Technical Report

Temporal Isolation with Preemption Delay Accounting

José Marinho
Vincent Nelis
Stefan M. Petters

CISTER-TR-130903
Version:
Date: 9/20/2013
Temporal Isolation with Preemption Delay Accounting

José Marinho, Vincent Nelis, Stefan M. Petters

CISTER Research Unit
Polytechnic Institute of Porto (ISEP-IPP)
Rua Dr. António Bernardino de Almeida, 431
4200-072 Porto
Portugal
Tel.: +351.22.8340509, Fax: +351.22.8340509
E-mail: jmssm@isep.ipp.pt, nelis@isep.ipp.pt, smp@isep.ipp.pt
http://www.cister.isep.ipp.pt

Abstract

Reservation systems are generally employed to enforce temporal isolation between applications. In the real-time context the corresponding temporal isolation requires not only the consideration of the direct interference due to execution of higher priority tasks, but also the indirect cost of e.g. cache-related preemption delay. The accounting of this in a server-based implementation of temporal isolation possesses special challenges, in particular when misbehaving in the form of overruns and violation of the minimum inter-arrival time of an application are to be covered. We present a novel approach to extend the fault coverage and reduce the pessimism when compared to the state of the art. Furthermore we demonstrate that the extra implementation of the introduced mechanisms over the state of the art can be very low on complexity.
1. INTRODUCTION

In today’s technology, the vast majority of the processors that are manufactured are not deployed in desktop or servers, but instead are built in devices and embedded systems. Besides having specific functional requirements many applications (called tasks hereafter) deployed in such systems are subject to stringent timing constraints; in particular their executions have to complete by a deadline associated to each task. Applications exposing such timing requirements are usually called “real-time” applications. A violation of a task deadline might bring about the same complications as a functional failure in the task execution, so correctness of a real-time functionality depends on both the logical results of its computations and the time at which these results are produced. Besides time, many embedded applications also have further non-functional requirements in terms of safety and reliability. Before real-time applications can be deployed in public appliances, timing guarantees must be derived at design time, consequently enforced during run-time and the entire system must be certified. These timing guarantees are obtained through timing and schedulability analysis techniques, which are typically more accurate and simpler when spatial and temporal isolation between tasks is provided. This is because timing analysis techniques must thoroughly examine every shared resource and identify a worst-case interference scenario in which the analyzed task incurs the maximum delay before accessing the shared resource. Without a proper isolation between the tasks: First, the number of interference scenarios to be explored may be out of proportion, hence compelling the analysis techniques to introduce an undesired pessimism into the computation by over-approximating these scenarios. Secondly, having a high number of possible interference scenarios naturally increases the probability of encountering a “pathological” case, where the delay incurred by the analyzed task in that particular scenario is far higher than in the average case. Since the analysis tools must capture the worst-case scenario, this pathological scenario will be retained and used in higher-level analyses (like schedulability analyses) which are built upon the results of timing analyses (thus propagating the pessimism all the way up to system-level analyses). More recently, designers of safety-critical applications also have shown a growing interest in spatial and temporal isolation mechanisms, because these critical applications must be certified before being deployed. The rigorously (and cost) of the statutory certification process increases typically with the “Safety Integrity Level” (SIL) of the task to be certified and, when deployed on the same multicore architecture where tasks of different SILs can share low-level hardware resources, unless these tasks are shown to be sufficiently independent, the hardware and software must be developed at the highest SIL among the SILs of all these tasks, which is de-facto very expensive. This requirement is clearly stated in the international standards such as ISO 26262-4 [1] (automotive domain, req. 7.4.2.3) and IEC 61508 [2] (industrial systems, req. 7.6.2.10). This is why substantial efforts are put to (i) render the tasks of a same SIL as independent and isolated as possible from the tasks with different SILs and (ii) upper-bound the impact that the execution of the tasks of a same SIL may have on the execution behavior of the tasks of different SILs, with the objective of certifying each subset of tasks at its own SIL level. Within this context of real-time systems the focus of mixed-criticality systems is set on temporal isolation.

1 Spatial partitioning ensures that a task in one partition is unable to change private data of another task in another partition. Temporal partitioning, on the other hand, guarantees that the timing characteristics of a task, such as the worst-case execution time (WCET), are not affected by the execution of another task in another partition.

2 Every task is assigned a SIL during the design of the system to reflect its level of “criticality”.

ABSTRACT

Reservation systems are generally employed to enforce temporal isolation between applications. In the real-time context the corresponding temporal isolation requires not only the consideration of the direct interference due to execution of higher priority tasks, but also the indirect cost of e.g. cache-related preemption delay. The accounting of this in a server-based implementation of temporal isolation poses special challenges, in particular when misbehaving in the form of overruns and violation of the minimum inter-arrival time of an application are to be covered. We present a novel approach to extend the fault coverage and reduce the pessimism when compared to the state of the art. Furthermore we demonstrate that the extra implementation of the introduced mechanisms over the state of the art can be very low on complexity.
This might be in the form of asymmetric isolation, where it must be ensured only that tasks with low SIL do not impact those of higher SILs. However, the more interesting case is symmetric isolation, where tasks do not impact each other irrespective of their individual SIL. This is motivated through the fact that higher criticality tasks are not necessarily the ones with shorter deadlines, which leads to the fact that even the behaviour of the higher priority tasks needs to be monitored and acted upon in order not to jeopardize the temporal behaviour of higher criticality task with lower priority. A standard implementation of the symmetric temporal isolation is by the usage of servers [3], [4]. These servers provide a certain share of the processing resource called budget, which is supplied to a task in a recurrent fashion. While the general concepts of servers have been well explored, the use of implicitly shared resources, like caches is still an open issue for server-based systems. When a task executing in a server is preempted by a higher priority task, it loses at least partially its set of useful memory blocks in the caches (working set) and other shared resources. This loss of working set leads to an additional execution time requirement on resumption of execution, which either needs to be accounted for in the sizing of the budgets of individual tasks, or treated through online mechanisms. Besides easing timing analyses, schedulability analyses, and the certification process, the main objective of temporal isolation via servers is also to isolate applications from other temporally misbehaving applications and their corresponding effects. This misbehavior can come in two distinct flavors. Firstly, in terms of a violation of the worst-case execution time (WCET) assumptions, secondly in the minimum inter-arrival time assumption made in the analysis step. While the former has been treated to reasonable extent [4], the latter is of equal importance.

