Gang FTP scheduling of periodic and parallel rigid real-time tasks

Joël Goossens, Vandy Berten

Gang FTP scheduling of periodic and parallel rigid real-time tasks

Joël Goossens Vandy Berten
Université libre de Bruxelles (U.L.B.)
CP212, 50 av. F.D. Roosevelt
1050 Brussels, Belgium
{joel.goossens,vandy.berten}@ulb.ac.be

Abstract

In this paper we consider the scheduling of periodic and parallel rigid tasks. We provide (and prove correct) an exact *schedulability test for Fixed Task Priority (FTP) Gang scheduler sub-classes: Parallelism Monotonic, Idling, Limited Gang, and Limited Slack Reclaiming. Additionally, we study the predictability of our schedulers: we show that Gang FJP schedulers are not predictable and we identify several sub-classes which are actually predictable. Moreover, we extend the definition of rigid, moldable and malleable jobs to recurrent tasks.*

1. Introduction

We consider the preemptive scheduling of real-time tasks on identical multiprocessor platforms (see [2, 1]). We deal with *parallel* real-time tasks, the case where each job may be executed on different processors *simultaneously*, i.e., we allow *job parallelism*. Nowadays, the design of parallel programs is common thanks to parallel programming paradigms like Message Passing Interface (MPI [13, 14]) or Parallel Virtual Machine (PVM [19, 12]). Even better, sequential programs can be parallelized using tools like OpenMP (see [5] for details).

Related Work. Few models and results in the literature concern hard real-time and parallel tasks. Manimaran et al. in [18] consider the *non-preemptive* EDF scheduling of periodic tasks. Han et al. in [16] considered the scheduling of a (finite) set of real-time jobs allowing job parallelism while we consider the scheduling of either infinite set of jobs or equivalently a set a periodic tasks. In previous work we contributed mainly to the *feasibility* problem of parallel tasks. In [6] we provided a task model which integrates job parallelism. We proved that the time-complexity of the feasibility problem of these systems is linear relatively to the number of (sporadic) tasks. More recently, we considered the scheduling of jobs which are composed of phases to be executed sequentially ; in [3] we provided a *necessary* feasibility test. Regarding the *schedulability* of recurrent real-time tasks, and to the best of our knowledge, we can only report the S. Kato et al. work (see [17] for details) which considers the Gang scheduling of rigid tasks (the number of processors is fixed beforehand) and provides a *sufficient* schedulability condition for Gang EDF scheduling.

This Research. In this paper we study the scheduling of recurrent and parallel rigid tasks. Our main contribution is an *exact* schedulability test for Fixed Task Priority (FTP) Gang schedulers. Additionally, we study the predictability of our schedulers: we show that Gang FJP schedulers are not predictable and we identify several sub-classes which are actually predictable. Our technique is based on previous work for the scheduling of periodic tasks upon multiprocessors [8, 7]. To summarize the technique, we characterized for FTP schedulers and for asynchronous periodic task models, *upper bounds* of the first time-instant where the schedule repeats. Based on the upper bounds and the *predictability* property, we provide *exact* schedulability tests for asynchronous constrained deadline periodic task sets. The predictability property is important in the technique and will be revisited in this work. We also extend the definition of rigid, moldable and malleable jobs to *recurrent* tasks.

Paper Organization. This paper is organized as follows. Section 2 introduces definitions, the model of computation and our assumptions. In Section 3 we study the predictability of our system, in particular we show that Gang FJP schedulers are not predictable and we identify several sub-classes which are actually predictable. We prove the periodicity of feasible schedules of periodic systems in Section 4. In Section 5 we combine the periodicity and predictability properties, to provide, for our Gang FTP sub-classes, an *exact* schedulability test. Lastly, we conclude in Section 6.

2. Model and Definitions

2.1. Parallel Terminology

The parallel literature [10, 4, 11] defines several kind of parallel *tasks*. But *tasks* in the non real-time parallel

terminology does not have the same meaning as *tasks* in real-time scheduling literature. Actually, *tasks* in the parallel literature corresponds to *jobs* in our real-time community (i.e., corresponds to task *instance*). Especially, the notion of rigid recurrent task is not defined and does not extend trivially from the non real-time literature, in this section we fill the gap.

Definition 1 (Rigid, Moldable and Malleable Job)**.** *A* job *is said to be:*

Rigid *if the number of processors assigned to this job is specified externally to the scheduler a priori, and does not change throughout its execution;*

Moldable *if the number of processors assigned to this job is determined by the scheduler, and does not change throughout its execution;*

Malleable *if the number of processors assigned to this job can be changed by the scheduler during the job's execution.*

Definition 2 (Rigid, Moldable and Malleable Recurrent Task)**.** *A periodic/sporadic* task *is said to be:*

Rigid *if all its jobs are rigid, and the number of processors assigned to the jobs is specified externally to the scheduler;*

Moldable *if all its jobs are moldable;*

Malleable *if all its jobs are malleable.*

Notice that a rigid task does not necessarily have jobs with the same size. For instance, if the user/application decides that odd instances require v processors, and even instances v' processors, the task is said to be rigid.

