CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Technical Report

Response time analysis of hard real-time tasks with STM transactions on multi-core platforms

António Barros
Patrick Meumeu Yomsi
Luis Miguel Pinho

CISTER-TR-150501

2015/05/01

Response time analysis of hard real-time tasks with STM transactions on multi-core platforms

António Barros, Patrick Meumeu Yomsi, Luis Miguel Pinho

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: +351.22.8321159

E-mail: amb@isep.ipp.pt, pamyo@isep.ipp.pt, lmp@isep.ipp.pt

http://www.cister.isep.ipp.pt

Abstract

Recent embedded processor architectures integrating multiple heterogeneous cores and non-coherent caches renewed attention to the use of Software Transactional Memory (STM) as a building block for developing parallelapplications. STM promises to ease concurrent and parallel software development, but relies on the possibility of abort conflicting transactions to maintain data consistency, which in turns affects the execution time of tasks carrying transactions. The possibility of a transaction abort-and-repeat incurs execution time overheads that have to be accounted for in the WCET of the task that executes the transaction.

In this paper we formalise a response time analysis for sets of non-independent tasks that use STM to share data, and in which transactions are scheduled by following the non-preemptive approaches NPDA, NPUC and the fully preemptive SRP-TM.

Response time analysis for hard real-time tasks with STM transactions on multi-core platforms

António Barros, Patrick Meumeu Yomsi and Luís Miguel Pinho

May 1, 2015

Abstract

Recent embedded processor architectures integrating multiple heterogeneous cores and non-coherent caches renewed attention to the use of Software Transactional Memory (STM) as a building block for developing parallel applications. STM promises to ease concurrent and parallel software development, but relies on the possibility of abort conflicting transactions to maintain data consistency, which in turns affects the execution time of tasks carrying transactions. The possibility of a transaction abort-and-repeat incurs execution time overheads that have to be accounted for in the WCET of the task that executes the transaction. In this paper we formalise a response time analysis for sets of non-independent tasks that use STM to share data, and in which transactions are scheduled by following the non-preemptive approaches NPDA, NPUC and the fully preemptive SRP-TM.

1 Introduction

Recent proposed architectures for embedded systems include tens and hundreds of cores, as the way to increase the processing power within the semiconductor thermal limits. Such architectures provide true application parallelism, which influences substantially in the way applications share data. Traditional lock-based synchronisation solutions do not cope with the ever increasing degree of parallelism, as coarse-grained locks serialise non-conflicting operations that could progress in parallel, degrading the system throughput, while fine-grained locks increase the complexity of system development, degrading the system composability.

The software transactional memory (STM) [1] is a concept in which a critical section – also referred to as the *transaction* – executes speculatively in isolation, without blocking, independently from other parallel transactions. An optimistic concurrency control mechanism is responsible for serialising concurrent transactions, maintaining the consistency of shared data objects. Conflicts are solved by applying a contention policy that selects the transaction that will commit, while the contenders will most likely abort and repeat. The execution time overhead resulting from aborts affects the response time of a task that executes a transaction.

This research: In this report, we formalise the response time analysis for sets of non-independent tasks that share data by means of STM, scheduled by following the non-preemptive approaches NPDA and NPUC, and SRP-TM[1]. The

[1]SRP-TM is a fully-preemptive scheduling strategy, based on the Stack Resource Protocol (SRP).

three aforementioned approaches extend the partitioned EDF (P-EDF) algorithm whenever a transaction is in progress. Furthermore, we assume that transactions are serialised by the STM system by chronological order of arrival.

Paper structure: The report is structured as follows. Section 2 sets the system model and the assumptions made in this work. Section 3 briefly summarizes the scheduling approaches that are subject to the response time analysis in this report: NPDA, NPUC and SRP-TM. The response time analysis for these three scheduling approaches are given in Section 4 (NPDA), Section 5 (NPUC) and Section 6 (SRP-TM).

2 System model

Task specification. We assume that the workload is carried by a set of n periodic tasks $\tau \stackrel{\text{def}}{=} \{\tau_1, \ldots, \tau_n\}$. Each task τ_i releases a potentially infinite number of jobs and is characterised by a worst-case execution time C_i, a relative deadline D_i, and a period T_i. These parameters are given with the following interpretation. The j^{th} job of task τ_i executing on processor π_k, referred to as $\tau_{i,j}^k$, is characterised by its release time $r_{i,j}$ such that $r_{i,j+1} \stackrel{\text{def}}{=} r_{i,j} + T_i, \quad \forall i \in \{1, \ldots, n\}, \forall j \geq 1$; and an absolute deadline $d_{i,j} \stackrel{\text{def}}{=} r_{i,j} + D_i$. We assume the system to be constrained-deadline, so $D_i \leq T_i, \forall i$ which imposes that each job must finish before the following job is released, so no consecutive jobs can overlap in time. We also assume that the system is synchronous, i.e., all tasks in τ release their first jobs at the same time instant, say $t = 0$. Formally, we assume that $r_{i,1} = 0, \forall i$. For such a system, interval $[0, P)$, where P is the hyper-period ($P = \text{lcm}(T_1, \ldots, T_n)$ – the least common multiple of the periods of all tasks), is a feasibility interval [3].

Tasks may not be independent, meaning that they may concurrently access common data located in shared memory. Accesses to these shared data are performed in the context of a *STM transaction* [4]. The outcome of a transaction must appear as if the operations that constitute it were performed atomically, isolated from the interference of concurrent jobs, i.e. as if performed in mutual exclusion. All reads and updates must be performed over a single state of the subset of STM objects that are accessed by the transaction: non intermediate concurrent updates on this subset can occur during the execution of the transaction, otherwise the results would be inconsistent. The mission of an STM system is to maintain the consistency of the share data, validating the outcome of transactions. If the outcome of a transaction results in an inconsistent state, then the transaction *aborts*; otherwise, the transaction *commits*.

We assume that each task τ_i performs *at most* one STM transaction, denoted as ω_i. Nevertheless, the results obtained throughout this paper are extensible to tasks that execute multiple non-nested transactions with minor efforts. The transaction of τ_i is characterised by:

- C_{ω_i}: the maximum time required to execute the sequential code of ω_i once, without any external interference from other tasks or the system itself, and try to commit.
- **The data set (DS_i)**: the collection of shared objects that are accessed by ω_i. This data set can be partitioned in two subsets: the *read set* (RS_i) and the *write set* (WS_i) where:

- RS_i is the subset of objects that are accessed by ω_i solely for reading, and
- WS_i is the subset of objects that are modified by ω_i during its execution.

The size of the data set, read set and write set are denoted as $|DS_i|$, $|RS_i|$ and $|WS_i|$, respectively.

Platform and Scheduler specifications. We assume that all the jobs are executed on a multi-core platform $\pi \stackrel{\text{def}}{=} \{\pi_1, \ldots, \pi_m\}$ composed of m homogeneous cores, i.e., all cores have the same computing capabilities and are interchangeable. The task set τ is scheduled by following a policy based on the *partitioned Earliest Deadline First* (P-EDF) scheduler, i.e., each task is statically assigned to a specific core at design time (and task migrations from one core to another at run-time are not allowed) and each core schedules its subset of tasks at run-time by following the classical EDF scheduler, where at each time instant the job with the earliest absolute deadline is selected for execution. Ties are broken in an arbitrary manner.

We define σ as a function that returns the core to which a task is assigned. Thus, if task τ_i is assigned to core π_k, then $\sigma(\tau_i) = \pi_k$.

STM specification. We assume that the STM manages a collection of p objects $O \stackrel{\text{def}}{=} \{o_1, \ldots, o_p\}$ are located at the globally shared memory and are accessible to all tasks carrying a transaction, independently from the core on which they are executing. Multiple simultaneous transactions are supported and for each object there is a chronologically ordered list that records all transactions currently accessing the object. The number of transactions that have a specific object, say o_j, in their data sets is denoted has $|o_j|$.

Contention occurs when two or more transactions, executing in parallel, have intersecting data sets and at least one transaction modifies the value of a shared object. Hence, we can map contentions by using a graph G in which *vertices* represent transactions and *edges* represent the shared objects between a pair of transactions.

Definition 1 (Contention group)**.** Given a contention graph G, a *contention group*, denoted as Ω_k (with $k \geq 1$), is defined as a set of connected vertices of G in which any two transactions are connected by a path.

Figure 1 illustrates a very simple example of contention groups in which a set of five transactions $\{\omega_1, \omega_2, \omega_3, \omega_4, \omega_5\}$ are sharing a set of three STM objects $\{o_1, o_2, o_3\}$.

In this example, the data sets of the transactions form two distinct contention groups: $\Omega_1 = \{\omega_1, \omega_5\}$ and $\Omega_2 = \{\omega_2, \omega_3, \omega_4\}$.

Each instance of a transaction has a life cycle that follows the states represented in Figure 2. When a transaction starts, it enters in the active state in which the transaction code executed. At the end of code execution, the STM system checks if the transaction is sound, in terms of data consistency and current data access conflicts. If the transaction is sound, then it commits and concludes, otherwise the transaction enters in the failed state, in order to restart. During the active state, the data accessed by a transaction can become inconsistent when a contender commits, and the transaction enters the zombie mode, in which it continues to execute, but it will inevitably fail the final validation, transiting to the failed state. A transaction may be aborted multiple times until it successfully commits.

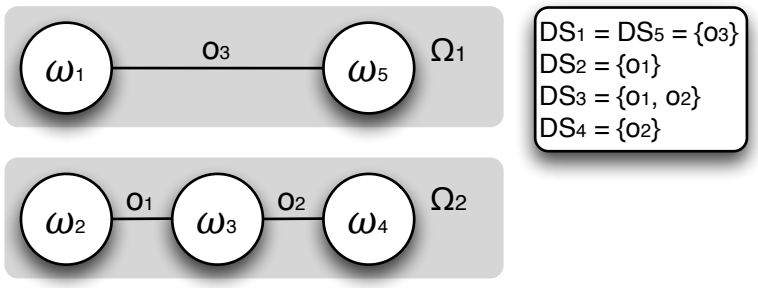

Figure 1: Transaction dependencies by object concurrency.