**Contribution of the paper:** Within this work we discuss analysis and online mechanisms which prove suitable to provide temporal isolation addressing two forms of misbehavior from a task:

1. Execution over-run (i.e. a job requesting a workload greater than the WCET assumed at design time)
2. Job released at a faster rate than assumed at design time

A simple solution for temporal isolation assurance in systems only subject to miss-behaviors of type 1) is achieved by setting the budgets of each server equal to the WCET of the task plus an upper bound on the maximum preemption delay that any job might suffer during its execution. We provide an alternate solution which ensures the temporal isolation for systems facing only type 1) miss-behaviors which consists of an online mechanism to transfer budgets from the server of the preempting task to the server of the preempted task at each preemption occurrence. We prove that our proposed solution is always less pessimistic than the over-dimensioning of the budgets of the preempted tasks. In order to maintain the temporal isolation in systems faced with miss-behaviors of type 1) and 2) we provide a further online mechanism for budget transferring. For this type of systems there exists no solution in the literature providing full temporal isolation.

**Organization of the paper:** In the next section, we provide an overview of related work followed by a description of our system model in Section 3. In this we also give a very brief review of the reservation-based framework which is used during the work discussion. The options for preemption delay accounting mechanisms in the reservation based system are discussed in Section 4 leading to the presentation of our budget augmentation approach in Section 5. Section 6 is devoted to the discussion of the violation of the minimum arrival assumption. The implementation concerns of our proposed techniques are presented in Section 7. Section 8 demonstrates the integration of preemption delay computation and evaluates our proposed techniques experimentally. Conclusions and indication of future work finalizes the document.

### 2. RELATED WORK

Reservation-based systems are quite a mature topic in real-time literature. Sporadic servers [3, 5] are used in real-time literature to ensure temporal isolation in fixed task priority systems. Each server reserves a budget for the task execution. A task can only execute if the server budget is not depleted. This ensures that the interference generated by a task in the schedule cannot exceed what is dictated by its server parameters and servers are the scheduled entities.

Solutions for temporal isolation in Earliest Deadline First (EDF) also exist employing the sporadic server concept, namely constant bandwidth server (CBS) [3] and Rate Based Earliest Deadline First (RBED) [4]. Each server has an absolute deadline associated to it which acts as the server priority. On top of the temporal isolation properties these frameworks also employ budget passing mechanisms which enhance the average-case response time of tasks in the system without jeopardizing the temporal guarantees [6, 7].

The previously employed execution models assume that the interference between workload occurs solely on the CPU. As it turns out, if other architectural subsystems are shared in the execution platform which present some state with non-negligible state transition times (e.g. caches), interference between task will be created (commonly referred to as preemption delay). The maximum preemption delay any task may endure may still be integrated into the task’s budget\(^3\), this solution is shown in this paper to be subject to heavy pessimism. When minimum inter-arrival times of tasks cannot be relied upon (i.e. tasks release jobs at a faster rate than assumed at design time), the previously mentioned frameworks fail to ensure temporal isolation between concurrent tasks in the system.

System with non-negligible CRPD have been a subject of wide study. Several methods have been proposed that provide an off-line estimation based on static analysis for this inter-task interference value. Lee et al. [8] presented one of the earliest contributions on CRPD estimation for instruction-caches. The authors introduced the concept of useful cache blocks, which describe memory blocks potentially stored in the cache at a given program point and that are potentially reused in the later program execution. They assume that the CRPD incurred by a task after the preemption by another task is constant throughout a basic block. By considering for the maximum quantity of information a task may have in the cache at every basic block and the maximum quantity of information that any preempting task can evict, they have devised a formulation for the computation of the maximum preemption delay a task may suffer. Several works followed, either by reducing the complexity of

\(^3\)we assign one server per task, the task and server term is used interchangeably in this document.
the schedulability analysis or by enhancing the definition of useful cache blocks \[9, 10, 11\].

Embedded in all the stated frameworks are schedulability tests. Scheduling analysis for \[8\] is based on response time analysis (RTA); Ju et al. \[12\] have provided a demand bound function-based procedure suitable for EDF schedulability with preemption delay awareness. The general approach of computing the CRPD is similar to Lee’s.

In order to ensure temporal isolation cache partitioning may be employed \[13\]. This technique has the disadvantage of decreasing the usable cache area available to each task, and as a consequence decreasing its performance. Other architectural subsystems exist (e.g. TLB, dynamic branch predictors, etc.) which cannot be partitioned in order to remove the interference source between tasks. Recently an approach has been proposed where, when a task starts to execute it stores onto the main memory all the contents of the cache lines it might potentially use \[14\]. After the pre-empting task terminates its execution it loads back from memory all the memory blocks that it has stored in the cache at its release. This indeed ensures temporal isolation among different applications but has several drawback. It unnecessarily moves all the memory blocks to main memory which reside in cache lines it might use even if the actual execution it does not access them. This mechanism significantly increases memory traffic which might be troublesome in multicore partitioned scheduling due to increased contention on the shared memory bus. In comparison our approach only passes budgets between servers and hence this budget is only used if it is required, leading to a more efficient system behaviour. As a last limitation of \[14\] it cannot cope with scenarios where a given task does not respect the minimum inter-arrival contract part. As a last resort non-preemptive scheduling policies may be employed. By nature, these are not subject to any preemption delay overhead. Even though fully preemptive fixed task priority and non-preemptive fixed task priority are incomparable with respect to schedulability (i.e. one does not dominate the other), the later presents lower schedulability capabilities \[15\].