2.2. Task and Job Model

We consider the preemptive scheduling of parallel jobs on a multiprocessor platform with m processors. We will focus on the problem of scheduling a set of parallel jobs, each job $J_j \stackrel{\text{def}}{=} (r_j, v_j, e_j, d_j)$ is characterized by a release time r_j, v_j a required number of processors, an execution requirement e_j and an absolute deadline d_j. The job J_j must execute for e_j time units over the interval $[r_j, d_j)$ on v_j processors. We consider the scheduling of *rigid* tasks since v_i is fixed externally to the scheduler. Actually, S. Kato et al. in [17] have the same assumption as we do, and, given the Definition 2, they consider *rigid* tasks, and not *moldable* tasks, as said in their paper — otherwise the scheduler would determine v_i on-line and at job-level.

As we will consider periodic systems, let $\tau = \{\tau_1, \ldots, \tau_n\}$ denote a set of n periodic parallel tasks. Each task $\tau_i = (O_i, v_i, C_i, D_i, T_i,)$ will generate an infinite number of jobs, where the k^{th} job of task τ_i is

$$(O_i + (k-1)T_i, v_i, C_i, O_i + (k-1)T_i + D_i).$$

The execution requirement of a job of τ_i corresponds as a $C_i \times v_i$ *rectangle*. In this document we assume $D_i \leq T_i$ for any τ_i, i.e., we consider constrained deadline systems. We consider multiprocessor platforms π composed of m identical processors: $\{\pi_1, \pi_2, \ldots, \pi_m\}$.

2.3. Priority Assignment and Schedulers

In this document we consider FTP and FJP schedulers with the following definitions.

Definition 3 (FTP)**.** *A priority assignment is a* Fixed Task Priority *assignment if it assigns the priorities to the tasks beforehand; at run-time each job priority corresponds to its task priority. (An FTP scheduler uses an FTP priority assignment.)*

We assume that tasks are indexed according to priority (lower the index, higher the priority).

Definition 4 (FJP)**.** *A priority assignment is a* Fixed Job Priority *assignment if and only if it satisfies the condition that: for every pair of jobs J_i and J_j, if J_i has higher priority than J_j at some time-instant, then J_i always has higher priority than J_j. (An FJP scheduler uses an FJP priority assignment.)*

Remark that any FTP assignment is also FJP.

Definition 5 (Gang FJP)**.** *At each instant, the algorithm schedules jobs on processors as follows: the highest priority (active) job J_i is scheduled on the first v_i available processors (if any). The very same rule is then applied to the remaining active jobs on the remaining available processors.*

Notice that the Definition 5 defines also the particular case of Gang FTP.

Priority inversion. Figure 1 illustrates the Gang FTP schedule (τ_1 is the highest priority task and τ_3 the lowest one) of $\tau_1 = (0, 2, 2, 5, 5), \tau_2 = (0, 2, 3, 5, 5), \tau_3 = (0, 1, 4, 5, 5)$. Notice that Gang FJP and FTP can produce schedules where a lower-priority job (J_j) is scheduled while an active higher-priority job (J_i) is not (typically if $v_i > v_j$ — which occurs at time 0 in our example, τ_2 and τ_3 are active, τ_3 is executing in $[0, 2)$ while τ_2 is not). This phenomenon, called *priority inversion* in this document, could be a drawback as we will see. Fortunately, we keep an important FTP-property: the scheduling of the sub-set $\{\tau_1, \ldots, \tau_i\}$ is not jeopardized by lower-priority tasks ($\{\tau_{i+1}, \ldots, \tau_n\}$).

Definition 6 (Schedule $\sigma(t)$)**.** *For any set of jobs $J \stackrel{\text{def}}{=} \{J_1, J_2, J_3, \ldots\}$ and any set of m identical processors $\{\pi_1, \ldots, \pi_m\}$ we define the* schedule $\sigma(t)$ *of system τ at time-instant t as $\sigma : \mathbb{N} \to \mathbb{N}^m$ where $\sigma(t) \stackrel{\text{def}}{=} (\sigma_1(t), \sigma_2(t), \ldots, \sigma_m(t))$ with*

$$\sigma_j(t) \stackrel{\text{def}}{=} \begin{cases} 0, & \text{if there is no job scheduled on } \pi_j \\ & \text{at time-instant } t; \\ i, & \text{if job } J_i \text{ is scheduled on } \pi_j \text{ at} \\ & \text{time-instant } t. \end{cases}$$

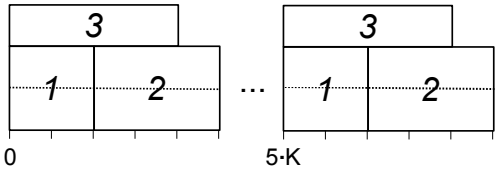

Figure 1. Gang FTP schedule with priority inversion at time 0.