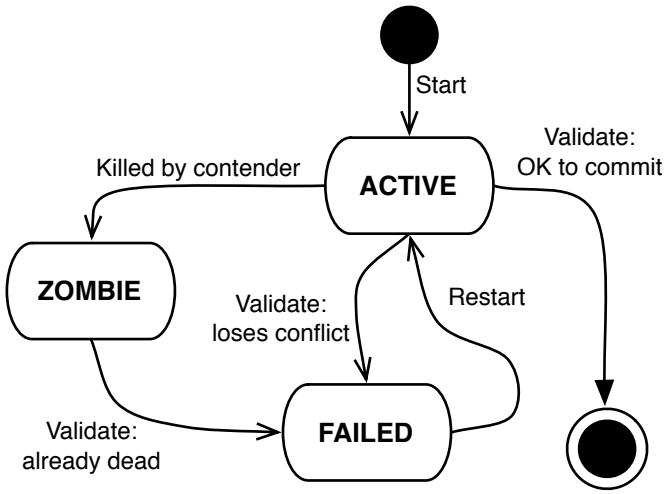

Figure 2: State diagram of a transaction.

Definition 2 (Cores allocated to a contention group)**.** Given a contention group Ω_a, then Π_a is the set of m_a cores allocated to transactions of Ω_a. Formally, $\Pi_a = \{\pi_k \mid \sigma(\omega_i) = \pi_k,\ \omega_i \in \Omega_a\}$ The size of this set is $m_a \leq m$.

Definition 3 (Direct contender of a transaction)**.** The *direct contender of a transaction* ω_i is defined as a transaction ω_j that shares at least one STM object with ω_i. Formally, that is $DS_i \cap DS_j \neq \emptyset$.

Definition 4 (Indirect contender of a transaction)**.** The *indirect contender of a transaction* ω_i is defined as a transaction ω_j that does not share any STM object with ω_i, but belongs to the same contention group as ω_i.

Definition 5 (Independent transactions)**.** Two transactions ω_i and ω_j are said to be *independent* when they belong to different contention groups.

Definition 6 (Transaction overhead of a job)**.** The *transaction overhead of job* $\tau_{i,j}$, denoted as $W_{i,j}$, is defined as the time wasted in executing aborted commit

attempts of ω_i. Formally, $W_{i,j}$ is given by:

$$W_{i,j} \stackrel{\text{def}}{=} A_{i,j} \cdot C_{\omega_i} \tag{1}$$

In Equation 1, $A_{i,j}$ represents the number of failed attempts of ω_i before it commits.

Definition 7 (Execution time of a job executing a transaction)**.** Given task τ_i executing a transaction ω_i, the *execution time of the j^{th} job*, denoted as $C_{i,j}$, is defined as the sum of four terms: (1) the time ($C_{\text{a-}\omega_i}$) required to execute the code of τ_i *before* ω_i starts, (2) the time ($C_{\text{p-}\omega_i}$) required to execute the code of τ_i after ω_i has committed, (3) the execution time (C_{ω_i}) of a successful transaction of ω_i, and (4) the transaction overhead ($W_{i,j}$) of that job. Formally, $C_{i,j}$ is given by:

$$C_{i,j} \stackrel{\text{def}}{=} C_{\text{a-}\omega_i} + C_{\omega_i} + C_{\text{p-}\omega_i} + W_{i,j} \tag{2}$$

Definition 8 (Task utilisation)**.** The *utilisation of task τ_i*, denoted as U_i, is defined as the execution ratio of the jobs of τ_i within one hyper-period. Formally:

$$U_i = \frac{1}{P/T_i} \cdot \sum_{j=1}^{P/T_i} \frac{C_{i,j}}{T_i} \tag{3}$$

Definition 9 (System utilisation)**.** The *utilisation of system τ*, denoted as U_s, is defined as the sum of the utilisations of all tasks in τ. Formally:

$$U_S = \sum_{i=1}^{n} U_i \tag{4}$$

3 Scheduling protocols description

Barros and Pinho [5] defended the idea that a STM contention management policy based on serialising transactions by their arrival order (FIFO) was an effective approach to avoid transaction starvation by some innate characteristic and, as consequence, achieve predictability on the expected time to commit. To achieve this, they defined the contention manager FIFO-CRT which selects the current running transaction with the earliest release timestamp upon the occurrence of a conflict. This selected transaction will thus be the first to commit.

In the same paper, it was shown how preemptions could have a negative impact on the contention management policy. To mitigate this problem, two scheduling variants based on the preemptive P-EDF called *Non-Preemptive During Attempt* (NPDA) and *Non-Premptive Until Commit* (NPUC) were proposed. For these scheduling variants, preemptions were temporarily disabled during the execution of a transaction.

Later, a fully preemptive approach based on the Stack Resource Policy (SRP) [2], was devised to incorporate the predictability of NPUC and the reponsiveness of NPDA. For this reason, this approach is called *SRP-based scheduling of Transactional Memory atomic sections* (SRP-TM).

For the convenience of the reader, the three approaches will be now summarised.

NPDA. In this approach, preemptions are disabled during the execution of the transactional section. However, when a transaction tries to commit and fails, preemptions are temporarily enabled again, allowing the scheduler to schedule pending jobs with earlier deadlines before the transactional section is retried. This approach allows blocking of higher priority jobs, but it is limited by the execution time of the longest transactional section of lower priority tasks, on each core. However, it allows multiple simultaneous transactions in progress on the same core, which complicates the timing analysis of transactions.

NPUC. In this approach, preemptions are simply disabled from the moment a job starts executing a transaction until the transaction commits. This approach has a worse impact on the responsiveness of higher priority tasks, because the blocking can extend to the longest time of a transaction to commit. However, since transactions cannot be preempted, then the time to commit of each transaction depends exclusively of the transactions that are in progress at the moment the transaction starts.

SRP-TM. This fully preemptive approach was devised to incorporate the predictability of NPUC and the reponsiveness of NPDA. In this approach, preemption levels are assigned to tasks in the reverse order of relative deadlines, such that $\lambda_i < \lambda_j$ iff $D_i > D_j$. In addition, each transaction ω_i is also assigned a preemption level λ_{ω_i} that is, by definition, the highest preemption level from all tasks with transactions that may have its progress delayed by the transaction. At every time, each core maintains a core ceiling that is the highest preemption level of a task that may have its progress delayed by the current transaction in progress. This approach does not allow multiple simultaneous transactions in progress on the same core.

When a job starts a transaction, the core ceiling is set to the preemption level of the transaction in progress, and can only be preempted by jobs with earlier deadlines, preemption levels higher than the core ceiling and without transactions. If a job with the earliest deadline and higher preemption level than the core ceiling is prevented to execute because it has a transaction, then this job rises the core ceiling to its preemption level and the scheduler forces the job with the current transaction in progress to execute, so the blocked job can be scheduled as soon as possible. When a transaction commits, the core ceiling is reset to zero, and tasks are again scheduled by following the classical EDF scheduler.

4 Response time analysis under NPDA

We recall that NPDA is a scheduling approach in which jobs are scheduled under P-EDF, but the transactional sections of jobs are protected from preemptions. However, when a transaction fails to commit, preemptions are enabled and the scheduler is allowed to schedule ready jobs with earlier deadlines.

NPDA does not restrict the number of transactions that can be simultaneously in progress in the same core, so the contention manager FIFO-CRT permits a committing transaction to abort a concurrent transaction with an earlier release time in a preempted job. On the one hand, the decision of the contention manager is determined exclusively by the concurrent transactions that are running at the time

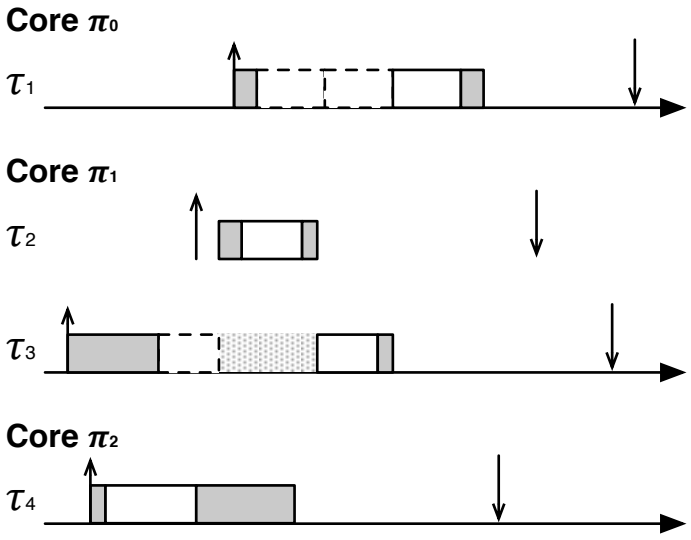

Figure 3: Transactions scheduled under NPDA.

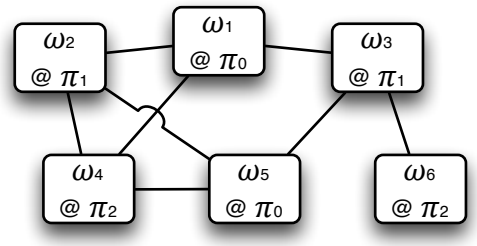

Figure 4: Graph representing transactions with intersecting data sets.

one transaction tries to commit. But, on the other hand, having multiple transactions in progress on the same core increases the complexity of determining an upper-bound on the number of times a transaction is aborted. Figure 3 illustrates how transaction ω_1 on core π_0 is aborted two times due to two different transactions (ω_2 and ω_3) executing on the same core π_1. The most extreme case is the scenario in which a transaction aborts because of all other concurrent transactions in the same contention group execute on other cores, before they eventually commit.

Figure 4 illustrates a set of transactions that belong to the same contention group, and their respective dependencies. Each vertex represents a transaction and, in addition, it is indicated the core allocated to the transaction. Vertices connected by edges mean that the pair of transactions share transactional data and execute in different cores, so there is a probability of conflicts between them (direct contenders).