3. SYSTEM MODEL

We model the workload by a task set \( \mathcal{T} = \{\tau_1, \ldots, \tau_n\} \) composed of \( n \) tasks, where each task \( \tau_i \) is characterized by the three-tuple \((C_i, D_i, T_i)\) with the following interpretation: \( \tau_i \) generates a potentially infinite sequence of jobs, with the first job released at any time during the system execution and subsequent jobs released at least \( T_i \) time units apart. Each job released by \( \tau_i \) must execute for at most \( C_i \) time units within \( D_i \) time units from its release. Hence, the parameter \( C_i \) is referred to as the “worst-case execution time”, \( D_i \) is the relative deadline of the task and \( T_i \) is its minimum inter-release time (sometimes called, its period). In this work we only focus on task sets where \( D_i \leq T_i \). These three parameters \( C_i, D_i \) and \( T_i \) represent an agreement between the task and the system. If the system can complete the execution of all the jobs by their respective deadline then the task set is said to be “schedulable”. Considering that the tasks are scheduled by a fixed-priority scheduling algorithm, e.g. Deadline-Monotonic, we assume that tasks are indexed by decreasing order of priority, i.e., \( \tau_1 \) has the highest priority and \( \tau_n \) the lowest one.

In this work, we use the notion of “contract”, where each task has a contract with the system. From the point of view of a task, this contract states that “as long as the task respect its parameters \( C_i \) and \( T_i \), its temporal deadline \( D_i \) will be met.” However, tasks may not always respect their contract. We say that a task \( \tau_i \) “behaves” if it does not require more CPU resources than indicated by its parameters \( C_i \) and \( T_i \). Otherwise, if any job of \( \tau_i \) comes to request more than \( C_i \) time units to complete, or if \( \tau_i \) releases two consecutive jobs in a time interval \( < T_i \) time units, then \( \tau_i \) is said to be “misbehaving”. The other party – the system – is assumed to never violate its contracts with any task. The system associates to each task \( \tau_i \) a sporadic server \( S_i \) defined by the two-tuple \((B_i, T_i^*)\). The parameter \( B_i \) encodes the execution budget that \( S_i \) provides to \( \tau_i \) in any time window of length \( T_i^* \). This budget is consumed as task \( \tau_i \) executes and a task can only execute if its budget is not depleted.

A sporadic server is, at any time instant, in either one of the two following states:

- **active** when there is pending workload from task \( \tau_i \) and \( B_i(t) > 0 \);
- **idle** when there is no pending workload from task \( \tau_i \) or \( B_i(t) = 0 \).

The sporadic server budget replenishment mechanics can be described succinctly by the protocol formulated with the two following rules:

- When the server becomes Active at a time \( t_1 \), a recharge event is set to occur at time instant \( t_1 + T_i^* \);
- when \( S_i \) becomes idle at a time \( t_2 \), the replenishment amount corresponding to the last recharging time is set to the amount of capacity consumed by \( S_i \) in the interval \([t_1, t_2]\).

At the start of the system \((t = 0)\) \( S_i \) is idle and \( B_i(t) = C_i \). We assume \( T_i = T_i^* \) for the sake of simplicity and hence, \( T_i \) is used throughout the document as a synonym for \( T_i^* \).

From this point onward, we assume that all the task deadlines are met at run-time as long as every job of each task \( \tau_i \) executes within the execution budget granted by \( S_i \) and respects its timing parameters \( C_i \) and \( T_i \). The framework proposed here ensures that, though any task \( \tau_i \) can misbehave by violating its stated parameters, the other tasks in the system will never miss their deadlines as long as they behave. Note that we assume throughout the document that each server has only a single task associated to it. The server and task terms are used interchangeably in the remainder of the document.

4. COMPARISON BETWEEN PREEMPTION DELAY ACCOUNTING APPROACHES

In a reservation-based system, as previously stated, each task \( \tau_i \) can only execute as long as \( B_i(t) \) is greater than 0 – Hereafter, \( B_i(t) \) denotes the remaining budget of server \( S_i \) at time \( t \). If every job is guaranteed to meet its deadlines, then at each time \( t \) where task \( \tau_i \) releases a job, it must hold that \( B_i(t) \) is greater than or equal to \( C_i \) plus the maximum preemption delay that the job may be subject to. We denote by \( \delta_j \) the maximum interference that a task \( \tau_i \) may induce in the execution time of task \( \tau_j \) by preempting it. This maximum interference can be computed by using methods such as the ones presented in \[9, 16, 17\].
Given all these $\delta_{j,i}$ values, we present below a naive solution to compute the budget $B_t$ of each task $\tau_i \in \mathcal{T}$. If we assume that task $\tau_i$ releases its jobs exactly $T_j$ time units apart, then the maximum number of jobs that $\tau_i$ can release in an interval of time of length $t$ is given by
\[
    n_j(t) \overset{\text{def}}{=} \left\lceil \frac{t}{T_j} \right\rceil
\] (1)

Therefore, during the worst-case response time of a task $\tau_i$ denoted by $R_i$, there are at most $n_j(R_i)$ jobs of task $\tau_j$, $j < i$, that can potentially preempt $\tau_i$. Since each of these preemptions imply an interference of at most $\delta_{j,i}$ time units on the execution of $\tau_i$, a straightforward way to compute the budget $B_t$ assigned to each task $\tau_i$ to meet all its deadlines is
\[
    B_t \overset{\text{def}}{=} C_i + \sum_{j=1}^{n-1} n_j(R_i) \times \delta_{j,i}
\] (2)

For the budget assignment policy defined in Equation (2), Equation (3) gives an upper-bound $PD_{\text{bgt}}^{\text{max}}(t)$ on the total CPU time that is reserved in any time interval $[0, t]$ to account for all the preemption delays.
\[
    PD_{\text{bgt}}^{\text{max}}(t) \overset{\text{def}}{=} \sum_{i=2}^{n} \left( n_i(t) \times \sum_{j=1}^{i-1} n_j(R_i) \times \delta_{j,i} \right)
\]
\[
    = \sum_{i=2}^{n} \left( n_i(t) \times \sum_{j=1}^{n} n_j(R_i) \times \delta_{j,i} \right)
\] (3)

as $\forall \ j \geq i$ it holds that $\delta_{j,i} = 0$.

It is worth noticing that Equation (2) assigns the budget of task $\tau_i$ by looking at how many times $\tau_i$ might get preempted during the execution of each of its jobs and how much each such preemption may cost. That is, this budget assignment policy implicitly considers the problem from the point of view of the preempted task.

