required at runtime by task instances. These techniques allow reducing both the number of times and the duration the system executes in high-criticality modes. By combining

the online gain time collection with BP, the authors introduced two new scheduling protocols that are named *Bailout Protocol-Gain Time* (BPG) and *Bailout Protocol-Slack and Gain Time* (BPSG).

3. System Model

In the following, the system model used for task sets is described. A dual-criticality system, which consists of multiple tasks, where each task has a criticality $l \in \{LO, HI\}$ with *HI* being of higher criticality than *LO*, is assumed. As discussed in Section 1, the criticality of a task can be derived by different means but no specific interpretation of criticality is assumed, as this is orthogonal to the scheduling method presented in this paper. Furthermore, it is assumed that the processor is the only resource that is shared among tasks, and that the overheads due to the scheduling operations and context switches can be bounded by a constant included within each task worst-case execution times.

We consider a set of independent and sporadic tasks τ that has to be processed on uniprocessor systems and that consists of two sub sets:

$$\tau = \tau_{LO} \cup \tau_{HI} \tag{1}$$

with

$$\tau_{LO} = \{\tau_i \in \tau \mid l_i = LO\} \tag{2}$$

$$\tau_{HI} = \{\tau_i \in \tau \mid l_i = HI\} \tag{3}$$

where τ_{HI} is the subset of tasks that are highly critical and τ_{LO} is the subset of tasks that are not highly critical within the system.

The tasks represent scheduling units that the system has to perform. An individual task $\tau_i \in \tau$ is represented by the following tuple:

$$\tau = \langle T, D, C_{LO}, C_{HI}, L \rangle$$

where T is the period, D is the relative deadline, C_{LO} and C_{HI} are, respectively, the optimistic and the pessimistic worst case execution times and $L \in \{LO, HI\}$ refers to the criticality. In this paper, for simplicity, we assume implicit deadlines, i.e., tasks with deadlines equal to their periods: $D = T$.

A job is an instance of a task at runtime, i.e., a job represents the actual object processed by the scheduler and inherits almost all properties from the task that generates it plus the arrival time A as below:

$$j_i = \langle A, D, C_{LO}, C_{HI}, L \rangle$$

The LO tasks and, as a consequence, their relative jobs do not have a known safe WCET bounds C_{HI}, since safe worst-case execution times are rather costly to obtain and thus provided only for HI tasks. Once it finishes its execution, each job j_i has got a computation time $et(j_i)$ that can vary for each specific job of the same task. The job set produced by an individual task τ_i is indicated by $J(\tau_i)$ while $J(\tau)$ is the job set produced by all tasks belonging to the task set τ. Therefore, τ represents the set of activities that have to be performed by the system while J represents the set of concrete process instances that have to be considered by the scheduler.

The jobs produced via the task set are scheduled according to the standard fixed-priority fully pre-emptive real-time scheduling. However, the traditional fixed-priority scheduling is unaware of criticality of task instances and scheduling decisions are only made according to priority that indicates the job timing requirements. Therefore, it is also used a protocol that considers the task's criticality to meet the mixed-criticality requirements. The following assumptions are made about the task set and the underlying real-time scheduler, i.e., fixed priority fully pre-emptive scheduling:

Assumption 1. *All HI and LO jobs together are schedulable with the underlying real-time scheduling method with respect to their* C_{LO}.

Assumption 2. *All HI jobs alone are schedulable with the underlying real-time scheduling method with respect to their C_{HI}. Since C_{HI} is a safe WCET bound, i.e., $et \leq C_{HI}$, this assumption also implies that the HI jobs alone are schedulable with respect to their actual execution time.*

Assumption 3. *All HI jobs are schedulable with respect to their C_{HI}, while also assuming the execution of all LO jobs arrived in Normal mode with respect to their C_{LO}.*

Note that Assumption 3 is required so that, while LO tasks are allowed to run within their C_{LO}, it is still ensured that all HI tasks are still schedulable within their C_{HI}. Assumption 3 is based on jobs rather than tasks as it covers the moment in time when a HI task overruns its C_{LO}. In addition, note that Assumption 2 is just a weaker case of Assumption 3, without the LO tasks considered.

4. The Lazy Bailout Protocol

The standard BP is an adaptive protocol to schedule mixed-criticality job sets. The strength of BP is providing an effective and fast control mechanism to go back to the low-criticality mode, where all jobs can start and being processed. However, the main weakness of BP is immediately abandoning LO jobs in case of resource shortage, which leads to a high percentage of jobs that miss their deadline.

The *Lazy Bailout Protocol* (LBP) is built upon BP and inherits from it the following three execution modes that work as specified below:

1. *Normal*: It is the starting system execution mode. It corresponds to a low-criticality mode where all jobs within the system are supposed to be processed correctly according to the C_{LO} threshold.
2. *Bailout*: It is the emergency mode that is entered whenever a HI job overruns its C_{LO}.
3. *Recovery*: It is the emergency mode that is entered to allow the last pending lowest priority HI job to complete before going back to *Normal* mode.

Figure 1 shows the components of LBP. The LBP filter is responsible for changing the execution modes. The system has two ready queues for jobs: the *high-priority queue* represents the BP ready jobs queue while the *low-priority queue* keeps the LO jobs that have been released during Bailout or Recovery modes or that have exceeded their C_{LO}. Note that LO jobs inserted into the low-priority queue run until their deadline and *only* when the high-priority queue is idle. Thus, such jobs cannot lead to any deadlines being missed. There are two job monitors to check, respectively, LO and HI jobs that overrun their C_{LO}. *ET-MonLO* signals to the real-time scheduler the LO jobs that have to be inserted within the low-priority queue while *ET-MonHI* communicates to the LBP filter when a HI job exceeds its optimistic WCET to switch the execution mode to Bailout.

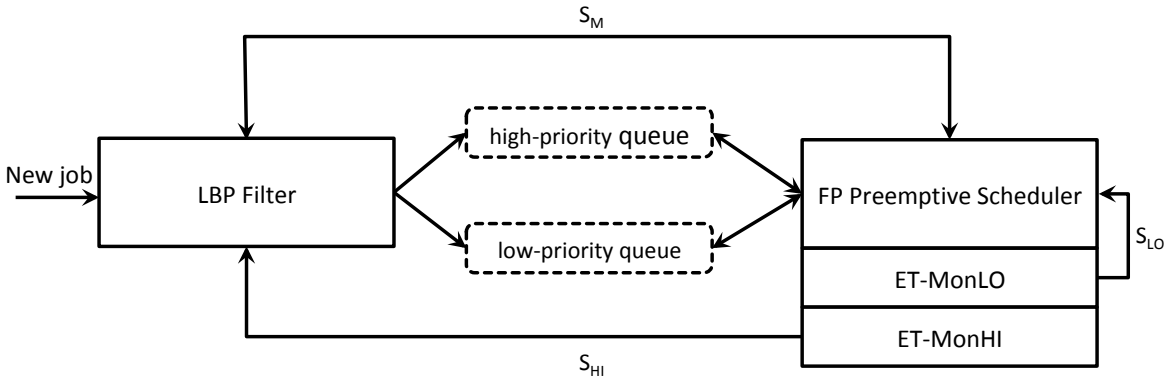


Figure 1. LBP architecture.

Similar to BP, LBP inherits from AMC the system execution behaviour, i.e., the system starts in a low-criticality execution mode and whenever a HI job exceeds its optimistic WCET, the system switches to a high-criticality execution mode where any LO job execution is prevented. Finally,

the system goes back the starting execution mode in case of idle instant. Furthermore, LBP inherits from BP the control mechanism that is in charge of the execution mode changes that permits a fast recovery from the Bailout/Recovery modes back to the Normal mode. Such mechanism is based not only on the detection of a idle instant but also on the value of a decision variable named *Bailout Fund* (BF). It is worth noting that LBP, as well as AMC and BP, implement dispatching policies that are independent and separated from the priority assignment used. Moreover, since a fixed-priority scheduler is used, no priority change is allowed. Figure 2 shows how the execution mode changes in the scheduling protocol. It contains the events that trigger the switch to a different execution mode together with the related update of the BF value. The system starts in Normal mode and then, if any HI job overruns its C_{LO}, the BF variable is initialised and there is a change to Bailout mode. Once the system is in this mode, the BF variable is updated with the earlier completion of jobs, the release of new LO jobs or the HI jobs overrunning their C_{LO}. If an idle instant occurs, then Normal mode is entered, whereas, if the BF becomes zero, then Recovery mode is entered. After the pending lowest priority HI job completes its execution in Recovery mode, the system goes back to Normal mode.

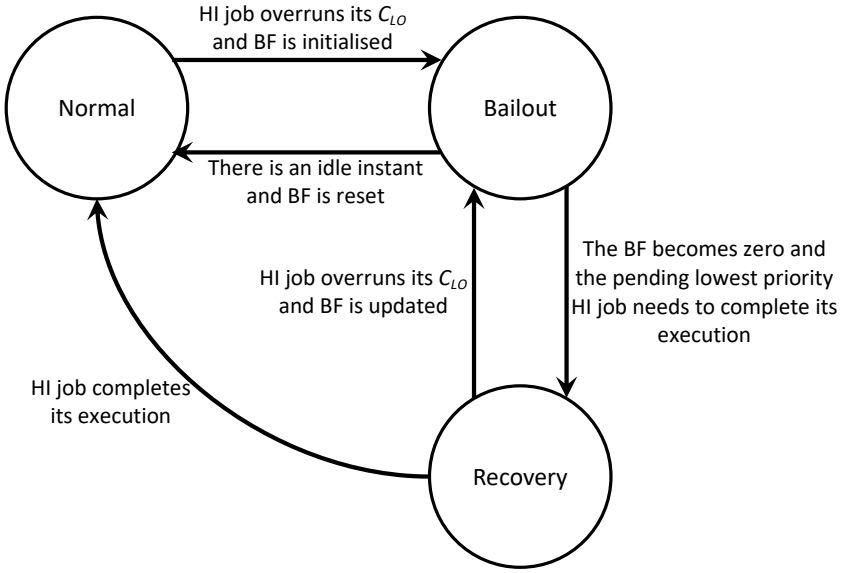


Figure 2. Execution mode changes in LBP.