In this example, the contention group covers three cores: π_0, π_1 and π_2. Transaction ω_1 is executed on core π_0, so it can abort in favour of transactions released earlier and executing on cores other than π_0. Since NPDA allows multiple transactions in progress simultaneously in the same core, one instance of ω_1 can abort

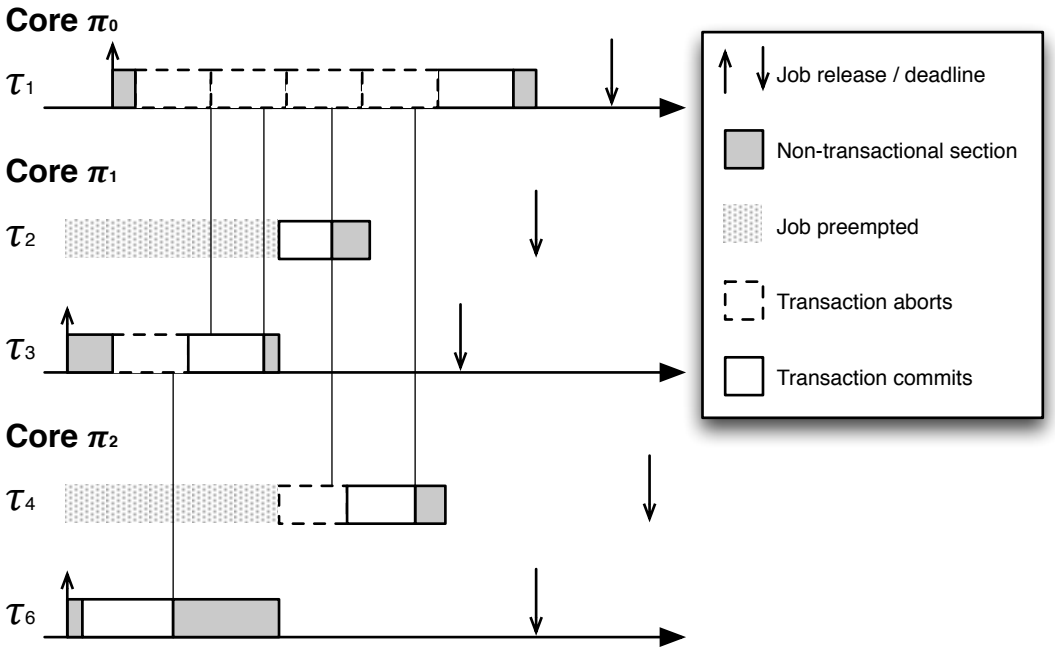

Figure 5: Preempted jobs of tasks τ_2 and τ_4 are rescheduled in precise times to abort transaction of job τ_1.

because of transactions ω_2, ω_3, ω_4 and ω_6, until eventually commits.

Figure 5 illustrates one case in which transaction ω_1 starts when all the four mentioned contenders are already in progress. Jobs of tasks τ_2 and τ_4 are preempted, but they have transactions that have not committed yet. Thin vertical lines indicate time instants at which a transaction is aborted or doomed to abort (turned into zombie); the transaction that aborts and the transaction that causes the abort are found on the extremes of each line.

Jobs of tasks τ_3 an τ_6 started their transactions and the first to commit is ω_6. That invalidates the first attempt of ω_3 and forces it to repeat; meanwhile, the first attempt of ω_1 to commit conflicts with the second attempt of ω_3 and aborts. Then, ω_1 gets successively aborted by ω_3, ω_2 and ω_4, until it finally commits. In the end, ω_1 had to wait for all its direct contenders (i.e. ω_2, ω_3 and ω_4) to commit, and that included waiting for the indirect contender (i.e. ω_6) to commit too.

Notice that in this example, the job of τ_1 never gets preempted. However, even if it suffered preemptions, it could be possible to find contrived cases in which the attempts of ω_1 coincide with the execution of its contenders.

While identifying the direct contenders that can delay a transaction to commit is a trivial task, matching the indirect contenders with direct contenders to find the sequence of transactions that produce the longest delay becomes a non-trivial exercise when the number of transactions (vertices), dependencies (edges) and allocated cores in the contention group grow. Therefore, a tight feasibility analysis for NPDA becomes computationally hard when the size of the problem grows.

Returning to the contention group as illustrated in Figure 4, it is easy to determine that the direct contenders of transaction ω_3 form the subset $\{\omega_1, \omega_5, \omega_6\}$,

and the unique indirect contender is ω_4, either via transaction ω_1 or transaction ω_5. In this very simple example, it would be easy to determine if ω_4 produces the longest delay on ω_3 through ω_1 or ω_5 (there are only two possible cases). But if the number of indirect contenders rises, as well as the depth of indirect contenders (if the group was allocated more cores), then the number of possible combinations to analyse would grow very fast.

A reduction of the complexity of the feasibility analysis could be achieved, considering all possible indirect contenders for each direct contender and, in the end, sum all the individual contributions of the direct contenders for the commit delay. However, the contribution of indirect contenders could be multiplied, if they were considered in the calculations of multiple direct contender contributions, and the results would be too pessimistic and far from the actual worst-case. Again, looking at the example of Figure 4, the effect of transaction ω_4 would be accounted twice in the calculation of the delay of transaction ω_3: one time in the contribution of ω_1 and a second time in the contribution of ω_5.

The high order of complexity to produce a useful upper bound that permits to infer about the feasibility of a task set scheduled under NPDA led to more deterministic approaches like NPUC and SRP-TM.

5 Response time analysis under NPUC

We recall that by following NPUC, once a job starts a transaction, it will execute free of preemptions until the transaction commits. This approach ensures two important predicates:

- At most one transaction can be in progress on a core, and
- The delay experienced by a transaction to commit depends exclusively on its direct contenders that, in their turn, depend on their own direct contenders.

As such, bounding the delay of a given transaction can be achieved by determining the sequence of transactions that will produce the longest delay. The longest delay is produced when the transaction under analysis is released when its contenders have their maximum workload to execute, i.e. when their release times are as closer to the release time of the transaction as possible. For the sake of simplicity, it will be assumed that the transaction under analysis is released one instant immediately after all of its contenders in progress, in the following analysis methods.

Determining an upper bound on the response time of a transaction under NPUC

This section discusses two methods to determine an upper bound on the delay a transaction can suffer before it commits. The first method provides a tight bound, but involves determining all possible sequences of transactions that produce a delay on the transaction under analysis. The second method provides a more pessimistic upper bound, but with a linear complexity.

Method 1: Exact determination of an upper bound on the response time of a transaction. A graph as the one in Figure 4, in which the vertices represent transactions, connected by edges when the data sets of a pair of

transactions (executing in different cores) intersect, is useful for finding the possible sequences of transactions, and to determine which sequence produces the longest delay. A sequence of transactions is determined by considering a *simple path*[2] converging to the vertex that represents the transaction under analysis, without including multiple transactions allocated to the same core. Therefore, if a contention group Ω_a covers, say m_a cores, then the maximum length of a sequence will be m_a vertices.

For a given sequence of transactions, the delay until the transaction commits can be upper bounded by computing the maximum times taken by each transaction in the sequence to commit, by chronological order. That is, from the furthest vertex in the path (first transaction) to the vertex of the transaction under analysis (last transaction).

Figure 6 depicts an example in which transactions ω_6, ω_3 and ω_1 start executing in this order. The time intervals between consecutive transaction releases are minimal, so these three transactions are released almost simultaneously. In this example, ω_6 is able to commit at the first attempt, which sets ω_3 as zombie in its first attempt. Afterwards, ω_3 commits at the second attempt, leaving ω_1 without contenders with earlier release times. Finally, ω_1 commits as it has the earliest release time. Since ω_1 is shorter than ω_3, it requires 5 attempts to finally commit. In order to formalise an analysis on the time required to commit a transaction, two points must be noted.

First, *each transaction can abort at most one more time after the previous transaction has committed*. The maximum time taken by a transaction in aborted attempts is never less than the time taken by the previous transaction in the sequence to commit. If two consecutive transactions in a sequence start immediately one after the other, then the earlier transaction will commit first and only then the later transaction will be free to commit, aborting at most one more time after the previous transaction committed, and before committing in the following attempt.

Second, *the first transaction in the sequence cannot start after the transaction under analysis*. In contrast to FIFO-CRT, at the time the transaction in analysis starts, the first transaction in the sequence may be already executing. Let us assume that at the moment the last transaction in the sequence starts, the first transaction in the sequence was executing long enough to have been set as zombie by an earlier transaction (that does not belong to the sequence, because it executed earlier on the same core as one of the transactions in the sequence) before the transaction under analysis started. In this case, this doomed attempt will prolong the delay of the second transaction in the sequence. This delay will cascade along the sequence of transactions, affecting the delay of the last transaction in the sequence. Although in this example the first transaction in the sequence (i.e. ω_6) requires only one attempt to commit, a generic worst case analysis requires to account for two attempts for the first transaction in the sequence.

With these two points put together, it is possible to formulate an upper bound on the time required to commit the last transaction in a given sequence, under NPUC.

Formally, let υ be defined as the function that returns the transaction on a determined position of a given sequence of transactions. Thus, if transaction ω_i is

[2] A simple path of a graph is a sequence of connected vertices in which no vertices nor edges are repeated.

