Temporal Isolation with Preemption Delay Accounting

José Marinho

Vincent Nélis

Stefan M. Petters

CISTER-TR-140703

09-16-2014

Temporal Isolation with Preemption Delay Accounting

José Marinho, Vincent Nélis, Stefan M. Petters

CISTER Research Center

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

Temporal Isolation with Preemption Delay Accounting

José Marinho* Vincent Nélis* Stefan M. Petters*
*CISTER/INESC-TEC, Polytechnic Institute of Porto, Portugal
Email: {jmsm,nelis,smp}@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 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.**

I. 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 embedded devices. Besides having specific functional requirements many functionalities (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. Tasks exposing such timing requirements are usually called "real-time" tasks. 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 task depends on both the logical results of its computations and the time at which these results are produced.

Before a safety-critical system can be deployed and marketed, a certification authority must validate that all the safety-related norms are met. All the components comprising that system (the software, the hardware, and the interfaces) are scrutinized to ensure conformance to safety standards. Timing guarantees must be derived at design time and consequently enforced during run-time for the system to 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[s]. 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 case 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).

A first step in the certification process is to categorize each component (software and hardware) by its level of criticality and assign a unitary safety integrity level[1] (SIL) to all components. When integrated in the same platform the components of different SILs share low-level hardware resources such as cores, cache subsystems, communication buses, main memory, etc. To provide the required degree of "sufficient independence" between components of different SILs, Industry and Academy have been seeking solutions for many years to (1) render the components of a same SIL as independent and isolated as possible from the components with different SILs and (2) upper-bound the residual impact that components of different SILs may have on each other after the segregation step, with the primary objective of certifying each subset of components at its own SIL.

Within this work, the focus is set on temporal isolation. We consider the scenario where *symmetric* isolation is enforced, i.e. where tasks do not impact each other irrespective of their individual SIL. A standard implementation of the symmetric temporal isolation is by the usage of servers [1], [2]. 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.

Contribution of the paper: 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. Within this work we discuss analysis and online mechanisms which prove suitable to pro-

[1]A Safety Integrity Level (SIL) is defined as a relative level of risk-reduction provided by a safety function, or to specify a target level of risk reduction. In simple terms, a SIL is a measurement of performance required for a safety instrumented function.

vide temporal isolation addressing two forms of misbehavior from a task: (a) Execution over-run, i.e. a job requesting a workload greater than the WCET assumed at design time, and (b) job released at a faster rate than assumed at design time. A simple solution to ensure the temporal isolation in systems only subject to miss-behaviors of type (a) is achieved by setting the budgets of each server equal to the worst-case execution time (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 for such systems: an online mechanism that augments the cpu-budget associated to a task whenever it resumes its execution (on returning from preemption). Then, in order to maintain the temporal isolation in systems faced with miss-behaviors of type (a) and (b) we provide a further online mechanism that transfers budgets from the server of the preempting task to the server of the preempted task at each preemption occurrence. 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 III. 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 IV leading to the presentation of our budget augmentation approach in Section V. Section VI is devoted to the discussion of the violation of the minimum arrival assumption. An experimental evaluation followed by conclusions and indication of future work finalizes the document.

II. Related Work

Reservation-based systems are quite a mature topic in real-time literature. Sporadic servers [1], [3] are proposed 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) [1] and Rate Based Earliest Deadline First (RBED) [2]. 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 [4], [5].

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[2], 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 computed at design time), the previously mentioned frameworks *fail* to ensure temporal isolation between concurrent tasks in the system.

[2]we assign one server per task, the task and server term is used interchangeably in this document

Systems with non-negligible cache related preemption delay (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. [6] 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 the schedulability analysis or by enhancing the definition of useful cache blocks [7], [8], [9].

Embedded in all the stated frameworks are schedulability tests. Scheduling analysis for [6] is based on response time analysis (RTA); Ju et al. [10] 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 [11]. This technique has the disadvantage of decreasing the usable cache area available to each task, and as a consequence impacting 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 [12]. After the preempting 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 drawbacks. It unnecessarily moves all the memory blocks to main memory which reside in cache lines it might use even if the actual execution does not access them. This mechanism significantly increases memory traffic which may 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. As a last limitation of [12] 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 to avoid CRPD. 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 [13].

