Real-Time Syst (2016) 52:566–597
DOI 10.1007/s11241-016-9251-6

Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems

Ernesto Massa[1,4] · **George Lima**[2] · **Paul Regnier**[2] · **Greg Levin**[3] · **Scott Brandt**[3]

Published online: 18 February 2016

Abstract We describe a new algorithm, called quasi-partitioned scheduling (QPS), capable of scheduling any feasible system composed of independent implicit-deadline sporadic tasks on identical processors. QPS partitions the system tasks into subsets, each of which is either scheduled by EDF on a single processor or by a set of servers on two or more processors. More precisely, QPS uses an efficient scheme to switch between partitioned EDF and global-like scheduling rules in response to system load variation, providing dynamic adaptation in the system. Extensive simulation compares QPS favorably against related work, showing that it has very low preemption and migration overheads.

Keywords Real time · Scheduling · Multiprocessor · Optmality

✉ Ernesto Massa
esmneto@uneb.br

George Lima
gmlima@ufba.br

Paul Regnier
pregnier@ufba.br

Greg Levin
glevin@soe.ucsc.edu

Scott Brandt
sbrandt@soe.ucsc.edu

1 State University of Bahia (UNEB), Alagoinhas, Brazil

2 Federal University of Bahia (UFBA), Salvador, Brazil

3 University of California, Santa Cruz, USA

4 UNIFACS, Salvador, Brazil

1 Introduction

Motivation and related work 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 solved by *global scheduling approaches*, where the scheduler manages a global task queue and tasks can migrate from one processor to another, *e.g.* Andersson and Bletsas (2008), Andersson and Tovar (2006), Baruah et al. (1996), Cho et al. (2006), Easwaran et al. (2009), Funaoka et al. (2008), Funk (2010), Levin et al. (2010), McNaughton (1959), and Zhu et al. (2003). Unfortunately, most global approaches incur an excessive overhead of preemptions and migrations by subdividing and overconstraining all tasks to run within small time intervals. Within these intervals, processor time is divided among tasks according to some fairness criteria (Levin et al. 2010).

Run-time overhead is minimized by partitioning the system tasks so that task subsets are entirely allocated to processors. However, it is unlikely that such a *partitioned scheduling approach* can deal with systems that fully utilize the m processors (Koren et al. 1998) and so when optimality is required, partitioned solutions are not an option.

There are also hybrid approaches offering a compromise between achievable system utilization and run-time overhead. Semi-partitioning is carried out by assigning most tasks to processors and by selecting a few of them to migrate between processors, *e.g.* Bletsas and Andersson (2011), Burns et al. (2011), Kato et al. (2009), and Santos-Jr et al. (2013). Task migration control mechanisms must be applied at run-time. Dividing the system into task/processor clusters is another option, *e.g.* Easwaran et al. (2009). In both cases, optimality can be achieved by some hybrid approaches by, again, enforcing short execution windows across the system *e.g.* Andersson and Bletsas (2008), Andersson and Tovar (2006), Bletsas and Andersson (2009), Easwaran et al. (2009), causing high run-time overheads (Bastoni et al. 2011).

Two recently described global scheduling algorithms, RUN (Regnier et al. 2011) and U-EDF (Nelissen et al. 2012), achieve optimality with low preemption and migration overheads. Interestingly, a recent implementation of RUN on Litmus^{RT} (Compagnin et al. 2014) has confirmed its low overhead in practice. However, although RUN provides the lowest known figures in terms of generated preemption and migration, in its current version sporadic tasks are not supported whereas U-EDF can manage sporadic tasks.

To the best of our knowledge, the quasi-partitioned scheduling (QPS) approach described in this paper is the first multiprocessor scheduling algorithm to date capable of adapting its scheduling strategy as a function of system load. When dealing with sporadic tasks in the system, for instance, the required processing resources may fluctuate over time. Taking advantage of this fact, QPS monitors the system load at run-time and moves from global-like to partitioned-like scheduling rules and *vice-versa* in response to system load fluctuations. This dynamic adaptation drastically reduces the number of migrations required by the scheduler.

Contribution QPS is a new algorithm capable of scheduling any feasible system composed of independent implicit-deadline sporadic tasks on identical processors.

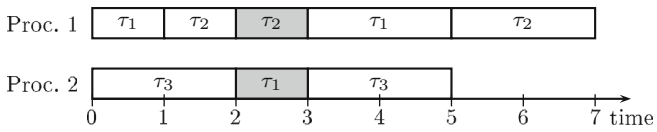

Fig. 1 QPS schedule for three tasks. All tasks arrive at time 0; tasks τ_1 and τ_3 are activated with period 3; but the second job of τ_2 is late. Task migration occurs in the period [0, 3); partitioned scheduling is applied during [3,6) due to a late task arrival

QPS first partitions the system tasks into subsets of two types, depending on whether they require one or multiple processors; we refer to these as *minor* and *major* execution sets, respectively. If all subsets are minor, QPS reduces to partitioned Earliest Deadline First (EDF). Major execution sets are scheduled either by a set of QPS servers on multiple processors (QPS mode), or by local EDF on a single processor (EDF mode) depending on their execution requirements. By monitoring major execution sets at run-time, QPS is able to switch between these two modes, providing dynamic adaptation to system load.

The dynamic adaptation capability of QPS is dependent on the specific type of partitioning used, the so-called *quasi-partition*. For illustration, consider a simple two-processor system with three tasks, τ_1, τ_2, and τ_3, where each job requires 2 units of work over 3 units of time, and jobs have a minimum inter-arrival time of 3 (see Fig. 1). QPS partitions the task set into major execution set $P_1 = \{\tau_1, \tau_2\}$ and minor execution set $P_2 = \{\tau_3\}$, which require 4/3 and 2/3 of a processor's capacity, respectively. Consequently, P_2 can execute on a single processor. The remainder capacity of that processor is used to handle the excess requirements of P_1. The QPS servers manage the execution of P_1 during the interval [0, 3), ensuring that τ_1 and τ_2 run in parallel when Processor 2 is available, and alternate on Processor 1 when it isn't. Now let's suppose that τ_1 and τ_3 arrive periodically, but that τ_2's second job doesn't arrive until time 5. We deactivate P_1's QPS servers, and schedule the remaining tasks of P_1 (namely, τ_1) on Processor 1 using EDF during [3, 7). When τ_2 arrives again, we reactivate the QPS servers, and once again allow for the parallel execution of τ_1 and τ_2 as shown in Fig. 1. To the best of our knowledge, QPS is the first multiprocessor scheduling algorithm that provides on-line adaptation between global and partitioned scheduling rules.

QPS was first described with a single adaptation strategy (Massa et al. 2014) which, despite achieving the primary goal of dynamic adaptation, is very conservative and sometimes remains working as a global-like scheduling while it could be working as partitioned EDF. This behaviour causes some extra preemptions and migrations, which could be avoided with a more accurate adaptation strategy. To achieve this secondary goal of better accuracy, it is necessary to predict at run-time system demand for a near future. In this paper we describe two new and more accurate adaptation approaches, which only adopt a global-like scheduling scheme when partitioned EDF can not successfully schedule system tasks, and compare all the three adaptation approaches in terms of generated preemption and migration. Additionally, we also examine in this paper the effects quasi-partitioning heuristics cause on the performance of QPS by considering two alternative implementations of quasi-partitioning.

Paper structure Section 2 defines the system model and establishes the notation used in the paper. We introduce the generalized *fixed-rate server* in Sect. 3, and present the concept of a *quasi-partition* in Sect. 3.2. Section 4 details the QPS scheduling algorithm and Sect. 5 proves it correct. Results obtained from simulations are discussed in Sect. 6. Final comments are given in Sect. 7.

2 System model and notations

Let Γ be a set of n sporadic tasks scheduled on a set of m identical processors. Each task τ in Γ releases a possibly infinite sequence of jobs, or workloads. The concept of a *job* is formally defined as follows:

Definition 1 (*Job*) A job $J : (c, r, d)$ is a sequence of instructions that consumes up to c units of processing time within time interval $[r, d)$. That is, c, r, and d represent the worst-case execution time (WCET), the release time, and the absolute deadline of J, respectively.

We assume that all jobs require their WCET. If a job were to require less work, we could simulate a WCET requirement by filling in the difference with idle processor time. Our definition of tasks is more generic than what is commonly found elsewhere since we need to represent possible task aggregations. We characterize a task τ by its rate, denoted $\mathrm{R}(\tau)$, which represents the fraction of a processor required by its jobs, namely the jobs it generates. When a task τ releases a job $J : (c, r, d)$, its execution time c equals $\mathrm{R}(\tau)(d - r)$, as usual. However, different from the classical sporadic task model, the time interval duration that a task remains active may vary for each of its jobs. That is, $d - r$ is not constant over the jobs of τ, and so neither is c. Each job of the same task can have a particular duration $(d - r)$, which induces a particular WCET $c = R(\tau)(d - r)$. As will be clearer latter on, the main purpose of this definition is to address task aggretation into a single entity, namely server. Note that this is a generalization of the typical sporadic task model. The concept is defined more formally below. We let $D(\tau, t)$ denote the next deadline of τ occurring after time t, and $E(\tau, t)$ the work remaining for τ's current job at time t.

Definition 2 (*Fixed-rate task*) A *fixed-rate task* (henceforth simply "task") τ, with rate $\mathrm{R}(\tau) \leq 1$, releases a possibly infinity sequence of jobs. A job $J : (c, r, d)$ of τ released at time r will have $d = D(\tau, r)$ and $c = E(\tau, r) = \mathrm{R}(\tau)(d - r)$. In the particular case when a task τ has constant minimum inter-release time T, we represent it as a tuple $\tau : (\mathrm{R}(\tau)T, T)$.

A job $J : (c, r, d)$ of τ is said to be active during $[r, d)$ and inactive otherwise. Accordingly, τ is active when it has an active job, and is inactive otherwise. If Γ is a set of tasks, we use $\mathrm{R}(\Gamma) = \sum_{\tau \in \Gamma} \mathrm{R}(\tau)$ to denote the total rate of tasks in Γ. If Γ is a set of active tasks, we also define $D(\Gamma, t)$ as the next deadline after t of its active jobs. Formally, $D(\Gamma, t) = \min_{\tau \in \Gamma}\{D(\tau, t)\}$. A job can be preempted at any time on a processor and may resume its execution on another processor. We make the incorrect but standard simplifying assumption that there is no cost associated with

such preemptions or migrations. We also assume that a task can only release a new job at or after the deadline of its previous job.

Finally, we say that a processing system has processing resource ρ if it can execute a system of tasks for $\rho\delta$ in any interval of length δ.

3 Fixed-rate servers and quasi-partitions

A server is a scheduling mechanism used to reserve processing bandwidth on a set of processors for a set of tasks or other servers, known as its *clients*. More precisely, we say that a server σ (resp. a scheduling algorithm A) provides processing bandwidth up to ρ to a task set Γ in a time interval of length δ if tasks in Γ can execute up to $\rho\delta$ when scheduled by σ (resp. A) during the interval on a system with ρ processing resources.

A server will present itself as a task to some external scheduling mechanism, like EDF, and will release a series of virtual jobs. When that scheduling mechanism chooses to execute the server, the server will, in turn, use its allocated execution time to schedule its own clients. After defining our servers, we introduce the concept of a particular type of partitions of tasks/servers, called **quasi-partitions**, and illustrate how servers are used in QPS to schedule a quasi-partitioned set of tasks.

3.1 Fixed-rate servers

Definition 3 (*Fixed-rate EDF server*) A *fixed-rate EDF server* σ (henceforth simply "server") is a scheduling mechanism instantiated to regulate the execution of a set of active tasks or other servers Γ, known as its *clients*. A server σ, denoted $\sigma = \sigma(\mathrm{R}(\sigma), \Gamma)$, has a *rate* $\mathrm{R}(\sigma) \leq 1$ which is the processing bandwidth it reserves for its clients. The following rules define the attributes and behavior of a server:

Deadline The next deadline of a server σ after time t is denoted $D(\sigma, t)$. These deadlines will include, but may not be limited to, the deadlines of σ's clients.