An alternative approach to analyze the maximum preemption delay that a task can incur consists in considering the problem from the point of view of the preempting task. An example of such an approach has been presented by Stachulat et al. [11]. The authors defined the multi-set $M_{j,i}(t)$ as the set of all costs $\delta_{j,k}$ that tasks $\tau_j$ may induce in the execution requirements of all the tasks $\tau_k$ with a priority between that of $\tau_j$ and $\tau_i$, in a time window of length $t$. A multi-set is a generalisation of the set concept where elements may be replicated (i.e. a multi-set may be for example $\{x, x, x, y, y\}$ whereas a set consists of a collection of elements such that no element is equal to any other in the same set). The multi-set $M_{j,i}(t)$ is formally defined at any time $t$ as follows:
\[
    M_{j,i}(t) \overset{\text{def}}{=} \left( w_{k=j+1}^{i-1} n_k(t) w_{k=j+1}^{i-1} n_k(R_k) \delta_{j,k} \right)
\] (4)

The operator $\otimes$ denotes the union over multi-sets. Let us look at a brief example to differentiate between the multi-set union and the set union. For the multi-set union we have $\{x, y\} \otimes \{x, y\} = \{x, x, y, y\}$ whereas for the set union the outcome is $\{x\} \cup \{x, y\} = \{x, y\}$.

Each set $M_{j,i}(t)$ enables the construction of the function $\Delta_{j,i}(t)$, denoting the maximum preemption delay caused by jobs from task $\tau_j$ in any time window of length $t$.
\[
    \Delta_{j,i}(t) \overset{\text{def}}{=} \sum_{\ell=1}^{t} \max(M_{j,i}(t))
\] (5)

where
\[
    q_{j,i}(t) \overset{\text{def}}{=} \sum_{k=j}^{i-1} \min(n_k(t), n_j(t))
\] (6)

and the function $\max(M_{j,i}(t))$ returns the $\ell$th highest value in the set $M_{j,i}(t)$ – the equation $\Delta_{j,i}(t) \overset{\text{def}}{=} \sum_{\ell=1}^{t} \max(M_{j,i}(t))$ thus represents the sum of the $q_{j,i}(t)$ highest values in $M_{j,i}(t)$.

We show below that, considering the preemption delay from the perspective of the preemtping task is always less pessimistic than considering the preemption delay from the point of view of the preempted task.

**THEOREM 1.** For each task $\tau_i \in \mathcal{T}$, it holds at any time $t$ that
\[
    \sum_{j=1}^{n} \Delta_{j,i}(t) \leq PD_{\text{bgt}}^{\text{max}}(t)
\] (7)

**PROOF.** From Equation (4) and since $\forall j \geq i$ it holds that $\delta_{j,i} = 0$, for all $\tau_j \in \mathcal{T}$ the sum of all elements in $M_{j,i}(t)$ is given by:
\[
    \sum_{\ell=1}^{t} \max(M_{j,i}(t)) = \sum_{\ell=1}^{t} \sum_{k=j+1}^{n} n_k(t) \times n_j(R_k) \times \delta_{j,k}
\] (8)

We now split the remainder of the proof into two lemmas that will straightforwardly yield Equation (7).

**LEMMA 1.** $\sum_{j=1}^{n} \sum_{\ell=1}^{t} \max(M_{j,i}(t)) \leq PD_{\text{bgt}}^{\text{max}}(t)$

**PROOF.** From Equation (8), we know that
\[
    \sum_{j=1}^{n} \sum_{\ell=1}^{t} \max(M_{j,i}(t)) \leq \sum_{j=1}^{n} \sum_{k=2}^{n} n_k(t) \times n_j(R_k) \times \delta_{j,k}
\]
\[
    \leq \sum_{k=2}^{n} n_k(t) \times \sum_{j=1}^{n} n_j(R_k) \times \delta_{j,k}
\]
from (3)
\[
    \leq PD_{\text{bgt}}^{\text{max}}(t)
\]

**LEMMA 2.** $\sum_{j=1}^{n} \sum_{\ell=1}^{t} \max(M_{j,i}(t)) \geq \sum_{j=1}^{n} \Delta_{j,i}(t)$

**PROOF.** On the one hand, one can observe from Equation (4) that the number of elements in the multiset $M_{j,i}(t)$ is given by
\[
    \#M_{j,i}(t) = \left( \sum_{k=j+1}^{i-1} n_k(t) \times n_j(R_k) \right) + n_j(t)
\] (9)

and since $n_j(R_k) \geq 1$, $\forall j, k \in [1, n]$, it holds that
\[
    \#M_{j,i}(t) \geq n_j(t) + \sum_{k=j+1}^{i-1} n_k(t)
\] (10)

On the other hand, since when $j = k$ we have
\[
    \min(n_k(t), n_j(t)) = \min(n_j(t), n_j(t)) = n_j(t)
\]

the function $\max(M_{j,i}(t))$ returns the $\ell$th highest value in the set $M_{j,i}(t)$ thus represents the sum of the $q_{j,i}(t)$ highest values in $M_{j,i}(t)$.
and thus this Equation (6) can be rewritten as:

\[ q_j, i(t) = n_j(t) + \sum_{k=j+1}^{i-1} \min(n_k(t), n_j(t)) \]

\[ \leq n_j(t) + \sum_{k=j+1}^{i-1} n_k(t) \quad (11) \]

By combining Equations (10) and (11), it thus holds \( \forall j, i \in [1, n] \) that

\[ q_j, i(t) \leq \#M_{j, i}(t) \quad (12) \]

Remember that \( \sum_{i=1}^{t} \max(M_{j, i}(t)) \) represents the sum of the \( q_{j, i}(t) \) highest values in \( M_{j, i}(t) \). From Inequality (12) and by definition of the function \( \max(M_{j, i}(t)) \), we can conclude that \( \forall t > 0 \) and for all \( \tau_j, \tau_i \in T \):

\[ \sum_{e \in M_{j, i}(t)} e \geq \sum_{t=1}^{\max(M_{j, i}(t))} e \quad (13) \]

By summing Inequality (13) over all \( j \in [1, n] \), we get

\[ \sum_{j=1}^{n} \sum_{e \in M_{j, i}(t)} e \geq \sum_{j=1}^{n} \sum_{t=1}^{\max(M_{j, i}(t))} e \]

\[ \sum_{j=1}^{n} \Delta_{j, i}(t) \geq \sum_{j=1}^{n} \max(M_{j, i}(t)) \]

Hence the lemma follows. \( \square \)