III. System Model

We model the workload by a task set $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ composed of n tasks, where each task τ_i is characterized by the three-tuple $\langle C_i, D_i, T_i \rangle$ with the following interpretation: τ_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 τ_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 (often called, its period). In this work we only focus on task sets where $D_i \leqslant T_i$. 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., τ_1 has the highest priority and τ_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 τ_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 τ_i comes to request more than C_i time units to complete, or if τ_i releases two consecutive jobs in a time interval $< T_i$ time units, then τ_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 τ_i a *sporadic* server S_i defined by the two-tuple $\langle B_i, T_i^s \rangle$. The parameter B_i encodes the execution budget that S_i provides to τ_i in any time window of length T_i^s. This budget is consumed as task τ_i executes and a task can only execute if its budget is not depleted. We shall resort to the function $B_i(t)$ to denote the remaining budget in the server S_i at every time instant t.

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

active when there is pending workload from task τ_i **and** $B_i(t) > 0$;

idle when there is no pending workload from task τ_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 transits to the **Active** state at a time t_1, a recharging event is set to occur at time instant $t_1 + T_i^s$;
- when S_i transits to the **idle** state 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 $[t1, t2)$.

At the start of the system ($t = 0$) S_i is **idle** and $B_i(t) = C_i$.

We assume $T_i^s = T_i$ for the sake of simplicity and hence, T_i is used throughout the document as a synonym for T_i^s.

From this point onward, we assume that all the task deadlines are met at run-time as long as every job of each task τ_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 τ_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.

IV. Comparison Between Preemption Delay Accounting Approaches

In a reservation-based system, as previously stated, each task τ_i can only execute as long as $B_i(t)$ is greater than 0. If every job is guaranteed to meet its deadlines, then at each time t where task τ_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 during its execution. We denote by $\delta_{j,i}$ the maximum interference that a task τ_j may induce in the execution time of task τ_i by preempting it. This maximum interference can be computed by using methods such as the ones presented in [7], [14], [15].

Given all these $\delta_{j,i}$ values, we present below a naive solution to compute the budget B_i of each task $\tau_i \in \mathcal{T}$. If we assume that task τ_j releases its jobs *exactly* T_j time units apart, then the maximum number of jobs that τ_j can release in an interval of time of length t is given by

$$n_j(t) \stackrel{\text{def}}{=} \left\lceil \frac{t}{T_j} \right\rceil \tag{1}$$

Therefore, during the worst-case response time of a task τ_i denoted by R_i, there are at most $n_j(R_i)$ jobs of task τ_j, $j < i$, that can potentially preempt τ_i. Since each of these preemptions imply an interference of at most $\delta_{j,i}$ time units on the execution of τ_i, a straightforward way to compute the budget B_i assigned to each task τ_i to meet all its deadlines is

$$B_i \stackrel{\text{def}}{=} C_i + \sum_{j=1}^{i-1} n_j(R_i) \times \delta_{j,i} \tag{2}$$

For the budget assignment policy defined in Equation (2), Equation (3) gives an upper-bound $\text{PD}_{\text{bgt}}^{\max}(t)$ on the total CPU time that is reserved in any time interval $[0, t]$ to account for all the preemption delays.

$$\begin{aligned} \text{PD}_{\text{bgt}}^{\max}(t) &\stackrel{\text{def}}{=} \sum_{i=2}^{n} \left(n_i(t) \times \sum_{j=1}^{i-1} n_j(R_i) \times \delta_{j,i} \right) \\ &= \sum_{i=2}^{n} \left(n_i(t) \times \sum_{j=1}^{n} n_j(R_i) \times \delta_{j,i} \right) \end{aligned} \tag{3}$$