Job release A job $J : (c, r, d)$ released by server σ at time r satisfies $c = \mathrm{R}(\sigma)(d - r)$ and $d = D(\sigma, r)$.

Execution order Whenever a job J of server σ executes, σ schedules the jobs of its clients for execution in EDF order.

The server mechanism used in QPS generalizes the one used in RUN (Regnier et al. 2011). RUN servers have rates equal to their clients' summed rates, and have exactly the same release times and deadlines as all their clients. QPS servers are more flexible. Two QPS servers may share a set of clients, or cooperatively schedule a larger set of clients. Thus QPS servers have rates no larger than their summed client rates, and while they will share all the release times and deadlines of their clients, they may include other release times and deadlines as well. QPS servers are only used to schedule active tasks, so if a server's sporadic client ever arrives late, that server will be deactivated.

We note that the above concept of a server is a generalization of a task. Indeed, a task can be seen as a server which schedules a single entity and just has one active client job at a time. Further, a server behaves similarly to a task, releasing jobs whose

execution time is a function of its rate. Hence, hereafter we use the term "server" to refer to both servers and tasks when there is no need to distinguish them; Γ is often referred to as a server set.

3.2 Quasi-partition

QPS partitions the system task set in a particular way, called **quasi-partition**, which is formally defined below.

Definition 4 (*Quasi-partition*) Let Γ be a task set or server set to be scheduled on m identical processors. A quasi-partition of Γ, denoted $\mathcal{Q}(\Gamma, m)$, is a partition of Γ such that:

(i) $|\mathcal{Q}(\Gamma, m)| \leq m$
(ii) $\forall P \in \mathcal{Q}(\Gamma, m), 0 < \mathrm{R}(P) < 2$; and
(iii) $\forall P \in \mathcal{Q}(\Gamma, m), \forall \sigma \in P, \mathrm{R}(P) > 1 \Rightarrow \mathrm{R}(\sigma) > \mathrm{R}(P) - 1$

Each element P in $\mathcal{Q}(\Gamma, m)$ is either a *minor execution set* (if $\mathrm{R}(P) \leq 1$) or a *major execution set* (if $\mathrm{R}(P) > 1$).

The first condition in the above definition rules out partitions with more execution sets than the number of processors. Condition (ii) means that each execution set in a quasi-partition does not require more than two processors to be correctly scheduled. When more than one processor is needed to schedule some P in $\mathcal{Q}(\Gamma, m)$, condition (iii) establishes that the extra processing bandwidth required is less than what is demanded by any server in P. It is worth mentioning that this last property is a cornerstone in QPS for dealing with sporadic tasks, as will be detailed in Sect. 4.2.

There are numerous possible ways of quasi-partitioning a given server set Γ. We define below two possible strategies.

FFD It starts by running First-Fit Decreasing bin packing. That is, the objects (tasks) are packed into bins (execution sets) so that no bin contains objects with summed sizes (rates) exceeding one. FFD packs each object, one at a time in decreasing size order, into the first bin into which it fits, or opens a new bin if the object does not fit into any old one. The number of bins is limited to m. If an object does not fit into any of these m bins, we "overpack" it into the bin which has the *largest* room remaining so as to minimize what exceeds the bin size.

EFD Aiming at distributing objects onto bins in a more evenly fashion, we created the Evenly-Fit Decreasing bin-packing strategy. Like FFD, objects are sorted in decreasing size order. With all m bins open, the first object is packed into the first bin, the second object into the second bin and so on. After using m bins, we start over the iteration considering the i-th object to be packed into the first bin it fits, $i = m + 1, \ldots, n$. If during this packing procedure some object does not fit into any bin, we over-pack the first under-packed bin by placing such an object into it and then return to the packing iteration.

In both heuristics, we observe that if no bin is overpacked, then we have a successful packing, and a proper partitioning of our n tasks onto m processors. Also, overpacking all bins is not possible since this would lead to a partition of Γ into m subsets each with

summed rate exceeding one; we would then have $\mathrm{R}(\Gamma) > m$, and an unfeasible system. Further, we note that conditions (i)–(iii) of Definition 4 are satisfied. In particular, (iii) holds because the smallest object in a bin is the last one added (we pack in decreasing order), and the size of that object is larger than the amount by which the bin is overpacked, *i.e.,* $\mathrm{R}(\sigma) > \mathrm{R}(P) - 1$. Thus, the described implementation leads to a correct quasi-partition and requires $O(n \log n + nm)$ steps, with $O(n \log n)$ steps being necessary for sorting the task set. We denote a quasi-partition implementation as function $\mathcal{Q}(.)$ in the following sections.

3.3 QPS servers

We describe now how servers are managed in QPS so that a valid schedule is generated on a multiprocessor platform.

Definition 5 *(QPS Servers)* Let P be a *major execution set*, as defined in Definition 4, with execution requirement $1 + x$, where $x < 1$ represents the exceeding execution requirement of P regarding its execution on a single processor. Given a bi-partition $\{P^A, P^B\}$ of P with $P^A \neq \emptyset$ and $P^B \neq \emptyset$, we define four QPS servers associated with P as $\sigma^A : (\mathrm{R}(P^A) - x, P^A)$, $\sigma^B : (1 - \mathrm{R}(P^A), P^B)$, $\sigma^S : (x, P)$ and $\sigma^M : (x, P)$. At any time t, all QPS servers associated with P share the same deadline $D(P, t)$. σ^A and σ^B are *dedicated servers* associated with P^A and P^B, respectively. σ^M and σ^S are the *master* and *slave servers*, respectively.

The execution requirement $1 + x$ of P in the above definition is time dependent varying with the load of servers in P. This requirement could be conservatively defined as its maximum value $\mathrm{R}(P)$, as it was the case in Massa et al. (2014). In this paper we extend the definition to provide alternative adaptation strategies. The details of how to compute the effective rate to serve P will be given in Sect. 4.2.2.

Servers σ^A and σ^B deal with the non-parallel execution of P^A and P^B, respectively, while σ^M and σ^S deal with their parallel execution. 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. Further, 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 P^A and one task from P^B execute in parallel; the choice of which executes on behalf of σ^M or σ^S is only a matter of efficiency.

Example 1 illustrates how servers are managed in QPS.

Example 1 Let Γ be a set of periodic tasks that fully utilizes two processors and consider $P = \{\tau_1 : (6, 15), \tau_2 : (12, 30), \tau_3 : (5, 10)\}$ a subset of Γ.

As $\mathrm{R}(P) = 1 + 0.3$, a bi-partition of P may be defined as $P^A = \{\tau_1, \tau_2\}$ and $P^B = \{\tau_3\}$. QPS servers $\sigma^A, \sigma^B, \sigma^M$, and σ^S can be defined as $\sigma^A : (0.5, P^A)$, $\sigma^B : (0.2, P^B)$, $\sigma^M : (0.3, P)$ and $\sigma^S : (0.3, P)$. Figure 2 illustrates how these servers would be scheduled within $[0, 10)$. Servers σ^A, σ^B and σ^S are allocated to the same processor while σ^M executes on another processor. Whenever σ^M is scheduled to execute (by EDF on its processor), σ^S also executes. Task migration decisions are carried out on-line and depend on which task is active when its server executes.

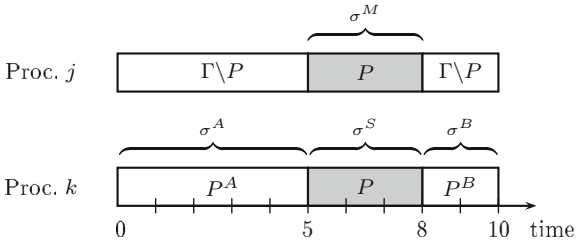

Fig. 2 Illustration of a QPS schedule for the tasks in Example 1. When Processor k schedules σ^M, then σ^S is scheduled on Processor k, and P^A and P^B are executed in parallel. When Processor j is running other tasks, Processor k schedules σ^A or σ^B

4 Quasi-partitioned scheduling

The QPS algorithm has three basic components: the *Partitioner*, the *Manager*, and the *Dispatcher*. The *Partitioner* is responsible for quasi-partitioning tasks/servers into execution sets and for allocating those sets to processors. The *Manager* activates and deactivates QPS servers (Definition 5) in response to system load changes. The *Dispatcher* is responsible for scheduling the active servers in the system.

4.1 The partitioner

The *Partitioner* is an off-line procedure which allocates execution sets to processors, as specified by Algorithm 1. It uses quasi-partitioning from Definition 4 as a subroutine; more precisely, let $\mathcal{Q}(\Gamma, m)$ be the quasi-partition generated for Γ on m processors via FFD or EFD heuristics. Starting with a quasi-partition of a set of n tasks (line 1), the Partitioner allocates an entire processor to any major execution set P and defines an external server σ_{n+j} responsible for reserving processor capacity time for the execution of P on another processor (lines 5–6). Note that the quasi-partition/allocation routine is actually iterative. The external servers from major execution sets are added to the pool Γ (line 6) of tasks/servers from minor execution sets (lines 8–9); this pool is itself

Algorithm 1: QPS Partitioner

Input: Set of n servers Γ_0 and $m \geq \lceil \mathrm{R}(\Gamma_0) \rceil$ processors
Output: Association between processors and execution sets

```
1  P ← Q(Γ_0, m); j ← 0
2  while ∃P ∈ P, R(P) > 1 do
3      Γ ← ∅
4      foreach P ∈ P such that R(P) > 1 do
5          j ← j + 1; x ← R(P) − 1
6          Γ ← Γ ∪ {σ_{n+j} : (x, P)}
7          Allocate dedicated processor j to P
8      foreach P ∈ P such that R(P) ≤ 1 do
9          Γ ← Γ ∪ P
10     P ← Q(Γ, m − j)
11 foreach P ∈ P do
12     j ← j + 1
13     Allocate dedicated processor j to P
```

quasi-partitioned (line 10), and the process is iterated. Once no major execution sets remain, each minor execution set may be given its own processor (lines 11–12). Note that when the initial execution of $\mathcal{Q}(\Gamma_0, m)$ in line 1 returns only minor execution sets, QPS reduces to Partitioned EDF.

We highlight that if P is a major execution set, it is explicitly allocated to a processor that is entirely dedicated to P (line 7), called its *dedicated processor*. The remaining $\mathrm{R}(P) - 1$ processing bandwidth is reserved by the external server on another processor. This server, in turn, could become part of another major execution set, and as a result, P could actually end up running on more than two processors, namely the *shared processors* of P. Although the allocation of shared processors does not appear explicitly in Algorithm 1, it is carried out in lines 7 or 13 whenever the execution set considered contains an external server created in line 6.

Example 2 Let $\Gamma = \{\sigma_1, \sigma_2, \ldots, \sigma_5\}$ be a server set with $\mathrm{R}(\sigma_i) = 0.6$ for $i = 1, 2, \ldots, 5$ scheduled on three processors.

The behavior of the Partitioner is better illustrated with Example 2. Let $\mathcal{P} = \{\{\sigma_1, \sigma_2\}, \{\sigma_3, \sigma_4\}, \{\sigma_5\}\}$ be the initial quasi-partition defined in line 1. As there are two major execution sets in $\mathcal{P}$, the loop in lines 4–7 executes twice. The first and second processors are dedicated to the major execution sets $P_1 = \{\sigma_1, \sigma_2\}$ and $P_2 = \{\sigma_3, \sigma_4\}$, respectively. External servers $\sigma_6 = (0.2, \{\sigma_1, \sigma_2\})$ and $\sigma_7 = (0.2, \{\sigma_3, \sigma_4\})$ are defined in line 6. At the end of the first iteration of the `while` loop, $\mathcal{P} = \{\{\sigma_5, \sigma_6, \sigma_7\}\}$. Since $\mathcal{P}$ contains no major execution sets, the `while` loop exits, and lines 11–12 assign the single remaining minor execution set to the third processor. Thus, at the end of the procedure, processors 1 and 2 are said to be dedicated to major execution sets P_1 and P_2, respectively, and processor 3 is dedicated to minor execution set $P_3 = \{\sigma_5, \sigma_6, \sigma_7\}$. Processor 3 is also said to be shared regarding P_1 and P_2.

