Multi-Processor Schedulability Analysis of Preemptive Real-Time Tasks with Variable Execution Times

Pavel Krcal[1], Martin Stigge[2], and Wang Yi[1]

[1] Uppsala University, Sweden
Email: {pavelk,yi}@it.uu.se
[2] Humboldt University Berlin, Germany
Email: mstigge@informatik.hu-berlin.de

Abstract In this paper, we study schedulability analysis problems for multi-processor real-time systems. Assume a set of real time tasks whose execution times and deadlines are known. We use timed automata to describe the non-deterministic arrival times of tasks. The schedulability problem is to check whether the released task instances can be executed within their given deadlines on a multi-processor platform where each processor has a task queue to buffer task instances scheduled to run on the processor. On the positive side, we show that the problem is decidable for systems with non-preemptive schedulers or tasks with fixed execution times. A surprising negative result is that for multi-processor systems with variable task execution times and a non-preemptive scheduler, the schedulability analysis problem is undecidable, which is still an open problem in the single-processor setting.

1 Introduction

Real-time systems are often designed as a collection of real-time tasks and a scheduling strategy implemented as a scheduler. Each of the tasks may have a set of task parameters such as release rate (or release pattern), best and worst-case execution times on the target hardware, priority, and deadline etc. In the operation of a system, the tasks will be released according to the pre-designed release rates, and the released task instances are scheduled to execute on a processor by the scheduler. Here the scheduler, namely the scheduling strategy, is the critical component for the correct functioning of a system. It makes the decision on which order of the released task instances should be executed, based on the task parameters and the current system state.

An important design problem is to analyze whether all the task instances can be executed within the given deadlines, which is essentially to estimate the worst-case response times of the tasks. This is the so-called schedulability analysis based on (1) the task release patterns, (2) the task parameters and (3) the scheduling policy, all of which are from the system design phase. The source of complexity in solving the analysis problem is in dealing with task releases and preemptions. A newly released task instance may preempt the running task which influences the response time for the preempted task. Classic solutions, e.g., Rate-Monotonic Analysis [LL73] often assume deterministic task release patterns such as periodic tasks or deterministic patterns with fixed type of non-determinism such as jitters [But97] and offsets [RC01]. A challenge is to solve the schedulability analysis problem for systems with dynamic and non-deterministic task releases and preemptions. In recent years, in a series of work, we have used timed automata [AD94] to model task release patterns and solved the problem for a large class of single-processor systems [FKPY07,FMPY06].

In [FKPY07], timed automata are extended with asynchronous tasks. Each location of a timed automaton may be associated with a task. As soon as the automaton visits a location, an instance of the associated task is released and put (scheduled) into a task queue for execution. Compared with classic task models studied in the literature on scheduling, extended timed automata provide a much more expressive model which, in fact, inherits the full expressive power of timed automata for modeling of dynamic and non-deterministic task releases and preemptions. In our previous work, the classic notion of schedulability analysis has been extended to the automata model, and it is shown that for a large class of systems, the problem can be solved automatically using algorithmic methods. However, the study has been restricted to the single-processor setting.

In this paper, we study the schedulability analysis problem of the extended automata model in the multi-processor setting. Basic scheduling algorithms for multi-processor systems can be found in, e.g., [Liu00]. For recent work on schedulability analysis for multi-processor systems and further references, see [ABJ01,BCL05]. We assume that a system has a fixed number of processors available. Each processor is associated a task queue, where the released tasks wait for being processed. A scheduling policy (modeled as a function) decides for a newly released task instance into which queue and at which position in the queue it will be inserted. However, task migration is not allowed, that is, once a task instance is assigned to a processor, it will remain in the associated queue until it finishes.

On the positive side, we show that the problem is decidable as for the single-processor case if

1. the scheduler runs a non-preemptive scheduling strategy, or
2. the tasks have fixed execution times, that is, the best and worst-case execution times coincide.

It is an open problem, whether for the class of systems with variable task execution times and a non-preemptive scheduler, the schedulability problem is decidable in the single-processor setting. This problem becomes undecidable when the scheduling policy has at least two processors with their task queues available. More precisely as a main technical result of this paper, we show that the schedulability problem will be undecidable for multi-processor systems if

1. the scheduler runs a preemptive scheduling strategy, and
2. the tasks have variable execution times ranging over an interval between the best and worst-case execution times.

2 Preliminaries

In this section, we introduce the concept of *task automata* (timed automata extended with tasks developed in [FKPY07], and the multi-processor schedulability problem.

2.1 Task Automata

Let $\mathcal{C}$ be a finite set of clocks. A function $\nu : \mathcal{C} \to \mathbb{R}_{\geq 0}$ is called a *clock valuation* while the set of all clock valuations over the clocks $\mathcal{C}$ is denoted by $\mathcal{V}(\mathcal{C})$. With $\nu[r]$ we denote the clock valuation which is equal to ν, except that all clocks in $r \subseteq \mathcal{C}$ are being reset to zero. $\mathcal{B}(\mathcal{C})$ is the set of *clock guards* g, which are conjunctions of expressions $x_1 \bowtie N$ and $x_1 - x_2 \bowtie N$ with $x_1, x_2 \in \mathcal{C}$, $N \in \mathbb{N}_{\geq 0}$ and $\bowtie\ \in \{<, =, >\}$. If g contains only $x_1 \bowtie N$ expressions, we say it is *diagonal-free*. A valuation ν satisfies a guard g (written $\nu \models g$) if for all expressions $x_1 \bowtie N$ and $x_1 - x_2 \bowtie N$ in g it holds that $\nu(x_1) \bowtie N$ and $\nu(x_1) - \nu(x_2) \bowtie N$, respectively.

