Revealing the secrets of RUN and QPS: new trends for optimal real-time multiprocessor scheduling

Ernesto Massa[†], George Lima[‡], Paul Regnier[‡]
[†]State University of Bahia and UNIFACS,
[‡]Federal University of Bahia
Salvador, Bahia, Brasil
Email: esmneto@uneb.br,{gmlima,pregnier}@ufba.br

***Abstract*—Until recently there has been a common belief that optimal multiprocessor real-time scheduling algorithms necessarily incur a high number of task preemptions and migrations. New scheduling algorithms have shown that this is not the case. In this paper we explain why two of these algorithms, RUN and QPS, achieve optimality with only a few preemptions and migrations. We also compare these two algorithms, exhibiting their similarities and differences. By putting RUN and QPS side-by-side, we bring about their fundamental properties and help in the understanding of the multiprocessor real-time scheduling problem.**

I. Introduction

The problem of optimally scheduling a set of n preemptible independent real-time tasks on m identical processors has extensively been studied. By *optimal scheduling* we mean producing a correct schedule (no missed deadlines) whenever it is possible to do so. When task deadlines are equal to their inter-release times (implicit deadlines), it is well known that this problem can be optimally solved by enforcing the execution of all tasks within short time intervals. The amount of execution in each interval is then assigned to system tasks following some fairness criteria [1]. Several approaches employ this principle *e.g.,* [1]–[8]. As a side effect, excessive overhead is usually found in terms of task preemption and migration.

Notably RUN (Reduction to UNiprocessor) [9] and QPS (Quasi-Partition Scheduling) [10] are two optimal scheduling approaches which employ different principles to obtain optimality. Both of them partition the system packing tasks into groups and each group is managed by *servers*, using the well known EDF policy [11] as an underline scheduling algorithm. A server can be thought of as a proxy entity which schedules their client tasks. If the packing produces up to m task groups, each of which requiring less than 100% of a processor, the system is feasibly scheduled by EDF in each processor. In this case, no task needs to migrate during its execution and the system behaves like partitioned-EDF [12]. Otherwise, RUN and QPS provide their own mechanisms to effectively deal with task migration, which is illustrated with the following example:

Example I.1. *Consider a 2-processor system with three tasks, τ_1, τ_2, and τ_3. Tasks τ_1 and τ_2 are periodically released every 3 time units and require 2 time units of execution. Task τ_3 requires 4 time units and its period is 6. All tasks are first released at time 0. Tasks deadlines are equal to their periods.*

Following the notation used in this paper, these tasks are represented as $\tau_1\!:(2,3)$, $\tau_2\!:(2,3)$, and $\tau_3\!:(4,6)$. As each task requires an execution rate of $2/3$, it is clear that any correct schedule for this example implies some task migration. In a nutshell, RUN and QPS deals with such a system as follows.

RUN transforms the multiprocessor scheduling problem into one or more uniprocessor scheduling problems. It is strongly based on the *duality* principle. Roughly speaking, if one knows when a task should not execute, then the derivation of when it must execute is straightforward. As each task requires 2/3 of a processor, it leaves out 1/3 unused. That is, each *primal* task τ_i is associated to a *dual* task τ_i^* requiring 1/3 of a (virtual) processor and keeping the same deadlines. A correct schedule of the dual tasks can be obtained by using EDF on a single (virtual) processor (see Figure 1). By the duality principle, the schedule for the primal tasks can be obtained. When a dual task is executing on the virtual processor, there are exactly $m = 2$ dual tasks not executing. The primal tasks related to these dual ones can then be chosen to execute.

QPS partitions the set of 3 tasks into two groups, $\{\tau_1, \tau_2\}$ and $\{\tau_3\}$ say, and explicitly deals with what exceeds 100% of the former group. As it requires 4/3 of processing resources (*i.e.,* more than one processor) and contains at least two tasks, QPS schedules them onto two processors by ensuring the parallel execution of τ_1 and τ_2 during 1/3 of the time. More specifically, QPS guarantees that 1/3 of processing resources are reserved to execute τ_1 on a processor; another 1/3 is reserved to τ_2's execution on the same processor as τ_1's; whereas 1/3 of the time both τ_1 and τ_2 are executed in parallel. Task τ_3 is allocated to a single processor and receives 2/3 of processing resources.