Note that the Partitioner defines a hierarchy on processors. For Example 2, the two (dedicated) processors to which servers are allocated are linked to the third (shared) processor via external servers. See Fig. 3 for illustration.

The Partitioner procedure runs in polynomial time. The worst-case size of $\mathcal{P}$ is $|\mathcal{P}| = \lceil \mathrm{R}(\Gamma) \rceil \leq m$, which corresponds to the largest quasi-partition by Definition 4. When defining servers, the rate of $\mathcal{P}$ is decreased by at least 1 (lines 5–6) at each iteration of the while-loop which takes $O(m)$ steps. As the quasi-partitioning procedure can be implemented in $O(n \log n + nm)$, the while-loop runs in $O(nm \log n + nm^2)$. Also, lines 11–12 takes $|\mathcal{P}| < n$ steps. Taking the initial quasi-partitioning in line 1 into consideration, the whole procedure runs in $O(nm \log n + nm^2)$, and so it takes $O(mn^2)$ steps.

We observe the following property:

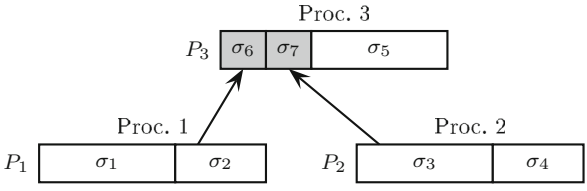

Fig. 3 Illustration of the hierarchy of processors defined by Algorithm 1 for Example 2. Servers σ_6 and σ_7 are external and define reserves for allocating master servers for major execution sets P_1 and P_2

Lemma 1 *Any server set Γ_0 with $\lceil \mathrm{R}(\Gamma_0) \rceil \leq m$ will be allocated no more than m processors by Algorithm 1.*

Proof In each pass through the `while` loop, the value of $\mathrm{R}(\mathcal{P})$ is decreased by the number of calls to lines 5–7 (each major execution set P is replaced with $\sigma_{n+j} : (\mathrm{R}(P) - 1, P)$, and minor execution sets are unchanged); this is also the number of processors allocated by the pass through the `while` loop. The condition of the `while` loop requires that lines 5–7 be called at least once, so the `while` loop can execute at most $\lfloor \mathrm{R}(\Gamma_0) \rfloor$ times. Suppose lines 5–7 have been called a total of j times when the `while` loop exits, so that j processors have been allocated, and $\mathrm{R}(\Gamma) = \mathrm{R}(\Gamma_0) - j \leq m - j$.

From Definition 4(i), it is known that the number of execution sets generated by the final call to $\mathcal{Q}(\Gamma, m - j)$ in line 10 is at most $m - j$. Since the `while` loop is exiting, they must all be minor execution sets. Thus when they are assigned their own processors by line 13, they will not require more than the $m - j$ processors remaining. □

4.2 The manager

Let P be a major execution set with a dedicated processor and an external server responsible for its execution on a shared processor. According to our sporadic task model, it may be possible that servers in P can be safely executed during a given time interval on P's dedicated processor, *i.e.,* with no need of its shared processor. This may happen when some task is not active, for instance. In such a case, the Manager deactivates the QPS servers (Definition 5) in charge of P so that P is simply managed by local EDF during the interval, similarly to a minor execution set.

On the other hand, whenever P is being scheduled by EDF, the arrival of a sporadic job may make necessary the use of part or all of the execution time reserved on P's shared processor via its external server. In such a case, the Manager activates the QPS servers for P, using the reserve defined by its external server to define the master server in charge of P. Thus, a major execution set P can be scheduled according to two modes, EDF and QPS. In QPS mode, QPS servers schedule servers in P on two or more processors while in EDF mode, servers in P are simply scheduled by EDF on a single processor. The transition from one mode to another is called *mode change*.

Note that a mode change carried out for a major execution set can cause mode changes on other major execution sets. This happens when activating/deactivating the master server associated with a major execution set which is part of another major execution set.

Algorithm 2 outlines the main procedure executed by the Manager. At all deadline and release instants t of servers in a major execution set P defined by Algorithm 1, Algorithm 2 determines which mode should be used, QPS or EDF, by calling function $qps\_mode(P, t)$ in line 2. This function returns *True* whenever it is detected that more than one processor is necessary to safely schedule P and QPS mode should be used, and *False* otherwise.

Upon the activation of QPS mode at time t, P is bi-partitioned into P^A and P^B, choosing for P^A a single server among those arrived at t (lines 6–7). Note that, as

Algorithm 2: QPS manager

Input: Major execution set P; release time or deadline of some server in P
Output: Activation/deactivation of QPS servers associated with P

```
1  Let t be a release time or deadline
2  if qps_mode(P, t) // QPS mode
3  then
4      if QPS servers are inactive at t then
5          Let σ ∈ P be the server released at t
6          P^A ← {σ}
7          P^B ← P\{σ}
8          Activate QPS servers
9      x ← qps_rate(P, t)−1
10     Set-up QPS servers:
11         σ^M : (x, P); σ^S : (x, P)
12         σ^A : (R(P^A) − x, P^A)
13         σ^B : (1 − R(P^A), P^B)
14 else // EDF mode
15     Deactivate the QPS servers associated with P
```

all servers are considered inactive before $t = 0$, the single server for A may be chosen arbitrarily if all tasks arrive initially. QPS servers can then be activated. While QPS servers are active, new rates are computed for these servers at all deadlines and release instants. Function $qps\_rate(P, t)$ computes the rate needed by P at time t and its implementation will be given in Sect. 4.2.2. As will be seen in Lemma 4, $qps\_rate(P, t)$ does not return values greater than $\mathrm{R}(P)$. Hence, the corresponding QPS servers will correctly fit into the processor hierarchy defined by Algorithm 1.

It is worth observing that Algorithm 2 takes $O(1)$ steps for managing the activation/deactivation of QPS servers in charge of a major execution set P. Since the activation/deactivation of QPS servers for a set P can cause the activation/deactivation of other QPS servers, which may involve all processor hierarchy (recall Sect. 4.1), the system-wide management operation takes no more than $O(m)$ steps.

There are some options to configure QPS, giving rise to different mode-change strategies, as we now explain.

4.2.1 Mode change strategies

We define three different strategies for carrying out mode change: Conservative with Full-rate (CF); Rate-based with Full-rate (RF); and Rate-based with Partial-rate (RP). If QPS is configured to CF, it is conservatively assumed that whenever all servers of any major execution set P are active, QPS servers providing full rate ($\mathrm{R}(P)$) are needed. If some server is inactive, though, EDF is used instead. According to the rate-based strategies, RF or RP, the rate required by active servers in P, namely $\rho(P, t)$, is accurately computed at release time or deadline instants t (Sect. 4.2.2 gives details on how this rate is computed). If more than one processor is needed, *i.e.,* $\rho(P, t) > 1$, QPS servers are activated by the Manager. Otherwise, EDF is used to schedule P. The

difference between RF and RP is the rate used for setting up QPS servers when it is needed to do so. Although both strategies use $\rho(P, t)$ to decide whether QPS servers are instantiated, RP strategy also uses $\rho(P, t)$ to set up their partial rates while RF always sets up QPS servers at their full rate.

Below we specify functions `qps_mode(P,t)` and `qps_rate(P,t)` that conform with the mode-change strategy chosen. The former determines which mode (QPS or EDF) is to be used. Note that this function indicates that QPS mode is to be used if all servers are active for CF and additionally when $\rho(P, t) > 1$ for RF or RP.

Function qps_mode(P, t)

$A \leftarrow$ all active servers at time t
case *CF:* **return** $(A = P)$

case *RF or RP:* **return** $(A = P$ and $\rho(P, t) > 1)$

Function `qps_rate(P,t)` is only called when in QPS mode (line 9 of Algorithm 2). As in EDF mode, P is to be executed on a single processor by EDF and there is no need to determine the execution rate.

Function qps_rate(P, t)

```
// qps_mode(P,t) equals True
```
case *CF or RF:* **return** $\mathrm{R}(P)$

case *RP:* **return** $\rho(P, t)$

4.2.2 Computation of $\rho(P, t)$

To calculate $\rho(P, t)$, the Manager estimates the maximum *service time* available after t by assuming that all servers in P will be executed at execution rate of $\mathrm{R}(P)$ after $D(P, t)$. Doing so, the Manager determines the maximum demand of P's servers at t that can be postponed after $D(P, t)$. Subtracting this quantity from the demand of servers in P at t, the urgent part of this demand that must execute during $[t, D(P, t))$ can be computed. Finally, $\rho(P, t)$ is obtained by dividing this urgent demand by $D(P, t) - t$. Since each server in P may have its own urgent demand, $\rho(P, t)$ is the maximum of those rates obtained for each server in P. We now detail this procedure.

Let the set of servers with higher or equal priorities than a server σ_i in P at time t be denoted as

$$\mathrm{hp}_i(P, t) = \left\{\sigma_j \in P,\, D(\sigma_j, t) \leq D\,(\sigma_i, t)\right\}$$

For each server σ_i in P, $\phi_i(P, t)$ denotes the *highest possible demand* due to jobs with priority not lower than σ_i's job that may have to be executed from time t

until deadline $D(\sigma_i, t)$. By "highest" we mean the demand due to the worst-case job releasing scenario by sporadic tasks *i.e.,* as if they were periodic. For some server σ_j in $\mathrm{hp}_i(P, t)$, we shall distinguish two kinds of job that contribute to $\phi_i(P, t)$: the jobs released but not yet finished by t and the jobs released from $D(\sigma_j, t)$ with deadlines not greater than $D(\sigma_i, t)$. This latter demand is no more than $\mathrm{R}(\sigma_j)(D(\sigma_i, t) - D(\sigma_j, t))$. Hence,

$$\phi_i(P, t) = \sum_{\sigma_j \in \mathrm{hp}_i(P,t)} \mathrm{E}(\sigma_j, t) + \mathrm{R}(\sigma_j)\big(D(\sigma_i, t) - D(\sigma_j, t)\big) \tag{1}$$

The *service time* available from $D(P, t)$ until $D(\sigma_i, t)$ regarding any σ_i in P is denoted $\psi_i(P, t)$. Assuming that QPS servers provide their full execution rate $\mathrm{R}(P)$ during interval $[D(P, t), D(\sigma_i, t))$ and that servers in P execute at this rate, we obtain

$$\psi_i(P, t) = \mathrm{R}(P)\big(D(\sigma_i, t) - D(P, t)\big) \tag{2}$$

The difference $\phi_i(P, t) - \psi_i(P, D(P, t))$ gives the urgent demand of server σ_i that must be executed in interval $[t, D(P, t))$. Thus, the minimum execution rate needed by P during $[t, D(P, t))$ is

$$\rho(P, t) = \frac{\max_{\sigma_i \in P} \big\{\phi_i(P, t) - \psi_i(P, t)\big\}}{D(P, t) - t} \tag{3}$$

As an example of how to compute the minimum execution rate needed by a major execution set P at a time instant t using Eq. (3), let us consider a major execution set $P = \{\tau_1 : (6, 15), \tau_2 : (12, 30), \tau_3 : (5, 10)\}$. Suppose an arbitrary scenario where, at time 16, tasks deadlines are $D(\tau_1, 16) = 30$, $D(\tau_2, 16) = 30$, and $D(\tau_3, 16) = 26$ whereas the remaining execution times are $E(\tau_1, 16) = 6$, $E(\tau_2, 16) = 4$, and $E(\tau_3, 16) = 5$. Figure 4 illustrates this scenario. Note that $\mathrm{R}(P) = 1.3$ and the next deadline of tasks in P after t is $D(P, 16) = 26$. As the objective of this example is to show how to compute $\rho(P, t)$ for a major execution set P with its respective remaining execution times, the schedule of this system until time 16 is not relevant and therefore will be omitted.