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

It is worth noticing that Equation (2) assigns the budget of task τ_i by looking at how many times τ_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 [9]. The authors defined the multi-set $M_{j,i}(t)$ as the set of all costs $\delta_{j,k}$ that tasks τ_j may induce in the execution requirements of all the tasks τ_k with a priority between that of τ_j and τ_i, in a time window of length t. A multi-set is a generalisation of the concept of "set" where the elements may be replicated (i.e. a multi-set may be for example $\{x, x, x, y, y\}$ whereas a regular 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 follow:

$$M_{j,i}(t) \overset{\text{def}}{=} \left(\uplus_{k=j+1}^{i-1} \uplus_{m=1}^{n_k(t)} \uplus_{\ell=1}^{n_j(R_k)} \delta_{j,k} \right) \uplus_{g=1}^{n_j(t)} \delta_{j,i} \qquad (4)$$

The operator $\uplus$ 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 $\{x, y\} \uplus \{x, y\} = \{x, x, y, y\}$ whereas for the set union the outcome is $\{x, y\} \cup \{x, y\} = \{x, y\}$.

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 τ_j on task τ_i in any time window of length t.

$$\Delta_{j,i}(t) \overset{\text{def}}{=} \sum_{\ell=1}^{q_{j,i}(t)} \overset{\ell}{\max}(M_{j,i}(t)) \qquad (5)$$

where

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

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

We show below that, considering the preemption delay from the perspective of the *preempting* 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

$$\sum_{j=1}^{n} \Delta_{j,i}(t) \leqslant \mathrm{PD}_{\mathrm{bgt}}^{\max}(t) \qquad (7)$$

Proof: From Equation (4) and since $\forall\ j \geqslant i$ 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:

$$\begin{aligned} \sum_{e \in M_{j,i}(t)} e &= \sum_{k=j+1}^{i} \sum_{m=1}^{n_k(t)} \sum_{\ell=1}^{n_j(R_k)} \delta_{j,k} \\ &= \sum_{k=j+1}^{i} n_k(t) \times n_j(R_k) \times \delta_{j,k} \qquad (8) \end{aligned}$$

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

Lemma 1: $\sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e \leqslant \mathrm{PD}_{\mathrm{bgt}}^{\max}(t)$

Proof: From Equation (8), we know that

$$\begin{aligned} \sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e &\leqslant \sum_{j=1}^{n} \sum_{k=2}^{n} n_k(t) \times n_j(R_k) \times \delta_{j,k} \\ &\leqslant \sum_{k=2}^{n} n_k(t) \times \sum_{j=1}^{n} n_j(R_k) \times \delta_{j,k} \\ &\overset{\text{from(3)}}{\leqslant} \mathrm{PD}_{\mathrm{bgt}}^{\max}(t) \end{aligned}$$

■

Lemma 2: $\sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e \geqslant \sum_{j=1}^{n} \Delta_{j,i}(t)$

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

$$\# M_{j,i}(t) = \left(\sum_{k=j+1}^{i-1} n_k(t) \times n_j(R_k) \right) + n_j(t) \qquad (9)$$

and since $n_j(R_k) \geqslant 1$, $\forall j, k \in [1, n]$, it holds that

$$\# M_{j,i}(t) \geqslant n_j(t) + \sum_{k=j+1}^{i-1} n_k(t) \qquad (10)$$

On the other hand, since when $j = k$ we have

$$\min(n_k(t), n_j(t)) = \min(n_j(t), n_j(t)) = n_j(t)$$

and thus this Equation (6) can be rewritten as:

$$\begin{aligned} q_{j,i}(t) &= n_j(t) + \sum_{k=j+1}^{i-1} \min(n_k(t), n_j(t)) \\ &\leqslant n_j(t) + \sum_{k=j+1}^{i-1} n_k(t) \qquad (11) \end{aligned}$$

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

$$q_{j,i}(t) \leqslant \# M_{j,i}(t) \qquad (12)$$

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

$$\sum_{e \in M_{j,i}(t)} e \geqslant \sum_{\ell=1}^{q_{i,j}(t)} \overset{\ell}{\max}(M_{j,i}(t)) \qquad (13)$$