The explanation above would still apply when the three tasks were servers that aggregate other tasks, as it would be clearer shortly. Although RUN and QPS employ distinct mechanisms, as can be seen by the generated schedule in Figure 1, they share similarities, which are not explicit at first glance. By delving into the underline principles of RUN and QPS we contribute to the better understanding of the multiprocessor scheduling problem since such principles are in the core of the good performance of these algorithms. Since experimental comparison between RUN and QPS has been

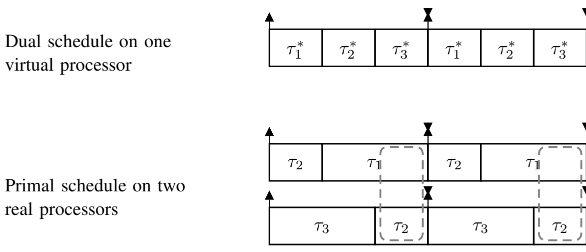


Figure 1. Possible schedule for tasks $\tau_1{:}(2,3)$, $\tau_2{:}(2,3)$ and $\tau_3{:}(2,3)$ on two processors. RUN applies the duality principle: whenever a dual task τ_i^* is executing on the virtual processor, the other real tasks are selected to execute on real processors. QPS partitions the system into $\{\tau_1,\tau_2\}$ and $\{\tau_3\}$, and enforces the parallel execution of τ_1 and τ_2 during $1/3$ of the time whereas τ_3 is statically allocated on a single processor and does not migrate. Enforced parallel execution is indicated by the dashed rectangles.

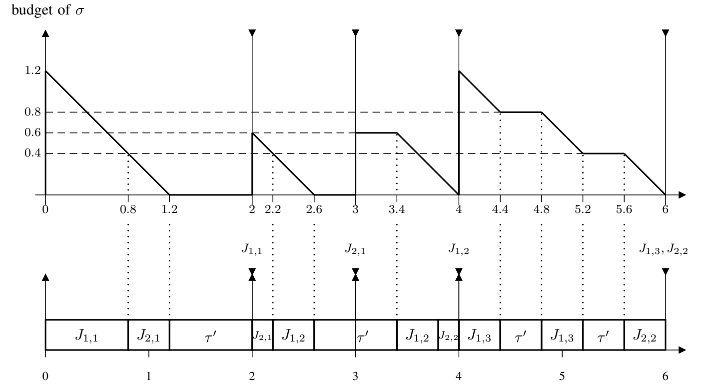


Figure 2. Budget management and schedule for an EDF-server σ with client set equal to $\Gamma = \{\tau_1{:}(0.8,2),\ \tau_2{:}(0.6,3)\}$ and $\mathrm{R}(\sigma) = 0.6$. $J_{i,k}$ is the kth job of τ_i. Task τ' represents the concurrent execution of other tasks or servers.

previously carried out [10], we chose to focus in this paper on the relation between these two algorithms, which has not been addressed up to now.

II. Terminology and basic concepts

A. Assumptions and Notation

Let Γ be a set of n tasks scheduled on a set of m identical processors. Each task τ in Γ releases a (possibly infinite) sequence of jobs, or workloads. A job is a sequence of instructions with a worst-case execution time, a release time and a deadline. A correct schedule is the one such that all released jobs are completely executed, they do not start executing before their release times, they finish execution no later than their deadlines, and they do not execute at the same time on more than one processor.

In this paper we assume that all tasks release their jobs periodically, *i.e.,* we focus on the periodic task model. This is because our main objective is to explain the scheduling principles behind RUN and QPS. Since RUN was designed for periodic tasks, we also limit the description of QPS for the same task model. Moreover, we restrict our explanation to systems of n tasks that require 100% of m processors. This is not a restriction of RUN or QPS since a fully utilized system can be thought of as an under utilized system with added dummy tasks. This assumption, however, makes the description of RUN and QPS easier for the our purposes.

In order to represent possible task aggregations into servers, the concept of task for RUN and QPS is slightly more generic than what is commonly found elsewhere. A task τ is characterized by its rate, denoted $\mathrm{R}(\tau)$, which represents the fraction of a processor it requires, and it is called a fixed-rate task. When τ releases a job at time r with deadline at time d, its execution time c equals $\mathrm{R}(\tau)(d-r)$. That is, c is no longer fixed but explicitly depends on τ's job deadline. Note that under the usual task model, $d-r$ is constant and so this is a particular case of our task model. For example, a periodic task τ with initial release time $r_0 = 0$, period T and execution time c, is represented as a fixed-rate task where $\mathrm{R}(\tau) = c/p$ and the set of all release instants of the jobs of τ is $\{r_k = kT, k \in \mathbb{N}\}$. If Γ is a task set, we use $\mathrm{R}(\Gamma) = \sum_{\tau_i \in \Gamma} \mathrm{R}(\tau_i)$ to denote the total rate of tasks in Γ.

We assume that jobs can be preempted at any time during their execution. When preempted in a processor, a job can resume its execution on another processor. There is no cost associated with preemption or migration. It is also assumed that a task can only release a new job after the deadline of its previous job.