The maximum demand until $D(\tau_1, 16) = 30$ due to jobs with priority not lower than τ_1's priority is $\phi_1 = 15 + 2 = 17$. The service time available from $D(P, t) = 26$ until $D(\tau_1, 16) = 30$ is $\psi_1 = 1.3(30 - 26) = 5.2$. This results in a demand equal to

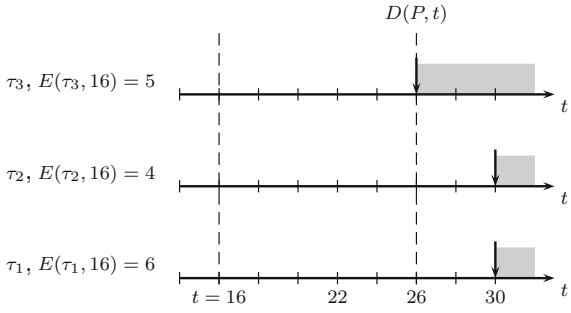

Fig. 4 Illustration of a scenario where $\rho(P, 16)$ is computed for a major execution set $P = \{\tau_1 : (6, 15), \tau_2 : (12, 30), \tau_3 : (5, 10)\}$, with respective deadlines $D(\tau_1, 16) = 30$, $D(\tau_2, 16) = 30$, $D(\tau_3, 16) = 26$ and respective remaining execution times $E(\tau_1, 16) = 6$, $E(\tau_2, 16) = 4$, $E(\tau_3, 16) = 5$

$17-5.2 = 11.8$ to be executed until $D(P, 16) = 26$. As $D(\tau_1, 16) = D(\tau_2, 16) = 30$, the computation for τ_2 yields the same urgent demand. The maximum demand until $D(\tau_3, 16) = 26$ is equal to $\phi_3 = 5 + 0 = 5$ and the service time available from $D(P, 16) = 26$ until $D(\tau_3, 16) = 26$ is null. Hence, the urgent demand regarding τ_3 is 5 time units, which needs to be executed until $D(P, 16) = 26$. Getting the maximum of these three values (11.8, 11.8, 5) and dividing it by the size of the interval $[t = 16, D(P, t) = 26)$ yields

$$\rho(P, 16) = \frac{\max\{11.8, 11.8, 5\}}{26 - 16} = 1.18$$

4.3 The dispatcher

The Dispatcher, outlined in Algorithm 3, visits each processor j for which there is an active server (lines 3–4); selects a server in each processor (lines 5–8); selects the task that must be dispatched (lines 9–14); and finally dispatches the selected tasks (line 16). The procedure is executed whenever a task or server has a job arrival or job complete event.

Recall from Sect. 4.1 that Algorithm 1 defines a hierarchy on processors. Algorithm 3 takes this hierarchy into account. It visits processor in reverse order from that produced by Algorithm 1. As a result, masters are selected before their respective slaves, ensuring their parallel execution.

Considering a specific execution set P allocated to processor j, the dispatching rules are as follows. If P is either a minor execution set or a major execution set in EDF mode, its servers are selected via local EDF on processor j, similarly to a uniprocessor system. On the other hand, if P is a major execution set in QPS mode, its dedicated QPS servers are scheduled according to the hierarchical relation between a master and its slave *i.e.,* whenever the master server executes on the shared processor of P, its associated slave executes on the dedicated processor j so as to ensure the parallel execution requirement of P (lines 5–6). Whenever the master server is not executing, its slave server does not execute either, and the two dedicated QPS servers are selected in arbitrary order since they share the same deadlines (line 8). In any case, servers themselves select their clients via EDF.

Once a server is selected for a given processor j, the task that should execute on j must be determined. As servers may encapsulate other servers, a server chain must be followed until a task is reached, which is done in lines 9–15. Recall that master and slave servers may potentially serve any client from the execution set. Line 14 prevents the same client from running simultaneously on two processors.

Each loop in Algorithm 3 (lines 3 and 9) takes no more than $O(m)$ iterations, limiting the total number of iterations to $O(m^2)$. We note that a more efficient implementation can be derived. For example, processors can be visited following either a sequential order or the processor hierarchy if a master is selected. Doing so, the same processor is not visited more than once, meaning that the server selection would take $O(m)$ steps. We prefer to present the Dispatcher as described in Algorithm 3 for the sake of

Algorithm 3: QPS Dispatcher

```
1  Let t be the current time and d the next deadline after t
2  Let k be the last processor allocated by Algorithm 1
3  for j ← k, k − 1, ..., 1 do
4      if there is an active server at t on processor j then
5          if there is a slave server on processor j whose master was previously selected then
6              Select slave server σ on processor j
7          else
8              Select a non-slave server σ on processor j in EDF order
9          while σ is not a task do
10             P ← the clients of σ
11             if σ is not a slave server then
12                 Select σ′ in P via EDF
13             else
14                 Select σ′ via EDF from whichever of P^A or P^B is not running on P's master server
15             σ ← σ′
16 Execute all tasks on processors they are selected onto
```

simplicity. As for the selection of the highest priority tasks in an ordered queue, it can be done in $O(\log n)$.

4.4 Illustration

The behavior of the QPS algorithm is illustrated by Example 3, which is a modification of Example 1 to fully utilize two processors and to take sporadic tasks into account.

Example 3 Consider a set of sporadic tasks $\Gamma = \{\tau_1 : (6, 15), \tau_2 : (12, 30), \tau_3 : (5, 10), \tau_4 : (3.5, 5)\}$ to be scheduled by QPS on two processors. Assume that all four tasks release their first jobs at time 0 and the second job of τ_3 arrives at $t = 16$ whereas the other tasks are released periodically.

Let $\mathcal{Q}(\Gamma, 2) = \{P_1, P_2\}$ with $P_1 = \{\tau_1, \tau_2, \tau_3\}$ and $P_2 = \{\tau_4\}$, be the quasi-partition for this example, and CF be the chosen mode-change strategy. The Partitioner (Algorithm 1) then allocates P_1 on processor 1 and its external server is allocated on processor 2 together with τ_4. Note that the external server has rate of 0.3 since $\mathrm{R}(P_1) = 1.3$. Since all tasks in P_1 are active at $t = 0$, the Manager (Algorithm 2) activates QPS servers σ^M, σ^S, σ^A and σ^B in charge of P_1. As τ_1, τ_2 and τ_3 were activated simultaneously, the choice of the bi-partition is arbitrary. It is only required that a single task must be assigned to server σ^A. In Fig. 5, which depicts the QPS schedule produced for this example, σ^A and σ^B initially serve $\{\tau_1\}$ and $\{\tau_2, \tau_3\}$, respectively. As $\mathrm{R}(P_1) = 1.3$, the Manager activates the QPS servers as: $\sigma^M : (0.3, P_1)$, $\sigma^S : (0.3, P_1)$, $\sigma^A : (0.1, \tau_1)$, and $\sigma^B : (0.6, \{\tau_2, \tau_3\})$. As the first deadline of tasks in P_1 is $t = 10$, servers σ^M, σ^S, σ^A, and σ^B receive, respectively, initial budgets equal to 3, 3, 1 and 6, all with deadlines at $t = 10$. At time 0 the Dispatcher (Algorithm 3) visits first processor 2 (the last one assigned by the Partitioner), where σ^M and τ_4 are allocated.

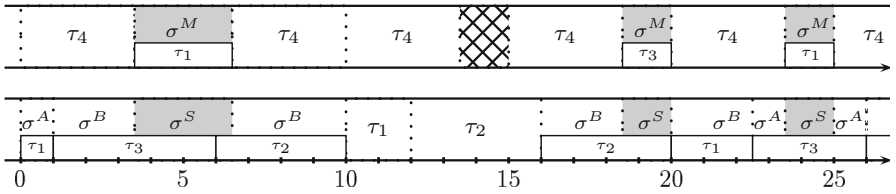

Fig. 5 Quasi-partitioned scheduling for Example 3 assuming that $\mathcal{Q}(\Gamma) = \{\{\tau_1, \tau_2, \tau_3\}, \{\tau_4\}\}$ and the mode-change strategy. The second job of τ_3 arrives late at t = 16

As τ_4 has the highest priority at time 0 (by EDF), it is chosen to execute during [0, 3.5). Also, either σ^A or σ^B can be selected at time 0 on the first processor.

The figure shows a scenario where σ^A is chosen. It then executes its single client, τ_1, until time 1. Then server σ^B starts executing its clients. Task τ_3 has the earliest deadline compared to τ_2 and so it is selected to execute by σ^B until time 3.5 when σ^S starts to execute due to the selection of σ^M on the second (shared) processor. During [3.5, 6.5), σ^M and σ^S execute one client each from partitions $\{\tau_1\}$ and $\{\tau_2, \tau_3\}$. As τ_3 has higher priority than τ_2 (by EDF) and it is already executing on the dedicated processor, it remains so but being served by σ^S. Note that there is no preemption here. Task τ_1 migrates and is served by σ^M on the shared processor.

The remainder of the schedule shown in the figure follows similar reasoning until $t = 10$, when τ_3 becomes inactive. Until the arrival of τ_3's late job (at $t = 16$) the QPS servers are kept deactivated by the Manager (EDF mode). That is, during interval [10, 16) tasks τ_1 and τ_2 are scheduled by EDF on the first processor.

At time 16, the second job of τ_3 arrives making all tasks active. According to Algorithm 2 and CF strategy, the QPS servers are then activated at time 16 as σ^M : (0.3, P_1), σ^S : (0.3, P_1), σ^A : (0.2, $\{\tau_3\}$) and σ^B : (0.5, $\{\tau_1, \tau_2\}$). Now, σ^A serves τ_3 since it is the last task to become active. Tasks τ_1 and τ_2 are now the clients of σ^B. Then, QPS servers release jobs with the same deadline $D(\sigma, 16) = 26$, *i.e.*, σ^M, σ^S, σ^A and σ^B release jobs with execution times 3, 3, 2, and 5, respectively. The next schedule decisions follow similar reasoning.

We observe that if either RF or RP were used instead of CF, the Manager would calculate $\rho(P_1, 16) = 1.18$ and would activate the QPS servers at time 16. Note that this is the same scenario of Fig. 4. Thus, P_1 would be managed by local EDF until time $D(P_1, 16) = 26$. Postponing the activation of the QPS servers this way is useful for reducing preemption and migration overheads, as will be shown in Sect. 6.

5 Proof of correctness

For the proofs below, we assume that the QPS algorithm consists of a set of execution sets created by Algorithm 1, managed by servers in Algorithm 2, and dispatched by Algorithm 3. For time interval $\Delta = [t, t')$, we say that Δ is a *complete EDF interval* for a major execution set P if the Manager activates EDF mode for P at time t (possibly $t = 0$), and next activates QPS mode at t' (so that P executes in EDF mode throughout Δ). Similarly, we say that Δ is a *complete QPS interval* if a phase of QPS execution

begins at t (again, possibly $t = 0$) and ends at t'. Thus in complete QPS intervals all tasks in P are active whereas in complete EDF intervals some task in P may be inactive. Finally, Δ is a *QPS job interval* if some task of P releases a job at t, and $t' = D(P, t)$ is the next deadline of any task of P. Thus a complete QPS interval is divided into a sequence of QPS job intervals, and P's four managing servers each release a new job within each QPS job interval.