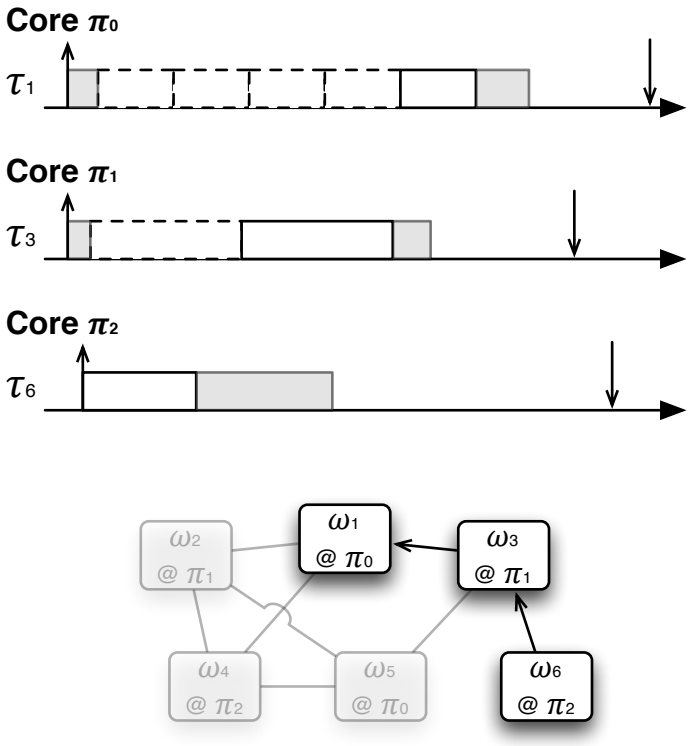

Figure 6: Sequence of transactions until ω_1 commits.

on the q^{th} position in a sequence, then $\upsilon(q) = \omega_i$. In a sequence with k transactions, an upper bound on the time required to transaction in the q^{th} position ($1 \leq q \leq k$) to commit can be computed by Equation 5.

$$\begin{cases} R_1 = 2 \cdot C_{\upsilon(1)} \\ \\ R_q = \left(\left\lceil \frac{R_{q-1}}{C_{\upsilon(q)}} \right\rceil + 1 \right) \cdot C_{\upsilon(q)} \quad \text{if } 1 < q \leq k. \end{cases} \tag{5}$$

Equation 5 computes an upper bound on the time to commit $\upsilon(q)$, the q^{th} transaction in the sequence, denoted as R_q, by considering the time to commit the previous transactions in the sequence. The first transaction in the sequence ($q = 1$) takes the time to execute at most two attempts. Subsequent transactions ($q > 1$) take the time for the previous transaction in the sequence to commit plus an additional attempt that finally commits. The time to commit the transaction under analysis, denoted as R_k, is determined for $q = k$.

At this point, an upper bound on the response time of a transaction under NPUC can be determined. Let us assume that $\mathcal{S}_i = \{S_{i,1}, S_{i,2}, \ldots S_{i,g}\}$ is the set of all g longest possible simple paths (i.e. sequences of transactions) without multiple transactions allocated to the same core, that converge to the vertex of transaction ω_i. Each path $S_{i,j}$ has a length of k_j vertices and produces an upper bound R_{k_j} for

ω_i to commit. An upper bound on the response time of this transaction, denoted as R_{ω_i}, is given by the maximum value of R_{k_j} from all possible paths in $\mathcal{S}_i$:

$$R_{\omega_i} = \max\{R_{\omega_i,k_j} \mid 1 \leq j \leq g\} \tag{6}$$

Although very precise, this method assumes a brute force approach to calculate the result for every possible path. It has combinational order of complexity and becomes impractical when the number of transactions and the number of cores in the system increase. This is because the number of possible sequences of transactions increases dramatically.

Method 2: Linear complexity estimation of an upper bound on the response time of a transaction. The difficulty to determine tight upper bounds on the response times of transactions using the previous method arises from an increase in the number of transactions and cores, as this in turn increase the number of possible transaction sequences in a combinatorial manner. Pessimistic methods that can produce results in practical computational time are possible.

To this end achievement, this second method also takes into consideration the two points noted as relevant for the previous method: a transaction can abort at most once after the previous transaction in the sequence committed, and the first transaction in the sequence may execute two attempts in the worst case. This method does not try to determine the sequence that produces the longest delay, as this is the source of the high complexity of the previous method. Instead, the main idea is to upper bound the delay caused on one transaction on a given core, by the execution times of the longest transactions in the same contention group executing on other cores, without considering a specific sequence. This delay is, in fact, the inter-core interference caused by higher-priority transactions executing on other cores.

This method is formalised as follows. Let us assume that a contention group Ω_a is allocated to a set of m_a cores, denoted as Π_a (see Definition 2). First, for each core π_ℓ allocated to the contention group is determined C_{Ω_a,π_ℓ}, the execution time of the longest transaction of the contention group that executes on that core, see Equation 7.

$$C_{\Omega_a,\pi_\ell} = \max\{C_{\omega_i} \mid \omega_i \in \Omega_a \ \wedge \ \sigma(\tau_i) = \pi_\ell\} \tag{7}$$

Then, an upper bound on the delay caused to any transaction of Ω_a on a given core π_k by concurrent transactions on other cores, denoted as I_{Ω_a,π_k}, is the sum of two execution times of the longest transaction on every other core, as formalised in Equation 8.

$$I_{\Omega_a,\pi_k} = \sum_{\pi_\ell \in \Pi_a \setminus \pi_k} 2 \cdot C_{\Omega_a,\pi_\ell} \tag{8}$$

This result accounts, for each core, the attempt in which the transaction aborts and the following attempt in which finally commits.

The result is common for all transactions of Ω_a allocated to core π_k, so it is computed only once for each core/contention group. The response time of a transaction ω_i allocated to core π_k and belonging to contention group Ω_a can be upper bounded by Equation 9, in which the execution time of two attempts of ω_i (the first aborts and the second commits) is added to the previously determined delay.

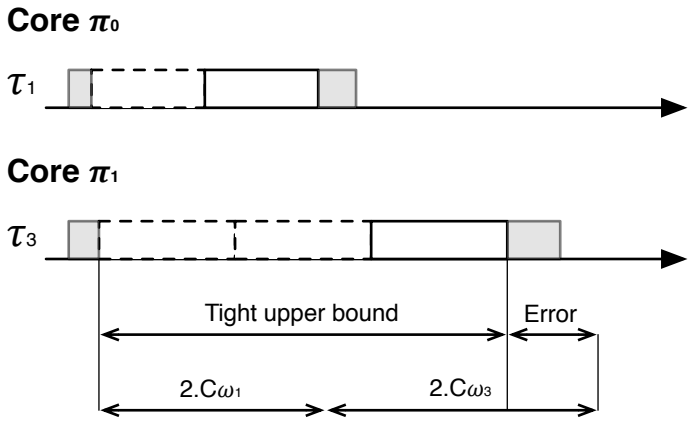

Figure 7: Error due to computation without considering overlapping concurrent attempts.

$$R_{\omega_i} = 2 \cdot C_{\omega_i} + I_{\Omega_a, \pi_k} \tag{9}$$

This method result is more pessimistic than the one provided by Method 1.

First, the transaction that demands the longest execution time is assumed on each core. Although this provides the largest accountable delay for a given core, the set of transactions selected for the computation may not form a simple path in the graph. Therefore, a practical sequence of conflicts with such transactions might be impossible. Considering the example illustrated in Figure 4, the delay caused on a transaction on core π_0 could account for transactions ω_2 and ω_6 on cores π_1 and π_2, respectively, even though they do not conflict, and are not part of any simple path converging to transactions ω_1 and ω_5.

Second, the computation of the response time of the transaction does not consider that the last attempt of a transaction (in which it commits) and the last aborted attempt of the next transaction in the sequence, may overlap in time. Figure 7 illustrates the error added to the result of this method, as compared with the tighter upper bound determined by Method 1. In this example, the last attempt of transaction ω_1 overlaps with a non-negligible portion of the last aborted attempt of ω_3. This overlapping is accounted for in Method 1, but is neglected in Method2. This has a negative impact on the precision of the upper bound. This error increases with the number of cores allocated to the contention group, because it is cumulative for each transaction in the sequence. However, Method 2 is capable of providing an upper bound on the response time of a transaction under NPUC in a reasonable amount of time in practice.

Response time of a task under NPUC

With an upper bound on the response time of a transaction under NPUC computed, it becomes possible to establish a formal response time analysis for tasks scheduled under NPUC. In the system model (see Section 2), a task carrying a transaction can be divided into three sections to be enumerated:

1. a$-\omega_i$, the section executing before the transaction (with execution time $C_{\mathrm{a}-\omega_i}$),
2. ω_i, the transaction, and
3. p$-\omega_i$, the section executing after the transaction (with execution time $C_{\mathrm{p}-\omega_i}$).

An upper bound on the response time of tasks scheduled under NPUC is determined by analysing each of these three sections separately. The non-transactional parts are scheduled under fully-preemptive P-EDF, while the transactional section is scheduled with preemptions disabled. Moreover, the possible concurrency scenarios, in terms of higher priority interference and lower priority blocking are diverse. Therefore, it is necessary to determine upper bounds on the response times of these sections.

1. Response time of a$-\omega_i$. The computation of an upper bound on the response time of a$-\omega_i$ is computed by adapting the method described by Spuri [6] to determine the response time of a transaction under fully-preemptive EDF. Therefore, before going into further details, let us recall the intuitive idea behind the method devised by Spuri.

This method assumes a single processor and that tasks are scheduled by following the fully-preemptive EDF policy, and do not share resources, i.e. tasks do not access concurrently data and/or devices so no synchronisation is required. The longest response time experienced by a job of a task τ_i occurs in a deadline-d busy period[3] in which all tasks except τ_i are released synchronously (e.g. at time $t = 0$) and at their maximum rate, i.e., all subsequent jobs of these tasks are released as soon as it is legally permitted to do so. The job of task τ_i that experiences the longest response time is released at time a, which lies inside but near the end of the busy period. The length of the deadline-d busy period is denoted as $L_i(a)$, and the busy period ends before the deadline of the job considered, such as $a < L_i(a) < a + D_i$, if the busy period starts at $t = 0$.

$L_i(a)$ is computed iteratively by using the fixed-point algorithm, as defined by Equation 10. The final result is obtained when $L_i^{(q+1)}(a) = L_i^{(q)}(a)$ or when $L_i^{(q+1)}(a) > D_i$ in which case a deadline is missed.

$$\begin{cases} L_i^{(0)}(a) = 0 \\ \\ L_i^{(q+1)}(a) = \sum\limits_{\substack{D_j < a + D_i \\ j \neq i}} \min\left\{ \left\lceil \frac{L_i^{(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor \right\} \cdot C_j + \left(1 + \left\lfloor \frac{a}{T_i} \right\rfloor\right) \cdot C_i \end{cases} \tag{10}$$

This computation accounts for the execution demand of jobs of tasks other than τ_i with a higher priority (i.e. with deadlines earlier than $a + D_i$), the execution demand of jobs of τ_i (i.e. with release times earlier than a) and the execution demand of the considered job of τ_i.