B. Servers

As mentioned earlier, both RUN and QPS packs tasks into servers. The type of server considered is called *fixed-rate EDF servers* [9]. Instead of formally defining servers, we illustrate the mechanism they provide.

A server reserves processing resources for a set of tasks, or even other servers, which are called its clients. It thus regulates the execution of its clients, scheduling them by EDF. The set of deadlines of a server contains all deadlines of its clients and a server releases a new job at any time which is also a release instant of some of its clients. If the accumulated rate of a set of tasks Γ is equal to $\mathrm{R}(\Gamma) \leqslant 1$, a server σ with rate $\mathrm{R}(\sigma) = \mathrm{R}(\Gamma)$ can schedule Γ in a way that no deadline is missed (for a formal proof refer to [9]).

Figure 2 provides some illustration. A server σ has rate equal to 0.6, which corresponds to the accumulated rate needed to execute their clients, $\mathrm{R}(\tau_1) + \mathrm{R}(\tau_2)$. At time 0, the server releases its job with deadline at time 2 and workload equal to $0.6 \times 2 = 1.2$. Once σ is scheduled by the system, their clients jobs are executed following EDF. At time 2 a new server job is released, but now with workload equal to 0.6 since the time interval between release time and deadline is equal to $3-2=1$. As can be seen, a server is actually an execution proxy. If the server rate was equal to 1, its schedule decisions would be the same as EDF running on a single processor.

C. Standard scheduling approaches for multiprocessors

We now call attention to the need of additional preemption points for achieving optimality in scheduling multiprocessor real-time systems. Consider again example I.1. If some usual

greedy scheduling algorithm (*e.g.,* EDF) is in place, the system is not schedulable. It would choose two tasks, τ_1 and τ_2, to execute from time 0 until 2 wasting processor resources during $[2, 3)$. This means that one needs to preempt either τ_1 or τ_2 in some point during $[0, 2)$ allowing for the execution of τ_3.

The standard strategy used by several scheduling algorithms is based on the *proportional execution* and *deadline propagation*. A classical approach is known as Proportional Fairness approach (PFair) [2] according to which quantum-based windows are defined and tasks are scheduled within short time intervals so as to emulate a fluid schedule. Each task executes proportionally to their rates within the intervals. More efficient versions of PFair [7] can be obtained if task deadlines are propagated to create scheduling windows. Within each window tasks execute proportional to their rates according to some fairness criterion [1]. Applying this strategy to Example I.1, this means that all tasks execute 2/3 during time windows $[0, 3)$ and $[3, 6)$. This creates the necessary preemption point at times 1 and 4, which leads to the same kind of schedule given in Figure 1. The problem is that the interval between any two deadlines may be arbitrarily short, causing the propagation of small scheduling windows throughout the system, greatly affecting performance.

Deadline propagation is indeed an effective synchronization mechanism for creating the necessary preemption points. RUN and QPS, nevertheless, restrict deadline propagation to some parts of the system. This is achieved by the way these algorithms partition the system and encapsulate the entities to be scheduled into servers. Moreover, RUN and QPS do not employ proportional execution of tasks (or fairness) to generate preemption points. In the next sections we provide details about the mechanisms used by RUN and QPS to achieve good performance.

III. The RUN Algorithm

RUN first carries out an off-line transformation of the multiprocessor system and uses the information from this transformation to generate the schedule on-line. In the off-line phase, RUN carries out a series of *DUAL* and *PACK* operations for iteratively reducing the number of processors in a multiprocessor system until an equivalent uniprocessor system is obtained. The *PACK* operation transforms low-rate tasks (or servers) into a set of high-rate servers. The *DUAL* operation creates the corresponding dual servers which require less processing resources than their primal ones. By carrying out both operations in a systematic way, RUN reduces both the number of entities to be scheduled (by packing) and the number of processors considered (by duality).

The *DUAL* operation on a (primal) server σ creates a (dual) server σ^*, by computing $\mathrm{R}(\sigma^*) = 1 - \mathrm{R}(\sigma)$. All deadlines of σ are also deadlines of σ^*. Hence, the execution time of σ^* during any time window represents the idle time of σ in that window and *vice-versa*. Also, the execution of σ^* in the dual system induces the non-execution of σ of the primal system and *vice-versa*. The reduction carried out by RUN was illustrated in Figure 1. We now make use of a slightly more complex example to give more details about the algorithm.