As described in Sect. 4, a major execution set P is alternately scheduled by QPS servers or by EDF. Hence, we need to show that this jointly generated schedule is valid. To do so, we first assume that QPS servers are consistently created by Algorithm 2, *i.e.,* QPS server rates for a major execution set P do not sum up more than $R(P)$. For the case when the mode change strategies RF or CF are being used, this assumption is trivially granted. As for RP, the assumption means that $\rho(P, t) \leq R(P)$. In any case, we refer to a QPS server considering that it was consistently created. Under this condition, we show that QPS correctly schedules P from the beginning of a complete QPS interval until the starting time of another (Lemmas 2, 3). Then, we prove that the assumed condition always holds (Lemma 4). Lemma 5 then proves that P is always correctly scheduled provided that its master server is correctly scheduled. Finally, Theorem 1 completes the proof by inductively extending the lemmas to all execution sets.

We begin to prove the correctness of QPS with Lemma 2 which states that whenever created by the QPS manager, as specified by Algorithm 2, QPS servers meet their deadlines so long as the master server of the same execution set also meets its deadlines.

Lemma 2 *Let P be a major execution set and consider a complete QPS interval $\Delta = [t, t')$. If the master server σ^M in charge of P is scheduled on its shared processor so that it meets all its deadlines in Δ, then the other three QPS servers will also meet theirs on P's dedicated processor.*

Proof Let $\sigma^M, \sigma^S, \sigma^A$ and σ^B be the QPS servers in charge of P during Δ. σ^M is part of some other execution set, and is being scheduled on a shared processor. We assume this schedule allows σ^M to meet all its deadlines during Δ. Algorithm 3 schedules σ^S whenever σ^M executes and so σ^S also meets all of its deadlines. The remaining time on the dedicated processor is filled with the execution of σ^A and σ^B in their correct proportion by Algorithm 3 since $\mathrm{R}(\sigma^S) + \mathrm{R}(\sigma^A) + \mathrm{R}(\sigma^B) = 1$. As all four QPS servers share the same release times and deadlines, they all met their deadlines.

Once the four QPS servers in charge of an execution set P meet their deadlines, Lemmas 3, 4 and 5 assure that the elements of P will also meet their deadlines.

Lemma 3 *Let P be a major execution set. If $\rho(P, t) \leq \mathrm{R}(P)$ where t is the starting time of a QPS interval, the individual tasks and servers in P will meet all their deadlines until the starting time of the next QPS interval provided that the master server in charge of P meets its deadlines.*

Proof As we are taking a local view of major execution set P, let us refer to its elements as "the tasks" (though some may actually be the master servers of other major execution sets), and σ^M, σ^S, σ^A, and σ^B as "the servers". By our assumption

that the master server in charge of P meets its deadlines and by Lemma 2, QPS servers in charge of P meet their deadlines in any complete QPS interval providing P an execution rate of $\rho(P, t)$. So, by assumption, Eq. (3) returns a value not greater than $\mathrm{R}(P)$ and so those four QPS servers can always be consistently defined at t. As per Algorithm 2, when we partition P into P^A and P^B, P^A will contain a singleton task τ which arrived at the beginning of the current complete QPS interval (possibly at $t = 0$ if all servers arrived then). Suppose that processor j is the dedicated processor of P.

We focus on proving the lemma by showing that all deadlines in P are met when QPS is configured to the RP mode-change strategy. As the mode-change strategies RF and CF provide higher execution rates to QPS servers, the correctness of the RP strategy implies that strategies RF and CF are also correct. Our proof of correctness here relies primarily on the observation that processor j is always running uniprocessor EDF on P^B, sometimes with a minor modification. Even though the elements of P^B may change as P goes into and out of QPS mode, these changes conform to standard EDF scheduling.

(A) QPS mode Let $[r, d)$ with $r \geq t$ be a QPS job interval, so that each of P's four servers releases a job at r with deadline d. Recall that σ^A, σ^B, and σ^S are scheduled on processor j. Now, while the Dispatcher may choose either P^A or P^B to be served by σ^M, since σ^M and σ^S only execute in parallel, let us suppose WLOG that σ^M serves P^A and σ^S serves P^B (swapping these may affect migration count, but never execution time allotted to tasks). Thus processor j is scheduling P^B via EDF whenever σ^A isn't running.

The QPS rate computed by Eq. (3) gets the maximum of its arguments, so the rate calculated at time r satisfies

$$\rho(P, r) \geq \frac{\phi_i(P, r) - \psi_i(P, r)}{d - r} \tag{4}$$

for any server σ_i.

Consider a server σ_i in P with deadline at d, *i.e.,* $D(\sigma_i, r) = d$. We will show that σ_i (and so any other server in P with deadlines at d) meets its deadline. As σ_i has the first deadline after r, for any server $\sigma_k \in \mathrm{hp}_i(P, r)$, we have that $D(\sigma_k, r) = D(\sigma_i, r) = d$. This means that $\psi_i(P, r) = 0$, reducing (4) to

$$\phi_i(P, t) \leq \rho(P, r)(d - r) \tag{5}$$

and making

$$\phi_i(P, r) = \sum_{\sigma_k \in \mathrm{hp}_i(P,r)} \mathrm{E}(\sigma_k, r) \tag{6}$$

Recall from Algorithm 2 that $x = \rho(P, r) - 1$ and $\mathrm{R}(\sigma^M) = x$. As σ^M executes for $x(d - r)$ in $[r, d)$ and by our assumption it meets all of its deadlines, the demand of P to be executed in processor j during $[r, d)$ is $\phi_i(P, r) - x(r - d)$. Using (5) and (6), we have that the total demand with deadline at d to be executed in processor j is

$$\sum_{\sigma_k \in \mathrm{hp}_i(P,r)} \mathrm{E}(\sigma_k, r) - x(d-r) \leq \rho(P,r)(d-r) - x(d-r) = (d-r)$$

which is successfully scheduled by EDF on processor j during interval $[r, d)$. That is, if we were to simply run an EDF scheduler on processor j, it would have to schedule σ^A and the servers in P^B for this same amount of time during $[r, d)$. This EDF schedule is the same schedule produced by QPS, with σ^B and σ^S both serving P^B, except that σ^A may not execute at the same time as σ^M. However, moving and/or subdividing the execution of σ^A within $[r, d)$ (so as to avoid overlap with σ^M) cannot affect the correctness of the schedule, as there are no job releases or deadlines within $[r, d)$. Thus all deadlines in P^B must be met while in QPS mode, as the schedule produced for P^B is equivalent to a (guaranteed correct) EDF schedule.

(B) Migrating task Recall from Algorithm 2 that $x = \rho(P, r) - 1$, $\mathrm{R}(\sigma^M) = x$, and $\mathrm{R}(\sigma^A) = \mathrm{R}(P^A) - x$, so that $\mathrm{R}(\sigma^M) + \mathrm{R}(\sigma^A) = \mathrm{R}(P^A)$. Since $P^A = \{\tau\}$, $\mathrm{R}(\sigma^M)$ and $\mathrm{R}(\sigma^A)$ will collectively do $\mathrm{R}(\tau)(d-r)$ units of work on τ during any QPS job interval $[r, d)$. This is assured since our assumption and Lemma 2 guarantee that σ^M and σ^A meet their deadlines. That is, at *any* deadline of some task of P, τ has received its correct proportion of work, and so will meet any deadline it has during QPS execution. Further, if P gets into EDF mode, when we switch τ to executing exclusively on processor j under EDF, the work remaining on its current job is in the correct proportion $\mathrm{R}(\tau)$ to the time remaining on that job. More precisely, if P enters EDF mode at time t', then $E(\tau, t') = \mathrm{R}(\tau)(D(\tau, t') - t')$.

(C) Entering EDF mode Here, $[r, d)$ is an EDF interval. Assume first that QPS is configured to the CF mode-change strategy. In this case, when a task of P fails to release a new job at its deadline and puts P into EDF mode, EDF execution of P^B [see (A) above] continues uninterrupted on processor j. As P's four servers are deactivated, we will no longer schedule σ^A along with P^B. If it is $\tau \in P^A$ that fails to arrive, then P^B continues to execute via EDF as before. If some $\tau \in P^B$ fails to arrive, then the remainder of the current job of $\tau \in P^A$ is moved exclusively to processor j. When this happens at time t, the EDF scheduler on processor j may treat this as if it were a newly arrived job with work $E(\tau, t)$, deadline $D(\tau, t)$, and rate $\mathrm{R}(\tau)$, as noted in (B). By construction, any proper subset of P (such as $P^B + \{\tau\} - \{\tau'\}$) must have rate less than one. As the set of jobs now being scheduled on processor j (σ^A and τ' are gone, but τ has "arrived") still has rate no more than one, EDF will continue to schedule it correctly. This will be the case while other jobs fail to arrive on time, and late jobs reappear, so long as not all jobs of P are active at once.

When QPS is configured to the RP/RF mode-change strategy and P gets into EDF mode, we have that $\rho(P, r) \leq 1$. The next deadline after r is (for some task σ_i) $D(\sigma_i, r) = d$ and all servers with deadline at d are in $\mathrm{hp}_i(P, r)$. Thus, using (5) and (6) again, we have that

$$\sum_{\sigma_k \in \mathrm{hp}_i(P,r)} \mathrm{E}(\sigma_k, t) \leq \rho(P,r)(d-r) \leq (d-r)$$

and there is no deadline miss during $[r, d)$. Similar reasoning applies to any EDF interval as long as the system runs in EDF mode.

(D) Leaving EDF mode Now suppose first that the last late task of P arrives at time t, when P switches from EDF mode back into QPS mode. Let's call this task τ, though it may be different from the task in P^A during the previous QPS execution phase. We now re-partition P so that $P^A =$ (this new) $\{\tau\}$, and $P^B = P - \{\tau\}$, and allot P's four managing servers accordingly. From the perspective of the EDF scheduler on processor j, all previous tasks from EDF mode are still present, and one new job of σ^A has arrived, which brings the total rate of tasks on processor j up to one. Hence QPS execution resumes as in (A).

Thus, except as noted in (B), all scheduling of P is handled by the persistent EDF scheduler running processor j (subject to some safe rearranging within QPS job intervals, as noted in (A)). Since the rate load of jobs being scheduled on processor j never exceeds one, EDF guarantees that all deadlines will be met. Further, the singleton task in P^A will meet its deadlines so long as σ^M does the same. □

Lemma 4 ensures that QPS servers can always be consistently instantiated since function $\rho(P, t)$, as defined by Eq. (3), is safe after a correct period scheduled by QPS.

Lemma 4 *At any time t at which $\rho(P, t)$ is computed for a major execution set P, $\rho(P, t) \leq \mathrm{R}(P)$ provided that P is correctly scheduled by QPS until t.*

Proof Assume by contradiction that t is the earliest instant such that $\rho(P, t) > \mathrm{R}(P)$. According to Algorithm 2, all servers in P are active at t; otherwise $\rho(P, t)$ would not be computed. Let σ_k be the server such that

$$\rho(P, t) = \frac{\phi_k(P, t) - \psi_k(P, t)}{D(P, t) - t} > \mathrm{R}(P) \tag{7}$$

and $S \subseteq \mathrm{hp}_k(P, t)$ a set of servers that release a set of jobs J before t with deadlines after t, which includes a job of σ_k.