Tasks and task queues. We define a task type as a tuple (P, B, W, D) written $P(B, W, D)$ where P is the task name (unique for each task type), $B, W \in \mathbb{N}_{\geq 0}$ the best and worst case calculation times (with $B \leq W$ and $W \geq 1$) and $D \in \mathbb{N}_{\geq 1}$ the relative deadline. A task instance $P_i(b_i, w_i, d_i)$ of type $P_i(B_i, W_i, D_i)$ is a "released copy" of this task type with $b_i, w_i, d_i \in \mathbb{R}$ being the remaining values. A task queue q is a list $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$ of task instances (of possibly the same type). By a discrete part of a queue $[P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)]$ we mean the list of the corresponding task names $[P_1, \ldots, P_n]$ (the information about the remaining computation times and deadline is projected out). Let $\mathcal{P}$ be the set of task types and $Q_{\mathcal{P}}$ be the set of queues over this task type set. We use a function $\mathsf{Run} : Q_{\mathcal{P}} \times \mathbb{R}_{\geq 0} \to Q_{\mathcal{P}}$ which given a non-negative real number t and a task queue q returns the task queue after t time units of execution on a processor. The result of $\mathsf{Run}(q, t)$ for $q = [P(b_1, w_1, d_1), Q(b_2, w_2, d_2), \ldots, R(b_n, w_n, d_n)]$ and $t \leq w_1$ is defined as $q' = [P(b_1 - t, w_1 - t, d_1 - t), Q(b_2, w_2, d_2 - t), \ldots, R(b_n, w_n, d_n - t)]$. For an empty queue, denoted by $[]$, $\mathsf{Run}([], t) = []$.

Definition 1. *A* task automaton *over actions $\mathcal{A}ct$, clocks $\mathcal{C}$, and task types $\mathcal{P}$ is a tuple $\langle N, l_0, E, I, M, x_{done} \rangle$ where*

- *N is a finite set of locations,*
- *$l_0 \in N$ is the initial location,*
- *$E \subseteq N \times \mathcal{B}(\mathcal{C}) \times \mathcal{A}ct \times 2^{\mathcal{C}} \times N$ is the set of edges,*
- *$I : N \to \mathcal{B}(\mathcal{C})$ is a function assigning a clock constraint to each location which is called* location invariant,
- *$M : N \hookrightarrow \mathcal{P}$ is a partial function assigning locations with task types,*[1] *and*
- *$x_{done} \in \mathcal{C}$ is the clock which is reset whenever a task finishes.*

We write $l \xrightarrow{g\ a\ r} l'$ for $\langle l, g, a, r, l' \rangle \in E$.

[1] Note that M is a partial function meaning that some of the locations may have no tasks.

Semantics. An important part of the operational semantics is the scheduling function, which decides, into which queue and at which position a newly released task should be inserted. For the sake of presentation, we first introduce the scheduling function $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \to Q_{\mathcal{P}}$ for the single-processor case. Given a task instance and a task queue, it returns the task queue with the task instance inserted and the order of the other task instances preserved. Depending on whether the scheduler is preemptive or non-preemptive, the function may insert new tasks as the first element or not. Since we want this function to be encodable in timed automata, the definition has the following important condition.

Definition 2. $\mathsf{Sch} : \mathcal{P} \times Q_{\mathcal{P}} \to Q_{\mathcal{P}}$ *is a* scheduling function*, if for each task type $P(B, W, D)$ and discrete part $[P_1, \ldots, P_n]$ of a queue there can be effectively constructed a diagonal-free timed automaton with*

- *Clocks $y_1^b, y_1^w, y_1^d, \ldots, y_n^b, y_n^w, y_n^d$,*
- *$n + 2$ locations $l_0, l_1, \ldots, l_{n+1}$, and*
- *$n + 1$ edges from l_0 to l_i for $1 \leq i \leq n + 1$,*

such that $\mathsf{Sch}(P(B, W, D), [P_1(b_1, w_1, d_1), \ldots, P_n(b_n, w_n, d_n)])$ inserts $P(B, W, D)$ into the queue at the k-th position if and only if l_k is the only location reachable from (l_0, u) where $u(y_i^b) = b_i, u(y_i^w) = w_i, u(y_i^d) = d_i$ for all $1 \leq i \leq n$.

This definition generalizes the notion of a scheduling policy. Most of the standard scheduling policies such as EDF, FPS, FIFO etc satisfy this condition (concrete cases are discussed later), but also ad hoc policies such as combinations of the standard ones are also included. Therefore, the results hold for a very general class of scheduling policies.