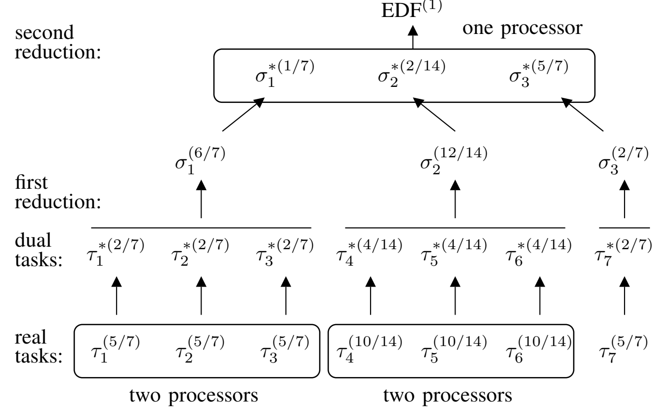


Figure 3. RUN off-line allocation phase for Example III.1, which needs two reduction levels. *DUAL* and *PACK* operations are indicated by arrows and horizontal lines, respectively. Solid rectangles indicate the number of processors used by task clusters.

Example III.1. *Consider a set of seven periodic tasks all initially released at time 0. Let τ_1, τ_2, τ_3, τ_7 be in the form $(5, 7)$ and τ_4, τ_5, τ_6 be defined as $(10, 14)$.*

During the off-line phase, the *PACK* operation does not group any system tasks because they have individual rate greater than 0.5 and RUN packs tasks to be scheduled by servers which cannot have rate greater than 1. The *DUAL* operation can be applied and it creates seven dual tasks $\tau_i^*{:}(2, 7)$, $i = 1, 2, 3, 7$ and $\tau_i^*{:}(4, 14)$, $i = 4, 5, 6$. Applying the *PACK* operation on these dual tasks, they can be grouped together as follows: Tasks τ_1^*, τ_2^* and τ_3^* can be associated to a server σ_1; τ_4^*, τ_5^* and τ_6^* to a server σ_2; and τ_7^* to server σ_3. This grouping actually creates *virtual clusters* $\{\tau_1, \tau_2, \tau_3\}$, $\{\tau_4, \tau_5, \tau_6\}$ and $\{\tau_7\}$. This is because the scheduling of server σ_i induces the schedule of its clients, which by duality is converted to the schedule of their primal servers (or simply tasks here).

As the server rates sum more than one ($\mathrm{R}(\sigma_1) = \mathrm{R}(\sigma_2) = 6/7$ and $\mathrm{R}(\sigma_3) = 2/7$), a second reduction level is needed. Carrying out the *DUAL* operation this time creates the dual system σ_1^*, σ_2^*, and σ_3^*, which can be scheduled on a single processor since $\mathrm{R}(\sigma_1^*) = 1/7$, $\mathrm{R}(\sigma_2^*) = 2/14$ and $\mathrm{R}(\sigma_3^*) = 5/7$. This ends the off-line phase.

Note that there are two virtual systems, one at the first reduction level with two processors obtained by the first reduction, and another at the second reduction level with a single processor, the result of the second reduction. The former is composed of servers σ_1, σ_2, and σ_3 and their client tasks obtained by duality of the real tasks τ_i, $i = 1, 2, \ldots, 7$. The latter virtual system is made of servers σ_1^*, σ_2^*, and σ_3^*. During the on-line phase, all virtual and real systems are scheduled.

An interesting aspect is that virtual clusters $\{\tau_1, \tau_2, \tau_3\}$, $\{\tau_4, \tau_5, \tau_6\}$ and $\{\tau_7\}$ have total rates respectively equal to $2 + 1/7$, $2 + 1/7$ and $5/7$. Hence, a correct schedule for the system can be obtained if each of the first two clusters are

assigned to two dedicated processors and the fifth processor manages the third cluster $\{\tau_7\}$ plus the excess of the others. RUN generates the schedule by doing exactly that.