The difference between LBP and BP is that LO jobs released in Bailout and Recovery modes or exceeding their C_{LO} are inserted into the low-priority queue instead of being abandoned. This allows increasing the amount of LO jobs scheduled without interfering with the execution of jobs in the high-priority queue. In fact, LO jobs in the low-priority queue run until their deadline when the high-priority queue is idle. The essential difference in scheduling behaviour is that, in those cases where BP would be idle, LBP might have some tasks preserved in the low-priority queue that can now successfully be executed.

Whenever a job in the low-priority queue misses its deadline, it is removed. LO jobs released in Normal mode can continue to execute in both Bailout and Recovery modes and they could even overrun their deadlines as long as they do not exceed their C_{LO}. Below, is a detailed description of LBP in each of its execution modes:

Normal mode:

- While all HI jobs execute for no more than their C_{LO} values, the system remains in this mode.
- If any HI job overruns its C_{LO} without signalling completion, then the system switches into the *Bailout* mode and the BF is initialised to $BF = C_{HI} - C_{LO}$.
- LO jobs that overrun their C_{LO} are interrupted and inserted into the low-priority queue.

- LO jobs that have been inserted into the low-priority queue are executed during idle instants. If they do not complete within their deadlines, then they are removed from the low-priority queue.

Bailout mode:

- If any HI job executes for its C_{LO} without signalling completion, then the bailout fund is updated by its maximum extra time budget: $BF = BF + (C_{HI} - C_{LO})$.
- If any HI job completes with an execution time e, with $e \leq C_{LO}$, then its time left is donated to the bailout fund: $BF = BF - (C_{LO} - e)$.
- LO jobs released in *Normal* mode that complete with an execution time of e, with $e \leq C_{LO}$, donate their time left to the bailout fund: $BF = BF - (C_{LO} - e)$.
- If any HI job that already exceeded its C_{LO} completes with an execution time of e, with $C_{LO} < e \leq C_{HI}$, then it donates its extra time left, reducing the bailout fund: $BF = BF - (C_{HI} - e)$.
- LO jobs released in *Bailout* mode are not started but inserted in the low-priority queue to be executed during idle instants in *Normal* mode. Furthermore, when the scheduler would otherwise have dispatched such a job, the job's budget of C_{LO} is donated to the bailout fund: $BF = BF - C_{LO}$.
- If the BF becomes zero, then the lowest priority HI job that did not complete its execution (let this job be j_k) is recorded and the *Recovery* mode is entered.
- If an idle instant occurs, then a transition is made to *Normal* mode, and BF is reset to zero.

Recovery mode:

- LO jobs released in this mode are not started but inserted within the low-priority queue to be executed during idle instants in *Normal* mode.
- If any HI job executes for its C_{LO} value without signalling completion, then the system switches back to *Bailout* mode and BF is initialised: $BF = C_{HI} - C_{LO}$.
- When the job j_k noted at the point when *Recovery* mode was last entered completes, then the system switches to *Normal* mode.

Figure 3 shows how the same task set is scheduled according to the On the one hand, the standard BP abandons all the LO jobs released during the HI criticality execution modes while the lazy approach allows to recover and schedule more LO jobs. In particular, in Figure 3b, jobs B_1 and B_4 are released, respectively, at times $t = 6$ and $t = 24$ and they have the highest priority. Such jobs are inserted in the low-priority queue to be removed, respectively, at times $t = 12$ and $t = 30$ when they miss their deadlines and the next instance of the same task arrives. Furthermore, the LO jobs B_2 and B_5 released respectively at times $t = 12$ and $t = 30$ are executed afterwards in Normal mode since there are idle instants to exploit before their deadlines. Such example highlights how LO jobs that are delayed, instead of being abandoned, are executed during idle instants in Normal mode to not influence the real-time behaviour of jobs in the high-priority queue. Overall, compared with LBP, the standard BP results in a decrease of the system utilisation because, whenever there is interference among HI and LO jobs released in Bailout or Recovery modes, LO jobs are simply abandoned. On the other hand, LBP increases the processor utilisation by exploiting the system idle time and, by doing this, it improves the overall service provided to LO tasks, which is achieved by increasing the number of LO jobs that are processed.

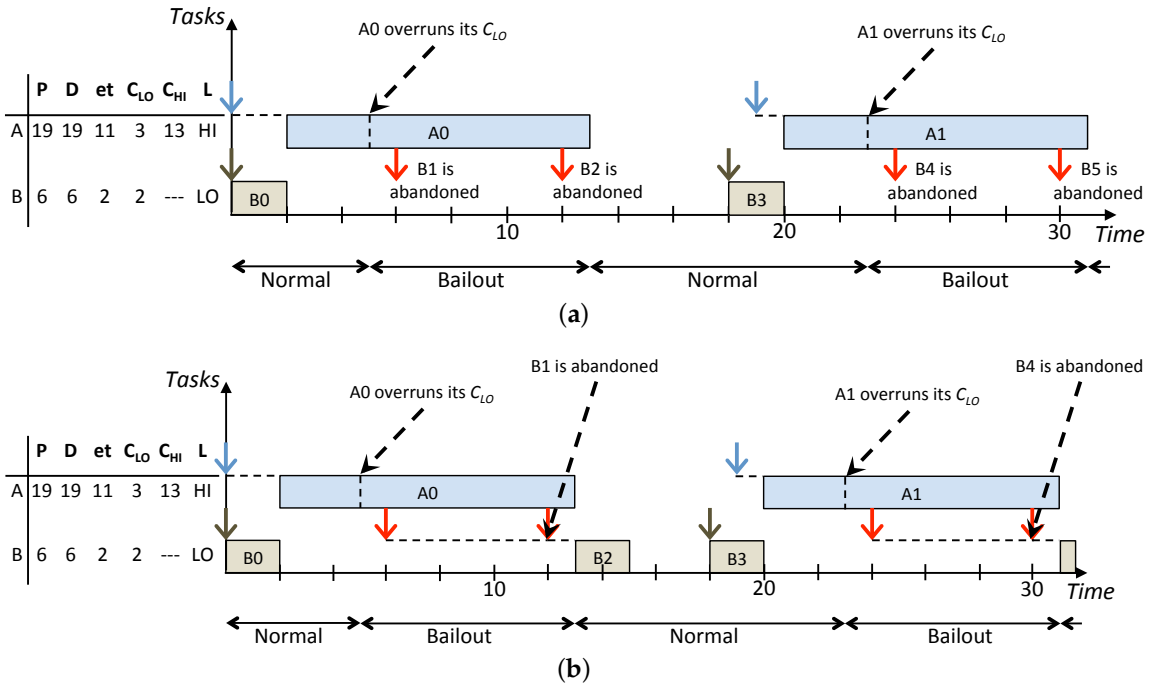


Figure 3. Comparison between BP and LBP: LBP schedules more LO jobs than BP: (**a**) BP abandons all LO jobs released in Bailout mode; and (**b**) LBP rescues the LO jobs $B2$ and $B5$, while $B1$ and $B4$ are abandoned after they miss their deadlines.

5. Proofs

In this section, we formalise a criterion to compare different mixed-criticality systems. Below are definitions and predicates used to prove the theorems afterwards.

STS, τ, JS**:**
: STS is a set of task sets τ. τ is an individual scheduling problem consisting of tasks. JS is a set of jobs created at runtime by scheduling a task set.

METHOD**:**
: This is the scheduling method applied, which can be BP, LBP or some of their derivatives resulting from the integration with the offline sensitivity analysis or with the online gain time collection.

HI(τ), LO(τ): $\tau \rightarrow \tau$**:**
: HI(τ) is a subset of τ containing only tasks of high-criticality. LO(τ) is a subset of τ containing only tasks of low-criticality.

Scheduled(mtd, τ)**:** METHOD $\times\ \tau \rightarrow$ JS**:**
: The list of jobs generated from a task set τ, which are *successfully* scheduled by method mtd, i.e., jobs which completed within their deadline.

ScheduledHI(mtd, τ)**:** METHOD $\times\ \tau \rightarrow$ JS**:**
: This includes only those jobs from Scheduled(mtd, τ) which are derived from tasks with high-criticality.

ScheduledLO(mtd, τ)**:** METHOD $\times\ \tau \rightarrow$ JS**:**
: This includes only those jobs from Scheduled(mtd, τ) which are derived from tasks with low-criticality.

Failed(mtd, τ)**:** METHOD $\times\ \tau \rightarrow$ JS**:**
: The list of jobs generated from a task set τ, which are *not successfully* scheduled by method mtd, i.e., jobs which were not completed within their deadline.

Abandoned(mtd, τ)**:** METHOD $\times\ \tau \rightarrow$ JS**:**
: This predicate returns the list of jobs generated from a task set τ that were never forwarded

by the mixed-criticality scheduling method *mtd* to its underlying real-time scheduler. This is a special case of failed jobs:

$$Abandoned(mtd, \tau) \subseteq Failed(mtd, \tau)$$

Abandoned jobs are also different from *dropped jobs*, which are jobs that failed after having started execution with the underlying real-time scheduler.

`LORated(`mtd, τ`):` METHOD $\times\ \tau \rightarrow$ JS**:**

This predicate returns the list of LO jobs, which were re-queued from the default high-priority queue to the low-priority queue (LBP-based methods only).

`IsBetterMCS(`mtd_1, mtd_2, τ`):` METHOD$^2 \times\ \tau \rightarrow$ BOOL:

This predicate tests whether a scheduling method mtd_1 is better than method mtd_2 for a task set τ with respect to mixed-criticality scheduling, which is formally defined as:

$$IsBetterMCS(mtd_1, mtd_2, \tau) \Rightarrow$$
$$= \begin{cases} \text{TRUE} & \text{if } (ScheduledHI(mtd_1, \tau) \supset ScheduledHI(mtd_2, \tau)) \vee \\ & ((ScheduledHI(mtd_1, \tau) == ScheduledHI(mtd_2, \tau)) \wedge \\ & (ScheduledLO(mtd_1, \tau) \supset ScheduledLO(mtd_2, \tau))) \\ \text{FALSE} & \text{otherwise} \end{cases}$$

This tests whether mtd_1 has a better performance than mtd_2 for HI jobs, or equal performance for HI jobs but better performance for LO jobs.