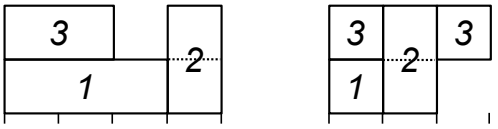

Figure 2. Non-predictability of Gang FJP schedulers. $J_1 > J_2 > J_3$, and they all arrive at time 0.

Definition 7 (State of the system $\theta(t)$)**.** *For any task system $\tau = \{\tau_1, \ldots, \tau_n\}$ we define the* state $\theta(t)$ *of the system τ at time-instant t as $\theta : \mathbb{N} \to (\mathbb{Z} \times \mathbb{N} \times \mathbb{N})^n$ with $\theta(t) \stackrel{\text{def}}{=} (\theta_1(t), \theta_2(t), \ldots, \theta_n(t))$ where*

$$\theta_i(t) \stackrel{\text{def}}{=} \begin{cases} (-1, x, 0), & \text{if no job of task } \tau_i \text{ was activated before or at } t. \text{ In this case, } x \text{ time units remain until the first activation of } \tau_i. \text{ We have } 0 < x \leq O_i. \\ (y, z, u), & \text{otherwise. In this case, } z \text{ is the time elapsed at time-instant } t \text{ since the arrival of the oldest active job of } \tau_i. \text{ If there are } y \neq 0 \text{ active jobs of } \tau_i \text{ then } u \text{ units were already executed for the oldest active job. If } y = 0, \text{ there is no active job for } \tau_i \text{ at } t, u \text{ is undefined in this case.} \end{cases}$$

Definition 8 (Availability of the processors)**.** *For any ordered set of jobs J and any set of m processors $\{\pi_1, \ldots, \pi_m\}$, we define the* availability of the processors $A(J, t)$ *of the set of jobs J at time-instant t as the set of available processors: $A(J, t) \stackrel{\text{def}}{=} \{\pi_j \mid \sigma_j(t) = 0\}$, where σ is the schedule of J.*

Definition 9 (Active, Ready and Running jobs)**.** *A job is said to be* active *if it has been released, but is not finished yet. An active job is* ready *if it is not currently served ; an active job is* running *otherwise.*

3. Predictability of Gang Scheduling

We consider the scheduling of sets of job $J \stackrel{\text{def}}{=} J_1, J_2, J_3 \ldots$, (finite or infinite set of jobs) and without loss of generality we consider jobs in a decreasing order of priorities ($J_1 > J_2 > J_3 > \cdots$). We suppose that the execution time of each job J_i can be any value in the interval $[e_i^-, e_i^+]$ and we denote by J_i^+ the job defined as $J_i^+ \stackrel{\text{def}}{=} (r_i, v_i, e_i^+, d_i)$. We denote by $J^{(i)}$ the set of the first i higher priority jobs. We denote also by $J_-^{(i)}$ the set $\{J_1^-, \ldots, J_i^-\}$ and by $J_+^{(i)}$ the set $\{J_1^+, \ldots, J_i^+\}$. Let $S(J)$ be the time-instant at which the lowest priority job of J begins its execution in the schedule. Similarly, let $F(J)$ be the time-instant at which the lowest priority job of J completes its execution in the schedule.

Definition 10 (Predictable algorithms [15])**.** *A scheduling algorithm is said to be* predictable *if $S(J_-^{(i)}) \leq S(J^{(i)}) \leq S(J_+^{(i)})$ and $F(J_-^{(i)}) \leq F(J^{(i)}) \leq F(J_+^{(i)})$, for all $i \geq 1$ and for all schedulable $J_+^{(i)}$ sets of jobs.*

Notice that the predictability of an algorithm implies that any system schedulable when all tasks use their worst case execution time is also schedulable when a task takes less time than expected. We then just need to show that the system is schedulable in the worst case to prove that the system is schedulable in all scenarios.

In previous work [9] we proved, for a quite general model, i.e., FJP priority schedulers on unrelated multiprocessors, the predictability for *sequential* jobs (i.e., $v_i = 1$ for any J_i). Unfortunately that property is not satisfied for parallel jobs.

Lemma 11. *Gang FJP schedulers are not predictable on multiprocessors.*

Proof. Here is an example task system, on 2 processors (see Figure 2):

$$J_1 = (0, 1, 3, 3), J_2 = (0, 2, 1, 4), J_3 = (0, 1, 2, 2) \ .$$

Upon two processors and using the priority assignment $J_1 > J_2 > J_3$, Gang FJP schedules the set of jobs (J_3 completes at time-instant 2). Unfortunately, if the actual duration of J_1 is 1, J_2 will preempt J_3 at time $t = 1$ and J_3 will complete *later*, at time-instant 3. Then, J_3 does not miss its deadline in the "worst case" scenario, but misses it if J_1 uses less than its worst case execution time e_1. □

This negative result implies that neither the DM, RM nor EDF are predictable for Gang scheduling.

The problem we highlight in this example occurs because some jobs which were not preempted in the worst case scenario are preempted in a scenario with shorter execution times. In other words, by allowing a job J_i taking advantage of some slack time given by another (higher priority) job, J_i interrupts a job J_j (with $J_j < J_i$) which would not have been suspended if we did not have any slack.