Finally, by combining Lemmas 1 and 2 it is easy to see that

\[ \sum_{j=1}^{n} \Delta_{j, i}(t) \leq \text{PD}_{\text{max}}(t) \]

\( \square \)

5. PROPOSED BUDGET AUGMENTATION FRAMEWORK

5.1 Description of the framework

Since a task may incur some delay due to a preemption, it is straightforward that an execution budget of \( B_i = C_i \) may be insufficient for the task \( \tau_i \) to complete if it gets preempted during its execution. On the other hand, the budget assignment policy defined by Equation (2) has been shown to be (potentially) pessimistic. Hence, we propose a run-time mechanism where every preemption makes \( \text{pay for the damage that it causes to the schedule}. \)

According to Theorem 1 by accounting for the preemption delay from the point of view of the preempting task leads to reduced wastage of system resources. Formally, the execution budget \( B_i \) of each task \( \tau_i \) is initially set to \( C_i \) and refilled according to the sporadic server definition. Then, each time a task \( \tau_i \) resumes its execution after being preempted by another task(s), the remaining budget \( B_i(t) \) of its associated server \( S_i \) is increased by \( \sum_{j \in I_i} \delta_{j, i}(t) \) (where \( I_i \) denotes the set of tasks that preempted \( \tau_i \)) to compensate for the potential extra execution requirement that \( \tau_i \) may incur.

An example of the described framework is presented in Figure 1. In that example the task set contains 4 tasks.

![Figure 1: Budget Augmentation Example](Image)

Task \( \tau_4 \) is first preempted by a job from \( \tau_2 \). When \( \tau_4 \) resumes execution at time \( t_1 \), immediately after \( \tau_2 \) terminates, its remaining budget \( B_4(t) \) is incremented by \( \delta_{2, 4} \) units. Then, two jobs of \( \tau_1 \) preempt both \( \tau_1 \) and (indirectly) \( \tau_4 \). Each time \( \tau_4 \) resumes its execution at the return from the preemption (at time \( t_2 + \delta_{4, 2} \)), the execution budget \( B_4(t_2) \) is incremented by \( \delta_{1, 3} \). Finally, when \( \tau_4 \) terminates its workload and \( \tau_4 \) resumes at time \( t_4 \), \( B_4(t_4) \) is incremented by \( \delta_{4, 3} + \delta_{3, 4} \) as both \( \tau_1 \) and \( \tau_4 \) might have evicted some of its cached data; hence forcing \( \tau_4 \) to reload it from the memory.

5.2 Schedulability Analysis

When the preemption delay is assumed to be zero (i.e., when the cache subsystem is partitioned for example), the authors of \([18]\) proposed the following schedulability test to check at design-time, whether all the task deadlines are met at run-time.

**Schedulability Test 1 (From [18]).** A task set \( T \) is schedulable if, \( \forall \tau_i \in T, \exists t \in (0, D_i) \) such that

\[ C_i + \sum_{j=1}^{i-1} \text{rbf}(S_j, t) \leq t \quad (14) \]

where

\[ \text{rbf}(S_j, t) \overset{\text{def}}{=} \left( \left\lceil \frac{t}{T_j} \right\rceil + 1 \right) \times B_j \quad (15) \]

**Theorem 2 (From [5]).** A periodic task-set that is schedulable with a task \( \tau_i \), is also schedulable if \( \tau_i \) is replaced by a sporadic server with the same period and execution time.

**Proof.** Follows from the proof of Theorem 2 in [5]. \( \square \)

The correctness of the schedulability test 1 comes as a direct consequence of the Theorem 2 as the presented test is the one for a task-set composed of periodic tasks \([18]\).

As introduced earlier, if every task \( \tau_i \) augments its budget for \( \delta_{j, i} \), time units after being preempted by a task \( \tau_j \), then an upper bound on the total budget augmentation in any time window of length \( t \) is given by \( \sum_{j=1}^{i-1} \Delta_{j, i}(t) \).

Besides, it can be shown that in any given time window of length \( t \), an upper-bound on the number of execution
resumptions in a schedule is given by \( q_{1,i}(t) \). Therefore, assuming that performing each execution of the budget augmentation consumes \( F_{\text{cost}} \) units of time, the time-penalty attached to the implementation of the proposed framework has an upper bound of

\[
\text{cost}(t) = q_{1,i}(t) \times F_{\text{cost}} \tag{16}
\]

Integrating these quantities into Schedulability Test 1 yields the following test:

**Schedulability Test 2.** A task set \( T \) is schedulable if, \( \forall \tau_i \in T, \exists \delta \in (0, D_i] \) such that

\[
C_i + \text{cost}(t) + \sum_{j=1}^{\delta-1} [\text{rbf}(S_j, t) + \Delta_{j,i}(t)] \leq t \tag{17}
\]

**Proof Correctness of Schedulability Test 2.** The Function (6) quantifies the maximum number of times that jobs from task \( \tau_j \) may preempt jobs of priority lower than \( \tau_j \) and higher or equal than \( \tau_j \) in a window of length \( t \). The Function 5 (\( \Delta_{j,i}(t) \)) is the summation over the \( q_{j,i}(t) \) largest values in the multi-set \( M_{j,i}(t) \). The function \( \Delta_{j,i}(t) \) is then an upper-bound on the amount of preemption delay compensation that can be extracted from task \( \tau_j \) from any task of priority lower than \( \tau_j \) and higher or equal than \( \tau_j \) in a window of length \( t \). Thus \( \sum_{j=1}^{\delta-1} \Delta_{j,i}(t) \) is an upper-bound on the preemption delay compensation budget used by tasks of priority higher or equal than \( \tau_i \) for any time \( t \). As a consequence and by the correctness of schedulability test 1, the correctness of this schedulability test is proven. \( \square \)

According to Schedulability Test 2 and as a consequence of Theorem 1, if we assume \( \text{cost}(t) = 0 \) then the proposed framework enables a higher schedulability than considering the budget \( B_i \) of each server \( S_i \) to be equal to \( C_i \) plus the maximum preemption delay that any job of \( \tau_i \) may be potentially subject to (see Equation (2)). However, in a scenario where \( \text{cost}(t) \) is non-negligible the dominance relation is no longer present.