`IsBetterMCS(`mtd_1, mtd_2`):` METHOD$^2 \rightarrow$ BOOL:

This predicate tests whether a scheduling method mtd_1 is better than method mtd_2 for all task sets with respect to mixed-criticality scheduling, which is formally defined as:

$$\begin{aligned} &IsBetterMCS(mtd_1, mtd_2) \Rightarrow \\ &\exists ts \in \text{STS}. \quad IsBetterMCS(mtd_1, mtd_2, \tau) \wedge \\ &\nexists \tau \in \text{STS}. \quad IsBetterMCS(mtd_2, mtd_1, \tau) \end{aligned}$$

It is worth noting that $IsBetterMCS(mtd_1, mtd_2, \tau)$ and $IsBetterMCS(mtd_1, mtd_2)$ are transitive:

$$IsBetterMCS(m_A, m_B) \ \wedge \ IsBetterMCS(m_B, m_C) \Rightarrow IsBetterMCS(m_A, m_C)$$

5.1. Comparison between BP and LBP

Theorem 1. *LBP has the same success rate of HI tasks than BP, which can be formally written as:*

$$\forall \tau \in \text{STS}.\ ScheduledHI(BP, \tau) == ScheduledHI(LBP, \tau)$$

Proof. (Theorem 1) BP and LBP behave the same way regarding the handling of HI jobs:

1. If a HI job is overrunning its C_{LO}, it is granted an execution budget until C_{HI}.
2. If a HI job does not finish within C_{HI} or within its deadline, then it is dropped.

The only difference between BP and LBP lies in the handling of LO jobs, where LBP puts them in a lower priority scheduling queue instead of abandoning them immediately when released in Bailout/Recovery modes or dropping them after the overrun of their C_{LO} as BP does. The content of the low-priority scheduling queue of LBP cannot influence the scheduling of the default scheduling queue. Thus, for any task set τ, it follows that $ScheduledHI(BP, \tau) == ScheduledHI(LBP, \tau)$. □

Theorem 2. *LBP can have a better success rate of LO tasks than BP, but never worse, which can be formally written as:*

$$\forall\, \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subseteq \mathit{ScheduledLO}(LBP, \tau)$$
$$\exists\, \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subset \mathit{ScheduledLO}(LBP, \tau)$$

Proof. (Theorem 2) The only difference between BP and LBP lies in the handling of LO jobs, where LBP puts them in a low-priority scheduling queue instead of abandoning them immediately or dropping them after the overrun of their C_{LO}. Hence, we have:

$$\forall\, \tau \in \text{STS}.\ \mathit{Abandoned}(BP, \tau) \subseteq \mathit{LORated}(LBP, \tau)$$

Thus, to prove Theorem 2, we only have to show that, among the tasks that BP abandons, there is at least one task that with LBP instead gets put into the low-priority queue and finally successfully scheduled:

$$\exists\, \tau \in \text{STS}.\ \mathit{Abandoned}(BP, \tau) \cap \mathit{LORated}(LBP, \tau) \cap \mathit{Scheduled}(LBP, \tau) \neq \emptyset$$

which means it is sufficient for the proof to show by example that it is possible to have task sets where LO-rated jobs can be scheduled within an idle time of the default scheduling queue. To do so, we use the following task set consisting of one HI task A and one LO task B:

Task	P	D	et	C_{LO}	C_{HI}	L
A	15	15	5	3	10	HI
B	4	4	2	2	-	LO

Task A is assumed to have an execution time $et = 5$, which always causes an overrun of the optimistic WCET estimate. The first time, job A_0 exceeds its C_{LO} at $t = 7$ and the system switches into Bailout mode. The LO job B_2 is released at time $t = 8.0$ during Bailout mode. Hence, the bailout fund BF is decreased by a quantity equal to the C_{LO} of B_2. However, BF remains positive. After A_0 is completed, the system experiences an idle instant and this causes a switch back to Normal mode.

As shown in Figure 4a, BP immediately abandons job B_2 at its arrival time during the high-criticality execution mode. In contrast, as shown in Figure 4b, LBP moves such job into the low-priority queue at its arrival and executes it when the default queue becomes idle. Thus, this example demonstrates the existence of a task set τ such that

$$\exists\, \tau \in \text{STS}.\ \mathit{Abandoned}(BP, \tau) \cap \mathit{LORated}(LBP, \tau) \cap \mathit{Scheduled}(LBP, \tau) \neq \emptyset$$

which demonstrates that there are cases where LBP can successfully schedule more jobs than BP. Here, we have to remind the fact that those jobs which are successfully scheduled by BP are processed exactly the same way by BP and LBP, meaning that, whenever BP successfully schedules a job, so does LBP. This property together with the existence of above example completes the proof of:

$$\forall\, \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subseteq \mathit{ScheduledLO}(LBP, \tau)$$
$$\exists\, \tau \in \text{STS}.\ \mathit{ScheduledLO}(BP, \tau) \subset \mathit{ScheduledLO}(LBP, \tau)$$

□

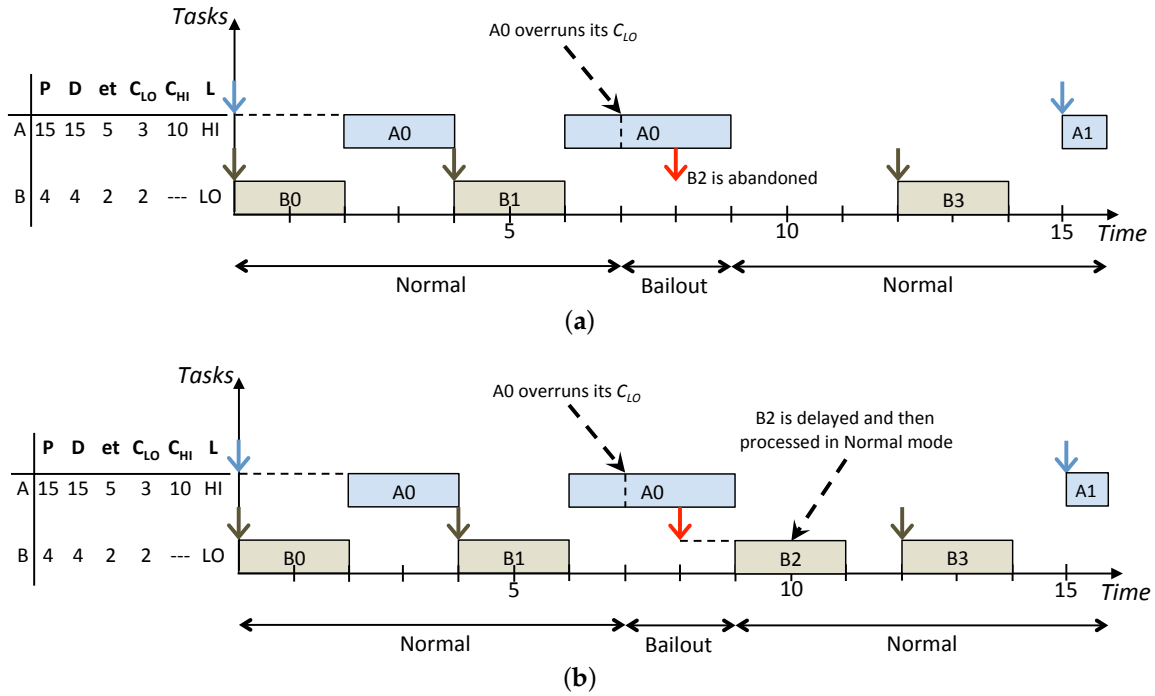


Figure 4. (Proof of Theorem 2) Example in which LBP successfully executes LO jobs that are abandoned by BP: (**a**) BP abandons LO jobs that are not released in Normal mode; and (**b**) LBP provides a delayed execution for job B_2.

From Theorems 1 and 2, it follows that:

Corollary 1. *LBP has a better mixed-criticality performance than BP, which can be formally written as:*

$$IsBetterMCS(LBP, BP)$$

5.2. Comparison between BPG and LBPG

Theorem 3. *LBPG has the same success rate of HI tasks than BPG, which can be formally written as:*

$$\forall \tau \in \text{STS}.\ ScheduledHI(BPG, \tau) == ScheduledHI(LBPG, \tau)$$

Proof. (Theorem 3) BPG and LBPG behave the same way regarding the handling of HI jobs:

1. If a HI job is overrunning its C_{LO}, it is granted an execution budget until C_{HI}.
2. If a HI job does not finish within C_{HI} or within its deadline, then it is dropped.

Moreover, any job that completes before its optimistic WCET during Normal mode gives its gain time to the next highest priority job in the ready queue. The only difference between BPG and LBPG lies in the handling of LO jobs that exceed their optimistic WCETs or that are released during Bailout and Recovery modes. BPG abandons such jobs while LBPG inserts them in the low-priority queue for later execution. Furthermore, the gain time collection only happens among jobs in the high-priority queue and no gain time is passed or happens among jobs in the low-priority queue. This guarantees that BPG and LBPG process and schedule jobs in the high-priority queue the same way. Thus, for any task set τ, it follows that

$$ScheduledHI(BPG, \tau) == ScheduledHI(LBPG, \tau).$$

□

Theorem 4. *LBPG can have a better success rate of LO tasks than BPG, but never worse, which can be formally written as:*

$$\forall\, \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPG}, \tau) \subseteq \mathit{ScheduledLO}(\mathit{LBPG}, \tau)$$
$$\exists\, \tau \in \text{STS}.\ \mathit{ScheduledLO}(\mathit{BPG}, \tau) \subset \mathit{ScheduledLO}(\mathit{LBPG}, \tau)$$

Proof. (Theorem 4) The only difference between BPG and LBPG lies in the handling of LO jobs, where LBPG puts them in a low-priority scheduling queue instead of abandoning them immediately or dropping them after the overrun of their C_{LO}. Hence, we have:

$$\forall\, \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPG}, \tau) \subseteq \mathit{LORated}(\mathit{LBPG}, \tau)$$

to prove Theorem 4 we only have to show that among the tasks that BPG abandons, there is at least one task that with LBPG instead gets put into the low-priority queue and finally successfully scheduled:

$$\exists\, \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPG}, \tau) \cap \mathit{LORated}(\mathit{LBPG}, \tau) \cap \mathit{Scheduled}(\mathit{LBPG}, \tau) \neq \emptyset$$

which means it is sufficient for the proof to show by example that it is possible to have task sets where LO-rated jobs can be scheduled within an idle time of the default scheduling queue.