Our proof arguments are based on a specific but legal scenario according to which the interference in the execution of the jobs in J is maximized; and so is the value of $\rho(P, t)$. To maximize the interference in the jobs of J, we consider that servers in $P' = P \setminus S$ release their jobs at time 0 with deadline at t. Likewise, we consider that all servers in S release their jobs at 0 with deadlines at r, where r is the release time of their jobs in J. Releasing all these jobs at 0 like we did, delays the execution of jobs in J, maximizing the value of $\rho(P, t)$, as we wished. Further, if our assumption that $\rho(P, t) > \mathrm{R}(P)$ holds for an arbitrary job release pattern, it will also hold for our constructed scenario. We distinguish two cases:

Case 1 $S = \{\}$. This means that all servers in P are released at t, since they are all active at t. As

$$\sum_{\sigma_j \in \mathrm{hp}_k(P,t)} \mathrm{E}(\sigma_j, t) = \sum_{\sigma_j \in \mathrm{hp}_k(P,t)} \mathrm{R}(\sigma_j)\big(D(\sigma_j, t) - t\big)$$

we have that

$$\begin{aligned}\phi_k(P,t) &= \sum_{\sigma_j \in \mathrm{hp}_k(P,t)} \mathrm{R}(\sigma_j)\big(D(\sigma_j,t)-t\big) + \mathrm{R}(\sigma_j)\big(D(\sigma_k,t)-D(\sigma_j,t)\big) \\ &= \sum_{\sigma_j \in \mathrm{hp}_k(P,t)} \mathrm{R}(\sigma_j)\big(D(\sigma_k,t)-t\big)\end{aligned}$$

Since $\mathrm{hp}_k(P,t) \subseteq P$, the above equation implies that

$$\begin{aligned}\phi_k(P,t) &\leq \sum_{\sigma_j \in P} \mathrm{R}(\sigma_j)\big(D(\sigma_k,t)-t\big) \\ &\leq \mathrm{R}(P)\big(D(\sigma_k,t)-t\big)\end{aligned}$$

From Eq. (2), we finally obtain,

$$\phi_k(P,t) - \psi_k(P,t) \leq \mathrm{R}(P)\big(D(P,t)-t\big)$$

which contradicts (7).

Case 2 $S \neq \{\}$. Let X be the workload executed by QPS before t. Observe that the value of X for the scenario we constructed is exactly $R(P)t$ because: (i) by assumption t is the earliest time when $\rho(P,t) > \mathrm{R}(P)$ and so, at any time $t' < t$ at which $\rho(P,t')$ was computed, its value is not above $R(P)$; (ii) By assumption, P is correctly scheduled by QPS during $[0,t)$; (iii) by construction of our scenario, all tasks are active during $[0,t)$ and the total workload released until t is not less than $R(P)t$. However, in the following, we show that $\rho(P,t) > \mathrm{R}(P)$ implies $X < \mathrm{R}(P)t$, contradicting $X = \mathrm{R}(P)t$.

Our contradiction assumption ($\rho(P,t) > \mathrm{R}(P)$, which keeps valid for our scenario) implies that

$$\phi_k(P,t) - \psi_k(P,t) > \mathrm{R}(P)\big(D(P,t)-t\big)$$

Substituting $\psi_k(P,t)$, defined by Eq. (2), in the above relation yields

$$\phi_k(P,t) - \mathrm{R}(P)\big(D(\sigma_k,t)-D(P,t)\big) > \mathrm{R}(P)\big(D(P,t)-t\big)$$

Hence,

$$\phi_k(P,t) > \mathrm{R}(P)\big(D(\sigma_k,t)-t\big) \tag{8}$$

Considering that $\phi_k(P,t)$ does not take into account jobs of servers in $P \setminus \mathrm{hp}_k(P,t)$, from Eq. (8) we have that the maximum workload of P that must be executed within $[t, D(\sigma_k,t))$, namely Φ, is

$$\Phi \geq \phi_k(P,t) > \mathrm{R}(P)\big(D(\sigma_k,t)-t\big) \tag{9}$$

The worst-case workload released by P during $[0, D(\sigma_k,t))$ to be executed up to $D(\sigma_k,t)$ is precisely $\mathrm{R}(P)D(\sigma_k,t)$, which occurs when all servers in P have a common deadline $D(\sigma_k,t)$. Also, Φ is the difference between the maximum workload released

by P to be executed up to $D(\sigma_k, t)$ and the workload already executed by QPS before t. Hence, $\phi = \mathrm{R}(P)D(\sigma_k, t) - X$ and from Eq. (9), we obtain

$$\mathrm{R}(P)D(\sigma_k, t) - X > \mathrm{R}(P)\big(D(\sigma_k, t) - t\big)$$

which implies that $X < \mathrm{R}(P)t$. A contradiction, since we know by construction that $X = \mathrm{R}(P)t$. □

Lemma 5 *Let P be a major execution set scheduled by QPS. The individual tasks and servers in P will meet all their deadlines provided that the master server in charge of P meets its deadlines.*

Proof Let t_i and t_{i+1} be the time instants at which the i-th and $(i+1)$-th QPS complete intervals begin. We apply a simple induction on the intervals $[t_i, t_{i+1})$.

Base case Consider the first and second complete QPS intervals, starting at t_1 and t_2, respectively. Either $t_1 = 0$ (all servers in P arrive at t_1); or before t_1 the system was continuously on EDF mode. In the former case, Lemma 4 states that $\rho(P, t_1) \leq R(P)$ (since no deadline up to t_1 is missed) and thus, from Lemma 3, we know that no deadline is missed until t_2 since by assumption the master server in charge of P meets its deadlines. For the sake of argumentation in the latter case, construct a scenario where all servers are released at some time $t_0 < t$, where t is when the first job by servers in P was released. Consider also that all jobs released at t_0 have deadlines at t. These artificial releases of jobs at t_0 does not change anything in the schedule from t onwards. As this scenario is similar to the former case, Lemmas 3 and 4 apply again, leading to the same conclusion.

Induction step Assume that all deadlines of servers in P were met until t_i. This means that $\rho(P, t_i) \leq \mathrm{R}(P)$ (by Lemma 4). Hence, as the master server in charge of P meets its deadlines, we have from Lemma 3 that all tasks and servers in P meet all their deadlines until t_{i+1}. □

The correctness of the QPS scheduler now follows easily by induction.

Theorem 1 *QPS produces a valid schedule for a set of implicit-deadline sporadic tasks Γ on $m \geq \lceil \mathrm{R}(\Gamma) \rceil$ identical processors.*

Proof By Lemma 1, given any task set Γ with $\mathrm{R}(\Gamma) \leq m$, Algorithm 1 will assign Γ to at most m processors. We must show that no task in Γ misses its deadlines. More generally, we will see that no task or server in the system misses deadlines.

Recall that minor execution sets are assigned to their own dedicated processors (see lines 11–13 of Algorithm 1), and that those processors are scheduled using uniprocessor EDF (line 8 of Algorithm 3). Since minor execution sets have rates no more than one, the optimality of EDF (Baruah et al. 1990) guarantees that no deadlines are missed in minor execution sets.

The `while` loop of Algorithm 1 cannot exit while major execution sets remain, so the last processor allocated (say, processor k) must to be a minor execution set. This, and all other minor execution sets, will serve as our base cases. Recall that Algorithm 3 schedules processors in reverse order. Suppose processors $k, k-1, \ldots, j+1$ are

scheduled at time t so that none of their next deadlines will be missed, and suppose that P is a major execution set assigned to dedicated processor j. From Lemma 4 we know that the on-line activations of QPS servers are safe since their rates never sum up more than $\mathrm{R}(P)$ implying that the master server σ^M of P will fit into the external server size defined by Algorithm 1. Server σ^M will be part of some "later" execution set which has been assigned some dedicated processor from $k, k-1, \ldots, j+1$. By our induction hypothesis, σ^M will meet its next deadline after t, and so by Lemma 5, the next deadline from P will also be met. Extending this to all processors and all scheduling instants, all deadlines in Γ will be met, and we conclude that QPS is an optimal scheduling algorithm for sporadic task sets. □

6 Evaluation

If preemptions and migrations were actually instantaneous, as we have assumed, then all optimal schedulers would be of roughly equal merit. However, since it is the time costs of these operations that limit the use of optimal schedulers in practice, we use the number of these operations as the primary metric for comparing various optimal schedulers. The performance of QPS in terms of task preemption and migration was assessed via simulation. During simulation, preemptions are counted only when the execution of a preempted task is resumed on the same processor as before. If the task execution is resumed on a different processor, we consider this as a migration event.

Random synthetic task sets were generated according to the procedure described by Emberson et al. (2010). The rate of each generated task was uniformly distributed in (0.00, 1.00) with integer period uniformly distributed within [1, 100]. Each simulation generated 1,000 task sets and ran for 1,000 time units. Since the task set generation procedure tends to produce small task rates in large task sets and this would favor QPS (due to partitioning), task sets had no more than $4m$ tasks, with m ranging from 2 to 32 processors.

Since the performance of QPS depends on the processor hierarchy created by quasi-partitioning, this property is examined in Sect. 6.1. Section 6.2 discusses the effects of different quasi-partition implementations on the performance of QPS. We do not intend to evaluate quasi-partitioning heuristics, though. The intention is to highlight which characteristics of these heuristics affect the performance of QPS. Sections 6.3 and 6.4 compare QPS against other optimal scheduling algorithms using periodic and sporadic task systems.

6.1 Processor hierarchy

A major execution set may initially appear on the first processor; its master server may then create a major execution set on the second processor, and so forth. In this case, quasi-partitioning forms a processor chain of maximum size. For example, quasi-partitioning a set of 10 tasks with rates equal to 0.9 to be scheduled on 9 processors forms a processor chain with 8 major execution sets and one minor execution set. Tasks belonging to the jth processor in the chain may migrate to any of the $m-j$ processors located upwards along the chain. This means that the larger the processor

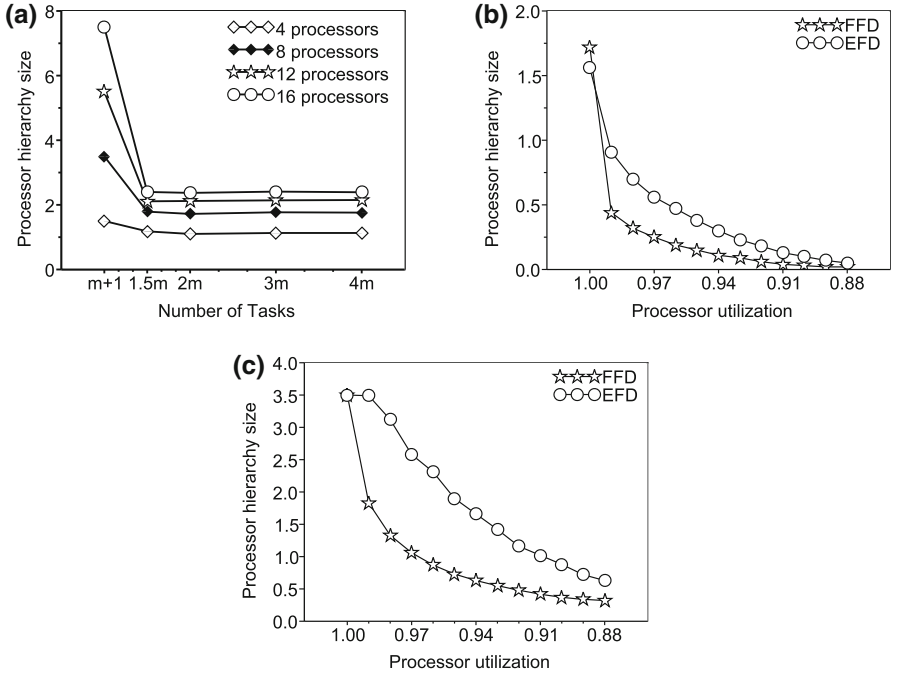

Fig. 6 Average processor hierarchy. **a** Systems that fully utilize m processors, **b** systems with 8 processors and 16 tasks, **c** systems with 8 processors and 9 tasks

hierarchy, the higher the expected overhead in terms of preemption and migration, mainly for tasks that are down in the processor chain. Conversely, if some sporadic task belonging to the first execution set becomes inactive, that execution set enters EDF mode; the removal of the master server from the next processor causes it (and, similarly, all other processors above it in the chain) to also enter EDF mode. In summary, the performance of QPS is dependent on the processor hierarchy produced during the quasi-partitioning and execution set allocation phase, but larger hierarchies may alter performance in positive or negative ways.