### 6. PROPOSED BUDGET DONATION FRAMEWORK

![Figure 2: Excessive Preemption Delay Due to Minimum Interarrival Time Violation](image)

The framework presented above is a combination of a reservation-based mechanism and a budget augmentation policy. This powerful combination ensures that the temporal isolation property is always met as long as none of the tasks violates its minimum inter-arrival constraint, i.e., as long as none of them release two consecutive jobs in a time interval shorter than its pre-defined period \( T_j \). This condition of not violating the minimum inter-arrival constraint is implicitly assumed by Equations (5) and (6), in which the upper-bound on the preemption delay interference inherently relies on the number of jobs released by every task in a given time window.

If any task violates its minimum inter-arrival time constraint, the temporal isolation property no longer holds. An example of this is depicted in Figure 2. On the left-hand side of the picture, task \( \tau_1 \) releases a single job in a time interval of length \( T_1 \). This job executes for \( C_1 \) time units and task \( \tau_2 \) suffers only from one preemption, leading to an increase of \( \delta_{1,2} \) on its execution requirement. In the right-hand side of the picture, \( \tau_1 \) releases more than one job in the same time interval of length \( T_1 \) and \( \tau_2 \) now suffers from 2 preemptions, leading to an increase of \( 2 \times \delta_{1,2} \) on its execution requirement. In the latter scenario, task \( \tau_2 \) augments its budget accordingly but may face a deadline miss, or a lower priority tasks may be subject to more interference than what was accounted for in the schedulability test.

In order to avoid this issue, the system associates a second server \( Y_i \) to each task \( \tau_i \). This server \( Y_i \) has parameters \((Z_i, T_i^Y) = Z_i\) and its budget and \( T_i^Y \) is the replenishment period. Unlike the server \( S_i \), the budget \( Z_i \) is not debited while \( \tau_i \) is running. The purpose of this second server \( Y_i \) is to “pay” for the damage caused by \( \tau_i \) in the system when \( \tau_i \) preempts another task. That is, each task \( \tau_i \) when it is preempted by \( \tau_i \), obtains a budget donations by transferring some execution budget from the server \( Y_i \) to its execution budget \( B_i \). These budgets \( Y_i \) impose a new condition to their associated task \( \tau_i \) in order to accommodate for the minimum inter-arrival misbehavior: task \( \tau_i \) is allowed to release a new job only if there is sufficient budget in \( Y_i \).

In this way the preemption delay that \( \tau_i \) may cause in the schedule is tightly monitored.

The Replenishment condition for server \( Y_i \) is defined as follows:

- At time instant \( t' \) when a task \( \tau_j \), after being preempted by \( \tau_i \), requests a preemption delay compensation of \( \delta_{i,j} \) from \( \tau_i \), it sets a replenishment event for \( Y_i \) at \( t' + T_i^Y \). The amount of budget replenished to server \( Y_i \) at the \( t' + T_i^Y \) event is equal to \( \delta_{i,j} \).

This replenishment mechanism is in accordance with the sporadic server replenishment rules [5] and hence the server \( Y_i \) is a sporadic server.

These two parameters \( Z_i \) and \( T_i^Y \) are set by the system designer and the question of how to define them will be discussed later. For now, bear in mind that these two parameters are given for each task \( \tau_i \in T \).

The purpose of each server \( Y_i \) is to ensure that new jobs of a task \( \tau_i \) can only be released as long as the maximum preemption delay that \( \tau_i \) can induce in the schedule (according to the schedulability test) is available in \( Y_i \). To effectively implement this solution, we reformulate the budget augmentation mechanism presented in the previous section as a budget transfer mechanism. The main concept remains simple:

1. To release a new job (say at time \( t \)), a task \( \tau_i \) is required to have at least \( P_{j}^{\max} \) time units in its budget \( Z_j(t) \). This quantity \( P_{j}^{\max} \) is the maximum delay that \( \tau_i \) can cause on the lower priority tasks by preempting them. It is straightforwardly defined as

\[
P_{j}^{\max} \triangleq \sum_{k=j+1}^{n} \delta_{j,k} \tag{18}
\]
If \( Z_i(t) < P_j^{\text{max}} \) then \( \tau_j \) is not authorized to release a new job at time \( t \) and must wait until the earliest time instant \( t' > t \) when \( Z_j(t') \geq P_j^{\text{max}} \).

2. Unlike the budget augmentation protocol proposed in the previous section, each time a task \( \tau \) resumes its execution (say at time \( t \)) after being preempted (let \( H(i) \) denote the set of tasks that preempted \( \tau_i \)), \( \tau \) does not see its execution budget \( B_i(t) \) being simply augmented by \( \sum_{\tau_j \in H(i)} \delta_{j,i} \) time units, with \( \sum_{\tau_j \in H(i)} \delta_{j,i} \) coming from thin air. Instead, \( \delta_{j,i}(t) \) time units are transferred from the budget \( Z_j(t) \) of each task \( \tau_j \in H(i) \) to its execution budget \( B_i(t) \).

Informally speaking, the underlying concept behind this budget transfer protocol can be summarized as follows: “a task \( \tau \) is allowed to release a new job only if it can pay for the maximum damage that it may cause to all the tasks that it may preempt”. If the task \( \tau \) can pay the required amount of time units, i.e., \( \tau \) has a provably sufficient amount of time units saved in its budget \( Z_i(t) \), then it can release its new job and the preempted tasks will claim their due preemption delay compensation when they will eventually resume their execution.

This simple concept makes the framework safe. Rather than a formal proof, we give below a set of arguments to illustrate the claim. Suppose that a task \( \tau \) starts misbehaving by frenetically releasing jobs that execute for an arbitrarily short time; hence clearly violating its minimum inter-arrival constraint.

1. From the point of view of a higher priority task (say, \( \tau \)): each job of \( \tau \) can preempt at most one job from \( \tau \) and before releasing each of its jobs, \( \tau \) makes sure that there is enough provision in its budget \( Z_j(t) \) to compensate for the damage caused to the lower priority tasks, including \( \tau \).

2. From the point of view of the misbehaving task \( \tau \): this task will keep on generating jobs until its budget \( Z_i(t) \) is depleted. For each job released, the framework makes sure that the job can actually pay for the damage caused to the lower priority tasks. Regarding the higher priority tasks, each job of \( \tau \) may be preempted and request some extra time units upon resumption of its execution. However, this extra budget requested has been accounted for when the higher priority jobs were allowed to be released – as mentioned in 1).