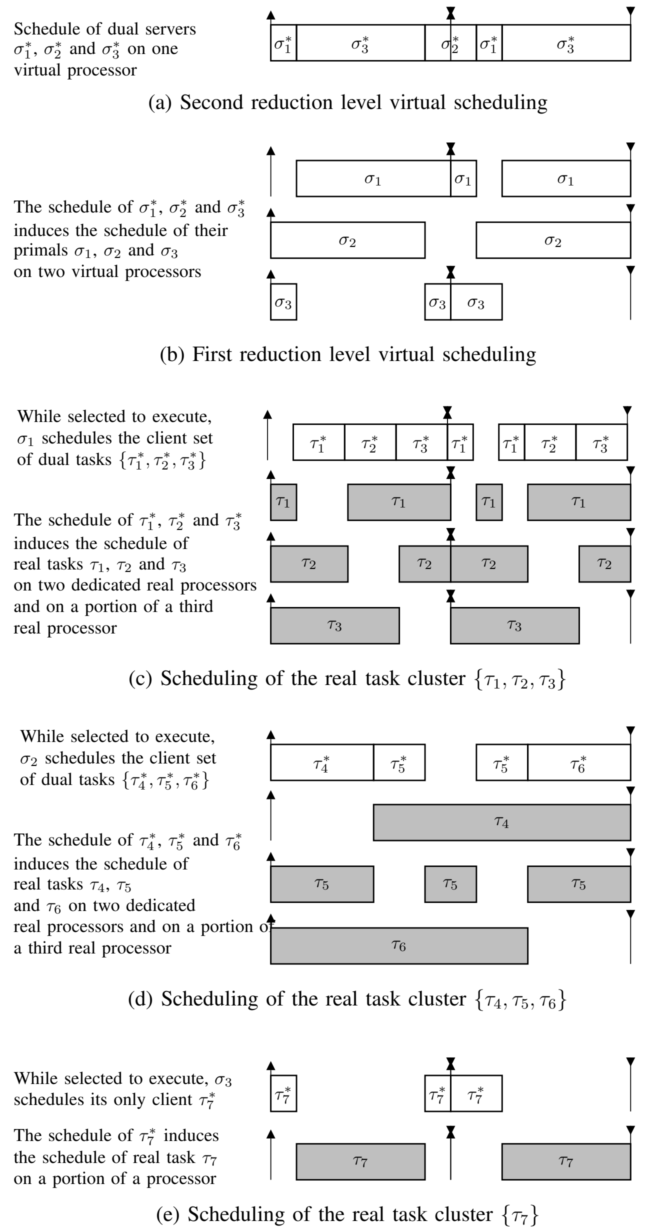
The on-line phase is somewhat straightforward, as illustrated in Figure 4. As servers have all deadlines of their clients, at time 0 the deadline of σ_1^* is 7 (first deadline for $\{\tau_1, \tau_2, \tau_3\}$), the deadline of σ_2^* is 14 (first deadline for $\{\tau_4, \tau_5, \tau_6\}$), and the deadline of σ_3^* is 7 since τ_7 is its only client. As scheduling choices must begin at the highest reduction level (in this case, the second one), Figure 4(a) shows that server σ_1^* is selected (via EDF) to execute at time 0 on the single virtual processor. This means that its primal σ_1 must not execute and servers σ_2 and σ_3 are then selected to execute on the two virtual processors at the first reduction level (Figure 4(b)). As σ_1 is not selected at the first reduction level, none of its dual clients (τ_1^*, τ_2^* and τ_3^*) are scheduled at time 0 and their respective primal tasks (τ_1, τ_2 and τ_3) are selected to execute at time 0 (Figure 4(c)).

Servers σ_2 and σ_3 select their highest priority clients (via EDF), τ_4^* and τ_7^* for instance (Figure 4(d) and Figure 4(e), respectively). By the duality principle, this means that tasks τ_4 and τ_7 must not execute.

The execution of σ_1^* ends at time 1 since its rate equals $1/7$ and its nearest deadline is 7. This creates a necessary preemption point. However, this preemption does not propagate throughout the whole system. At time one, σ_3^* is selected at the second reduction level (Figure 4(a)) and then σ_1 and σ_2 occupy both virtual processors at the first reduction level (Figure 4(b)), scheduling respectively their dual clients τ_1^* (Figure 4(c)) and τ_4^* (Figure 4(d)). Hence, τ_1 and τ_4 are the only tasks in clusters $\{\tau_1, \tau_2, \tau_3\}$ and $\{\tau_4, \tau_5, \tau_6\}$ (Figure 4(c)-(d)) not to execute, respectively. As σ_3 does not execute at the first reduction level (Figure 4(b)), its only client τ_7^* cannot be selected to execute, implying that τ_7 must execute (Figure 4(e)).

We now observe a key feature of the reduction procedure explained above. Consider cluster $\{\tau_1, \tau_2, \tau_3\}$ for illustration. Its tasks require $2 + 1/7$ (> 2) processing resources. That is, their cumulative rate *exceeds* two dedicated processors by $1/7$. This excess rate is exactly what appears at the second reduction level as a rate of σ_1^*. Observe that whenever σ_1^* executes (Figure 4(a)), τ_1, τ_2 and τ_3 also execute in parallel (Figure 4(c)). In other words, σ_1^*'s rate can be interpreted as the need for parallel execution for cluster $\{\tau_1, \tau_2, \tau_3\}$. Thus, when σ_1^* is selected to execute at the second reduction level, RUN actually ensures that three processors are assigned to its cluster $\{\tau_1, \tau_2, \tau_3\}$. The mechanism to ensure this behavior is the duality principle associated to the virtual clustering.