If, as we will do and prove in this work, we find a way to avoid the priority inversion phenomenon, we will never have those problematic preemptions. Indeed, if no lower priority job J_j is authorized to start between the arrival of J_i ($J_j < J_i$) and its start time, then if J_i starts anywhere between its arrival time, and its start time in the worst case scenario, it will not interrupt tasks that it would not have interrupt in the worst case.

The "problematic preemptions" can be avoided by (at least) three ways:

- By avoiding the priority inversion;
- By avoiding any slack (or by not using it);
- By using the slack, but in a "smart" way.

In order to obtain a predictable system, we identify two ways of modifying the system:

- First, we will propose to constraint the priority assignment. We introduce the *Parallelism Monotonic FTP assignment*, and prove the predictability of these priority assignments;
- Second, we will propose three variants of the scheduler, giving a predictable behavior. Those variants are the *idling scheduler* (not using the slack), the *limited Gang FJP scheduler* (avoiding priority inversion), and the *Gang FJP scheduler with limited slack reclaiming* (smartly using the slack).

3.1. Parallelism Monotonic FTP Assignment

In this section we will consider a sub-class of Gang FTP assignments which are predictable.

Definition 12 (Parallelism Monotonic)**.** *An FTP priority assignment is* Parallelism Monotonic *iff* $i < j \Rightarrow v_i \leq v_j$.

Notice that this class is very interesting from the theoretical point of view, but might be not a good choice for some implementation. Indeed, it gives a low priority to highly parallel jobs, which makes them more difficult to schedule. In general, it might be useful to first schedule the very parallel jobs, and then to fill the available processors with smaller jobs.

We will now prove that any Parallelism Monotonic assignment are predictable.

In [9] we showed that $A(J_+^{(i)}, t) \subseteq A(J^{(i)}, t)$, for all t and all i. In other words, that at any time-instant the processors available in $\sigma_+^{(i)}$ are also available in $\sigma^{(i)}$. The counterexample used in the proof of Lemma 11 violates that property as well. In the following we will consider another kind of processors availability.

Definition 13 (Level-(i) availability of the processors)**.** *For any ordered set of* $i-1$ *jobs* $J = J_1, \ldots, J_{i-1}$ *and any set of* m *processors, we define the* level-(i) availability of the processors $A_i(J, t)$ *of the set of jobs* J *at time-instant* t *as follows*

$$A_i(J,t) \stackrel{\text{def}}{=} \begin{cases} \#A(J,t) & \text{if } \#A(J,t) \geq v_i; \\ 0 & \text{otherwise.} \end{cases}$$

Informally speaking, $A_i(J, t)$ is the the number of available processors if the latter is sufficient to schedule J_i, otherwise $A_i(J, t)$ is null.

Lemma 14. *For any schedulable ordered set of jobs* J*, using a Gang FJP Parallelism Monotonic on* m *processors, we have* $A_i(J_+^{(i-1)}, t) \leq A_i(J^{(i-1)}, t)$*, for all* t *and all* i*. (We consider that the sets of jobs are ordered in the same decreasing order of the priorities, i.e.,* $J_1 > J_2 > \cdots > J_\ell$ *and* $J_1^+ > J_2^+ > \cdots > J_\ell^+$*.)*

Proof. The proof is made by induction on i (the number of jobs). Our inductive hypothesis is the following: $A_k(J_+^{(k-1)}, t) \leq A_k(J^{(k-1)}, t)$, for all t and $1 < k \leq i+1$.

The property is true in the base case since $A_2(J_+^{(1)}, t) \leq A_2(J^{(1)}, t)$, for all t. Indeed, $S(J^{(1)}) = S(J_+^{(1)})$. Moreover J_1 and J_1^+ are both scheduled on the (same) first v_1 processors, but J_1^+ will be executed for the same or a greater amount of time than J_1.

We will show now that $A_{i+2}(J_+^{(i+1)}, t) \leq A_{i+2}(J^{(i+1)}, t)$, for all t.

Since the jobs in $J^{(i)}$ have higher priority than J_{i+1}, then the scheduling of J_{i+1} will not interfere with higher priority jobs which have already been scheduled. Similarly, J_{i+1}^+ will not interfere with higher priority jobs of $J_+^{(i)}$ which have already been scheduled. Therefore, we may build the schedule $\sigma^{(i+1)}$ from $\sigma^{(i)}$, such that the jobs $J_1, J_2, \ldots, J_i$, are scheduled at the very same instants and on the very same processors as they were in $\sigma^{(i)}$. Similarly, we may build $\sigma_+^{(i+1)}$ from $\sigma_+^{(i)}$.

Note that the property is straightforward for time-instants where $J^{(i+1)}$ is not scheduled since the processor availability is not modified and by definition of level-$(i+2)$ processor availability.