In the multi-processor case, the scheduling function takes k queues and a task type as an input and returns the queues with the new task instance inserted at some position in one of the queues. It can use the information from all the queues for its decision. Each timed automaton corresponding to a scheduling policy then contains clocks for all the instances in all the queues. To simplify the notation, we assume that for k queues $q_1, \ldots, q_k$ with discrete parts $\tilde{q}_1, \ldots, \tilde{q}_k$, all the task instances are indexed by one index i ranging from 1 to $\sum_{1 \leq j \leq k} |\tilde{q}_j|$, that is, we view all the queues as one global queue, where $|\tilde{q}_j|$ denotes the number of the task instances in the queue q_j. E.g., the index of the first task instance in q_2 is $|\tilde{q}_1| + 1$. Also, all possible positions where a new task can be inserted are indexed by one index ranging from 1 to $\sum_{1 \leq j \leq k} |\tilde{q}_j| + k$. E.g., the index of the head position of the second queue (the first task instance in q_2 after insertion) is $|\tilde{q}_1| + 2$. By b_i, w_i and d_i we denote as before the continuous queue information for the task instance p_i.

Definition 3. *(Multi-processor Scheduler) Let $k \in \mathbb{N}$ be the* number of processors*. Then $\mathsf{Sch}_k : \mathcal{P} \times (Q_{\mathcal{P}})^k \to (Q_{\mathcal{P}})^k$ is a* multi-processor scheduling function*, if for each task type $P(B, W, D)$ and* discrete parts *of queues $\tilde{q}_1, \ldots, \tilde{q}_k$, there can be effectively constructed a diagonal-free timed automaton with*

– *Clocks* $y_1^b, y_1^w, y_1^d, \ldots, y_K^b, y_K^w, y_K^d$ *where* $K = \sum_j |\tilde{q}_j|$,
– $K + k + 1$ *locations* $l_0, l_1, \ldots, l_{K+k}$ *and*
– $K + k$ *edges from* l_0 *to* l_j *for* $1 \leq j \leq K + k$,

such that the function $\mathsf{Sch}(P(B, W, D), q_1, \ldots, q_k)$ *inserts* $P(B, W, D)$ *at the* m*-th position if and only if* l_m *is the only location reachable from* (l_0, u) *where* $u(y_i^b) = b_i, u(y_i^w) = w_i, u(y_i^d) = d_i$ *for all* $1 \leq i \leq K + k$.

Note that this definition does not allow for moving tasks between processors after they got assigned to one processor at the moment when they are released, that is, we do not allow task migration.

A task automaton may – just as a timed automaton – perform event and delay transitions, and additionally *task finishing* transitions. An event transition corresponds to the arrival of a new task, and a delay transition corresponds to active tasks being executed while the others are waiting, or just processor idling in case some queue is empty. The third type of transitions deals with the finishing of a task. We give now the formal definition of the operational semantics for a system with k processors as a labeled transition system (LTS), where $S := N \times \mathcal{V}(\mathcal{C}) \times (Q_{\mathcal{P}})^k$ is the state space, thus incorporating the queues into the state information. ν_0 is a clock valuation assigning all clocks the value 0, and $\Sigma := \mathcal{A}ct \cup \mathbb{R}_{\geq 0} \cup \{\mathsf{fin}\}$ a set of labels (for events, time pass values and task finishing).

Definition 4. *Given a scheduling strategy* Sch_k *over k processors, the* semantics *of a task automaton* $A = \langle N, l_0, E, I, M, x_{done} \rangle$ *is a labeled transition system* $[\![A_{\mathsf{Sch}_k}]\!] = \langle S, s_0, \Sigma, T \rangle$ *with* $s_0 = (l_0, \nu_0, [])$ *and the set of transitions* T *defined by the following rules:*

– $(l, \nu, q_1, \ldots, q_k) \xrightarrow{a}_{\mathsf{Sch}_k} (l', \nu[r], \mathsf{Sch}_k(M(l'), q_1, \ldots, q_k))$ *if* $l \xrightarrow{g\ a\ r} l'$, $\nu \models g$, *and* $\nu[r] \models I(l')$,
– $(l, \nu, q_1, \ldots, q_k) \xrightarrow{t}_{\mathsf{Sch}_k} (l, \nu + t, \mathsf{Run}(q_1, t), \ldots, \mathsf{Run}(q_k, t))$ *if* $t \in \mathbb{R}_{\geq 0}$, $(\nu + t) \models I(l)$, *and for all* i *with* $q_i = P(b, w, d) :: q_i'$ *it holds that* $t \leq w$, *and*
– $(l, \nu, q_1, \ldots, P(b, w, d) :: q_i, \ldots, q_k) \xrightarrow{\mathsf{fin}}_{\mathsf{Sch}_k} (l, \nu[x_{done}], q_1, \ldots, q_i, \ldots, q_k)$ *if* $b \leq 0 \leq w$ *and* $\nu[x_{done}] \models I(l)$,

where $P(b, w, d) :: q$ *denotes the queue with the task instance* $P(b, w, d)$ *on the first position and* q *being the (possibly empty) tail.*

Finally, we can also define the main question this work deals with, namely whether an automaton models schedulable sequences of task releases. As all deadlines in our model are hard, we say that a task automata is schedulable for a given scheduling strategy if no matter how does the (non-deterministic) computation evolve, all deadlines are met. We use q_{err} to denote queues containing a task instance $P(b, w, d)$ with $d < 0$.