We use the task set in Figure 5 to show that LBPG outperforms the standard BPG. The HI task A is assumed to have an execution time $et = 9$, which always causes an overrun of the optimistic WCET estimate. On the other hand, the LO task B always has an execution time of $et = 3$ apart from its first instance that runs for only two time units which allows to have a gain time of 1. Job B_0 completes earlier at time $t = 2$ and gives its gain time to job A_0 for which the optimistic time budget is now updated to A5. A_0 enters the Bailout mode at time $t = 7$ and then runs until its completion. No other gain time is collected during the schedule showed in the figure. Figure 5a,b show, respectively, that BPG abandons job B_1 and B_3, while LBPG runs them in Normal mode during idle time. Thus, this example demonstrates the existence of a task set τ such that

$$\exists\, \tau \in \text{STS}.\ \mathit{Abandoned}(\mathit{BPG}, \tau) \cap \mathit{LORated}(\mathit{LBPG}, \tau) \cap \mathit{Scheduled}(\mathit{LBPG}, \tau) \neq \emptyset$$

which completes the proof. □

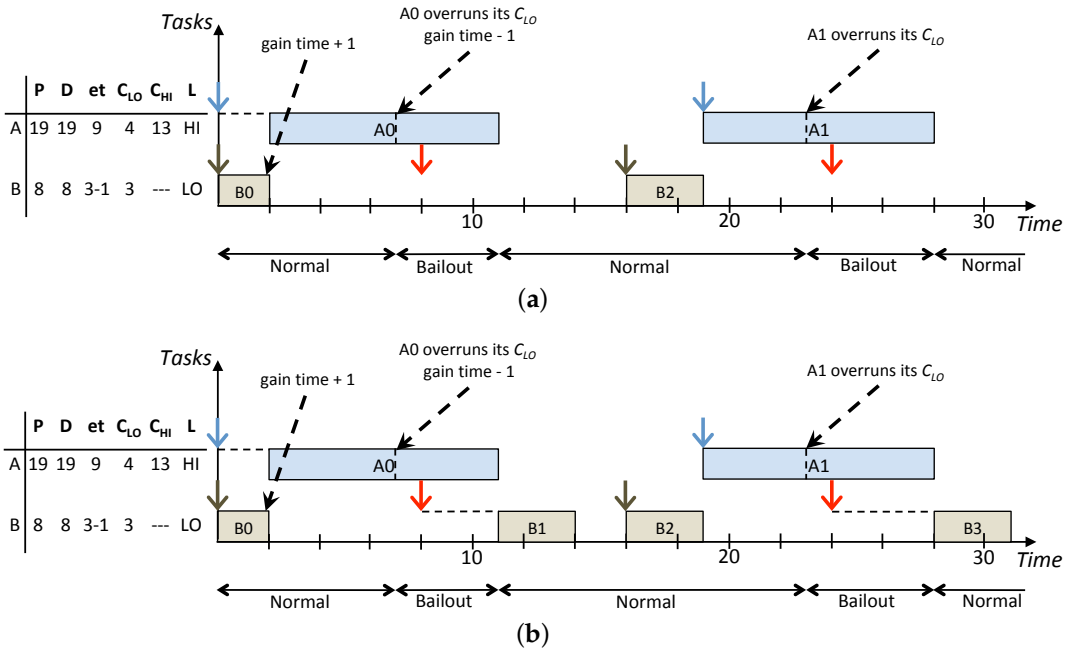


Figure 5. (Proof of Theorem 4) Example in which LBPG successfully executes LO jobs that are abandoned by BPG: (**a**) BPG abandons LO jobs in Bailout mode; and (**b**) LBPG provides a delayed execution for job B_1 and B_3

From Theorems 3 and 4 it follows that:

Corollary 2. *LBPG has a better mixed-criticality performance than BPG, which can be formally written as:*

$$IsBetterMCS(LBPG, BPG)$$

5.3. Comparison between BPS and LBPS

Theorem 5. *LBPS has better mixed-criticality performance than BPS, which can be formally written as:*

$$IsBetterMCS(LBPS, BPS)$$

Proof. (Theorem 5) The proof has two parts:

1. Proving that there is no task set τ such that

$$IsBetterMCS(BPS, LBPS, \tau)$$

2. Showing by concrete example that there exists a task set τ such that

$$IsBetterMCS(LBPS, BPS, \tau)$$

Part 1:

The strength of BPS over BP consists of the scaling up of the optimistic WCETs of HI tasks to increase the duration of Normal mode and to decrease the amount of time the system runs in Bailout mode. On the one hand, this leads to abandon a smaller amount of LO jobs due to the decrease in high-criticality mode duration. On the other hand, the increase of the Normal mode duration allows releasing and processing more LO jobs. The only difference between BPS and LBPS is in the handling of LO jobs released exceeding their C_{LO} or released in high-criticality modes, i.e., BPS suddenly abandons them while LBPS inserts them in a low-priority queue for later execution during system idle instants.

Therefore, LBPS keeps the BPS advantage, but it adds also the Lazy Bailout approach, which allows recovering LO jobs during high-criticality modes execution. This increases the amount of LO jobs processed and eventually scheduled during the Normal mode execution. This means that LBPS can never have worse performances than BPS.

Part 2:

To prove the second part, it is necessary to show that there exists one task set in which LBPS schedules more LO jobs than BPS. As an example, we use the task set given in Figure 6, for which the optimistic WCET C_{LO} of the HI task A has been already scaled up by sensitivity analysis. BPS increases the duration in which the system runs in Normal mode. However, it still abandons LO jobs released during high-criticality execution modes. Conversely, LBPS runs them afterwards during idle instants. Figure 6 displays how the LO job B_1 released at time $t = 9$ is abandoned with BPS while LBPS manages to execute it later at time $t = 10$.

This concludes the proof of Theorem 5. □

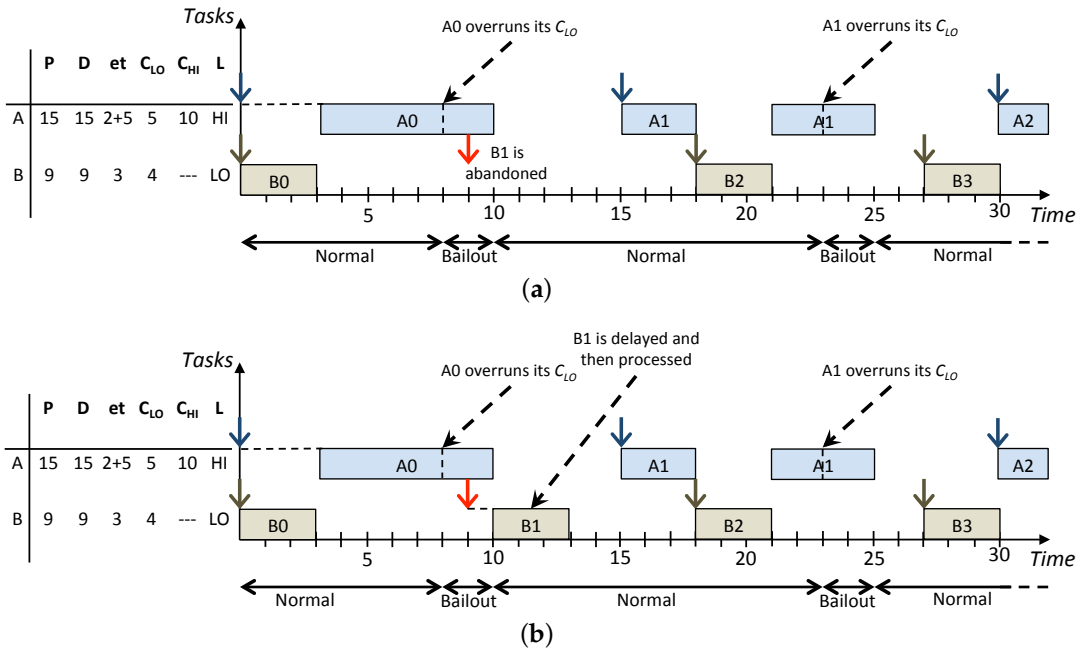


Figure 6. (Proof of Theorem 5, Part 2) LBPS always schedules more LO jobs than BPS: (**a**) BPS abandons job B_1; and (**b**) LBPS schedules all LO jobs.

5.4. Comparison between BPSG and LBPSG

With Corollary 2 and Theorem 5, we proved,, respectively that the LBPG always outperforms LBP and LBPS always outperforms BPS. This is because the gain time collection made at runtime and the offline scaling up of the C_{LO} have the same benefits in the lazy Bailout method as in the standard Bailout protocol. On the other hand, from Corollary 1, we know that LBP always outperforms BP. It follows that:

Corollary 3. *LBPSG has a better mixed-criticality performance than BPSG, which can be formally written as:*

$$IsBetterMCS(LBPSG, BPSG)$$

In this section, we introduce the criterion `IsBetterMCS`(mtd_1, mtd_2) to compare the performance of mixed-criticality scheduling methods with priority given to HI jobs scheduled. Using this criterion, we show that LBP always performs better than BP. Moreover, we also show that the offline sensitivity

analysis and online gain time collection always contribute to increase the amount of LO jobs scheduled and that they achieve better performances if used with LBP rather than BP. To conclude, the proposed LBPSG is consistently better than the existing BPSG [11].