The longest deadline-d busy period for task τ_i is given by Equation 11.

[3] A deadline-d busy period is a time interval in which the processor continually executes a sequence of jobs, delimited by two consecutive idle periods in which there are no pending jobs, that ends before the deadline d of a job of τ_i.

$$L_i = \max_{a<L_i(a)} \{L_i(a)\} \tag{11}$$

In order to drastically reduce its computational complexity and thus the search-space, Spuri used the results of three lemmas where the value of a varies, and provided an algorithm that determines the longest deadline-d busy period for all tasks in a task set, in finite time of computation. The response time of task τ_i is then determined by Equation 12, in which a_m is the release time of a job of τ_i that experienced the longest response time, defined as $a_m = \text{argmax}(L_i(a))$.

$$R_i = \max\{C_i, L_i - a_m\} \tag{12}$$

The results from [6] cannot directly be transferred to the system model assumed in this work, unfortunately. They require three adaptations to determine an upper bound on the response time of tasks under NPUC and FIFO-CRT.

1st adaptation: WCET of tasks with transactions. We know that transactions can abort and repeat. Thus, the computation of the length of a deadline-d busy period requires the WCET of the tasks that have jobs executing during that busy period. The execution time of a task with a transaction must include the overhead introduced by aborted attempts of the transaction. Therefore, for the purpose of this analysis, the WCET of a task τ_i is given by Equation 13.

$$C_i = C_{\mathrm{a}-\omega_i} + R_{\omega_i} + C_{\mathrm{p}-\omega_i} \tag{13}$$

The execution times of the non-transactional sections are known, and are assumed to be free of concurrency issues. The execution time of the transaction section is upper bounded by the response time of the transaction, calculated by one of the methods in Section 5. Thus, the determination of the deadline-d busy period considers upper bounds on the WCET values of tasks with transactions.

2nd adaptation: Extension of the deadline-d busy period. This part of the analysis determines an upper bound on the response time of a$-\omega_i$ and, therefore, the deadline-d busy period is relevant until the end of the execution of this section of the task. To this end, Equation 10 must only include the section before the transaction in the execution time of the job considered, resulting in Equation 14.

$$\begin{cases} L^{(0)}_{\mathrm{a}-\omega_i}(a) = 0 \\ \\ L^{(q+1)}_{\mathrm{a}-\omega_i}(a) = \displaystyle\sum_{\substack{D_j<a+D_i \\ j\neq i \\ \sigma(\tau_i)=\sigma(\tau_j)}} \min\left\{ \left\lceil \frac{L_i^{(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a+D_i-D_j}{T_j} \right\rfloor \right\} \cdot C_j + \left\lfloor \frac{a}{T_i} \right\rfloor \cdot C_i + C_{\mathrm{a}-\omega_i} \end{cases} \tag{14}$$

In this equation, the deadline-d busy period until the transaction, denoted as $L_{\mathrm{a}-\omega_i}$, accounts for the complete execution times of jobs of τ_i that were released prior to the considered job; only $C_{\mathrm{a}-\omega_i}$ is accounted for the last job of τ_i.

3rd adaptation: lower priority blocking. Let us assume that a job is executing a transaction and, as a consequence, preemptions are disabled. If a concurrent job with a shorter deadline is released at that moment, then it will be blocked, because tasks are scheduled by following EDF and the scheduler is not able to preempt the currently running job until the transaction commits. Only when the transaction

commits and preemptions are enabled again, do blocked jobs have the opportunity to be scheduled. Hence, the response time of a$-\omega_i$ of task τ_i must consider a possible blocking occurring at the moment the job is released. The maximum blocking time that a job of τ_i can experience is given by the longest response time of a transaction executed by a job with longer relative deadline and allocated to the same core, as formalised in Equation 15.

$$B_i = \max\left\{R_{\omega_j} \mid D_i < D_j \ \wedge \ \sigma(\tau_i) = \sigma(\tau_j)\right\} \tag{15}$$

Therefore, the response time of a$-\omega_i$, denoted as $R_{\mathrm{a}-\omega_i}$, is upper bounded by the length of the longest deadline-d busy period until the transaction, added the longest possible blocking that can occur at the time that the considered job is released, as formalised in Equation 16.

$$R_{\mathrm{a}-\omega_i} = B_i + \max\left\{C_{\mathrm{a}-\omega_i}, L_{\mathrm{a}-\omega_i} - a_m\right\} \tag{16}$$

It must be noted that if task τ_i does not have a$-\omega_i$, i.e., there is no non-transactional section prior to the transaction ($C_{\mathrm{a}-\omega_i} = 0$), then this partial analysis provides the longest delay before transaction ω_i starts executing once the job is released. This accounts for the overall response time of task τ_i.

2. Response time of p$-\omega_i$**.** Two methods to determine an upper bound on the response time of the transaction under non-preemptive scheduling have already been described in this section. To determine the response time of the non-transactional part that succeeds the transaction, it must be understood how concurrent jobs can affect the scheduling of this section.

Let us assume that a job of task τ_i is released and requests to be scheduled. This job will have to wait until all running and pending jobs with earlier deadlines have finished their executions. So, at the moment this job is scheduled, no more concurrent jobs with earlier release times and earlier deadlines are pending, awaiting for execution. From this moment on, the job of τ_i can only be preempted by later released jobs that have earlier deadlines. However, once the transaction ω_i starts, later released concurrent jobs with earlier deadlines are not able to preempt the job, because preemptions are disabled until ω_i commits. Therefore, p$-\omega_i$ may suffer interference from concurrent jobs with earlier deadlines that were released while ω_i was in progress; naturally, it will also suffer interference from jobs with earlier deadlines that are released prior to the job's completion.

Determining an upper bound on the response time of p$-\omega_i$ requires to account for the execution time demand of all jobs with earlier deadlines that are released since the transaction starts until the job considered ends.

In order to provide a conservative response time analysis from task τ_i viewpoint, we need to maximise the interference it may suffer. Let us assume that transaction ω_i starts executing at the furthest legally possible time instant from the deadline of the host job, say $\tau_{i,b}$. This allows us to accommodate the maximum release of concurrent jobs with earlier deadlines between the time the transaction starts and the deadline of the job. The relative deadline associated to transaction ω_i is the time interval between the earliest transaction start time ($r_{\omega_i,b}$) and the absolute deadline of the job ($d_{i,b}$), denoted as D_{ω_i} in Figure 8, and formalised as in Equation 17.

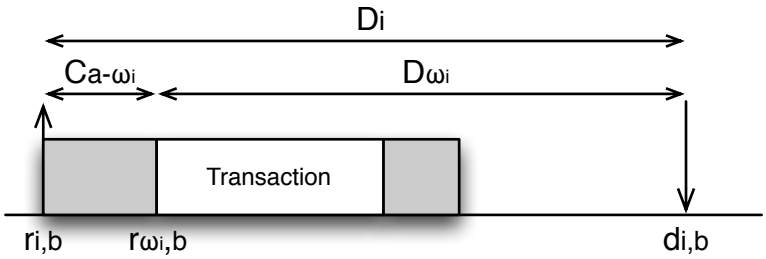

Figure 8: Maximum transaction relative deadline.

$$\begin{aligned} D_{\omega_i} &= d_{i,b} - r_{\omega_i,b} \\ &= D_i - C_{\mathrm{a}-\omega_i} \end{aligned} \tag{17}$$

The response time of $\mathrm{p}{-}\omega_i$ is computed by considering a busy period that starts at the moment at which transaction ω_i starts (and executes non-preemptively) until the job finally ends. This allows us to simplify Equation 12 as the jobs τ_j that are able to interfere with $\mathrm{p}{-}\omega_i$ are those that have deadlines such that $D_j < D_{\omega_i}$, because these jobs can be released after the transaction starts and present earlier deadlines than the job of τ_i. Therefore, an upper bound on the response time of $\mathrm{p}{-}\omega_i$ is given by $C_{\mathrm{p}-\omega_i}$ augmented by the execution times of jobs that are released with deadlines earlier than $d_{i,b}$ since ω_i started until the job ends. This upper-bound can be computed in an iterative manner by using Equation 18.

$$\begin{cases} R^{(0)}_{\mathrm{p}-\omega_i} = C_{\mathrm{p}-\omega_i} \\ \\ R^{(q+1)}_{\mathrm{p}-\omega_i} = C_{\mathrm{p}-\omega_i} + \displaystyle\sum_{\substack{D_j < D_{\omega_i} \\ i \neq j \\ \sigma(\tau_i)=\sigma(\tau_j)}} \min\left\{ \left\lceil \frac{R^{(q)}_{\mathrm{p}-\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j \end{cases} \tag{18}$$

Response time of a task. Now that we have the response times for the three sections (i.e. $\mathrm{a}{-}\omega_i$, ω_i and $\mathrm{p}{-}\omega_i$) for task τ_i, it is possible to determine an upper-bound on the response time of the complete task.

The response time of $\mathrm{a}{-}\omega_i$ ($R_{\mathrm{a}-\omega_i}$) accounts for the interference of concurrent jobs with earlier deadlines, that are released before this section ends. This includes the delay before the job is executed for the first time, and the delay caused by preemptions. It also accounts for a possible delay caused by a concurrent job with later deadline executing a transaction at the moment the job is released.

The response time of the transactional section (R_{ω_i}) accounts for the abortions caused by concurrent transactions executing in parallel on other cores. In this section, preemptions are disabled, so interference caused by concurrent jobs on the same core is pushed to the following non-transactional section.

Finally, the response time of $\mathrm{p}{-}\omega_i$ ($R_{\mathrm{p}-\omega_i}$) includes interference caused by concurrent jobs released during the execution of ω_i until the job completion time.