We will consider time-instant t, from r_{i+1} to the completion of J_{i+1} (which is actually not after the completion of J_{i+1}^+, see below for a proof), we distinguish between three cases:

1. $A_{i+1}(J_+^{(i)}, t) = A_{i+1}(J^{(i)}, t) = 0$: in both situations not enough processors are available for $J_{(i)}$ (and $J_{(i)}^+$). Therefore, both jobs, J_{i+1} and J_{i+1}^+, do not progress and we obtain $A_{i+2}(J_+^{(i+1)}, t) = A_{i+2}(J_+^{(i)}, t) = A_{i+2}(J^{(i+1)}, t) = A_{i+2}(J^{(i)}, t) = 0$, since $v_{i+2} \geq v_{i+1}$. The progression of J_{i+1} is identical to J_{i+1}^+.
2. $0 = A_{i+1}(J_+^{(i)}, t) < v_{i+1} \leq A_{i+1}(J^{(i)}, t)$: J_{i+1} progress on the v_{i+1} first available processors in

$A(J^{(i)},t)$ (not available in $A(J_+^{(i)},t)$). J_{i+1}^+ does not progress. $A_{i+2}(J_+^{(i)},t)=0$ since $v_{i+2}\geq v_{i+1}$. The progression of J_{i+1} is strictly larger than J_{i+1}^+.

3. $v_{i+1}\leq A_{i+1}(J^{(i)},t)\leq A_{i+1}(J_+^{(i)},t)$, J_{i+1} and J_{i+1}^+ progress on the same processors. The property follows by induction hypothesis.

Therefore, we showed that $A_{i+2}(J_+^{(i+1)},t)\leq A_{i+2}(J^{(i+1)},t)$, for all t, from r_{i+1} to the completion of J_{i+1} and that J_{i+1} does not complete after J_{i+1}^+. For the time-instant after the completion of J_{i+1} the property is trivially true by induction hypothesis. □

Theorem 15. *Gang FJP schedulers are predictable on identical platforms with Parallelism Monotonic priority assignment.*

Proof. In the framework of the proof of Lemma 14 we actually showed extra properties which imply that Gang FJP Parallelism Monotonic schedulers are predictable on identical platforms: (i) J_{i+1} completes not after J_{i+1}^+ and (ii) J_{i+1} can be scheduled either at the very same instants as J_{i+1}^+ or may progress during *additional* time-instants (case (2) of the proof) these instants may precede the time-instant where J_{i+1}^+ commences its execution. □

3.2. Idling Scheduler

Instead of giving constraints on the priority assignment, we can also adapt our scheduler in order to make it predictable. A first way of doing that is to force tasks to run *exactly* up to their worst case. If a task does not use its worst case, then the scheduler idles the processor(s) up to the expected end time.

Definition 16 (Idling scheduler)**.** *An* idling scheduler *idles any processor that was used by a task which finished earlier than its worst case, up to the time the processor would have been released in the worst case scenario.*

Lemma 17. *Gang FJP Idling schedulers are predictable on identical platforms.*

Proof. The proof is very straightforward in this case: any task starts at the same time in the worst case J^+, and in the case where some tasks use less than than their worst case. And if we consider the completion time of a job as the time at which its (possibly empty) idle period finishes, then the end time will be the same in the worst case scenario as in any case. Then,

$$S(J_-^{(i)})\leq S(J^{(i)})\leq S(J_+^{(i)}),$$

and

$$F(J_-^{(i)})\leq F(J^{(i)})\leq F(J_+^{(i)}).$$

□

3.3. Limited Gang FJP Scheduler

The predictability can also be ensured if we avoid the priority inversion phenomenon reported in Section 1, more precisely by restricting Gang FTP/FJP as follows:

Definition 18 (Limited Gang FJP scheduler)**.** *At each instant, the algorithm schedules jobs on processors as follows: the highest priority (active) job J_i is scheduled on the first v_i available processors (if any). The very same rule is then applied to the remaining active jobs on the remaining available processors* only if *J_i was scheduled (i.e., if at least v_i processors were available).*

We now prove that limited Gang FJP are predictable but first an additional definition.

Definition 19 (Limited level-(i) availability of the processors)**.** *For any ordered set of $i-1$ active jobs $J=J_1,\ldots,J_{i-1}$ and any set of m processors, we define the* limited level-(i) availability of the processors $\hat{A}_i(J,t)$ *of the set of jobs J at time-instant t as follows ($\hat{A}_0(J,t)=m$ for all J,t):*

$$\hat{A}_i(J,t)\stackrel{\text{def}}{=}\begin{cases}\#A(J,t) & \text{if } \#A(J,t)\geq v_i \text{ and}\\ & \hat{A}_{i-1}(J,t)\neq 0;\\ 0 & \text{otherwise.}\end{cases}$$

Lemma 20. *For any schedulable ordered set of jobs J, using a Limited Gang FJP on m processors, we have $\hat{A}_i(J_+^{(i-1)},t)\leq\hat{A}_i(J^{(i-1)},t)$, for all t and all i. (We consider that the sets of jobs are ordered in the same decreasing order of the priorities, i.e., $J_1>J_2>\cdots$ and $J_1^+>J_2^+>\cdots$.)*

Proof. The property follows using a similar reasoning as the proof of Lemma 14 and the fact that $\hat{A}_i(J^{(i-1)},t)\geq\hat{A}_{i+1}(J^{(i-1)},t)$. □

Theorem 21. *Limited Gang FJP schedulers are predictable on identical platforms.*

Proof. The proof is similar to the proof given for Theorem 15. $\hat{A}_i(J^{(i-1)},t)$ describes the number of processors available to schedule J_i. As this number is at any time higher in $J_+^{(i-1)}$ than in $J^{(i-1)}$, then J_i will never start later in $J^{(i-1)}$ than in $J_+^{(i-1)}$, and will never finish later either. □

Remark that by using limited Gang scheduling we accept to lose efficiency in the resource utilization to ensure system predictability.