Definition 5. *(Schedulability) A task automaton* A *with initial state* $(l_0, u_0, [], \ldots, [])$ *is* non-schedulable *with* Sch_k *if* $(l_0, u_0, [], \ldots, []) \longrightarrow^*_{\mathsf{Sch}_k} (l, u, q_1, \ldots, q_{err}, \ldots, q_k)$ *for some* l *and* u. *Otherwise, we say that* A *is* schedulable *with* Sch_k.

We call a queue *non-schedulable* if it will inevitably lead to a deadline miss provided that all tasks take their worst case computation times. Otherwise, a queue is said to be *schedulable*. The important observation for schedulable queues is that their length is bounded ([FKPY07]):

Lemma 1. *Given a task type set $\mathcal{P}$, one can effectively construct a natural number $B_{\mathcal{P}}$ such that $|q| \leq B_{\mathcal{P}}$ for all schedulable queues q.*

2.2 Decidability for Task Automata

A system, modeled by a task automaton and a scheduling function, can have the following three properties:

Preemption: The scheduler may insert a newly released task to the head of a non-empty queue, thus preempting all tasks already in the queue (a non-preemptive scheduler may insert only at other positions).

Variable task execution times: It may have task types $P_i(B_i, W_i, D_i)$ such that $B_i < W_i$, meaning that the task may non-deterministically finish execution at a time within the interval $[B_i, W_i]$.

Feedback: The precise finishing time of a task may influence the new task releases (by means of using x_{done} in a guard).

Note that the more of these properties are dropped for a task automaton, the "easier" the schedulability analysis problem becomes.

In [FKPY07], the schedulability problem is studied for single-processor systems. It is shown that (even with only one processor) schedulability becomes undecidable if a task automaton has all three properties. In turn, it was also shown that if there is no preemption, the problem becomes decidable. The same holds if there is no variable execution time. The open question remained, whether schedulability is decidable if x_{done} is not used in the guards for creating feedback. This last variant has also been proven decidable for certain types of schedulers in [FKPY07], but there is no result for the general case.

We study these questions for multi-processor scheduling. Since single-processor systems are just a special case of multi-processor systems, the negative result, namely that the schedulability becomes undecidable if a task automaton has all three properties, transfer to our setting. We show that the problem remains decidable for the following classes:

1. A non-preemptive scheduler (but possibly variable execution time and feedback) or
2. Fixed execution time tasks ($B = W$ for all task types, but possibly a preemptive scheduler and feedback).

On the other hand, we show that the schedulability analysis problem becomes undecidable for the third case:

3. No feedback (but possibly a preemptive scheduler and variable execution time).

3 Decidable Cases of Multiprocessor Scheduling

In this section, we show that the multiprocessor schedulability problem is decidable for the first two cases. The proof is done by reduction of this problem to the reachability problem for timed automata. For a given number of processors, a task automaton, and a multiprocessor scheduling policy, we construct a timed automaton with an error location such that the system is unschedulable if and only if the error location is reachable. Our construction is based on the construction for the single-processor case from [FKPY07] and [EWY99].

3.1 Fixed computation time

First, we handle the case where all tasks have fixed computation time (but the scheduler can be preemptive and there can be feedback from the scheduler back to the automaton).

Theorem 1. *The problem of checking whether a task automaton A with fixed computation times of tasks together with a multiprocessor scheduler Sch_k is schedulable, is decidable.*

Proof (Sketch). The given task automaton A is transformed into a standard timed automaton $E(A)$ by taking the underlaying timed automaton of A, removing the labels and adding new labels $\mathsf{release}_P$ on transitions where A would release an instance of task type P. Because of Definition 2 and Lemma 1, the whole scheduling strategy for schedulable queues can be encoded as a timed automaton $E(\mathsf{Sch}_k)$ (the *scheduler automaton*) as follows. The continuous part of the queues, namely the best/worst case computation times and remaining relative deadlines, are encoded in clocks. There are two clocks x_i^c and x_i^d for each task instance p_i in the queues, which measure how long this task instance has been computed and how long it has been released. The discrete parts of the queues (the order of the task instances) are encoded in locations of $E(\mathsf{Sch}_k)$. Since the queue length of schedulable queues is bounded (Lemma 1) and the number of the queues is fixed, there are only finitely many of locations needed. Once a queue becomes non-schedulable, $E(\mathsf{Sch}_k)$ will enter a dedicated error location. The edges and their guards correspond to the comparisons which the scheduling function uses for its decisions.

The values of b_i, w_i and d_i for each task instance p_i in the queue can be expressed using the clocks x_i^c and x_i^d, which is ensured by the construction in the following way. Whenever a new instance p_i of type P_j is released (event $\mathsf{release}_{P_j}$), the clock x_i^d is reset. Before a task instance p_i is put to the head of its queue (which models the beginning of the task execution), the values b_i, w_i and d_i from the queue can be expressed as B_j, W_j and $D_j - x_i^d$, respectively. The clock x_i^c is not used at all.

As soon as the task instance p_i is scheduled for execution for the first time, the clock x_i^c is reset, keeping track of its computation progress. For a running task instance p_i, the values b_i, w_i and d_i from its queue can be expressed as $B_j - x_i^c$,

$W_j - x_i^c$ and $D_j - x_i^d$, respectively. Therefore, this task instance may finish whenever "$B_j \leq x_i^c \leq W_j$" holds. To model task finishing, $E(\mathsf{Sch}_k)$ removes p_i from the queue and resets x_{done}. Whenever a constraint "$x_i^d > D_j$" is met for a released task instance p_i, an error location "non-schedulable" is entered.