3. From the point of view of a lower priority task (say, \( \tau \)): each job of \( \tau \) may be preempted multiple times by the frenetic job release pattern of \( \tau \). However, upon each resumption of execution, \( \tau \) will be compensated for the delay incurred by receiving some extra time units from the budget \( Z_i(t) \) of the misbehaving task – as guaranteed in 2).

As seen, the sole purpose of each server \( Y_i \), \( \forall i \in [1, n] \) is to control the preemption delay that the task \( \tau \) induces on the schedule. Since the upper-bound on the preemption delay related interference is now dictated by these servers, Schedulability Test 2 presented in the previous section can be rewritten as:

\[
\text{Schedulability Test 3. A task set } \mathcal{T} \text{ is schedulable if, } \forall \tau_i \in \mathcal{T}, \exists t \in (0, D_i] \text{ such that }
\]

\[
C_i + \text{cost}(t) + \sum_{j=1}^{i-1} [\text{rbf}(S_j, t) + \text{rbf}(Y_j, t)] \leq t
\]

**Proof Correctness of Schedulability Test 3.** The replenishment mechanism of server \( Y_i \) is in accordance with the sporadic server replenishment rules. As a consequence of this fact and according to the Theorem 2 the maximum amount of budget consumed in any interval of length \( T_i^\ell \) is \( Z_i \). This means that \( \text{rbf}(Y_i, t) \) is an upper-bound on the budget used for execution by any task that was preempted by task \( \tau \) and got the due compensation in any interval of length \( t \). By this reasoning and the correctness of schedulability tests 1 and 2 the correctness of this schedulability test is thus proven.

The choice of the parameters of each \( Y_i \) server is left at the criteria of the system designer. However, in a given period \( T_i \) any task \( \tau_i \) will require at least the execution of one job. As a consequence the budget \( Z_i \) of \( Y_i \) must necessarily be greater than or equal to \( P_i^{\text{max}} \). The simpler approach would be to define each server \( Y_i \) as \((Z_i = P_i^{\text{max}}, T_i = T_i)\) as \( Y_i \) would have enough budget to compensate for all the interference that \( \tau_i \) may cause in the schedule, assuming that the minimum inter-arrival constraint is not violated. However, the system designer may prefer \( T_i^\ell > T_i \) to provide more flexibility in case the task \( \tau_i \) is expected to violate its minimum inter-arrival constraint, or even to accommodate intended bursty arrival of requests.

As a last note it is important to state the advantage of this framework with respect to a simple mechanism imposing a limitation on the number of jobs a task may release in a given time window. With our framework the number of jobs that a task may release without breaking the temporal isolation guarantees is variable since this depends on the quantity of lower priority jobs that it has preempted so far. This allows for a more dynamic system with overall better responsiveness.

### 7. Implementation Issues

An efficient implementation is key for a useful approach to server-based scheduling. The general principle pursued is that the preemption delay repayment budget is only granted when an actual preemption occurs and is only transferred on resumption of execution.

![Preemption Delay Compensation Array](image)

**Figure 3: Preemption Delay Compensation Array**

As a first step we arrange the preemption delay repayment budget values \( \delta_{i,j} \) in a linear preemption delay array (Figure 3). As a second step the concept of preemption queue is introduced. Whenever a job is first executed it is entered as the new head of the preemption queue. On completion of a job, the highest priority job of the ready queue is picked for execution and, if executed for the first time, inserted as the new head of the ready queue, claiming the pointer of the run queue relationship of the completed job. If the job is
have preempted corresponding preemption delay donations are deduced from considered framework is the one ensuring temporal isolation $\tau$. The preemption queue provides fundamental information to guide the system in the budget augmentation procedure. The completion or overrun of budget of jobs is indicated since the last time instant at which it was executing. The bit relative to $\tau$ is stored (Figure 3). When a job $\tau_i$ finishes its execution. At this time instant, the out bitfield is initialized with only set (i.e. $\text{out}_{bf}(\tau_1) = 0x0$). These three procedures need to be carried out atomically.

Let us use a different schedule example to better visualize the bitfield related operations. In Figure 5 an example of the bit-field evolution as a response to a given set of events is presented. There are 9 events present. When a task has no active job in the system its bit field is by default $0$. At events 1–3, tasks $\tau_1$ to $\tau_3$ release jobs, consequently their bit fields are set to their respective index. At time $t_4$, the first job of task $\tau_2$ finishes its execution. At this time instant, and since $\tau_2$ is the prior preemption queue head and $\tau_3$ is the task after the queue head, the bit field from task the $\text{in}_{bf}(\tau_3) = \text{in}_{bf}(\tau_1) \lor \text{out}_{bf}(\tau_2)$. After this operation is performed $\text{in}_{bf}(\tau_2) = 0$. At $t_5$ and $t_6$ jobs from $\tau_1$ and $\tau_2$ are released correspondingly. At each event the corresponding bitfield is set accordingly. The job from task $\tau_1$ terminates at $t_7$. At this time instant, the out bitfield of task $\tau_1$ is passed onto the subsequent task in the preemption queue ($\tau_3$). Hence $\text{in}_{bf}(\tau_3) = \text{in}_{bf}(\tau_1) \lor \text{out}_{bf}(\tau_1)$. Since $\tau_2$ was not in the ready queue the out bitfield from $\tau_1$ does not get passed to its in bitfield.

The algorithm describing the bitfield inspection and budget augmentation is presented in Algorithm 1. Notice that in the scenario where minimum inter-arrival times cannot be relied upon, after the budget is augmented (line 6), the same value has to be decreased from the $Y_{\text{index}}$ server of the preempting task. In the same framework it is then necessary to set up a replenishment event for server $Y_{\text{index}}$, $T_{\text{index}}$ units after the execution resumption (line 7). At this replenishment event $Z_{\text{index}}$ is set to be replenished by $\delta_{\text{array\_index}}$ units. The variable offset is an offset with respect to the base of the $\delta$ array, where the first element preemption delay compensation value ($\delta_{1,i}$) from task $\tau_i$ is stored (Figure 3).