Therefore, the response time of a task can be upper bounded by the following equation:

$$R_i = R_{\mathrm{a}-\omega_i} + R_{\omega_i} + R_{\mathrm{p}-\omega_i} \tag{19}$$

Note that the response time of tasks without transactions is a special case of Equation 19 and is given by $R_{\mathrm{a}-\omega_i}$, assuming that $C_{\mathrm{a}-\omega_i} = C_i$ and either R_{ω_i} and $R_{\mathrm{p}-\omega_i}$ are null.

6 Response time analysis under SRP-TM

In the previous non-preemptive approaches (NPDA and NPUC), preemptions are disabled during the execution of the transactional code in order to avoid a transaction to be aborted by later released transactions once the hosting job is preempted.

SRP-TM achieves the same goal, without disabling preemptions. To this end, it applies the following rules:

1. Each transaction is protected from being preempted by concurrent jobs that are considered less urgent than itself.
2. Each transaction has a preemption level that indicates its urgency (computed from the relative deadline of the task) relative to a possible concurrent transaction that is waiting for the current transaction to commit, before it can progress.
3. Any concurrent job is allowed to preempt the running job during the execution of a transaction only if it has a higher preemption level than the transaction, i.e. it is more urgent than any other job that may be waiting for the transaction in progress.

These rules do not eliminate interference but, instead, is likely to reduce it during the transactional section. Note that the interference that is avoided during the transactional section is deferred to the following non-transactional section. Therefore, the analysis on the response time of a task under SRP-TM, just like in NPUC, involves analysing the two non-transactional sections (the anterior and posterior sections to the transaction) and the transactional section separately.

Determining an upper bound on the response time of a transaction under SRP-TM

Let us assume two transactions ω_i and ω_j with intersecting data sets and executing in parallel on cores π_k and π_ℓ, respectively. In a system in which transactions are serialised according to the chronological order of release, we assume without any loss of generality that ω_j starts first. In the worst case, in which ω_i starts at the same time as ω_j but the arbitrary tie breaking criteria decides in favour of ω_j, then the execution of ω_i is dependent on the progress of ω_j. Once ω_j commits, ω_i is free to commit, and requires at most two attempts to do so (as illustrated in Figure 9, see task τ_3): this is explained by the fact that in the worst case the current attempt might fail due to the success of ω_j, and a following attempt will finally commit.

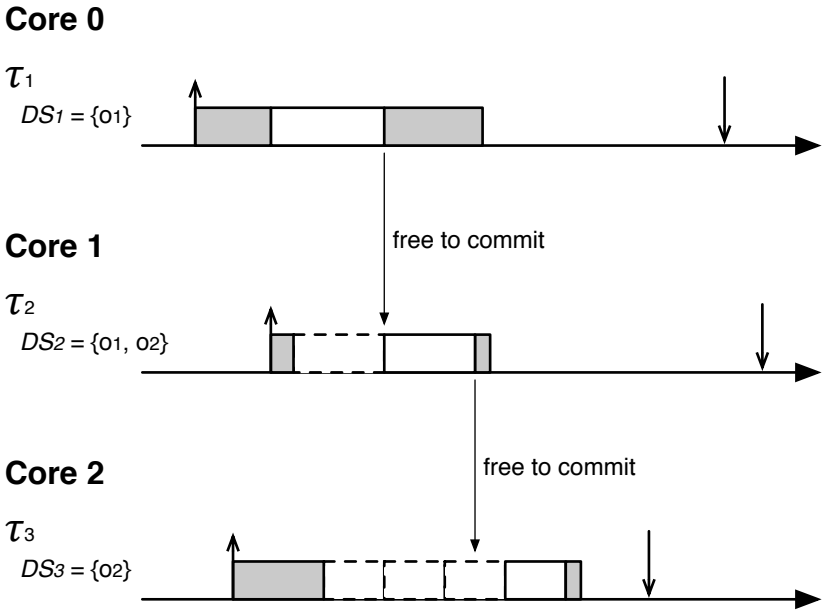

Figure 9: Contention cascades.

The response time of these last two attempts is denoted as $R^*_{\omega_i}$. For this particular example, the response time of transaction ω_i, denoted as R_{ω_i}, satisfies Equation 20.

$$R_{\omega_i} \leq R^*_{\omega_j} + R^*_{\omega_i} \tag{20}$$

This is because, in the worst case, ω_i needs to wait for the whole execution of ω_j, and then execute two more attempts before finally commit.

It must be noted that the response time of ω_i excluding the last two attempts, depends exclusively on the amount of time required by concurrent parallel transactions executing on other cores to commit (*inter-core interference*). Once all the earlier concurrent parallel transactions have committed, the transaction is free to commit, and the response time of the last two attempts depends exclusively on the scheduling of jobs on the same core as τ_i (*intra-core interference*). By induction, it can be assumed that for any given serialised sequence of transactions S_i executing concurrently in different cores, the response time of the last transaction in the sequence – ω_i – is upper bounded by the sum of the response times of the last two attempts of all transactions in the sequence, as formalised in Equation 21.

$$R_{\omega_i} = \sum_{\omega_j \in S} R^*_{\omega_j} \tag{21}$$

The first transaction in the sequence may require two attempts to commit, because it can fail at the first attempt due to a previous transaction executed on any of the cores allocated in the sequence. Thus, to compute an upper bound on the response time of a transaction, it is necessary to determine the response time of the last two attempts of each transaction, affected by intra-core interference.

Intra-core interference. Let us consider that a current running job $\tau_{i,b}$ starts executing a transaction on core π_k by following SRP-TM. From that moment until the transaction ω_i commits, SRP-TM will restrict the ability to preempt $\tau_{i,b}$ to later concurrent released jobs, say $\tau_{j,c}$, meeting the following conditions:

- the released job $\tau_{j,c}$ has an earlier deadline than $\tau_{i,b}$, so $d_{j,c} < d_{i,b}$,
- $\tau_{j,c}$ does not have a transaction, so $\lambda_{\omega_j} > 0$ (SRP-TM does not allows, more than one transaction in progress, per core), and finally
- $\tau_{j,c}$ has a higher preemption level than the current core ceiling, so $\lambda_j > \Lambda_k \geq \lambda_{\omega_i}$.

An upper bound on the response time of the last two attempts of ω_i is determined by maximising the possible intra-core interference during these two attempts. The intra-core interference is upper bounded in a pessimistic manner by assuming that the two following conditions are simultaneously met:

- the last two attempts start executing at the earliest legally possible time instant (i.e. the farthest from the deadline $d_{i,b}$), which allows us to accommodate the maximum number of releases of concurrent jobs that are capable of preempting $\tau_{i,b}$, and
- the concurrent jobs that are capable of preempting $\tau_{i,b}$ are released synchronously at the same time as the first of the two last attempts, which causes the longest delay to these two last attempts.

The earliest time instant at which these two attempts can start executing occurs when $\tau_{i,b}$ is immediately scheduled at the release time and is not preempted until ω_i starts; in addition, ω_i requires only two attempts to commit (i.e. there are no previous attempts of ω_i before these two). Under these premisses, $\tau_{i,a}$ starts executing ω_i at time $r_{\omega_i,b}$, as defined in Equation 22.

$$r_{\omega_i,b} = r_{i,b} + C_{\mathrm{a}-\omega_i} \tag{22}$$

The relative deadline associated to the transaction is then the resulting time interval between the earliest transaction start time and the absolute deadline of the job, denoted as D_{ω_i}. It is defined as in Equation 23.

$$\begin{aligned} D_{\omega_i} &= d_{i,b} - r_{\omega_i,b} \\ &= D_i - C_{\mathrm{a}-\omega_i} \end{aligned} \tag{23}$$

Figure 8 provides a graphical representation of this maximum interval.

The response time of the last two attempts of transaction ω_i, denoted as $R^*_{\omega_i}$, is given by the of execution time taken by the two attempts of the transaction, augmented by the execution time required by the concurrent jobs that are able to preempt $\tau_{i,b}$ during those two attempts, as formally defined by the fixed-point expression in Equation 24. The calculation of $R^*_{\omega_i}$ ends when the result converges to $R^{*(q+1)}_{\omega_i} = R^{*(q)}_{\omega_i}$ or when $R^*_{\omega_i}$ exceeds D_{ω_i}.

$$\begin{cases} R_{\omega_i}^{*(0)} = 2C_{\omega_i} \\ \\ R_{\omega_i}^{*(q+1)} = 2C_{\omega_i} + \sum\limits_{\substack{D_j < D_{\omega_i} \\ i \neq j \\ \sigma(\tau_i)=\sigma(\tau_j) \\ \lambda_j > \lambda_{\omega_i} \\ \lambda_{\omega_j}=0}} \min\left\{ \left\lceil \frac{R_{\omega_i}^{*(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j \end{cases} \tag{24}$$

This result is pessimistic because it does not consider the effect of a job with a transaction released with the earliest deadline to raise the core ceiling to its preemption level, if blocked. If the core ceiling is raised, then some releases of the concurrent jobs that are accounted for in Equation 24 may not be able to preempt $\tau_{i,b}$.

Inter-core interference. After the maximum response time of the last two attempts is computed for each transaction in the task set, it becomes possible to determine an upper bound for the inter-core interference of every transaction, based on the general approach (for some given sequence of transactions) provided by Equation 21.

Determining a tight upper bound on the response time of a given transaction requires us to find the sequence of transactions that produces the longest delay. Similar as in NPUC, such an operation has a high combinational complexity and, while it is feasible with small numbers of cores and transactions, it becomes a non-trivial exercise when the number of cores, the number of transactions allocated to each core and contention probability increases.

A more pessimistic upper bound in $O(n)$ complexity can be provided, if we assume a sequence that includes, for all the cores allocated to the contention group of the transaction in analysis, the longest response times of the transactions that belong to the same contention group. Even if the selected transactions are not able to form a practical sequence of conflicting transactions, the result upper bounds the exact value.