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

$$\begin{aligned} \sum_{j=1}^{n} \sum_{e \in M_{j,i}(t)} e &\geqslant \sum_{j=1}^{n} \sum_{\ell=1}^{q_{i,j}(t)} \overset{\ell}{\max}(M_{j,i}(t)) \\ &\geqslant \sum_{j=1}^{n} \Delta_{j,i}(t) \end{aligned}$$

Hence the lemma follows. ■

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

$$\sum_{j=1}^{n} \Delta_{j,i}(t) \leqslant \mathrm{PD}_{\mathrm{bgt}}^{\max}(t)$$

□ ■

V. Proposed Budget Augmentation Framework

A. 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 τ_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 preempting task has to pay for the damage that it causes to the schedule*. According to Theorem 1, accounting for the preemption delay from the point of view of the preempting task enables a reduction on the over-provisioning of system resources. Formally, the execution budget B_i of each task τ_i is initially set to C_i and refilled according to the sporadic server definition. Then, each time a task τ_i resumes its execution after being preempted by other task(s), the remaining budget $B_i(t)$ of its associated server S_i is increased by $\sum_{\tau_j \in H(i)} \delta_{j,i}(t)$ (where $H(i)$ denotes the set of tasks that preempted τ_i) to compensate for the potential extra execution requirement that τ_i may incur.

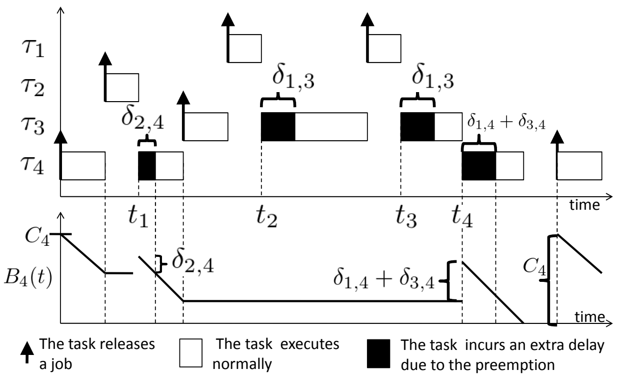


Fig. 1. Budget Augmentation Example

An example of the described framework is presented in Figure 1. In that example the task set contains 4 tasks. Task τ_4 is first preempted by a job from τ_2. When τ_4 resumes execution at time t_1, immediately after τ_2 terminates, its remaining budget $B_4(t_1)$ is incremented by $\delta_{2,4}$ units. Then, two jobs of τ_1 preempt both τ_3 and (indirectly) τ_4. Each time τ_3 resumes its execution at the return from the preemption (at time t_2 and t_3), the execution budget $B_3(t_2)$ and $B_3(t_3)$ is incremented by $\delta_{1,3}$. Finally, when τ_3 terminates its workload and τ_4 resumes at time t_4, $B_4(t_4)$ is incremented by $\delta_{1,4}+\delta_{3,4}$ as both τ_1 and τ_3 may have evicted some of its cached data; hence forcing τ_4 to reload it from the memory.

B. Schedulability Analysis

When the preemption delay is assumed to be zero (i.e., when the cache subsystem is partitioned for example), the authors of [16] proposed the following schedulability test to check at design-time, whether all the task deadlines are met at run-time.

Schedulability Test 1 (From [16]): A task set $\mathcal{T}$ is schedulable if, $\forall \tau_i \in \mathcal{T}$, $\exists t \in (0, D_i]$ such that

$$C_i + \sum_{j=1}^{i-1} \mathrm{rbf}(S_j, t) \leqslant t \tag{14}$$

where

$$\mathrm{rbf}(S_j, t) \overset{\mathrm{def}}{=} \left(\left\lfloor \frac{t}{T_j^s} \right\rfloor + 1 \right) \times B_j \tag{15}$$

Theorem 2 (from [3]): A periodic task-set that is schedulable with a task τ_i, is also schedulable if τ_i is replaced by a sporadic server with the same period and execution time

Proof: Follows from the proof of Theorem 2 in [3] ■

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 [16].