Although not applying duality, QPS also manages this need for parallel execution of groups of tasks, as will be clearer in the next section.



(a) Second reduction level virtual scheduling

(b) First reduction level virtual scheduling

(c) Scheduling of the real task cluster $\{\tau_1, \tau_2, \tau_3\}$

(d) Scheduling of the real task cluster $\{\tau_4, \tau_5, \tau_6\}$

(e) Scheduling of the real task cluster $\{\tau_7\}$

Figure 4. The schedule generated by RUN for Example III.1. Gray and white boxes represent the real and virtual task/server execution, respectively.

IV. The QPS Algorithm

Like RUN, QPS partitions the system off-line and generates the schedule on-line. However, QPS allows for partitions containing sets of tasks with accumulated rate greater than 1 as long as it is lower that 2. Also, QPS requires that what exceeds 1 on a given partition set be lower than the rate of each element in that partition. We note that if the excess rate was not less than the rate of some task on a partition set, it would always be possible to find a better partitioning by re-allocating such a task. This kind of partitioning is called quasi-partitioning [10]. There are a number of heuristics in line with such quasi-partition requirements that can be used as implementation procedure, as the First-Fit Decreasing Bin-Packing for instance.

We return to Example III.1 for an illustrative description of QPS using a possible way of quasi-partitioning the system. The reader interested in a more formal description may refer to [10]. The seven tasks of the example must be assigned to the

five processors available. The system can be quasi-partitioned as $\Gamma_1 = \{\tau_1, \tau_2\}$, $\Gamma_2 = \{\tau_3, \tau_4\}$, $\Gamma_3 = \{\tau_5, \tau_6\}$, $\Gamma_4 = \{\tau_7\}$ and Γ_i assigned to processor $i = 1, 2, \ldots, 4$. At this stage, processor 5 is still empty. As the execution rate of the former three sets is $10/7$, each of them is clearly not schedulable on a single processor. QPS manages the execution of such sets, called major execution sets, using what is called QPS servers, which actually are four fixed-rate EDF servers. Minor execution sets, namely those that do not require more than one processor, are managed by local EDF on the processors they are assigned to.

If Γ is a major execution set, $x = \mathrm{R}(\Gamma) - 1$ is what exceeds the capacity of a processor. In this case, Γ is bi-partitioned into two subsets $\Gamma = \Gamma^a \cup \Gamma^b$ and QPS servers σ^M, σ^S, σ^A and σ^B are created to manage the execution of these tasks. σ^A is dedicated to serve tasks in Γ^a at a rate of $\mathrm{R}(\sigma^A) = \mathrm{R}(\Gamma^a) - x$ and σ^B is dedicated to serve tasks in Γ^b at a rate of $\mathrm{R}(\sigma^B) = \mathrm{R}(\Gamma^b) - x$. σ^M and σ^S can serve any task in Γ at a rate of $\mathrm{R}(\sigma^M) = \mathrm{R}(\sigma^S) = x$. At any time, all QPS servers associated with Γ share the same deadlines. σ^A and σ^B are called *dedicated* servers associated to Γ^a and Γ^b, respectively, and σ^M and σ^S are called *master* and *slave* servers, respectively. Servers σ^A and σ^B respectively deal with the non-parallel execution of Γ^a and Γ^b, while σ^M and σ^S deal with their parallel execution. Whenever σ^M is scheduled to execute, σ^S also executes, which explains their names. Also, whenever σ^M and σ^S execute, one task from Γ^a and one task from Γ^b execute in parallel; the choice of which executes on behalf of σ^M or σ^S is only a matter of efficiency. As $\mathrm{R}(\sigma^A) + \mathrm{R}(\sigma^B) + \mathrm{R}(\sigma^S) = 1$, σ^A, σ^B and σ^S can execute on a single processor. Server σ^M, meanwhile, executes on a different processor.

Using our illustrative example and considering major execution sets Γ_1, Γ_2 and Γ_3, we proceed as follows. QPS servers σ_i^M, σ_i^S, σ_i^A and σ_i^B, with rates equals to $\mathrm{R}(\sigma_i^M) = \mathrm{R}(\sigma_i^S) = 3/7$ and $\mathrm{R}(\sigma_i^A) = \mathrm{R}(\sigma_i^B) = 2/7$, are associated to Γ_i, $i = 1, 2, 3$. Masters and slaves servers schedule any client in their respective major execution set and masters are allocated to a different processor from their associated slave and dedicated servers. For example, σ_1^M can be exported to execution set Γ_4 making $\Gamma_4 = \{\sigma_1^M, \tau_7\}$ a major execution set. This in turn requires the creation of σ_4^M, σ_4^S, σ_4^A and σ_4^B, and master servers σ_2^M, σ_3^M and σ_4^M can be allocated to the empty processor 5. The final server allocation becomes as illustrated in Figure 5.