Let us assume that transaction ω_i is assigned to core π_k and belongs to contention group Ω_a, and transactions in Ω_a are allocated to a subset of m_a cores denoted as Π_a. We can pessimistically select for each core in Π_a except π_k the transaction that presents the longest response time to execute two attempts, and sum these response times. The result is the maximum amount of time a transaction in core π_k will have to wait if all other cores in Π_a have a transaction in progress that must commit before ω_i. The inter-core interference, denoted as $I_{\omega_i}^k$, can be upper bounded by Equation 25.

$$I_{\omega_i}^k = \sum_{\pi_\ell \in \Pi_a \setminus \pi_k} \max\left\{ R_{\omega_j}^* \mid \omega_j \in \Omega_a \ \wedge \ \sigma(\tau_j) = \pi_\ell \right\} \tag{25}$$

Response time of a transaction. With the upper bounds on the time that ω_i has to wait for transactions in other cores to commit (given by $I_{\omega_i}^k$) and for the time required to execute two attempts of ω_i, considering exclusively intra-core

interference (given by $R^*_{\omega_i}$), the overall response time of ω_i can be upper bounded by combining the two previous results, as defined in Equation 26.

$$R_{\omega_i} = I^k_{\omega_i} + R^*_{\omega_i} \tag{26}$$

Response time of a task under SRP-TM

This section formalises the response time analysis of tasks scheduled under SRP-TM.

P-EDF allows interference when a job with an earlier deadline preempts a running job. SRP-TM adds blocking, when the scheduler decides that a job executing the transaction in progress should run, instead of the job with the earliest absolute deadline. Both of these scheduling characteristics affect the response time of a task. However, blocking and interference differ for tasks with and without transactions, as discussed in this section. Therefore, the response time analysis for tasks, with and without transactions, is formalised in two distinct approaches.

Response time analysis of tasks with transactions

A task τ_i with a transaction has three distinct segments: the segment anterior to the transaction, the transaction and the segment posterior to the transaction.

Any job of such task can get directly blocked at the time it is released, if a transaction with lower preemption level executing in a job with further absolute deadline is already in progress. Because SPR-TM does not alloe more than one transaction in progress per core, this job will not be scheduled before the transaction in progress commits. Once the transaction in progress commits, job $\tau^k_{i,j}$ becomes unblocked and no other job with further absolute deadline will be able to execute before $\tau^k_{i,j}$ has finished.

A task with a transaction suffers the usual interference that is natural in EDF: any concurrent job with an earlier absolute deadline can preempt it. However, when a job of such task is executing a transaction, it becomes protected from being preempted by concurrent jobs with transactions; furthermore, it can only be preempted by jobs with earlier absolute deadline and higher preemption level than of the transaction. Thus, the analysis of the response time differs for the time the transaction is in progress.

Therefore, upper bounds for the direct blocking time, the response time of the task before starting the transaction and the response time of the job after the transaction committed are derived in order to determine an upper bound for the response time of a task with a transaction, scheduled under SRP-TM.

Blocking of jobs with transactions. SRP-TM does not allow multiple transactions progressing simultaneously on the same core. This rule is enforced by not allowing a job with a transaction with the earliest deadline to be scheduled until the current transaction in progress commits. Consequently, any job with a transaction that holds the earliest deadline of all jobs at the instant it is released, it will have to wait for the current transaction in progress to commit: the job is directly blocked. After the transaction in progress eventually commits, the waiting job becomes unblocked and no other job with later deadline will be scheduled before

it. Therefore, any job with a a transaction can be directly blocked at most once, at the moment it is released.

In addition, SRP-TM rules out indirect blocking because it ensures that there is no transaction in progress from a concurrent job with later deadline, when a job with a transaction is executing. This eliminates the possibility of a job with a transaction to be preempted so another job with later deadline can finish the current transaction in progress (as no such transaction can be in progress).

Therefore, tasks that execute a transaction can only suffer direct blocking .

Thus, for a task with a transaction, the maximum blocking time, denoted as DB_i, is defined by longest response time from all the transactions that execute on the same core, from the subset of tasks with lower preemption levels, as formalised in Equation 27.

$$\mathrm{DB}_i \stackrel{\text{def}}{=} \max\left\{R_{\omega_j} \mid \lambda_j < \lambda_i \ \wedge \ \sigma(\tau_i) = \sigma(\tau_j)\right\} \tag{27}$$

Response time of the section anterior to the transaction. Under P-EDF, any job can suffer interference from jobs released on the same core with earlier absolute deadline. The work of [6], briefly described in Section 5, provides a method to determine the response time of independent tasks scheduled under EDF.

Like in the NPUC response time analysis, the response time of the section of a task that is executed before the transaction can be determined by adapting the method of Spuri to the restrictions imposed by SRP-TM to the general EDF policy. This method determines the length of a deadline-d busy period that produces the longest delay in the response time of a job of a task that is released at time a. The computation of the length of this busy period requires the WCET of the tasks that have jobs in this busy period. For the purpose of this analysis, the WCET of a task τ_i that has a transaction must include the overhead due to aborted attempts of the transaction, and is therefore defined by Equation 28.

$$C_i = C_{\mathrm{a}-\omega_i} + R_{\omega_i} + C_{\mathrm{p}-\omega_i} \tag{28}$$

The extension of the busy period goes from the time all concurrent jobs were synchronously released ($t = 0$) until the end of the initial non-transactional section of the job. This busy period includes the executions times taken by all concurrent jobs that executed with earlier deadlines than the job considered, the execution times used by previous releases of τ_i and the execution time of the initial non-transactional part of the job considered. This is the same case considered in the response time analysis of this section, under NPUC. Logically, the original iterative equation is adapted in the same way as in the NPUC response time analysis of the same non-transactional section, resulting in Equation 29. The computation for each value of a ends when $L_{\mathrm{a}-\omega_i}(a)$ converges to a constant value such that $L^{(q+1)}_{\mathrm{a}-\omega_i}(a) = L^{(q)}_{\mathrm{a}-\omega_i}(a)$.

$$\begin{cases} L^{(0)}_{\mathrm{a}-\omega_i}(a) = 0 \\ \\ L^{(q+1)}_{\mathrm{a}-\omega_i}(a) = \sum\limits_{\substack{D_j < a+D_i \\ j \neq i \\ \sigma(\tau_i)=\sigma(\tau_j)}} \min\left\{ \left\lceil \frac{L_i^{(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a+D_i-D_j}{T_j} \right\rfloor \right\} \cdot C_j + \left\lfloor \frac{a}{T_i} \right\rfloor \cdot C_i + C_{\mathrm{a}-\omega_i} \end{cases} \tag{29}$$

Spuri provides a method to reduce the search space so that this method provides a solution in practical time for the longest deadline-d busy period, defined as in Equation 30.

$$L_{\mathrm{a}-\omega_i} = \max_{a < L_{\mathrm{a}-\omega_i}(a)} L_{\mathrm{a}-\omega_i}(a) \tag{30}$$

Once the longest delay produced by a deadline-d busy-period to a job of τ_i released in instant $a_m = \mathrm{argmax}(L_i(a))$ is computed, the response time of the section anterior to the transaction is upper bounded by the execution time of this non-transactional section, added to the delay produced by the busy period, and the possible blocking that the job can suffer when released, as defined in Equation 31.

$$R_{\mathrm{a}-\omega_i} = \mathrm{DB}_i + \max\left\{C_{\mathrm{a}-\omega_i}, L_{\mathrm{a}-\omega_i} - a_m\right\} \tag{31}$$

Response time of the section posterior to the transaction. Under SRP-TM, once the section of a task that is executed after a transaction starts, it can only be preempted by any later released job presenting an earlier absolute deadline. To determine an upper bound for the response time of this section, we must define a time window limited by the instant the segment starts executing and the absolute deadline of the host job. The most critical time instant in which the segment can start executing is when the blocking and prior interference were experienced to the maximum extent. In such situation, represented in Figure 10, the job is facing the highest risk to miss its deadline due to interference. The length of this critical time interval, denoted as $D_{\mathrm{p}-\omega_i}$ and represented in Figure 10, is formally defined by the longest response times of the first non-transactional section (including possible direct blocking) and of the transactions section, as formalised in Equation 32.

$$D_{\mathrm{p}-\omega_i} = D_i - (R_{\mathrm{a}-\omega_i} + R_{\omega_i}) \tag{32}$$

The response time of this last section depends on the execution times of the concurrent jobs that are able to preempt this job until it finishes executing. Such jobs are those that are released inside this time interval but not after the job of τ_i ended, with deadlines earlier than the job of τ_i. The response time of this section can be defined by the execution time demand of the section added the execution time demands of the concurrent jobs with earlier deadlines that are released after this section started. Therefore, the response time of this section can be defined by Equation 33. The iterative computation ends when the result converges to a value such that $R^{(q+1)}_{\mathrm{p}-\omega_i} = R^{(q)}_{\mathrm{p}-\omega_i}$.

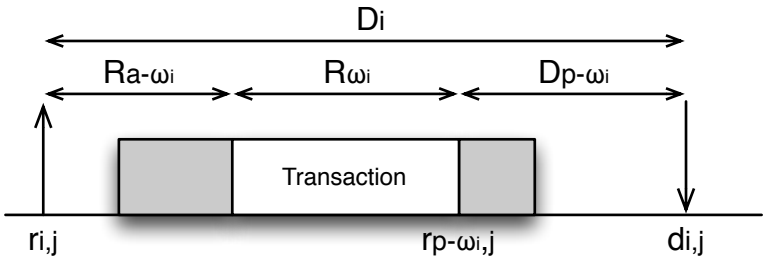

Figure 10: Relative deadline of last code segment.

$$\begin{cases} R^{(0)}_{\mathrm{p}-\omega_i} = C_{\mathrm{p}-\omega_i} \\ R^{(q+1)}_{\mathrm{p}-\omega_i} = C_{\mathrm{p}-\omega_i} + \sum\limits_{\substack{D_j < D_{\mathrm{p}-\omega_i} \\ i \neq j \\ \sigma(\tau_i)=\sigma(\tau_j)}} \min\left\{ \left\lceil \frac{R^{(q)}_{\mathrm{p}-\omega_i}}{T_j} \right\rceil, 1 + \left\lfloor \frac{D_{\mathrm{p}-\omega_i} - D_j}{T_j} \right\rfloor \right\} \cdot C_j \end{cases} \tag{33}$$

Response time of a task with a transaction. The response time of a task that executes a transaction is given by the combination of direct blocking and the response times of the three code segments that compose the task. Thus, the response time of a task τ_i can be upper bounded by the sum of the maximum response times of the three code segments of the task, as defined in Equation 34. Note that the response time of the initial non-transactional section already includes the possible direct blocking time.

$$R_i = R_{\mathrm{a}-\omega_i} + R_{\omega_i} + R_{\mathrm{p}-\omega_i} \tag{34}$$

Response time analysis of tasks without transactions

A task that does not execute a transaction can experience either direct or indirect blocking, besides interference from jobs (with or without transactions) with earlier absolute deadlines.