Because of the preemption, tasks waiting in the queues may have already been executed for some time, but they are stopped now. Since it is not possible to stop their x_i^c clocks, the time for which a preempted task p_i was already computed is represented not just by its x_i^c clock, but by a difference $x_i^c - x_j^c$. Here, p_j is the task which directly preempted p_i. When a running task p_m finishes, all x_i^c of the preempted tasks p_i are updated by subtracting x_m^c, i.e., the computation time which was needed by p_m. In general, the reachability for timed automata with such updates is undecidable, but because of the fixed computation time property, $x_m^c = B_j = W_j$ is a constant natural number (assume that the type of p_m is P_j). Therefore, clocks in the resulting timed automaton $E(\mathsf{Sch}_k)$ (and in the product automaton $E(\mathsf{Sch}_k) \parallel E(A)$) are updated only by subtractions of integers. There is also a bound on the values of the clocks which are subtracted (the deadline), which makes the reachability problem for this type of automata (*Timed Automaton with Bounded Subtraction*) decidable, as proven in [FKPY07].

The product automaton $E(\mathsf{Sch}_k) \parallel E(A)$ described above is a timed automaton with bounded subtraction for which it holds that that the "non-schedulable" error state is reachable if and only if A is non-schedulable with scheduling strategy Sch_k.

The proof of this fact for the multi-processor case is the same as for the single-processor case in [FKPY07], with the difference that there can be several tasks running at the same timepoint. Therefore, it is necessary to check that Condition C_3 from the proof in [FKPY07], i.e., correctness of the invariant $b_i = B_m - (x_i^c - x_j^c)$, also holds here.

But from the fact that the scheduling function cannot move tasks from one queue to another one it follows that whenever a task instance p_i of type P_m is preempted by another instance p_j of type P_n, it will stay preempted (i.e., not being computed) until p_j finishes. Moreover, for p_j it holds that $b_j = B_n, w_j = W_n$ at the timepoint when it preempts p_i. This together implies the correctness of C_3 and thus also of the updates at the end of preemption.

□

Note that here, the diagonal-freeness in guards from the scheduler (see Definition 2) is important, because it is necessary to use clock differences to express the computation time of tasks. (A comparison of a clock difference to a constant is already a diagonal constraint which cannot be further extended by another clock.)

3.2 Non-preemptive scheduler

In this case, tasks are not allowed to preempt other already running tasks, which makes handling the computation time of all tasks in the queues easier. Therefore, even variable computation time of tasks can be allowed.

Theorem 2. *The problem of checking whether a task automaton together with a non-preemptive multi-processor scheduler is schedulable, is decidable.*

Proof (Sketch). Here, the same construction as above is used, i.e., two automata $E(A)$ and $E(\mathsf{Sch}_k)$ are created. Again, $E(\mathsf{Sch}_k)$ keeps track of computation progress and time pass since the release in clocks x_i^c and x_i^d for each released task p_i.

Whenever a new instance p_i of type P_j is released (event $\mathsf{release}_{P_j}$), the clock x_i^d is reset. Before a task instance p_i is put to the head of a queue, the values b_i, w_i and d_i from the queue can be expressed as B_j, W_j and $D_j - x_i^d$, respectively. The clock x_i^c is not used at all.

As soon as the task instance p_i is scheduled for execution, the clock x_i^c is reset, keeping track of the computation progress of the task. For a running task instance p_i, the values b_i, w_i and d_i from its queue can be expressed as $B_i - x_i^c$, $W_i - x_i^c$ and $D_i - x_i^d$, respectively.

Because the scheduler is non-preemptive and the scheduling function cannot move tasks from one queue to another one, only the running task in each queue has already started its computation. The tasks which are not at the head of some queue did not start their computations yet. For this reason, we do not need to use x_i^c of the waiting tasks to compute b_i and w_i.

□

Note that for this result, the definition of the scheduling function (Definition 2) does not have to be that restrictive in the sense that also diagonal constraints can be allowed for the decisions in the scheduling function. The reason is that the computation time and deadline values in the queues are encoded into (at most) one clock each. The possibility of using diagonal constraints in the scheduler's decisions makes (for the non-preemptive case) encoding even of the Least Slack First scheduling policy possible [But97].

4 Undecidability

In this section we show, that the schedulability problem is undecidable for multiprocessor systems with at least two scheduling queues. This holds even if the precise task finishing times cannot influence releases of new tasks.

Theorem 3. *The problem of checking whether a task automaton using a k-multi-processor scheduler without feedback is schedulable, is undecidable for $k \geq 2$.*

To develop the proof, we first sketch a construction used in [FKPY07] for the single-processor case, where the automaton was allowed to use task feedback. We then extend the result to the multi-processor case, even if no task feedback is involved.

4.1 Undecidability on single-processor using feedback

The undecidability proof for the general single-processor schedulability problem is done by a reduction from the halting problem for two-counter machines. A *two-counter machine* consists of a finite state control unit and two unbounded non-negative integer counters. The three possible instructions are counter-increment, counter-decrement and conditional branching (checking whether a counter value is zero). After each step, the machine state is changed deterministically. One of the states is a dedicated *halt state*. It is known, that the problem whether this halt state is reachable (the *halting problem for two-counter machines*) is undecidable.