The whole procedure for partitioning the system and for defining QPS servers explained above can be carried out off-line.[1] During the on-line phase, servers and tasks are scheduled according to the following rules: (a) visit each processor in order reverse to that of its allocation to select via EDF the highest priority server/task; (b) if a master server is selected on a processor when applying (a), select also its corresponding slave server; (c) for all selected servers, select their highest priority client to be executed; (d) dispatch all selected tasks to execute.

[1]When sporadic tasks are being considered, QPS servers are activated on-line. This is to accommodate the fact that not all tasks are always active.

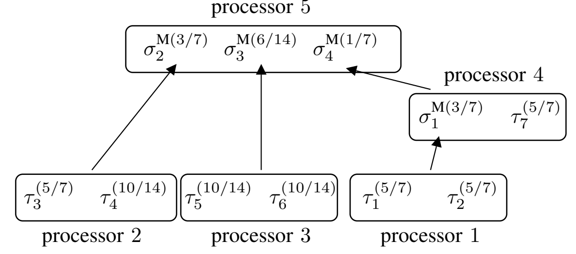


Figure 5. QPS off-line allocation phase for Example III.1. Allocated processors are represented by boxes and the arrows indicate the processor hierarchy. Processors with two entities are allocated to the major execution set bi-partitions.

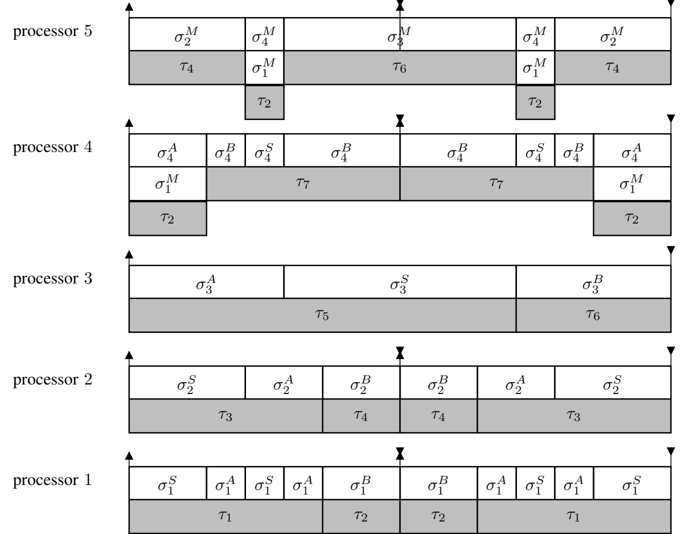


Figure 6. The schedule generated by QPS for Example III.1. Task executions are represented by gray boxes. Servers are positioned on the top of their clients.

The processor visiting order in rule (a) is needed because the selection of a master implies the selection of its slave, as stated in rule (b). From Figure 5 this is clear because the servers to be selected on processors 4 to 1 depend on which server was selected in processor 5. Rule (c) is for recursively seeking for tasks, which are the ones to actually be dispatched for execution, as stated in rule (d).

The schedule generated by QPS using rules (a)-(d) is illustrated in Figure 6. The allocation shown in Figure 5 was considered. Processor 5 was the last one to be allocated and so it is visited first. Note that σ_2^M is selected by EDF on this processor and this activates the selection of σ_2^S on processor 2. Server σ_4^A is chosen on processor 4. Its highest priority client (by EDF) is σ_1^M, enforcing the selection of σ_1^S on processor 1. Following the client chain for all selected servers, the actual tasks are dispatched. It worth observing that tasks executing on processor 5 belong to different clusters (recall Figure 5).

V. Discussion

RUN and QPS use similar strategies but implement different mechanisms, as will be clear in this section.

A. The Role of Servers

Both algorithms partition system tasks encapsulating partitioned elements (tasks or servers) into servers. Inner-server scheduling is provided very like it is in an uniprocessor system. The choice of using fixed-rate EDF servers, which are actually EDF schedulers with limited budget, is natural since EDF is a single-processor optimal algorithm.

Moreover, servers provide higher level entities on which multiprocessor scheduling rules apply. Interestingly, when tasks are packed into up to m servers, both RUN and QPS converges to partitioned EDF. In the case of QPS, since only major execution sets are subject to multiprocessor scheduling rules (QPS servers), parts of the system may be scheduled with no task migration.