Note that this result is strictly bound to our definition of `IsBetterMCS`(mtd_1, mtd_2), which is motivated by systems where a sacrifice of HI jobs to increase performance of LO jobs is not acceptable.

6. Experimental Evaluation

In this section, we describe how we conducted our experiments and what were the final outcomes. We start by explaining how the experiments were structured. Section 6.2 explains how task sets were created, what were our application scenarios and what scheduling methods we compared. Next, Section 6.3 describes what type of performance metrics we considered to evaluate and compare different mixed-criticality scheduling methods. Finally, Section 6.4 contains the description and discussion of results according to what is shown within tables and charts.

6.1. Task Set Generation

The aim of the task set generation was to simulate situations in which the system easily switches to the Bailout execution mode in order to show the effectiveness of the LBP methods over BP and its derivatives in systems with high load (using a utilisation factor of 0.6 or more). The system load computed according to the pessimistic WCET of all HI tasks within each task set was always greater than that computed according to the optimistic WCET of all its tasks. Furthermore, the execution times of HI tasks almost always exceeded their C_{LO} in order to trigger the mode change.

As a result, task sets were generated to have an overall utilisation factor that varied randomly between 0.60% and 0.75% with respect to the optimistic WCET C_{LO} of all tasks while the utilisation factor computed according to the HI tasks was always 0.75%. The number of tasks per task set varied randomly between 4 and 20. Within each task set, the amount of HI tasks varied randomly between 20% and 70% of the task set. Moreover, the execution times of HI tasks varied between the 90% of their C_{LO} and their C_{HI}, while the execution times of LO tasks was between 40% of their C_{LO} and 10% more than the C_{LO}. Priorities were assigned to tasks according to the *Deadline Monotonic* (DM) strategy: task instances with shorter deadlines had higher priorities.

6.2. Description of Experiments

We conducted different experiments, each consisting of a group of 3000 task sets randomly generated. Tasks within every task set have implicit deadlines and their periods varied randomly between 3 and 22. Every task set group belonged to one of the three scenarios specified below:

HC-LP contains job sets where all HI jobs have larger deadlines than all LO jobs. More precisely, LO tasks' periods varied randomly between 3 and 10, while HI tasks' periods varied between 14 and 22. Therefore, all HI jobs had lower priority than all LO jobs:

$$\forall j \in J_{HI}.\ \forall j' \in J_{LO}.\ pr(j) < pr(j')$$

HC-MP contains job sets where HI jobs have deadlines that are smaller or larger than those of LO ones. In fact, periods of tasks, either HI or LO, varied randomly between 3 and 22. Therefore, HI and LO jobs had mixed priorities:

$$\forall j \in J_{HI}.\ \forall j' \in J_{LO}.\ pr(j) \leq pr(j')\ \vee\ pr(j) > pr(j')$$

HC-HP contains job sets where all HI jobs have smaller deadlines than all LO ones. More precisely, HI tasks' periods varied randomly between 3 and 10 while LO tasks' periods varied between 14 and 22. This implies that all HI jobs have higher priority than LO jobs:

$$\forall j \in J_{HI}.\ \forall j' \in J_{LO}.\ pr(j) > pr(j')$$

We compared the following scheduling protocols:

- The standard *Fixed-Priority Preemptive Scheduling* with DM as priority assignment (FPPS-DM).
- The standard *Bailout Protocol* (BP).
- The *Bailout Protocol with Gain Time* (BPG), where each job that finishes before its optimistic time threshold in Normal mode gives its gain time to increase the time budget of next job ready to be scheduled.
- The *Bailout Protocol-Slack* (BPS) and the *Bailout Protocol-Slack and Gain Time* (BPSG) that represent the execution of BP and BPG on task sets in which the C_{LO} of HI tasks is appropriately increased via sensitivity analysis [30,31] while the schedulability is guaranteed according to AMC-rtb [15].
- The *Lazy Bailout Protocol* (LBP).
- The *Lazy Bailout Protocol with Gain Time* (LBPG), the *Lazy Bailout Protocol-Slack* (LBPS) and the *Lazy Bailout Protocol-Slack and Gain Time* (LBPSG) that represent extensions of LBP made by using the offline scaling of C_{LO} of HI tasks with sensitivity analysis and the gain time collection at runtime.

We finally show the benefit of the lazy bailout approaches with respect to the former methods.

It is important to note that, if HI tasks all have higher priority than LO ones, then the scheduling problem so created becomes equivalent to the standard real-time scheduling problem since there is no criticality inversion. The same applies to those cases in which higher priority is assigned to the highest criticality tasks regardless of their periods or deadline as in *Criticality As Priority Assignment* (CAPA) [32].

Results of experiments are collected in Tables 1 and 2, which refer to the different scenarios described above. For each scenario, we show the results with different scheduling protocols.

Table 1. BP and LBP variants: comparison of task set schedulability (%).

	HC-LP			***HC-MP***			***HC-HP***		
Method	*TSSched*	*TSSchedHI*	*TSSchedLO*	*TSSched*	*TSSchedHI*	*TSSchedLO*	*TSSched*	*TSSchedHI*	*TSSchedLO*
FPPS-DM	83.03	83.03	100.0	76.87	98.33	77.27	78.67	100.0	78.67
BP	2.20	100.0	2.20	0.97	100.0	0.97	0.87	100.0	0.87
BPG	4.87	100.0	4.87	1.17	100.0	1.17	0.93	100.0	0.93
BPS	7.23	100.0	7.23	11.17	100.0	11.17	12.30	100.0	12.30
BPSG	11.87	100.0	11.87	17.23	100.0	17.23	20.00	100.0	20.00
LBP	13.93	100.0	13.93	22.53	100.0	22.53	46.43	100.0	46.43
LBPG	21.17	100.0	21.17	23.57	100.0	23.57	46.63	100.0	46.63
LBPS	20.73	100.0	20.73	30.77	100.0	30.77	52.97	100.0	52.97
LBPSG	29.57	100.0	29.57	37.60	100.0	37.60	58.57	100.0	58.57

Table 2. BP and LBP variants: comparison of jobs scheduled within their deadline (%).

	HC-LP			***HC-MP***			***HC-HP***		
Method	*GJSched*	*GJSchedHI*	*GJSchedLO*	*GJSched*	*GJSchedHI*	*GJSchedLO*	*GJSched*	*GJSchedHI*	*GJSchedLO*
FPPS-DM	99.19	88.64	100.0	98.51	99.55	97.91	99.11	100.0	98.18
BP	62.81	100.0	55.99	73.63	100.0	54.78	85.41	100.0	60.20
BPG	67.22	100.0	61.12	74.38	100.0	55.89	85.63	100.0	60.73
BPS	66.21	100.0	60.38	79.06	100.0	64.68	88.44	100.0	69.96
BPSG	71.03	100.0	66.07	81.64	100.0	69.41	90.05	100.0	75.13
LBP	83.64	100.0	80.94	92.71	100.0	88.71	97.87	100.0	95.16
LBPG	85.87	100.0	83.54	92.91	100.0	88.99	97.88	100.0	95.18
LBPS	85.23	100.0	82.92	93.24	100.0	89.54	97.99	100.0	95.51
LBPSG	87.57	100.0	85.66	93.77	100.0	90.48	98.08	100.0	95.78

6.3. Description of Performance Metrics

This subsection introduces the criteria we used to assess performances of scheduling protocols that process dual-criticality task sets. To evaluate the results, we defined two types of performance parameters, i.e., *task set schedulability* that is relative to the whole amount of task sets and the *global job set schedulability* that is relative to jobs within each individual task set.

We conducted our experiments on three sets of 3000 task sets *STS*, one per scenario (HC-LP, HC-MP and HC-HP). The task set schedulability formula *tsched* is defined as follows:

$$tsched(S, cat) = \frac{|STSsucc(S, cat)|}{|S|} \tag{4}$$

where S could be either a simple task set τ or set of task sets *STS* and the category $cat \in \{HI + LO, HI, LO\}$ represents the type of tasks within a set that is HI to indicate HI tasks, LO to indicate LO tasks and either when we use HI+LO. The function *STSsucc* depends on the scheduling protocol that is actually used and returns as output the set of task sets *STS* in which there are no jobs missed of category *cat*. The absolute values within the formula give the set cardinality. Equation (4) allows deriving the percentages of tasks set in *STS* that are successfully processed according to the category *cat* as follows:

TSSched is the amount of task sets scheduled with no jobs missing their deadlines.

$$TSSched = tsched(STS, HI + LO)$$

TSSchedHI is the amount of task sets scheduled with no HI jobs missing their deadlines.

$$TSSchedHI = tsched(STS, HI)$$

TSSchedLO is the amount of task sets scheduled with no LO jobs missing their deadlines.

$$TSSchedLO = tsched(STS, LO)$$

The task set schedulability permits showing the percentage of task sets in which no job of category *cat* misses its deadline. However, whenever a task set contains some jobs, either HI or LO, that miss their deadline, it is also useful to assess the level of service provided in terms of jobs completed within their deadlines and jobs abandoned or aborted. The job set completion rate method *jsched* returns the percentage of jobs of category *cat* generated by a specific task set that complete within their deadlines.

The job set schedulability *jsched* is formally written as below:

$$jsched(\tau_{cat}) = \frac{|Jsucc(J(\tau_{cat}))|}{|J(\tau_{cat})|} \quad | \; cat \in \{LO, HI\} \tag{5}$$

The formula of the global job set schedulability $gjsched(STS, cat)$ returns the average amount of jobs of category *cat* processed within their deadline that has been generated by a set of task sets *STS*:

$$gjsched(STS, cat) = \frac{\sum_{\tau \in STS} jsched(\tau_{cat})}{|STS|} \quad | \; cat \in \{LO, HI\} \tag{6}$$

As in the previous case, it is possible to filter the jobs meeting their deadline according to the category *cat* as below:

GJSched is the average number of jobs (either HI or LO) that is scheduled within a task set.

$$GJSsched = gjsched(STS, HI + LO)$$

GJSchedHI is the average number of HI jobs that is scheduled within a tasks set.