Direct blocking occurs when job $\tau^k_{i,b}$ is released and is not allowed to preempt a running job $\tau^k_{e,c}$ that has a further absolute deadline ($d_{i,b} < d_{e,c}$), but is executing a transaction with a higher preemption level than τ_i ($\lambda_i < \lambda_{\omega_e}$). This type of blocking can occur at most once, when the job is released. Once the blocking transaction commits and $\tau^k_{i,b}$ becomes unblocked, then no further job with later absolute deadline will be able to execute before $\tau^k_{i,b}$ has finished. Thus, after being directly blocked, $\tau^k_{i,b}$ can only suffer interference from any released jobs with earlier absolute deadlines.

Indirect blocking occurs when when a transaction ω_e of job $\tau^k_{e,c}$ is in progress at the moment $\tau^k_{i,b}$ is released; however, unlike in the case of direct blocking, $\tau^k_{i,b}$ has the necessary scheduling conditions to execute ahead of the transaction, that is, $d_i < d_e$ and $\lambda_{\omega_e} < \lambda_i$, and so $\tau^k_{e,c}$ is preempted. However, transaction ω_e can be forced to execute before $\tau^k_{i,b}$ has finished if a job $\tau^k_{g,h}$ with a transaction and an earlier

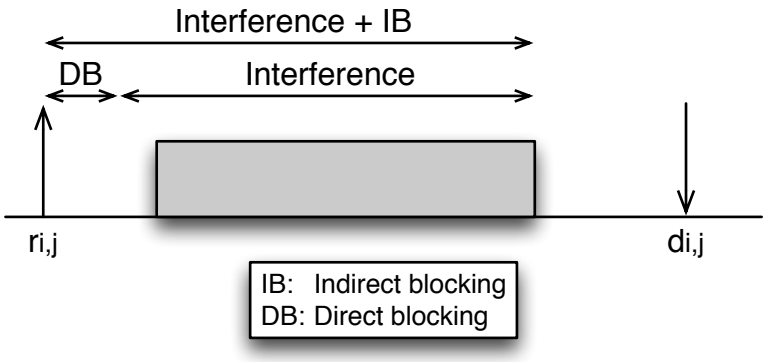

Figure 11: Relative deadline of last code segment.

absolute deadline than d_i is released, and rises the core ceiling above λ_i. In such situation, the scheduler prioritises the execution of the transaction in progress on behalf of the more urgent job, preempting any job that has a task preemption level lower than the raised core ceiling, if necessary. Once the transaction is successfully committed, the corresponding job is preempted and $\tau_{i,j}^k$ will be exclusively subject to higher priority interference from jobs with or without transactions.

Figure 11 illustrates the two possible patterns of blocking and interference described, that will be further discussed in this section.

Blocking of tasks without transactions. As previously referred, direct blocking occurs when the arriving job $\tau_{i,b}^k$ has earlier absolute deadline than of the job $\tau_{e,c}^k$ currently executing a transaction, but its task premption level is lower than the preemption level of the transaction in progress ($\lambda_i < \lambda_{\omega_e}$). In this case, $\tau_{i,b}^k$ can not preempt $\tau_{e,c}^k$ until the transaction successfully commits, when the core ceiling is reset to zero and the SRP-TM restrictions are dropped.

The longest direct blocking a task τ_i can experience, denoted as DB_i, is given by the longest response time from all transactions with higher preemption levels in tasks with lower preemption levels (respective to the preemption level of τ_i), allocated to the same core. This is formally expressed in Equation 35.

$$\mathrm{DB}_i = \max\left\{R_{\omega_j} \mid \lambda_{\omega_j} > \lambda_i \ \wedge \ \lambda_j < \lambda_i \ \wedge \ \sigma(\tau_j) = \sigma(\tau_i)\right\} \tag{35}$$

Alternatively, a job $\tau_{i,b}^k$ can be indirectly blocked if it is released while a transaction ω_e with lower preemption level is in progress. If another concurrent job $\tau_{g,c}^k$ with a transaction is released meeting the necessary scheduling conditions – with earlier absolute deadline and preemption level higher than the core ceiling –, it will rise the core ceiling to a level higher than λ_i. In this case, the scheduler will execute the job with the transaction in progress, so that the job with the earliest deadline can be scheduled as soon as possible. In this case, $\tau_{i,b}^k$ is indirectly blocked.

Once the transaction in progress commits, no other job with later absolute deadline will be able to execute before $\tau_{i,b}^k$: thus, this type of blocking can occur only once to job $\tau_{i,j}^k$, but at any moment between the release and finish times f $\tau_{i,b}^k$.

The longest indirect blocking time is given by the longest response time from all transactions with lower preemption levels than τ_i allocated to the same core, as long as there is a task τ_g with a transaction that is able to preempt τ_i, by having an earlier absolute deadline. This is formally defined in the Equation 36.

$$\mathrm{IB}_i = \max\left\{R_{\omega_j} \mid \lambda_{\omega_j} < \lambda_i \ \wedge \ \sigma(\tau_j) = \sigma(\tau_i)\right\}, \quad \exists \tau_g \ : \ \lambda_g > \lambda_i \ \wedge \ \lambda_{\omega_g} > 0 \tag{36}$$

As previously mentioned, once a job without a transaction $\tau_{i,b}^k$ is released, then only one transaction carried by a job with later deadline can ever execute before $\tau_{i,b}^k$ finishes execution. Therefore, direct and indirect blocking are mutually exclusive, so any job without a transaction can only suffer blocking at most once. Therefore, the longest time a job can be blocked is given by the maximum value between direct and indirect blocking, as defined in Equation 37.

$$B_i = \max\left\{\mathrm{DB}_i, \mathrm{IB}_i\right\} \tag{37}$$

Response time of a task without a transaction. When job $\tau_{i,j}^k$ is not blocked (either directly or indirectly), it is subject to suffer interference from any released job with earlier absolute deadline. The method described by [6] can be used to determine an upper bound for tasks that do not have a transaction. Unlike the analysis for tasks with transaction that involved dividing the analysis by transactional and non-transactional sections, the method of Spuri can be directly applied to the whole extent of the task execution. Like in the Spuri-based analysis of non-transactional sections, the execution times of tasks with transactions are defined as in Equation 28. The iterative equation is the originally provided by [6], reproduced in Equation 38.

$$\begin{cases} L_i^{(0)}(a) = 0 \\ \\ L_i^{(q+1)}(a) = \displaystyle\sum_{\substack{D_j < a + D_i \\ j \neq i \\ \sigma(\tau_i) = \sigma(\tau_j)}} \min\left\{ \left\lceil \frac{L_i^{(q)}}{T_j} \right\rceil, 1 + \left\lfloor \frac{a + D_i - D_j}{T_j} \right\rfloor \right\} \cdot C_j + \left\lfloor 1 + \frac{a}{T_i} \right\rfloor \cdot C_i \end{cases} \tag{38}$$

Once the length of the deadlline-d busy period that produces the longest delay on a job of τ_i released at instant $a_m = \mathrm{argmax}(L_i(a))$, denoted as L_i and defined in Equation 11, the result from the Spuri method must be combined with the blocking delay that can be produced by an earlier released job with transaction in progress. The response time of a task without a transaction can be upper bounded by Equation 39.

$$R_i = B_i + \max\left\{C_i, L_i - a_m\right\} \tag{39}$$

7 Conclusions

This report provides methods to determine upper bounds for for the response times of tasks scheduled by following three different approaches: NPDA, NPUC and SRP-TM. It is assumed that conflicts between concurrent transactions executed by these

tasks are solved by a contention manager that selects the transaction with the earlier release time to commit.

NPDA is an approach in which transactions are executed non-preemptively, but preemption points are inserted between attempts, which retains responsiveness of higher priority tasks. However, NPDA allows multiple transactions in progress per core which, consequently, provides very pessimistic upper bounds for the response times of tasks with transactions.

NPUC schedules transactions non-preemptively without any preeemption points from the instant the transaction starts until the transaction commits. This as a relevant impact on the maximum blocking time of higher priority tasks. However, it automatically ensures that on each core there is no more than one transaction in progress. Consequently, the response time of a transaction is more predictable, allowing a more realistic timing analysis than NPDA.

SRP-TM is a fully-preemptive approach that combines the predictability of NPUC and the responsiveness of higher priority tasks as NPDA. Based on the SRP, SRP-TM uses preemption levels to determine if a job should be preempted by a concurrent higher priority job, when it is executing a transaction. Although the response time analysis for SRP-TM is more pessimistic than for NPUC, it still provides realistic results, as opposed to NPDA.

References

[1] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th annual ACM symposium on Principles of distributed computing (PODC)*, pages 204–213, Ottawa, Canada, August 1995.

[2] Theodore P. Baker. Stack-based scheduling of realtime processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[3] Liliana Cucu-Grosjean and Joël Goossens. Exact schedulability tests for real-time scheduling of periodic tasks on unrelated multiprocessor platforms. *Journal of Systems Architecture*, 57(5):561–569, 2011.

[4] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, February 1997.

[5] António Barros and Luís Miguel Pinho. Non-preemptive scheduling of Real-Time Software Transactional Memory. In *Proceedings of the 27th International Conference on Architecture of Computing Systems (ARCS)*, pages 25–36, Lübeck, Germany, February 2014. Springer International Publishing.

[6] Marco Spuri. Analysis of Deadline Scheduled Real-Time Systems. Research Report RR-2772, INRIA - Institut National de Recherche en Informatique et en Automatique, Paris, France, 1996.