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

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_{cost} units of time, the time-penalty attached to the implementation of the proposed framework has an upper bound of

$$\mathrm{cost}(t) = q_{1,i}(t) \times F_{\mathrm{cost}} \tag{16}$$

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

Schedulability Test 2: A task set $\mathcal{T}$ is schedulable if, $\forall \tau_i \in \mathcal{T}$, $\exists t \in (0, D_i]$ such that

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

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

According to Schedulability Test 2 and as a consequence of Theorem 1, if we assume $\mathrm{cost}(t) = 0$ 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 τ_i may be potentially subject to (see Equation (2)). However, in a scenario where $\mathrm{cost}(t)$ is non-negligible the dominance relation does no loger hold.

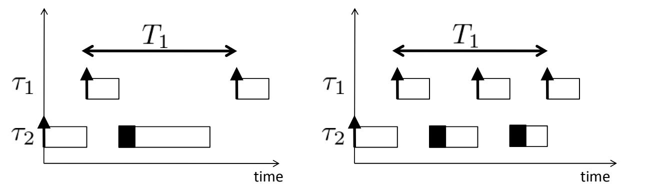


Fig. 2. Excessive Preemption Delay Due to Minimum Interarrival Time Violation

VI. Proposed Budget Donation Framework

The framework presented above is a combination of a reservation-based mechanism (budget initially assigned to each task) and a budget augmentation policy (budget inflated at return from preemption). This 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_i. 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 τ_1 releases a single job in a time interval of length T_1. This job executes for C_1 time units and task τ_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, τ_1 releases more than one job in the same time interval of length T_1 and τ_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 τ_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, we associate a second server Y_i to each task τ_i. This server Y_i has parameters $\langle Z_i, T_i^Y \rangle$ – Z_i is the budget and T_i^Y is the replenishment period. Unlike the server S_i, the budget Z_i is not consumed while τ_i is running. The purpose of this second server Y_i is to "pay" for the damage caused by τ_i in the system when τ_i preempts another task. That is, *each* task τ_j, when it is preempted by τ_i, obtains a budget donations by transferring some execution budget from the server Y_i to its execution budget B_j. These budgets Y_i impose a new condition to their associated task τ_i in order to accommodate for the minimum inter-arrival misbehavior: task τ_i is allowed to release a new job only if there is sufficient budget in Y_i. In this way the preemption delay that τ_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 τ_j, after being preempted by τ_i, requests a preemption delay compensation of $\delta_{i,j}$ from τ_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 [3] 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 $\tau_i \in \mathcal{T}$.

The purpose of each server Y_i is to ensure that new jobs of a task τ_i can only be released as long as the maximum preemption delay that τ_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 τ_j 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 τ_j can cause on the lower priority tasks by preempting them. It is straightforwardly defined as
$$P_j^{\max} \overset{\text{def}}{=} \sum_{k=j+1}^{n} \delta_{j,k} \tag{18}$$
If $Z_i(t) < P_j^{\max}$ then τ_j is not authorized to release a new job at time t and must wait until the earliest time instant $t' > t$ when $Z_j(t') \geqslant P_j^{\max}$.
2) Unlike the budget augmentation protocol proposed in the previous section, each time a task τ_i resumes its execution (say at time t) after being preempted (let $H(i)$ denote the set of tasks that preempted τ_i), τ_i 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 τ_i 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 τ_i can pay the required amount of time units, i.e., τ_i 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 τ_i 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, τ_j): each job of τ_j can preempt *at most* one job from τ_i and before releasing each of its jobs, τ_j makes sure that there is enough provision in its budget Y_j to compensate for the damage caused to the lower priority tasks, including τ_i.