3.4. Gang FJP and Limited Slack Reclaiming

As highlighted previously, the problem of using the slack caused by a job finishing earlier than expected is that it could cause a preemption that would not have occurred if the job had used its worst case execution time. But with a closer look, we can see that the problem only occurs when a job J_i wider than a job J_j ($v_i>v_j$) takes

advantage of the slack created by J_j early completion. A way of avoiding this is to only allow job not larger than the early completed job to use the slack. This is what we propose in this technique.

Definition 22 (Slack server)**.** *A* slack server *of level ℓ, width w and length λ is a job of priority ℓ, on w processors, running for λ units of time, serving jobs with a priority lower than ℓ which do not require more than w processors. If no task are available, the server stays idle until the end of the λ units of time.*

It may be noticed that:

- We do not give any constraint about the way the "slack server scheduling" (the way jobs are scheduled inside the slack server) is performed;
- Within a slack server, we might run several jobs in parallel, as long as they never need more than w processors simultaneously;
- If a job being served by the slack server becomes eligible by the "global scheduler", then it should be interrupted in the slack server, and made available to the global scheduler;
- All jobs served by the slack server should still stay in the ready state (but not running) from the "global scheduler" point of view.

Regarding this definition of a slack server, we can now define how our schedule will work.

Definition 23 (Gang FJP and limited slack reclaiming)**.** *A* Gang FJP scheduler with limited slack reclaiming*, works as follows: At each scheduling point (the completion of a job or an arrival):*

- *If this corresponds to the end of a job J_i, and this job used $e' < e_i$ units of time, starts a slack server of level i, width v_i, length $e_i - e'$;*
- *Otherwise, the highest priority (active) job J_i is scheduled on the first v_i available processors (if any). The very same rule is then applied to the remaining active jobs on the remaining available processors.*

We can make an important observation about this scheduling algorithm. Jobs that are run in the slack server will not have any other impact on the global scheduler that reducing the execution time of those jobs. So if we consider the slack server as a black box, the schedule will be exactly the same as if this black box was just idling. The only impact will be the proportion between actual and worst case execution time.

Remark also that will we cause priority inversion: some jobs will be run inside the slack server, while other higher priority ready job (but wider than the slack server) will be kept suspended.

Figure 3 illustrates how the slack server works. We consider the following set of jobs (they all have the same arrival time 0, and the same deadline 6:

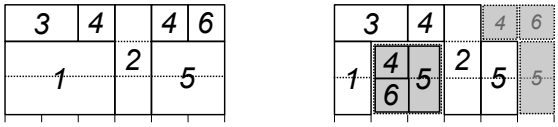

Figure 3. Slack server example. Left: all jobs use their worst case execution time. Right: Job 1 is shorter than expected, and a slack server is set up (gray part).

	J_1	J_2	J_3	J_4	J_5	J_6
v_i	2	3	1	1	2	1
e_i	3	1	2	2	2	1

The left side of Figure 3 shows the schedule where all jobs use their worst case execution time. On the right side, J_1 finishes at time 1 (instead of 3). The schedulers launches then a slack server (in gray on the figure) of level 1, width 2 and length 2, in order to fill the space that would have been used by J_1 in the worst case scenario. This server is then scheduled as a job of priority 1, as was J_1. At time 1, the slack server sees that jobs J_2, J_4, J_5, J_6 are *ready* (J_3 is running). But J_2 is too wide, so the slack server can for instance choose (arbitrarily) to run J_4 and J_6. After one unit of time, the scheduler sees that it can run J_4, the highest priority task which can run on the only available processor released by the end of J_3. J_4 is then "preempted" inside the slack server, and run normally. Then the slack server choses to run J_5 (J_6 is done). At time 3, the slack server ends, and J_5 is preempted.

At time 4 and 5, we need to start slack servers for J_4, J_5 and J_6, but they do not receive any work to perform.

Remark that if we compare both schedules of Figure 3, and see the slack server as part of the concerned job, all tasks start end and at the same time in both scenarios.

Theorem 24. *Gang FJP schedulers with limited slack reclaiming are predictable on identical platforms.*

Proof. From the schedulability point of view, this behaves exactly the same way as the Idling server. But instead of being idle, the slack server decreases the actual execution time of some ready (but not running) jobs.