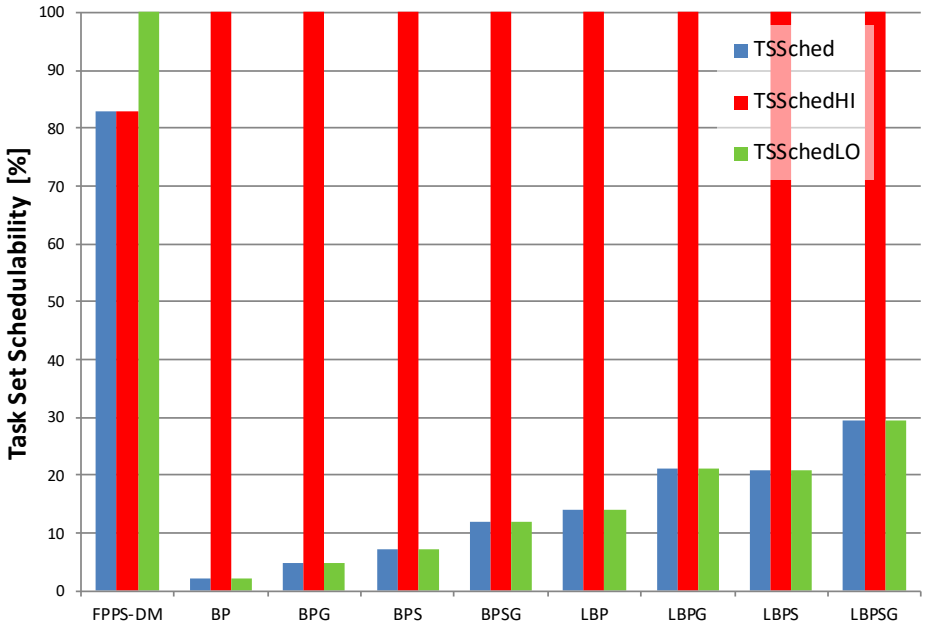
$$GJSchedHI = gjsched(STS, HI)$$

GJSchedLO is the average number of LO jobs that is scheduled within a task set.

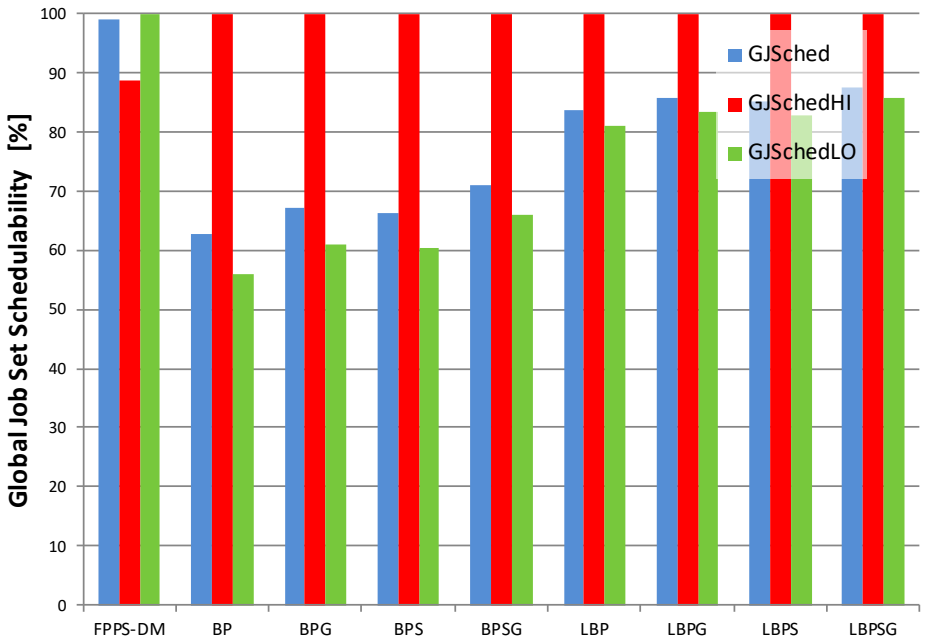
$$GJSschedLO = gjsched(STS, LO)$$

Tables 1 and 2 contain, respectively, the results about task set and global job set schedulabilities. It is possible to comment on the data according to scenario or scheduling protocol. However, we use figures to describe graphically what is contained within the tables and to allow an easier and quicker comparison among the results.

Task set and global job set schedulabilities are averages and do not give information about how data are distributed and about outliers. Therefore, we use also boxplots charts to show the distribution of LO jobs scheduled per task set. The information shown in Table 1 is contained in Figures 7a, 8a and 9a. On the other hand, Figures 7b, 8b and 9b displays the average percentages of LO jobs completed within their deadlines.



(**a**)



(**b**)

Figure 7. BP and LBP variants: schedulability in HC-LP scenario: (**a**) task sets with no jobs missed; and (**b**) average of jobs scheduled per task set.

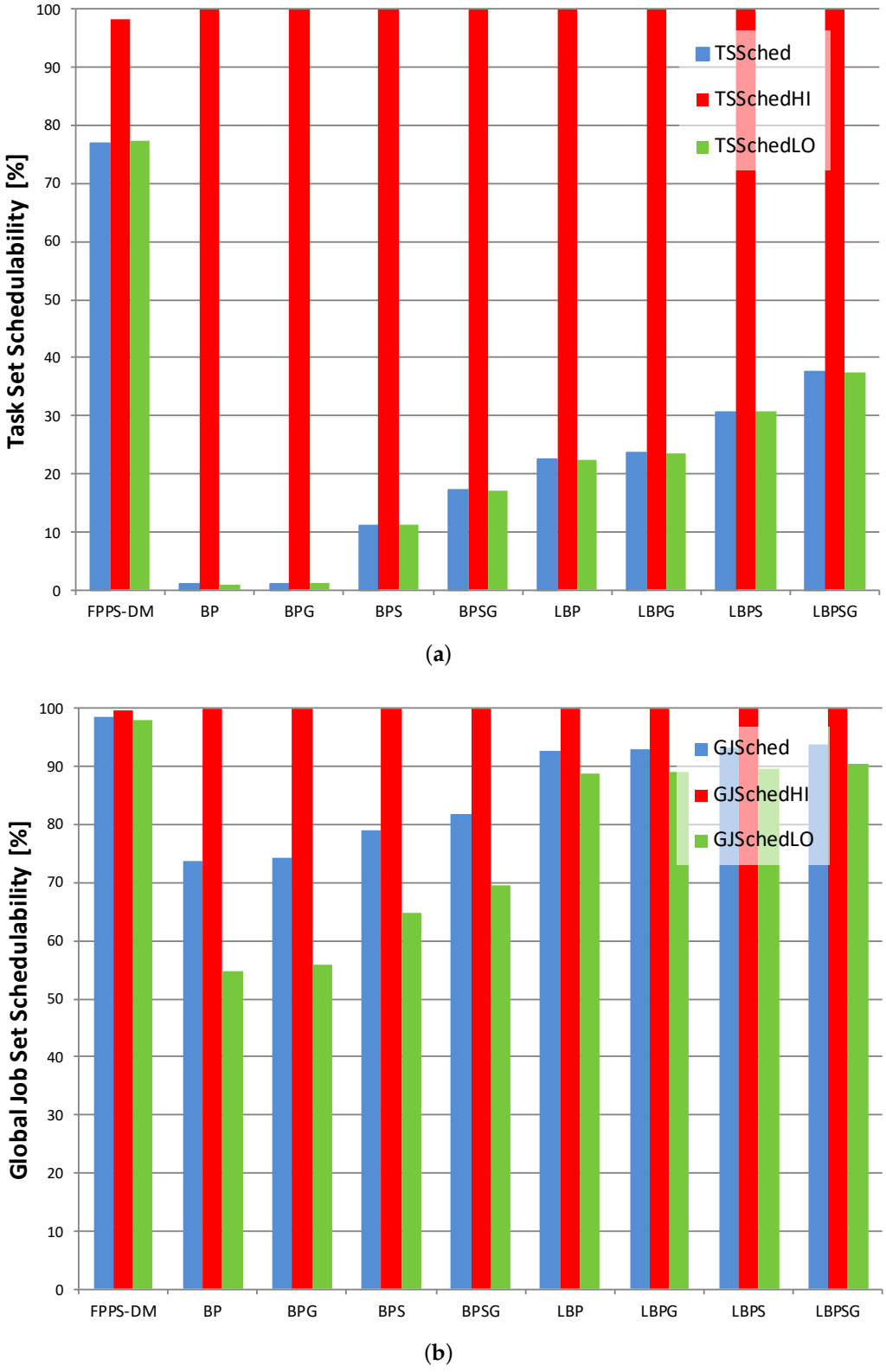


Figure 8. BP and LBP variants: schedulability in HC-MP scenario: (**a**) task sets with no jobs missed; and (**b**) average of jobs scheduled per task set