The idea is, given a two-counter machine M, to construct a task automaton A_M and a scheduling policy such that a dedicated halt location in A_M is reachable if and only if the halt state of M is reachable. It will be ensured that no task can miss its deadline as long as the halt location is not visited. In turn, the queue can unboundedly grow in the halt location, making tasks miss their deadlines. The result of these two properties is that A_M will be non-schedulable if and only if M can reach its halt state.

For each state of M's control unit there is one corresponding location l_i in A_M. These locations are connected depending on the operation which M would execute at the corresponding state (increment, decrement, branching), through auxiliary locations "executing" this instruction.

To encode the counters into clock values, an N*-wrapping* construction from [HKPV98] is used. All clocks x stay within the interval $[0, N]$ for a constant N by resetting each clock x as soon as $x = N$ (*wrapping reset*). For a dedicated system clock x_{sys} these are the only resets (which makes x_{sys} periodic). In this way, *wrapping values* for all other clocks can be defined as their values at the (periodic) times where $x_{sys} = 0$. Thus, between any two consecutive *non-wrapping resets* of the clock x (i.e., when $x < N$), its wrapping value does not change.

Using this construction, the value v of a counter C can be kept as the wrapping-value 2^{1-v} of a clock x_C. Therefore, this wrapping-value of x_C is always smaller than or equal to 2. The conditional branching is done by directly comparing the value of x_C to 2, and the increment (decrement) operation is implemented as dividing (multiplying) the wrapping value of x_C by 2, respectively. For the implementation of the decrement, a task Q with fixed computation time is released at a non-deterministically chosen time and a clock x_{Cnew} is reset at the same time. This task is then preempted by a task P of higher priority with variable execution time. P is released when x_C is reset to zero by a wrapping edge. A guard with $x_{done} = 0$ is used to check if its execution time is equal to the wrapping value of x_C, i.e., if it finishes when x_{sys} is reset. This preemption is repeated, and by using the $x_{done} = 0$ guard again afterwards to check that Q finishes when x_{sys} is reset, the wrapping-value of x_{Cnew} is forced to be twice bigger than the wrapping value of x_C. The response time of Q is a constant time (its computation time) plus two times the wrapping value of x_C – because of the two preemptions from P. If any step in the construction fails (some non-deterministic choice was wrong), the automaton enters a sink location where no task is released. Figure 1 illustrates the whole decrement procedure.

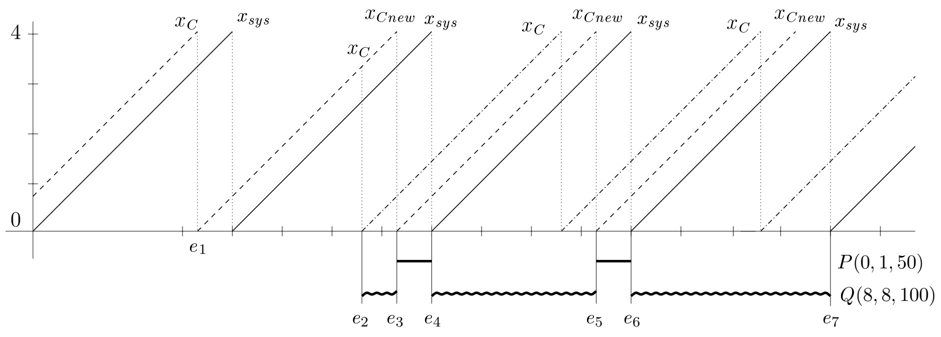

Figure1. Time chart of the doubling procedure using the *N-wrapping* construction.

An increment for C needs to halve x_C. To achieve this, the wanted new value is simply guessed in a clock x_{Cnew} and then checked using the above decrement procedure. Only if the double of x_{Cnew} is x_C, the computation will continue.

During all three operations, all used tasks meet their deadline (using a fixed priority scheduler) and therefore the automaton is schedulable as long as it executes these instructions. A special location l_{halt} is used to represent the halt state of M. This location releases the task Q and it has an unguarded selfloop. Therefore, if A_M reaches l_{halt}, it will be able to release an unbounded amount of task instances at the same timepoint, making the system unschedulable.

Lemma 2 ([FKPY07]). *For a given two-counter machine M, the constructed task automaton A_M is schedulable with the fixed priority scheduler if and only if M halts.*

Note that the construction uses all three properties (preemption, variable execution time, feedback) given in Section 2.

4.2 Undecidability on multi-processor without feedback

We extend this result to the multi-processor case for the systems with preemptive schedulers, variable computation time of tasks, but the finishing time of a task is never tested in the guards of a task automaton, that is, task feedback is not allowed (x_{done} is never used in guards). To achieve the same effect as task feedback, we develop two mechanisms based on the status of task queues representing the status of the processors. First, we describe how the guards $x_{done} = 0$ can be replaced by certain task releases so that the system is still able to detect if the task in question finished at the right time. Secondly, a construction will be given to store this information in a task queue.

Both mechanisms will ensure that the demand for task feedback in the construction is removed. There is no branching upon the fact whether $x_{done} = 0$, the automaton continues its computation even if some non-deterministic choice of the system is wrong. However, to store the information about the fact that all non-deterministic choices are correct so far, at least one additional processor with its own queue is needed.