One job will never preempt a job that would not have been preempted in the worst case scenario. □

Notice that with this kind of scheduler, we might considered the system as being not fully FTP anymore. Some jobs are indeed eligible (enough resource to run them), but are left waiting. In the presentation of this section, we said that we did not give any constraint on the scheduler of the slack server. Indeed, the method we use does not have any impact on the predictability of the system. But we can of course use a FTP scheduling algorithm. This does not make the global system to be strictly FTP, but it makes closer.

Notice also that we present a system with two level of scheduler: one global, and one inside the slack server.

This distinction was used for the sake of presentation, but in a real implementation, the global scheduler can of course also do the job of the slack server scheduler.

4. Periodicity

In this section we prove the periodicity of feasible Gang FTP schedules. It is important to note that we assume in this section that each job of the same task (say τ_i) has an execution requirement which is *exactly* C_i time units. Thanks to the predictability property this situation corresponds to the worst case.

Remark that, as we consider only the case where each job has its worst case, the schedule of an idling, slack reclaiming or general scheduler is exactly the same.

Theorem 25. *For any preemptive (limited or not) Gang FTP algorithm $\mathcal{A}$, if an asynchronous constrained deadline system τ is $\mathcal{A}$-feasible, then the $\mathcal{A}$-feasible schedule of τ on m identical processors is periodic with a period $P \stackrel{\text{def}}{=} \text{lcm}\{T_1, \ldots, T_n\}$ from instant S_n where S_i is defined inductively as follows:*

- $S_1 \stackrel{\text{def}}{=} O_1$;
- $S_i \stackrel{\text{def}}{=} \max\{O_i, O_i + \left\lceil \frac{S_{i-1} - O_i}{T_i} \right\rceil T_i\}, \forall i \in \{2, 3, \ldots, n\}$.

(Assuming that the execution time of each task is constant.)

Proof. The proof is made by induction on n (the number of tasks). We denote by $\sigma^{(i)}$ the schedule obtained by considering only the task subset $\tau^{(i)}$, the first higher priority i tasks $\{\tau_1, \ldots, \tau_i\}$, and by $A(J^{(i)}, t)$ the corresponding availability of the processors. Our inductive hypothesis is the following: the schedule $\sigma^{(k)}$ is periodic from S_k with a period $P_k \stackrel{\text{def}}{=} \text{lcm}\{T_1, \ldots, T_k\}$ for all $1 \leq k \leq i$.

The property is true in the base case: $\sigma^{(1)}$ is periodic from $S_1 = O_1$ with period P_1, for $\tau^{(1)} = \{\tau_1\}$: since we consider (feasible) constrained deadline systems, at instant $P_1 = T_1$ the previous request of τ_1 has finished its execution and the schedule repeats.

We shall now show that any $\mathcal{A}$-feasible schedule of $\tau^{(i+1)}$ is periodic with period P_{i+1} from S_{i+1}.

Since $\sigma^{(i)}$ is periodic with a period P_i from S_i the following equation is verified:

$$\sigma^{(i)}(t) = \sigma^{(i)}(t + P_i), \forall t \geq S_i. \quad (1)$$

We denote by $S_{i+1} \stackrel{\text{def}}{=} \max\{O_{i+1}, O_{i+1} + \left\lceil \frac{S_i - O_{i+1}}{T_{i+1}} \right\rceil T_{i+1}\}$ the first request of τ_{i+1} not before S_i.

Since the tasks in $\tau^{(i)}$ have higher priority than τ_{i+1}, then the scheduling of τ_{i+1} will not interfere with higher priority tasks which are already scheduled. Therefore, we may build $\sigma^{(i+1)}$ from $\sigma^{(i)}$ such that the tasks $\tau_1, \tau_2, \ldots, \tau_i$ are scheduled at the very same instants and on the very same processors as they were in $\sigma^{(i)}$. We apply now the induction step: for all $t \geq S_i$ in $\sigma^{(i)}$ we have $A(J^{(i)}, t) = A(J^{(i)}, t + P_i)$ the availability of the processors repeats. Notice that at those instants t and $t + P_i$ the available processors (if any) are the same. Consequently at only these instants where $\#A(J^{(i)}, t) \geq v_{i+1}$, task τ_{i+1} *may* be executed. Notice that the scheduler can decide to leave one or several processor(s) to be idle intentionally in a deterministic and memoryless way. Notice also, in the "non limited case", that τ_{i+1} might start executing before a higher priority task τ_j (with $j < i + 1$), if $v_j > \#A(J^{(i)}, t) > v_{i+1}$. But as soon as v_j processors are available in $A(J^{(i)}, t)$, τ_{i+1} is preempted (if still running) and the CPU is given to τ_j.