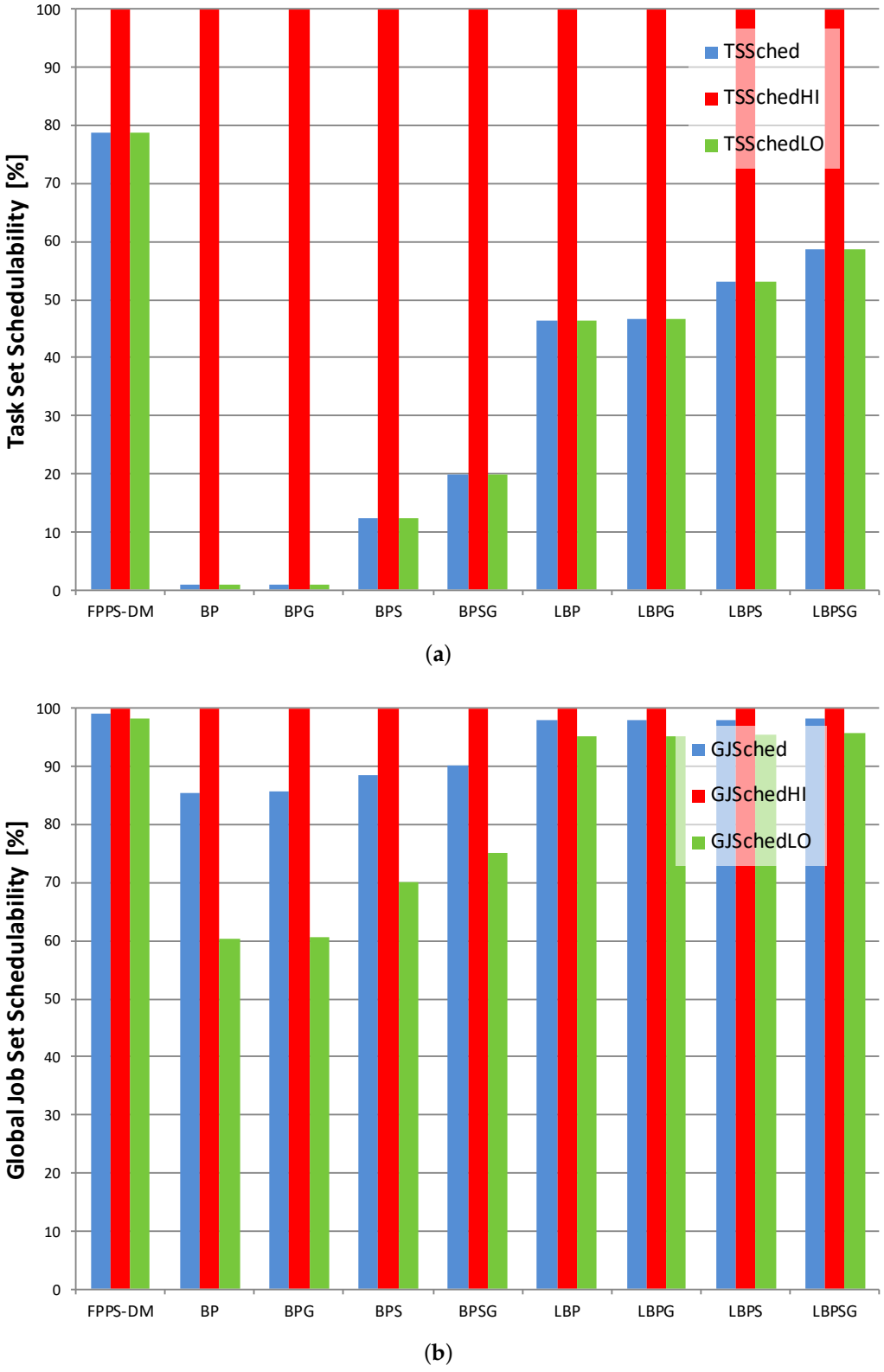


Figure 9. BP and LBP variants: schedulability in HC-HP scenario (as priority and criticality values have the same order, this is essentially a standard real-time scheduling problem): (**a**) task sets with no jobs missed; and (**b**) average of jobs scheduled per task set.

6.4. Discussion of Results

This subsection is dedicated to study the outcome of the experiments. It contains figures that summarise the results of our experiments with dual-criticality job sets. Figures 7 and 8 summarise the results in cases where there is criticality inversion. In these situations, if no HI job completes within its optimistic threshold estimate C_{LO}, then very likely there will be some new incoming higher priority LO jobs that will interfere with it. Then, Figure 9 contains information about cases in which all HI jobs have higher priority than LO jobs since all the critical jobs have smaller deadlines. This basically leads to having no interference between HI and LO jobs and thus no criticality inversion occurrence during the scheduling process.

Looking both at task set and job set schedulabilities results in Figures 7–9, it is possible to notice that, compared with mixed-criticality methods, the standard deadline monotonic approach always schedules jobs only according to priorities. In this case, the percentages of HI and LO jobs successfully scheduled mainly depend only on their priority, with all LO jobs always meeting their deadlines in HC-LP scenario and all HI jobs always meeting their deadlines in HC-HP scenario. Conversely, the mixed-criticality protocols always assure that there are no HI job missed regardless of job priorities. The experiments confirm what is stated in Section 5 with LBP in which the percentage of task set scheduled with no jobs missed is between 13% and 46% while BP schedules no more than 2.20% of task sets with no jobs missed.

Then, where the offline and online complementary techniques are used, there is an increase in LO jobs successfully processed. Furthermore, the usage of sensitivity analysis and the gain time mechanism have the same effects when applied both to the standard or to the lazy bailout method. A noticeable result is that each LBP-based approach always increases the amount of LO jobs completed within their deadlines compared with the corresponding standard BP-based protocol. Overall, according to the criteria defined in Section 5, LBPSG is the protocol that outperforms all other mixed-criticality scheduling methods with an amount of task set scheduled with no jobs missed that is between 29.57% and 58.57%.

Figures 10–12 display the distribution of the LO jobs percentages per task set that are completed within their deadlines. Each scheduling protocol is represented by a box-and-whisker diagram with the box itself representing the range in which at least 50% of results tend to be concentrated. The box also contains the indication of the median and the mathematical average of all the LO jobs scheduled by the related protocol. The results highlight how the LBP-based methods always increase the LO jobs success rate, as defined in Section 5, compared with the former BP ones.

In conclusion, the experiments confirm what is stated in Section 5 with lazy approaches increasing the amount of LO jobs successfully scheduled while guaranteeing the correct completion of all HI jobs. In other words, LBP has better mixed-criticality performances than BP, while LBPS, LBPG and LBPSG have, respectively, better mixed-criticality performances than BPS, BPG and BPSG. Finally, the usage of mixed-criticality protocols is recommended in HP-LP and HC-MP scenarios, i.e., when HI jobs could have lower priorities than LO jobs.

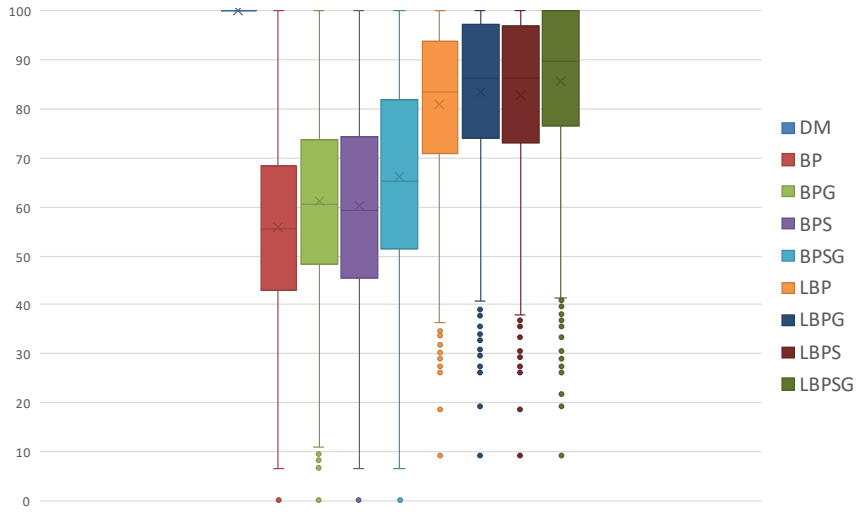


Figure 10. BP and LBP variants: LO jobs scheduled per task set in HC-LP scenario.

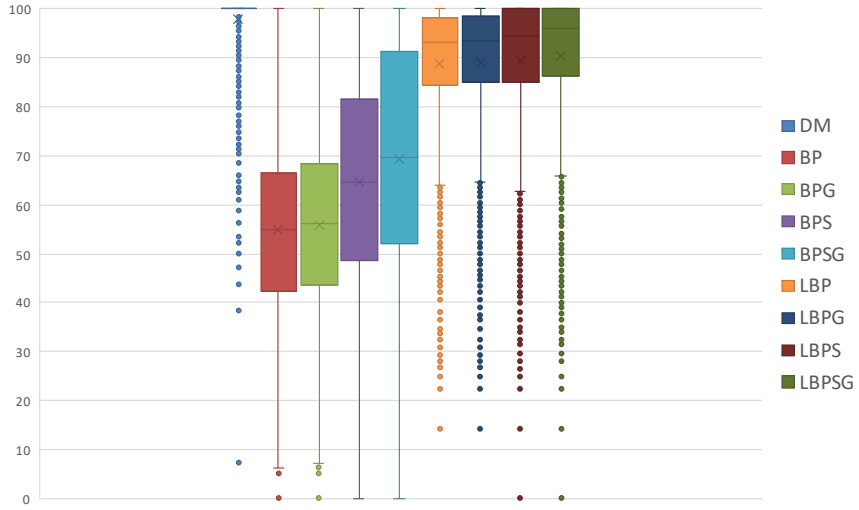


Figure 11. BP and LBP variants: LO jobs scheduled per task set in HC-MP scenario.

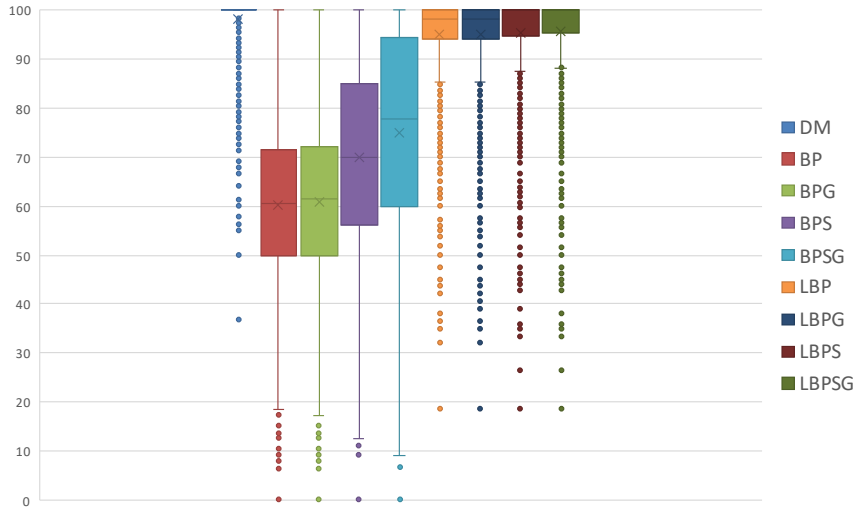


Figure 12. BP and LBP variants: LO jobs scheduled per task set in HC-HP scenario.