Checking the precise finishing time. The construction of A_M for a given two-counter machine M is exactly the same as before, but we change three aspects. Note that we are now in the multi-processor scenario with at least two queues q_1 and q_2. The tasks P and Q used in the construction above will be scheduled to q_1, P always preempts Q.

To check the finishing times of P and Q, two tasks T^P_{chk1} and T^P_{chk2} are released at the same time point (when we expect P or Q to finish). The scheduler then detects, if the task P finishes (is removed from the queue) between both task releases. Future decisions of the scheduler (like scheduling tasks in the halt location in an schedulable or unschedulable order) can be based on this. The construction works in detail as follows.

First, at each position where an edge contains the guard $x_{done} = 0$ for synchronizing on the finishing of a task P, remove the guard and add two additional locations l_1 and l_2. The first one releases T^P_{chk1}, the second one T^P_{chk2}. Both released instances will be scheduled to the queue q_2. The original edge which contained $x_{done} = 0$ will go to l_1. The edges from l_1 to l_2 and from l_2 to the old location will contain guards ensuring that the automaton does not delay in l_1 and l_2. The whole fragment is depicted in Figure 2.

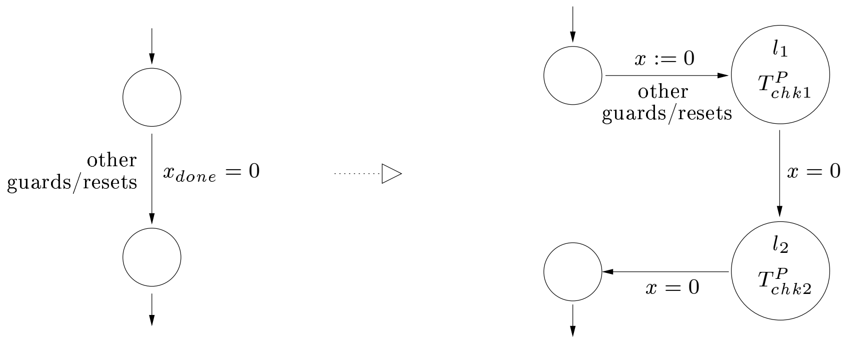

Figure2. Replacing the check for $x_{done} = 0$ with two additional locations l_1 and l_2 and an additional clock x.

Secondly, the scheduler can use the task releases of T^P_{chk1} and T^P_{chk2} to check if P stayed in the queue until that time point, but not longer. If P *is* in the queue at the release of the first "checking task" T^P_{chk1} and if it *is not* in the queue when the second "checking task" T^P_{chk2} is released, the scheduler knows that it has finished at the correct timepoint. Such a successful scenario is described in Figure 3. If it is not the case, then the automaton cannot guarantee that P has finished at the correct timepoint and thus it cannot guarantee that the simulation of the two-counter machine is correct.

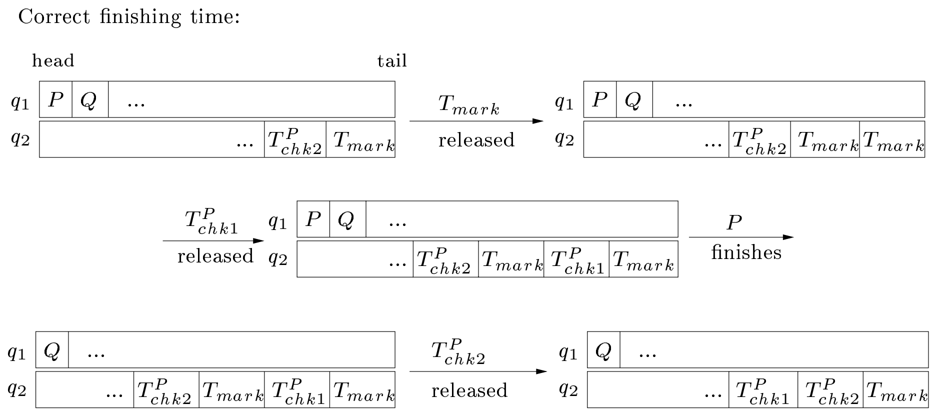

Figure3. Releases of tasks T^P_{chk1}, T^P_{chk2}, and T_{mark} detect that the finishing time of P is correct.

Using the queue as a memory cell. The remaining problem is to remember the information that all finishing times were correct so far. Note, that it is impossible to send this information back to the automaton and that the scheduling function is stateless. However, we now describe how the second queue q_2 can be used to remember this.

Except for T^P_{chk1} and T^P_{chk2}, there will be also tasks of an additional type T_{mark} released by the automaton and put into q_2 in such a way that the utilization of the second processor is 100%. Directly before (and at the same time of) the release of T^P_{chk1}, an instance of T_{mark} is released and put to the end of q_2. If the following release of T^P_{chk1} detects an early finishing (P is not in the queue q_1), the scheduler puts T^P_{chk1} directly *after* T_{mark} in q_2, otherwise directly *before* T_{mark}. The same holds for T^P_{chk2}, if the scheduler detects a late task finishing (P is still in q_1). In this way, the fact that M has been simulated correctly so far is equivalent to the fact that the last task in q_2 is of type T_{mark}.