In this section we compute the average hierarchy size for various task set batches considered during the simulation. The average hierarchy size is given by the average processor level. In the example of 9 processors given above, the average hierarchy size would be $\frac{1}{9}(0 + 1 + \cdots + 8) = 4$.

Figure 6a summarizes the results found for the considered task sets for the FFD quasi-partitioning heuristics. The results for EFD present the same basic behavior, although they differ in the absolute figures. As can be seen, each task set has rate equal to m to promote large hierarchy formations. As expected, when there are $m+1$ tasks in the system, the average hierarchy size was $\frac{m-1}{2}$. Interestingly, this value rapidly drops for larger task sets. Figure 6b and c show the difference between FFD and EFD for systems with $m = 8$ processors considering different processor utilization values. As expected, the lower the system utilization, the lower the average processor hierarchy sizes. FFD is in general a better heuristic. When the system becomes fully utilized,

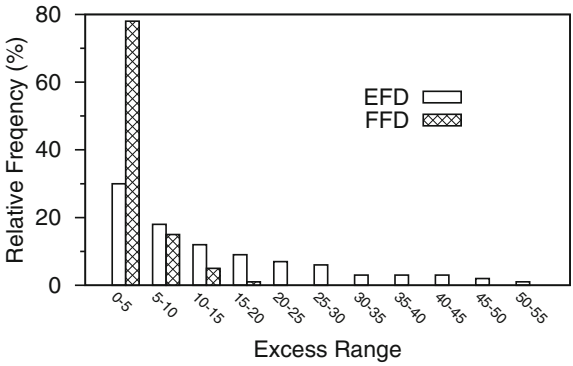

Fig. 7 Distribution of the excess in execution sets due to packing policy for fully utilized task sets with 16 tasks scheduled on 8 processors

EFD is shown to be slightly better than FFD for $2m$ tasks and exhibit the same value for $m + 1$ tasks. As will be seen shortly, there is a correlation between what is shown in these graphs and the performance of QPS for periodic task systems (Sect. 6.3).

6.2 Quasi-partitioning heuristics

The histogram-like graph of Fig. 7 shows the distribution of excess rates used to set up external servers in Algorithm 1 (line 6) for systems with 16 tasks using 100 % of 8 processors. These rate values represent the maximum rates of master/slave servers, activated in Algorithm 2 and so they play a role in QPS performance. Indeed, the lower the master server rate, the better dealing with sporadic tasks since major execution sets may be kept running on EDF mode more often mainly when QPS is configured to RF or RP. In this context, as can be seen in the graph, FFD tends to favor the scheduling of sporadic tasks: about 80 % of excess rates generated for FFD are not larger than 5 %; EFD distributes the generated excess more evenly. For system with lower processor utilization values, both FFD and EFD present similar distributions.

6.3 Performance for periodic task systems

Figure 8 shows the performance of QPS (using EFD) against other optimal scheduling algorithms considering periodic tasks that fully utilize the system processors and so mode-changes do not take place. For FFD, the found results were very similar due to their marginal difference in the generated hierarchy size for fully utilized systems, as pointed out in Sect. 6.1. Each point in the graphs corresponds to the average of results for 1, 000 task sets. We observe that as DP-Wrap (DPW) (Levin et al. 2010) and EKG (Andersson and Tovar 2006) use deadline-equality enforcement to achieve optimality, they tend to perform worse than QPS, RUN (Regnier et al. 2011) and U-EDF (Nelissen et al. 2012), which use different strategies. Overall, RUN presents the best results for these periodic systems, whereas the performance of U-EDF lies in between those found for RUN and QPS. As all servers are always active (periodic tasks) and the systems are fully utilized, the adaptation strategies of QPS do not apply. Even though, QPS achieves good results as compared to the best known algorithms to date.

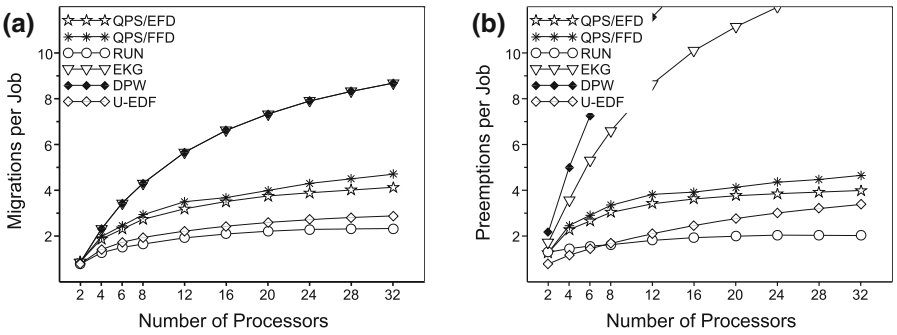

Fig. 8 Average number of preemptions and migrations for periodic systems with $2m$ tasks that fully utilize the processors making use of the EFD quasi-partitioning heuristic. **a** Migrations, **b** preemptions

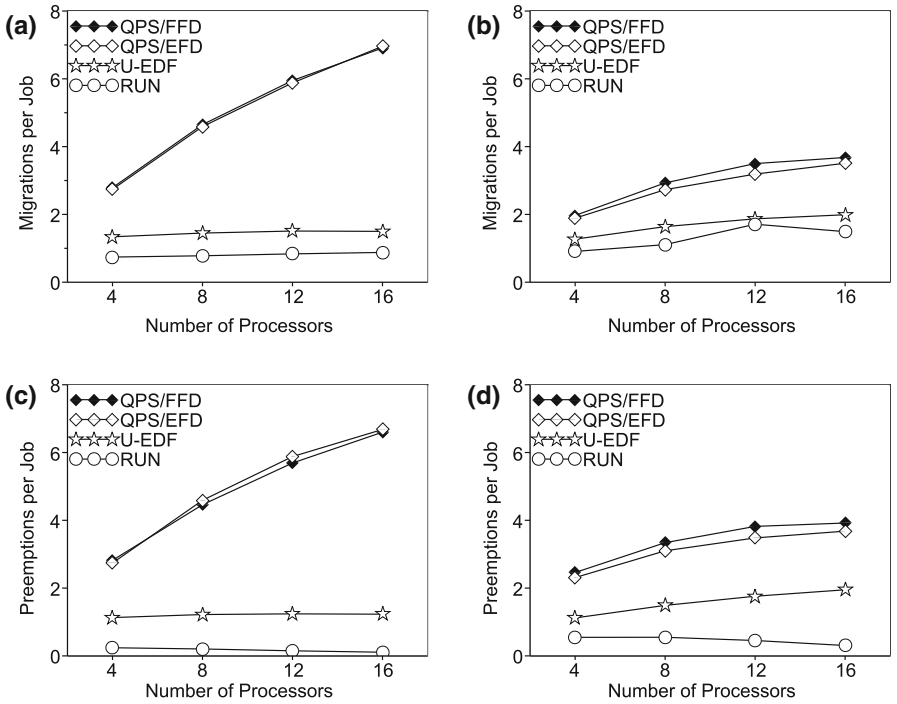

Fig. 9 Average number of migrations and preemptions for periodic task systems which fully utilize m processors. **a** Migrations with $m + 1$ tasks, **b** migrations with $2m$ tasks, **c** preemptions with $m + 1$ tasks, **d** preemptions with $2m$ tasks

To the best of our knowledge, RUN and U-EDF are the best optimal scheduling algorithms for periodic and sporadic task systems known to date, respectively. Therefore, they will be taken hereafter as our baseline comparison.

Figure 9 shows how QPS compares against RUN and U-EDF for periodic systems with a varying number of processors, and either $m+1$ or $2m$ tasks. Again, all systems in the figures require 100 % of m processors. As can be seen in the graphs, the performance

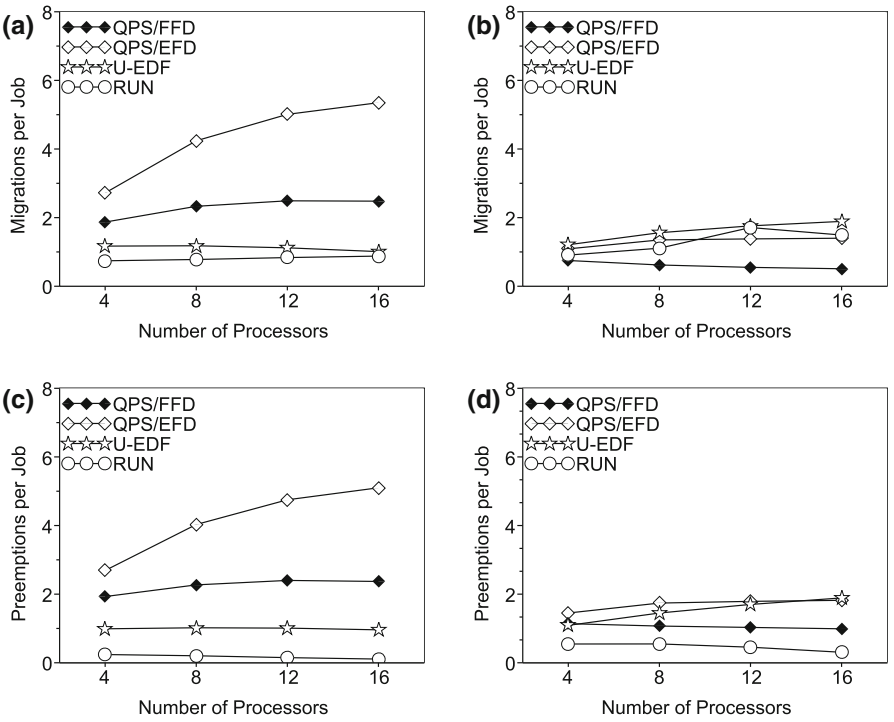

Fig. 10 Average number of migrations and preemptions for periodic task systems which utilize 98 % of m processors. **a** Migrations with $m + 1$ tasks, **b** migrations with $2m$ tasks, **c** preemptions with $m + 1$ tasks, **d** preemptions with $2m$ tasks

of QPS for $m + 1$ tasks is not as good as in the other scenarios. This is expected due to the fact that quasi-partitioning systems with $m + 1$ tasks fully utilizing m processors produces large processor hierarchies (recall Sect. 6.1). The performance of QPS significantly improves when larger task sets are considered. In the graphs we show only systems with up to $2m$ tasks. The behavior for larger task sets does not change significantly. Even considering that the adaptation strategies used in QPS do not apply (all servers are always active and the system is fully utilized), QPS performs very well.

Consider in Fig. 9 the points corresponding to $m = 8$ processors for both quasi-partitioning heuristics, FFD and EFD. As can be seen, for $m + 1$ tasks, the found number of preemption and migration is the same for both heuristics. EFD exhibits a slightly better result. This behavior can be explained using the data plotted in Fig. 6b and c, which show that the processor hierarchy size: Whereas FFD and EFD produce the same values for $m + 1$ tasks, EFD behaves slightly better for $2m$ tasks.

Interestingly, for systems that do not require 100 % of the processors the performance of QPS improves significantly, and can be comparable to that of RUN in some cases. This is illustrated in Fig. 10, which presents results obtained for systems utilizing 98 % of m processors. This considerable improvement corresponds to reduced hierarchy sizes, as no processor chain was found to be larger than 2 at 98 % utilization. As can be seen by comparing Figs. 9 and 10, this slight drop in utilization has very

little effect on the performance of RUN and U-EDF. QPS, by comparison, benefits greatly when even a little slack is provided to its off-line partitioner.

6.4 Performance for sporadic task systems

The great advantage of using QPS comes when sporadic task systems are considered, as it will be evident in this section. As RUN does not deal with sporadic tasks, we compare QPS against U-EDF only.