2) From the point of view of the misbehaving task τ_i: 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 τ_i 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, τ_k): each job of τ_k may be preempted multiple times by the abnormal job release pattern of τ_i. However, upon each resumption of execution, τ_k 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 τ_i 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:

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

$$C_i + \text{cost}(t) + \sum_{j=1}^{i-1} [\text{rbf}(S_j, t) + \text{rbf}(Y_j, t)] \leqslant t \qquad (19)$$

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^Y is Z_i. This means that $\text{rbf}(Y_j, t)$ is an upper-bound on the budget used for execution by any task that was preempted by task τ_j 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_j server is left at the criteria of the system designer. However, in a given period T_i any task τ_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 $P_i^{\max}$. The simpler approach would be to define each server Y_i as $\langle Z_i = P_i^{\max}, T_i^Y = T_i \rangle$ as Y_i would have enough budget to compensate for all the interference that τ_i may cause in the schedule, assuming that the minimum inter-arrival constraint is not violated. However, the system designer may prefer $T_i^Y > T_i^S$ to provide more flexibility in case the task τ_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 (and always greater than the worst-case that had to be considered if a static number of jobs had to be enforced) since this depends on the number of lower priority jobs that it has actually preempted so far. This allows for a more dynamic system with overall better responsiveness.

A. Experiemental Results

In order to assess the validity of the contribution a set of experiments was conducted. 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 (UUniFast) [17].

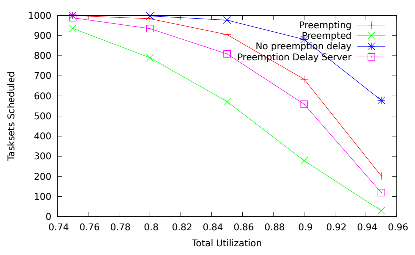
Task systems are randomly generated for every utilization step in the set $\{0.75, 0.8, 0.85, 0.9, 0.95\}$, their maximum execution requirements (C_i) were uniformly distributed in the interval $[20, 400]$. Knowing C_i and the task utilization U_i, T_i is obtained. At each utilization step 1000 task 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 ($D_i = T_i$).

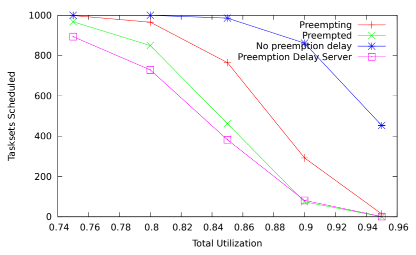
When each task is randomly generated preemption delay cost is obtained for each task pair in the task-set. The cache considered is composed of 10 cache lines. For each task a set of useful cache lines is computed, the usage of each cache line follows a uniform distribution. Similarly for each task a set of cache lines which are accessed during its execution is computed. The cardinality of interception of the useful set of τ_i with the accessed set of τ_j upper-bounds the maximum number of cache lines that τ_i has to refetch has a result of a preemption by τ_j.

The servers S_i have been attributed parameters $B_i = C_i$ and $T_i^S = T_i$. For the situation where the preemption delay server Y_j is put to use, its parameters are $Z_j = \sum_{j<k} \delta_{j,k}$ and $T_j^Y = T_j$.

The results are depicted in Figures 3(a) to 3(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 and there exists no other solution apart from this one 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 on average a higher scheduling performance. The schedulability reduction attached to the framework for misbehaving tasks


(a) 4 tasks


(b) 8 tasks

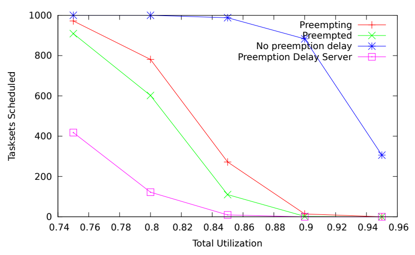

(c) 16 tasks

Fig. 3. Schedulability Comparison With CRPD

with respect to the minimum inter-arrival time is the price to pay for the added guarantees.

VII. Conclusion and Future Work

Reservation-based systems are one fundamental way to enforce temporal isolation in safety critical real-time systems. We show that, when preemption delay is present in the system, the state of the art mechanism of reservation-based systems induce 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 a provably reduction on the budget overprovisioning in platforms with non-negligible preemption delay overheads. 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).