In case that this checking discovered that the simulation was not correct (the last task in q_2 is not T_{mark} as the result of the checking procedure), we want to remember this information for the rest of the computation. We encode it by the fact that after any further checking, the last task in q_2 will not be T_{mark}. In the following, we explain how does this information get propagated from one checking to another.

All task types T^P_{chk1}, T^P_{chk2} and T_{mark} have fixed computation times. Since the checking times for each instruction are known in advance, it is possible to adjust these computation times so that the processor is never idle, but also no deadline is missed. Therefore, when new tasks T^P_{chk1}, T^P_{chk2} and T_{mark} arrive to the queue, the task instance from the previous checking which was scheduled as last is still in the queue. The scheduler can then take into consideration whether it is T_{mark}. If it is the case, the scheduler inserts the new tasks into q_2 as described above. Otherwise, the scheduler just makes sure that the last task is not T_{mark}. Therefore, if A_M does not cheat during the whole computation (all task instances finish when we wanted them to finish) then all checks with T^P_{chk1} and T^P_{chk2} invocations are successful and the last task in q_2 is T_{mark}.

The last thing which is changed from the single-processor case with feedback is the halt location. Here, instead of a selfloop releasing unboundedly many task instances, two tasks $R_1(1, 1, 1)$ and $R_2(1, 1, 2)$ are released at the same time, and then the automaton enters a sink location where no tasks are released anymore, as depicted in Figure 4.

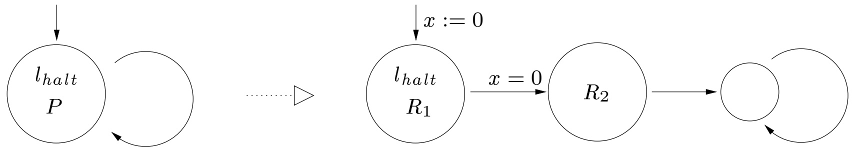

Figure4. The new encoding of the halt state.

The scheduler puts both instances to the head of the same queue and lets R_2 be computed first, making R_1 miss its deadline, if and only if A_M simulated M correctly, i.e., the last task in q_2 is T_{mark}. In this way, the automaton A_M will only be non-schedulable, if it can reach l_{halt} without cheating at the task finishing transitions. The following lemma states the correctness of the construction.

Lemma 3. *For a given two-counter machine M, the task automaton A_M is not schedulable with the constructed multi-processor scheduler if and only if M halts.*

Proof (Sketch). If the two-counter machine M reaches the halt state then A_M can simulate the same instructions leading to the halt location. During the whole (correct) simulation, the executions of P and Q always finish at the correct timepoint, i.e., when x_{sys} is reset. This keeps the wrapping values of the clocks x_C and x_D correctly encoding the values of the counters C and D, respectively, and q_2 is in a "not cheated" state all the time. When A_M then enters its halt location, R_1 and R_2 will be executed in an order which makes the queue unschedulable.

If M does not reach its halt state then A_M has to cheat to reach its halt location, which means that the following holds for the task instances of P and Q. There is an instance of P or Q which does not finish at the time when

x_{sys} is reset. Consequently, this cheating of A_M is detected by the scheduler through releases of T^P_{chk1} and T^P_{chk2}. From this timepoint on, the last task in q_2 will not be T_{mark}. This means that A_M can only reach the halt location with q_2 expressing that a task finished at a wrong time, making the scheduler execute R_1 before R_2, which means that both of them meet their deadlines. Provided that during the run up to this point all tasks met their deadlines (which is ensured by construction), the automaton A_M is schedulable. Also, no task misses its deadline along a run of A_M which does not reach the halt location.

5 Conclusion

We have shown that the schedulability problem stays decidable in the multi-processor setting for the classes of task automata with a non-preemptive scheduling strategy or with fixed computation times of tasks. On the negative side, the problem turns out to be undecidable for preemptive multi-processor schedulers when the computation times of tasks may vary within an interval. It is still an open problem, whether this problem is decidable in the single-processor setting. As a future work, will try to close this decidability gap.

References

[ABJ01] Bjorn Andersson, Sanjoy Baruah, and Jan Jonsson. Static-priority scheduling on multiprocessors. In *Proc. of RTSS '01*, page 93. IEEE Computer Society, 2001.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[BCL05] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *Proc. of ECRTS '05*, pages 209–218. IEEE Computer Society, 2005.

[But97] G. C. Buttazzo. *Hard Real-Time Computing Systems. Predictable Scheduling Algorithms and Applications.* Kulwer Academic Publishers, 1997.

[EWY99] C. Ericsson, A. Wall, and W. Yi. Timed automata as task models for event-driven systems. In *Proc. of RTCSA'99.* IEEE Computer Society, 1999.

[FKPY07] E. Fersman, P. Krcal, P. Pettersson, and Wang Yi. Task automata: Schedulability, decidability and undecidability. *Information and Computation*, 2007. To appear.

[FMPY06] E. Fersman, L. Mokrushin, P. Pettersson, and W. Yi. Schedulability analysis of fixed priority systems using timed automata. *Theoretical Computer Science*, 354(2), March 2006.

[HKPV98] T.A. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.

[Liu00] Jane W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[LL73] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[RC01] Jorge Real and Alfons Crespo. Offsets for scheduling mode changes. In *Proc. of ECRTS'01*, pages 3–10. IEEE Computer Society, 2001.