7. Summary and Conclusions

Mixed-criticality scheduling is important for cyber-physical systems to provide robustness against resource shortage. In this paper, we have introduced the mixed-criticality scheduling protocol *Lazy Bailout Protocol (LBP)*. LBP is a scheduling protocol for uni-processor platforms and is a refinement of Bailout Protocol (BP). We have also introduced a formal criterion to compare performances among mixed-criticality scheduling protocols. This criterion prioritises HI jobs against LO jobs, where HI indicates high-criticality and LO stands for low-criticality. Based on that criterion, we have proven that the complementary techniques used in [11] always contribute to increase the performances of the scheduling protocol. Similar to BP, LBP and its derivatives always guarantee the correct completion of HI jobs. Moreover, we have shown that LBP always schedules more LO jobs than BP and that each LBP derivative always outperforms the corresponding BP-based approach.

Besides these formal results, we have also presented experiments that give quantitative values of the comparisons between the different mixed-criticality scheduling protocols. LBP schedules between 13.93% and 46.63% of task sets with no jobs missed while BP at maximum schedules no more than 2.20% of task sets with no jobs missed. Finally, the experiments confirm that LBP is equivalent to BP in guaranteeing HI jobs and that the derivatives of LBP (LBPS, LBPG and LBPSG) outperform all the equivalent BP-based protocols by increasing the amount of LO jobs successfully scheduled. Overall, LBPSG has shown the best mixed-criticality performance with an amount of task sets processed with no jobs missed that is between 29.57% and 58.57%.

Future work will be on extending LBP towards support for many-core platforms.

Author Contributions: R.K. has contributed the majority of formal proofs, S.I. has done the implementation and experiments and refined some proofs.

Funding: The research leading to these results in its early stages has received funding from the FP7 ARTEMIS-JU research project *"ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems"* (CRAFTERS) under contract no 295371.

Acknowledgments: The authors would like to thank Olga Tveretina for general comments on how to improve proofs.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, C.L.; Layland, J.W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* **1973**, *20*, 46–61. doi:10.1145/321738.321743.
2. Vestal, S. Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance. In Proceedings of the 28th IEEE International Real-Time Systems Symposium (RTSS'07), Tucson, AZ, USA, 3–6 December 2007; pp. 239–243. doi:10.1109/RTSS.2007.47.
3. Burns, A.; Davis, R.I. *Mixed Criticality Systems—A Review*; Research Report V4-31/7/2014; Department of Computer Science, University of York: York, UK, 2014.
4. Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D.; Bernat, G.; Ferdinand, C.; Heckman, R.; Mitra, T.; et al. The Worst-Case Execution Time Problem—Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst. (TECS)* **2008**, *7*. doi:10.1145/1347375.1347389
5. Kirner, R.; Iacovelli, S.; Zolda, M. Optimised Adaptation of Mixed-criticality Systems with Periodic Tasks on Uniform Multiprocessors in Case of Faults. In Proceedings of the 11th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'15), Auckland, New Zealand, 13 April 2015.
6. RTCA SC-205. Software Considerations in Airborne Systems and Equipment Certification. RTCA/DO-178C. Available online: https://www.rtca.org (accessed on 20 December 2011).
7. International Organization for Standardization (ISO). *Road Vehicles—Functional Safety*; ISO Standard 26262; ISO: Geneva, Switzerland, 2011.

8. International Electrotechnical Commission. *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*; IEC Standard 61508; International Electrotechnical Commission: Geneva, Switzerland, 1998.
9. Esper, A.; Nelissen, G.; Nélis, V.; Tovar, E. How Realistic is the Mixed-criticality Real-time System Model? In Proceedings of the 23rd Int'l Conference on Real Time and Networks Systems (RTNS), Lille, France, 4–6 November 2015; ACM: New York, NY, USA, 2015; pp. 139–148. doi:10.1145/2834848.2834869.
10. Bate, I.; Burns, A.; Davis, R.I. A Bailout Protocol for Mixed Criticality Systems. In Proceedings of the 27th Euromicro Conference on Real-Time Systems, Lund, Sweden, 8–10 July 2015.
11. Bate, I.; Burns, A.; Davis, R.I. An Enhanced Bailout Protocol for Mixed Criticality Embedded Software. *IEEE Trans. Softw. Eng.* **2017**, *43*, 298–320.
12. Crespo, A.; Alonso, A.; Marcos, M.; de la Puente, J.A.; Balbastre, P. Mixed Criticality in Control Systems. In Proceedings of the 19th World Congress of The International Federation of Automatic Control, Cape Town, South Africa, 24–29 August 2014.
13. Ernst, R.; Natale, M.D. Mixed Criticality Systems—A History of Misconceptions. *IEEE Des. Test* **2016**, *33*, 65–74.
14. Burns, A.; Davis, R.I. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.* **2017**, *50*, 82:1–82:37. doi:10.1145/3131347.
15. Baruah, S.K.; Burns, A.; Davis, R.I. Response-Time Analysis for Mixed Criticality Systems. In Proceedings of the 2011 IEEE 32nd Real-Time Systems Symposium (RTSS '11), Vienna, Austria, 29 November–2 December 2011; IEEE Computer Society: Washington, DC, USA, 2011; pp. 34–43. doi:10.1109/RTSS.2011.12.
16. Burns, A.; Davis, R.I. Response Time Analysis for Mixed Criticality Systems with Arbitrary Deadlines. In Proceedings of the 5th International Workshop on Mixed Criticality Systems (WMC 2017), Paris, France, 5 December 2017.
17. Fleming, T.; Burns, A. Extending Mixed Criticality Scheduling. In Proceedings of the 1st International Workshop on Mixed Criticality Systems (WMC), Vancouver, BC, Canada, 3–6 December 2013.
18. Su, H.; Zhu, D. An Elastic Mixed-Criticality task model and its scheduling algorithm. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE), Grenoble, France, 18–22 March 2013; pp. 147–152.
19. Burns, A.; Baruah, S. Towards A More Practical Model for Mixed Criticality Systems. In Proceedings of the 1st International Workshop on Mixed Criticality Systems (WMC), Vancouver, BC, Canada, 3–6 December 2013; pp. 1–6.
20. Baruah, S.; Burns, A.; Davis, R. An Extended Fixed Priority Scheme for Mixed Criticality Systems. In *Workshop on Real-Time Mixed Criticality Systems (ReTiMics)*; George, L.; Lipari, G., Eds.; University of York: York, UK, 2013; pp. 18–24.
21. Audsley, N.C.; Burns, A.; Richardson, M.F.; Wellings, A.J. Hard Real-Time Scheduling: The Deadline-Monotonic Approach. In Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software (RTOSS), 1991; pp. 133–137.
22. Zuhily, A.; Burns, A. Optimal (D-J)-monotonic priority assignment. *Inf. Process. Lett.* **2007**, *103*, 247–250.
23. Fleming, T.; Burns, A. Incorporating the Notion of Importance into Mixed Criticality Systems. In Proceedings of the 2nd International Workshop on Mixed Criticality Systems (WMC), Rome, Italy, 2 December 2014; pp. 33–38.
24. Prasad, D.; Burns, A.; Atkins, M. The Valid Use of Utility in Adaptive Real-Time Systems. *Real-Time Syst.* **2003**, *25*, 277–296. doi:10.1023/A:1025184411567.
25. Audsley, N. *Optimal Priority Assignment and Feasibility of Static Priority Tasks With Arbitrary Start Times*; Technical Report YCS 164; Department of Computer Science, University of York: York, UK, 1991.
26. Gettings, O.; Quinton, S.; Davis, R.I. Mixed criticality systems with weakly-hard constraints. In Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS '15), Lille, France, 4–6 November 2015; ACM: New York, NY, USA, 2015; pp. 237–246.
27. Bernat, G.; Burns, A.; Llamosi, A. Weakly Hard Real-Time Systems. *IEEE Trans. Comput.* **2001**, *50*, 308–321. doi:10.1109/12.919277.
28. Frehse, G.; Hamann, A.; Quinton, S.; Woehrle, M. Formal Analysis of Timing Effects on Closed-Loop Properties of Control Software. In Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium. IEEE, 2014 (RTSS '14), Rome, Italy, 2–5 December 2014; pp. 53–62.

29. Santy, F.; George, L.; Thierry, P.; Goossens, J. Relaxing Mixed-Criticality Scheduling Strictness for Task Sets Scheduled with FP. In Proceedings of the 24th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, 11–13 July 2012; pp. 155–165.
30. Bini, E.; Natale, M.D.; Buttazzo, G.C. Sensitivity analysis for fixed-priority real-time systems. *Real-Time Syst.* **2008**, *39*, 5–30. doi:10.1007/s11241-006-9010-1.
31. Punnekkat, S.; Davis, R.; Burns, A. Sensitivity Analysis of Real-Time Task Sets. In *Advances in Computing Science—ASIAN'97: Third Asian Computing Science Conference Kathmandu, Nepal, December 9–11, 1997 Proceedings*; Springer: Berlin/Heidelberg, Germany, 1997; pp. 72–82. doi:10.1007/3-540-63875-X_44.
32. De Niz, D.; Lakshmanan, K.; Rajkumar, R.R. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In Proceedings of the 2009 30th IEEE Real-Time Systems Symposium (RTSS '09), Washington, DC, USA, 1–4 December 2009; pp. 291–300.