The instants t with $S_{i+1} \leq t < S_{i+1} + P_{i+1}$, where τ_{i+1} may be executed in $\sigma^{(i+1)}$, are periodic with period P_{i+1} since P_{i+1} is a multiple of P_i. Moreover since the system is feasible and we consider constrained deadlines, the only active request of τ_{i+1} at S_{i+1}, respectively at $S_{i+1} + P_{i+1}$, is the one activated at S_{i+1}, respectively at $S_{i+1} + P_{i+1}$. Consequently, the instants at which the deterministic and memoryless algorithm $\mathcal{A}$ schedules τ_{i+1} are periodic with period P_{i+1}. Therefore, the schedule $\sigma^{(i+1)}$ repeats from S_{i+1} with period equal to P_{i+1} and the property is true for all $1 \leq k \leq n$, in particular for $k = n : \sigma^{(n)}$ is periodic with period equal to P from S_n and the property follows. □

5. Exact Schedulability Test

Now we have the material to define an exact schedulability test for rigid and asynchronous periodic systems.

Corollary 26. *For any preemptive Gang FTP predictable algorithm $\mathcal{A}$ (i.e., Parallelism Monotonic, Idling, Limited Gang, and Limited Slack Reclaiming variants) and for any asynchronous rigid constrained deadline system τ on m identical processors, τ is $\mathcal{A}$-schedulable if and only if*

- *all deadlines are met in $[0, S_n + P)$ and*
- *$\theta(S_n) = \theta(S_n + P)$*

where S_i are defined inductively in Theorem 25.

Proof. Corollary 26 is a direct consequence of Theorem 25 and the predictability of Parallelism Monotonic (Theorem 15), Idling (Lemma 17), Limited Gang (Theorem 21), and Limited Slack Reclaiming (Theorem 24) variants. □

6. Conclusion and Future Work

In this paper we considered the scheduling of periodic and parallel rigid tasks. We provided and proved

correct an *exact* schedulability test for Fixed Task Priority (FTP) Gang scheduler sub-classes: Parallelism Monotonic, Idling, Limited Gang, and Limited Slack Reclaiming. Additionally, we studied the predictability of our schedulers: we show that Gang FJP schedulers are not predictable and we identify several sub-classes which are actually predictable. We also extended the definition of rigid, moldable and malleable jobs to recurrent tasks.

In future work we aim to extend the model by considering *moldable* tasks — task can be executed in a varying number of processors — that is the scheduler can determine, on-line, the rectangle of each task instance (job) based upon parallel performance model (e.g., the one defined in [6]).

References

[1] T. P. Baker. An analysis of EDF scheduling on a multiprocessor. *IEEE Trans. on Parallel and Distributed Systems*, 15(8):760–768, 2005.

[2] T. P. Baker and S. Baruah. Schedulability analysis of multiprocessor sporadic task systems. *Handbook of Real-Time and Embedded Systems*, 2006.

[3] V. Berten, S. Collette, and J. Goossens. Feasibility test for multi-phase parallel real-time jobs. In D. Zhu, editor, *Proceedings of the Work-in-Progress session of the IEEE Real-Time Systems Symposium 2009*, pages 33–36, 2009.

[4] R. Buyya. *High Performance Cluster Computing: Architectures and Systems*, chapter Scheduling Parallel Jobs on Clusters, pages 519–533. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999.

[5] R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[6] S. Collette, L. Cucu, and J. Goossens. Integrating job parallelism in real-time scheduling theory. *Information Processing Letters*, 106(5):180–187, May 2008.

[7] L. Cucu and J. Goossens. Feasibility intervals for fixed-priority real-time scheduling on uniform multiprocessors. *Proceedings of the 11th IEEE International Conference on Emerging Technologies and Factory Automation*, pages 397–405, 2006.

[8] L. Cucu and J. Goossens. Feasibility intervals for multiprocessor fixed-priority scheduling of arbitrary deadline periodic systems. *Proceedings of the 10th Design, Automation and Test in Europe*, pages 1635–1640, 2007.

[9] L. Cucu-Grosjean and J. Goossens. Predictability of fixed-job priority schedulers on heterogeneous multiprocessor real-time systems. *Information Processing Letters*, 110:399–402, 2010.

[10] M. Drozdowski. Scheduling parallel tasks — algorithms and complexity. *Handbook of Scheduling*, pages 25–1–25–25, 2005.

[11] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Job Scheduling Strategies for Parallel Processing*, pages 1–26. Springer-Verlag, 1996.

[12] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[13] S. Gorlatch and H. Bischof. A generic MPI implementation for a data-parallel skeleton: Formal derivation and application to FFT. *Parallel Processing Letters*, 8(4):447–458, 1998.

[14] W. Gropp, editor. *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MIT Press, 2nd edition, 1999.

[15] R. Ha and J. Liu. Validating timing constraints in multiprocessor and distributed real-time systems. *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 162–171, 1994.

[16] C. Han and K.-J. Lin. Scheduling parallelizable jobs on multiprocessors. *Proceedings of the 10th IEEE Real-Time Systems Symposium (RTSS'89)*, pages 59–67, 1989.

[17] S. Kato and Y. Ishikawa. Gang EDF scheduling of parallel task systems. In *30th IEEE Real-Time Systems Symposium*, pages 459–468. IEEE Computer Society, 2009.

[18] G. Manimaran, C. Siva Ram Murthy, and K. Ramamritham. A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. *Real-Time Systems*, 15:39–60, 1998.

[19] V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience*, 2(4):315–339, 1990.