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.

Acknowledgements

This work was partially supported by National Funds through FCT (Portuguese Foundation for Science and Technology) and by ERDF (European Regional Development Fund) through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project(s) FCOMP-01-0124-FEDER-037281 (CISTER), ARTEMIS/0003/2012, JU grant nr. 333053 (CONCERTO), by the North Portugal Regional Operational Programme (ON.2 O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF), and by National Funds through FCT (Portuguese Foundation for Science and Technology), within project ref. NORTE-07-0124-FEDER-000063 (BEST-CASE, New Frontiers), by FCT (Portuguese Foundation for Science and Technology) and by ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH/BD/81085/2011

References

[1] L. Abeni and G. Buttazzo, "Resource reservation in dynamic real-time systems," *Real-Time Systems*, vol. 27, pp. 123–167, 2004. 10.1023/B:TIME.0000027934.77900.22.

[2] C. Lin, T. Kaldewey, A. Povzner, and S. A. Brandt, "Diverse soft real-time processing in an integrated system," in *27th RTSS*, pp. 369–378, Dec. 2006.

[3] B. Sprunt, L. Sha, and J. Lehoczky, "Aperiodic task scheduling for hard-real-time systems," *Real-Time Systems*, vol. 1, pp. 27–60, 1989. 10.1007/BF02341920.

[4] L. Nogueira and L. Pinho, "Shared resources and precedence constraints with capacity sharing and stealing," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, April 2008.

[5] C. Lin and S. Brandt, "Improving soft real-time performance through better slack reclaiming," in *26th RTSS*, pp. 12 pp.–421, Dec. 2005.

[6] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Analysis of cache-related preemption delay in fixed-priority preemptive scheduling," *IEEE Transactions on Computers*, vol. 47, pp. 700–713, 1998.

[7] S. Altmeyer and C. Burguiere, "A new notion of useful cache block to improve the bounds of cache-related preemption delay," in *21th ECRTS*, 2009.

[8] S. Altmeyer and G. Gebhard, "WCET analysis for preemptive scheduling," in *8th WCET*, 2008. Austrian Computer Society (OCG),.

[9] J. Staschulat, S. Schliecker, and R. Ernst, "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay," in *17th ECRTS*, pp. 41–48, July 2005.

[10] L. Ju, S. Chakraborty, and A. Roychoudhury, "Accounting for cache-related preemption delay in dynamic priority schedulability analysis," in *DATE 2007*, pp. 1–6, April 2007.

[11] B. Bui, M. Caccamo, L. Sha, and J. Martinez, "Impact of cache partitioning on multi-tasking real time embedded systems," in *Embedded and Real-Time Computing Systems and Applications, 2008. RTCSA '08. 14th IEEE International Conference on*, pp. 101 –110, aug. 2008.

[12] J. Whitham and N. Audsley, "Explicit reservation of local memory in a predictable, preemptive multitasking real-time system," in *RTAS 2012*, pp. 3–12, 2012.

[13] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo, "Cache-aware scheduling with limited preemptions," tech. rep., SSSUP, Pisa, Italy, 2010. http://feanor.sssup.it/∼marko/LP RTSS09.pdf accessed on 10th of February, 2010.

[14] H. Ramaprasad and F. Mueller, "Bounding preemption delay within data cache reference patterns for real-time tasks," in *12th RTAS*, pp. 71–80, April 2006.

[15] C.-G. Lee, K. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, "Bounding cache-related preemption delay for real-time systems," *Software Engineering, IEEE Transactions on*, vol. 27, pp. 805–826, Sep 2001.

[16] J. Lehoczky, L. Sha, and Y. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Real Time Systems Symposium, 1989., Proceedings.*, pp. 166 –171, dec 1989.

[17] E. Bini and G. Buttazzo, "Biasing effects in schedulability measures," in *ECRTS 2004*, Jun 2004.