8. Cache Related Preemption Delay Proof of Concept

The previously described theory may be exploited in order to ensure temporal isolation with respect to any subsystem holding state concurrently accessed by tasks in the system. In this section we describe how temporal isolation is ensured when caches are present in the execution platform by usage
of the framework described in this document. At each program point a task has a set of useful memory blocks (UCB) in cache. These memory blocks were loaded into the cache at some prior program point and will be reused in the future. At each program point p of task τk we have UCBp,k [9]. Additionally, the set of cache lines accessed during the execution of task τj (ECBj) [9] may be constructed.

The maximum CRPD which a preemption by task τj may induce in task τk is then defined as [9]:

\[ \delta_{j,k} \triangleq BRT \times \max_p \left\{ |\text{UCB}_p,k \cap \text{ECB}_j| \right\} \]  \hspace{1cm} (20)

BRT is a constant, denoting the worst case latency for a cache miss to be served.

8.1 Temporal Isolation Assumptions

In order to ensure full temporal isolation with the used preemption delay estimation mechanisms, when using information from the preemptioning tasks to compute the \( \delta_{j,i} \), it is mandatory to ensure that the ECB sets of each preempting task are met at run-time. This is because it is assumed that a task can execute for longer than expected, which implies that it might in turn also use a larger cache footprint.

Both instruction and data cache footprints have to be relied upon. In order to ensure the worst-case cache footprint, it suffices to ensure that each task only accesses the basic blocks assumed in the static analysis. This can be easily achieved with the work by Martín Abadi et al. [19], where each jump instruction is guarded against jumps into illegal memory locations. Since at run-time, all the possible basic blocks accessed are the exact same as the ones assumed in the static analysis the instruction cache ECB set is never violated. When using [19], all the jumps in the code are statically defined. Jump to function pointers in the code are not allowed. These are generally forbidden in safety critical workload.

In the case of data caches, since the basic blocks accessed at run-time are enforced, it is sufficient to ensure that no dynamic memory is used, which is generally the case in real-time workload. Another assumption is that any access to an array is guarded against accesses outside of the array bounds. As a last assumption, recursive calls are forbidden in order to ensure that the stack footprint is known offline.

If the ECB set of any task cannot be relied upon, it is always safe to consider that the maximum preemption delay induced in task τk when preempted by any task τj is the time to reload the entire UCBk set:

\[ \forall j : \delta_{j,k} = BRT \times |\text{UCB}_k| \]  \hspace{1cm} (21)

8.2 Experimental Results

In order to assess the validity of the contribution a set of experiments was extracted. The schedulability guarantees from the proposed framework is trialled against the scenario where the maximum preemption delay is integrated into the budget of the execution server. Results for the same task sets are also displayed for a schedulability test oblivious of preemption delay. In each model all tasks are generated using the unbiased task set generator method presented by Bini (UUniFust) [20].

Task systems are randomly generated for every utilization step in the set \{0.75, 0.8, 0.85, 0.9, 0.95\}, their maximum execution requirements (Ci) were uniformly distributed in the interval [20, 400]. Knowing Ci and the task utilization Ui, Ti is obtained. At each utilization step 1000 task sets are trialled and checked whether the respective algorithm considers it schedulable. Task set sizes of 4, 8, and 16 tasks have been explored. The relative deadline of tasks is equal to the minimum inter-arrival time (Di = Ti). When each task is randomly generated the set of useful cache blocks is randomly obtained. The cache considered is composed of 10 cache lines. In the computation of each task τk UCBj set, the usage of each cache line is uniformly distributed. Without loss of generality, only one UCB set is generated for each task τk. Taking the UCB set as input, all the cache lines which were not occupied in the UCB set have a 0 probability of being occupied in the ECB set.

The servers Sj have been attributed parameters Bi = Ci and Ti = Ti. For the situation where the preemption delay server Yi is put to use, its parameters are Zj = \( \sum_{j<k} \delta_{j,k} \) and TiY = Ti.

The results are depicted in Figures 6(a) to 6(c). The number of task considered is in the set \{4, 8, 16\}. In the plots the scenario where the preemption delay is incorporated into the task execution budget is displayed is presented by the green line with “x” points. The presented framework for well-behaving minimum inter-arrival times is represented by the red line with “+” points. The purple line with square points represents the framework performance for situations where the minimum inter-arrival times cannot be trusted. Finally the blue line with “star” points displays the results for fixed task priority schedulability test disregarding preemption delay.

From the displayed results it is apparent that the schedulability achieved with the proposed framework is generally much higher than the one enabled by the simpler version considering the preemption delay as part of the execution budget. When the minimum inter-arrival times cannot be relied upon the schedulability degrades. It is important to note that the proposed framework ensures temporal isolation whereas there exists no other solution apart from the proposed framework which ensures the temporal isolation property for the given system model. Furthermore, when the number of tasks is small, the framework which provides the stronger guarantees appears to have average a higher scheduling performance. The schedulability reduction attached to the framework for misbehaving tasks with respect to the minimum inter-arrival time is the price to pay for the added guarantees.
9. CONCLUSION AND FUTURE WORK

Reservation-based systems are one fundamental way to enforce temporal isolation in mixed-criticality real-time systems. We show that, when preemption delay is present in the system, the state of the art mechanism of reservation-based systems require pessimism in the analysis and in the budget allocation procedures. Due to this inherent limitation, a run-time budget augmentation mechanism is presented in this paper. This framework enables the reduction of this pessimism with respect to the preemption delay to be accounted for in the dimensioning of the reservation systems when tasks may face budget overruns. For a model in which the minimum inter-arrival time of tasks cannot be relied upon, there existed no prior result in the literature ensuring temporal isolation (for systems where preemption delay is non-negligible). We propose a second framework which by relying on budget transfers between preempting and preempted tasks effectively enforces the temporal isolation property in such a considerably challenging system model (where execution requirements and minimum inter-arrival times cannot be trusted upon). Furthermore we have devised a very promising implementation strategy.

As future work we intend to implement the proposed mechanics in a real-time system scheduler in order to assess the overhead associated with both frameworks. As an outcome, we expect to shown that the overhead of such a system is generally less demanding than the pessimism which would otherwise have to be included in the analysis if the on-line mechanism would not be present. We further intend to show the robustness of the solution in scenarios where several tasks misbehave with respect to their declared minimum inter-arrival times.

10. REFERENCES