B. Virtual Clustering

Multiprocessor scheduling rules for RUN are fundamentally based on the duality principle. If there are only k processors and $k+1$ entities to be scheduled, the selection of the entity that does not execute is straightforwardly obtained by duality. This is basically what is illustrated in Figure 1. When this is not the case, like in Example III.1, RUN groups $k+1$ tasks into clusters as illustrated in Figure 3 so that their scheduling is possible via duality. What exceeds the capacity of k processors is then exported to be executed in the context of another server on another processor. In turn, this excess rate is exactly the portion of processing resources reserved to execute all tasks in the clusters execute in parallel. This is illustrated in Figure 4. The schedule of σ_1 in interval $[0, 1)$, for instance, defines that all tasks in cluster $\{\tau_1, \tau_2, \tau_3\}$ must execute in parallel in this interval. At all time σ_1 is not scheduled, only two of these tasks should execute, *e.g.,* during $[1, 7)$. And the schedule decisions in this case follow the duality principle applied to dual tasks $\{\tau_1^*, \tau_2^*, \tau_3^*\}$.

The way QPS clusters the system can be seen as the particular case of RUN, restricting $k = 1$. That is, QPS manages the schedule of 2 entities to be scheduled on a single processor and exports what exceeds 1 as a reserve of another processor. The two entities are defined as the result of bi-partitioning a major execution set. Figure 5 clearly shows this clustering. Unlike RUN, which manages the parallel execution of servers via dual scheduling, the parallel execution of these two entities are enforced in QPS by the master/slave servers relation, as illustrated in Figure 6.

C. Deadline Propagation

Another important characteristic observed in both algorithms is related to the provided isolation between clusters. This explains the superior behavior of both RUN and QPS in terms of preemption and migrations when compared to standard scheduling approaches (see [9], [10]). Figure 4 illustrates this effect for RUN, where task deadlines in cluster $\{\tau_1, \tau_2, \tau_3\}$ (Figure 4(c)) cause preemptions that are not propagated to cluster $\{\tau_4, \tau_5, \tau_6\}$ (Figure 4(d)). The same observation is noticed for QPS in Figure 6 as for processors 2 and 3, for instance.

D. Practical relevance

Due to their non-standard infra-structures, RUN and QPS may be seem as not practical approaches. The hierarchies due to both the reduction of RUN and the master/slave relationship in QPS indeed require special implementation data structures at the operating system level. However, very low overhead has been found in an implementation of RUN [13], which shows that RUN execution overhead is actually comparable to those found in partitioned EDF. Being RUN an optimal algorithm, this result is indeed very appealing. Due to the similarities between RUN and QPS, similar performance behavior is expected for QPS.

VI. Conclusion

We have provided an informal and intuitive description of RUN and QPS, two recently published multiprocessor scheduling algorithms. Key aspects of these algorithms have been identified and their main similarities and differences have been pointed out. We believe that the information provided in this paper contributes to better understanding the design of optimal and efficient scheduling algorithms for multiprocessor real-time systems.

References

[1] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: a simple model for understanding optimal multiprocessor scheduling," in *Euromicro Conference on Real-Time Syst.*, 2010, pp. 3–13.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," in *ACM Symp. on the Theory of Computing*. New York, NY, USA: ACM, 1993, pp. 345–354.

[3] K. Funaoka, S. Kato, and N. Yamasaki, "Work-conserving optimal real-time scheduling on multiprocessors," in *Euromicro Conference on Real-Time Syst.*, 2008, pp. 13–22.

[4] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *IEEE Real-Time Syst. Symp.*, 2006, pp. 101–110.

[5] S. Funk, "LRE-TL: An optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines," *Real-Time Systems*, vol. 46, no. 3, pp. 332–359, 2010.

[6] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiprocessor scheduling," *Real-Time Syst.*, vol. 43, no. 1, pp. 25–59, 2009.

[7] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *IEEE Embedded and Real-Time Computing Syst. and Applications*, 2006, pp. 322–334.

[8] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *Euromicro Conference on Real-Time Syst.*, 2008, pp. 243–252.

[9] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *IEEE Real-Time Syst. Symp.*, 2011, pp. 104–115.

[10] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt, "Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach," in *Euromicro Confer. on Real-Time Syst.*, 2014, pp. 291–300.

[11] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *IEEE Real-Time Syst. Symp.*, 1990, pp. 182–190.

[12] S. K. Baruah, "The partitioned edf scheduling of sporadic task systems," in *IEEE Real-Time Syst. Symp.*, 2011, pp. 116–125.

[13] D. Compagnin, E. Mezzetti, and T. Vardanega, "Putting run into pratice: implementation and evaluation," in *Euromicro Conference on Real-Time Syst.*, 2014, pp. 75–84.