All task sets in this section have rates summing to m. When a task is to be activated, the simulator generates a task activation delay, whose values are uniformly distributed in the continuous interval [0, max.delay]. This causes system load fluctuations. The Manager deactivates the QPS servers in charge of any major execution set P whenever some of their tasks/servers are late (CF mode-change strategy); or $\rho(P, t) < 1$ (RP or RF mode-change strategies).

During the experiments we first observed that the performance of QPS does not significantly vary with the number of processors in the system. The pattern found is very similar to the one shown in Fig. 11, which presents the results for $m = 16$ processors. Interestingly, systems with $m + 1$ tasks present a lower number of migrations than systems with $1.5m$ tasks, so long as average arrival delay is sufficiently high (generally more than 10 time units). Under the periodic case (or low maximum delay), the large processor hierarchies lead to more migrations between related processors. However, any late arrival lower in a long processor chain will cause transitions to EDF mode all the way up the chain, with all affected processors behaving like Partitioned EDF, and suffering no migrations during this time.

Figure 12 better illustrates how QPS converges very nicely from global to partitioned scheduled systems, except for a few residual migrations, when task activation delays increase. U-EDF, on the other hand, does not vary as a function of activation delays, an expected behavior (Nelissen 2013). As expected, under the conservative mode-change strategy (CF), higher figures are observed. The effect of using rate-based approaches (RF or RP) is indeed evident. However, the simplicity of the CF approach (based on monitoring whether or not all active tasks in a major execution set

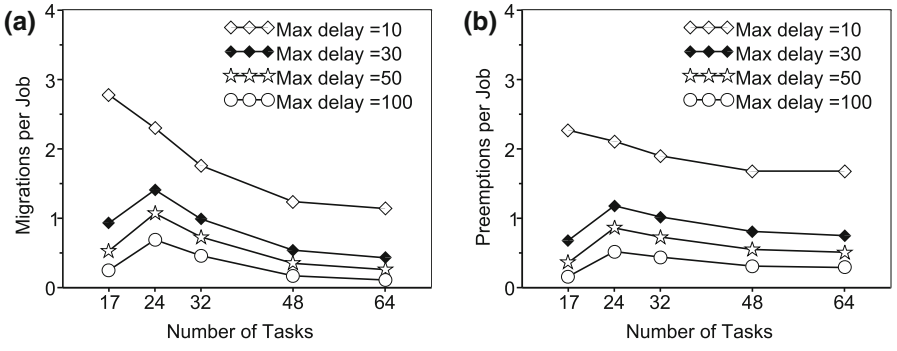

Fig. 11 Average number of preemptions and migrations for systems with $m = 16$ processors and maximum task activation delay ranging from 10 to 100. **a** Migrations, **b** preemptions

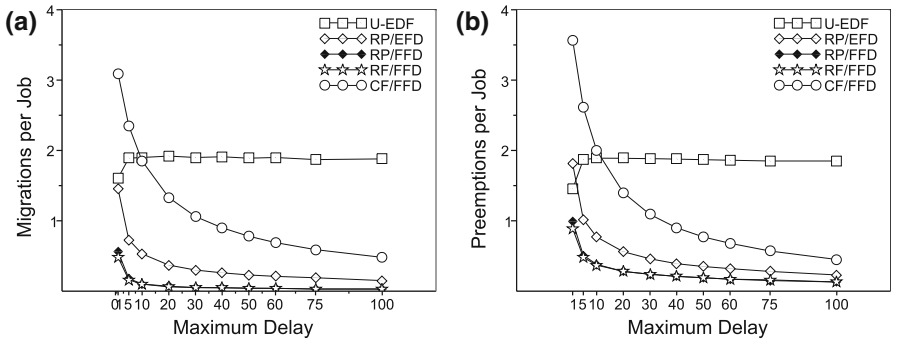

Fig. 12 Average number of migrations and preemptions for sets with 16 sporadic tasks scheduled on 8 processors. **a** Migrations, **b** preemptions

are active) favors the cost of implementation. In the graphs of Fig. 12 we also plotted data related to using the EFD packing heuristics. As mentioned in Sect. 6.2 due to the generated excess rates distribution, EFD tends to respond worse than FFD when sporadic tasks are being considered. However, as can be observed, the difference is not that significant.

7 Conclusion

We have described QPS, a new optimal algorithm for scheduling real-time sporadic implicit-deadline tasks. By switching between partitioned EDF and global scheduling rules at run-time, QPS performs competitively with state-of-the-art global schedulers on periodic task sets, and outperforms similar schedulers on sporadic task sets. Different configurations of QPS have been described, including two quasi-partitioning heuristics and three adaptation strategies. Results from simulation have evaluated several combinations of such configurations against related work. It has been shown that when sporadic tasks are considered, QPS is able to provide on-line adaptation, dynamically tuning between the partitioned-to-global multiprocessor scheduling spectrum. The results shown in this paper may well motivate further developments. Adaptation may be explored in different dimensions, *e.g.* slack sharing between servers would favor scheduling soft real-time tasks. Other interesting questions are related to implementation issues in an actual operating system or to mechanisms to manage resource sharing between tasks. The results showed here are solid foundations for these and other developments.

Acknowledgments This work has been funded by CNPq and CAPES. The authors would like to thank to Geoffrey Nelissen for his comments about U-EDF and his help in its simulations.

References

Andersson B, Bletsas K (2008) Sporadic multiprocessor scheduling with few preemptions. In: Euromicro conference on real-time systems (ECRTS), pp 243–252. doi:10.1109/ECRTS.2008.9

Andersson B, Tovar E (2006) Multiprocessor scheduling with few preemptions. In: IEEE embedded and real-time computing systems and applications (RTCSA), pp 322–334
Baruah SK, Mok AK, Rosier LE (1990) Preemptively scheduling hard-real-time sporadic tasks on one processor. In: IEEE real-time systems symposium (RTSS), pp 182–190
Baruah SK, Cohen NK, Plaxton CG, Varvel DA (1996) Proportionate progress: a notion of fairness in resource allocation. Algorithmica 15(6):600–625
Bastoni A, Brandenburg BB, Anderson JH (2011) Is semi-partitioned scheduling practical? In: Proceedings of 23rd euromicro conference real-time systems, pp 125–135
Bletsas K, Andersson B (2009) Notional processors: an approach for multiprocessor scheduling. In: IEEE real-time and embedded technology and applications symposium (RTAS), pp 3–12. doi:10.1109/RTAS.2009.25
Bletsas K, Andersson B (2011) Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. Real-Time Syst 47(4):319–355
Burns A, Davis R, Wang P, Zhang F (2011) Partitioned EDF scheduling for multiprocessors using a C=D scheme. Real-Time Syst 48(1):3–33
Cho H, Ravindran B, Jensen ED (2006) An optimal real-time scheduling algorithm for multiprocessors. In: IEEE real-time systems symposium (RTSS), pp 101–110
Compagnin D, Mezzetti E, Vardanega T (2014) Putting run into practice: implementation and evaluation. In: Euromicro conference on real-time system, pp 75–84
Easwaran A, Shin I, Lee I (2009) Optimal virtual cluster-based multiprocessor scheduling. Real-Time Syst 43(1):25–59
Emberson P, Stafford R, Davis RI (2010) Techniques for the synthesis of multiprocessor tasksets. In: Workshop on analysis tools and methodologies for embedded and real-time systems (WATERS), pp 6–11
Funaoka K, Kato S, Yamasaki N (2008) Work-conserving optimal real-time scheduling on multiprocessors. In: Euromicro conference on real-time systems (ECRTS), pp 13–22
Funk S (2010) LRE-TL: an optimal multiprocessor algorithm for sporadic task sets with unconstrained deadlines. Real-Time Syst 46(3):332–359
Kato S, Yamasaki N, Ishikawa Y (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: Proceedings of 21st Euromicro conference on real-time systems (ECRTS), pp 249–258
Koren G, Amir A, Dar E (1998) The power of migration in multi-processor scheduling of real-time systems. In: ACM-SIAM symposium on discrete algorithms (SODA), pp 226–235
Levin G, Funk S, Sadowski C, Pye I, Brandt S (2010) DP-FAIR: a simple model for understanding optimal multiprocessor scheduling. In: Euromicro conference on real-time systems (ECRTS), pp 3–13
Massa E, Lima G, Regnier P, Levin G, Brandt S (2014) Optimal and adaptive multiprocessor real-time scheduling: the quasi-partitioning approach. In: Euromicro conference on real-time system, pp 291–300
McNaughton R (1959) Scheduling with deadlines and loss functions. Manag Sci 6(1):1–12
Nelissen G (2013) Private communication
Nelissen G, Berten V, Nelis V, Goossens J, Milojevic D (2012) U-edf: an unfair but optimal multiprocessor scheduling algorithm for sporadic tasks. In: 24th Euromicro conference on real-time systems. IEEE Computer Society, Los Alamitos, pp 13–23. doi:10.1109/ECRTS.2012.36
Regnier P, Lima G, Massa E, Levin G, Brandt S (2011) Run: optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: IEEE real-time systems symposium (RTSS), pp 104–115
Santos-Jr JAM, Lima G, Bletsas K, Kato S (2013) Multiprocessor real-time scheduling with a few migrating tasks. In: Proceedings of the 34th IEEE real-time systems symposium, pp 170–181
Zhu D, Mossé D, Melhem R (2003) Multiple-resource periodic scheduling problem: how much fairness is necessary? In: IEEE real-time systems symposium (RTSS), pp 142–151

Ernesto Massa is Researcher and Professor of Computer Science at the State University of Bahia (UNEB), Brazil. Ernesto main research focus is on real-time scheduling theory. He is member of the STER Group (Real-Time Systems Research Group - UFBA) and co-author of some awarded works in the area. In 2015, Dr. Massa created the GRAAL Research Group (UNEB) with the aim of promoting the development of research in computer science in the countryside. Ernesto received his Ph.D. degree in Computer Science, his M.Sc. degree in Mathematics and his B.Sc. degree in Computer Science from the Federal University of Bahia in 2014, 2004 and 1991, respectively.

George Lima is an Associate Professor at Federal University of Bahia (UFBA), Brazil. At UFBA, he has assumed leadership positions such as Head of Department and Graduate Program Coordinator. More recently he has created STER, a research group at UFBA whose focus is on both practical and theoretical aspects of real-time systems. Dr. Lima received his Ph.D. in 2003 from The York University, UK, his M.Sc. in 1996 from State University of Campinas, and his B.Sc. in 1993 from UBFA. All degrees in Computer Science.

Paul Regnier received a B. Physics degree from the University Paris XI, France, in 1990 and the M.Sc. degree and the Ph.D. degree in Computer Science from the Federal University of Bahia (UFBa), Brazil, in 2008 and 2012, respectively. Since 2014, he has taken a permanent position at Federal University of Bahia as Professor of Computer Science. His main research interests are in the area of real-time systems, covering topics such as scheduling, operating systems, communication networks, formal specification and distributed systems. He recently coauthored the papers "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor.", which received the best paper award at RTSS 2011 and "Optimal and Adaptive Multiprocessor Real-Time Scheduling: The Quasi-Partitioning Approach.", which received the best paper award at ECRTS 2014.

Greg Levin holds a Ph.D. in Computer Science from University of California, Santa Cruz, where worked with Dr. Scott Brandt in the Systems Research Lab, dealing with the Hard Real-Time Multiprocessor Scheduling problem. He also holds a Ph.D. in Mathematical Sciences from the Johns Hopkins University, where he wrote his dissertation on Fractional Graph Theory. Currently, he works at Google on the Chrome OS team.

Scott Brandt is Vice Chancellor for Research and Professor of Computer Science at the University of California, Santa Cruz. As Vice Chancellor for Research, Dr. Brandt provides executive leadership and coordination for UCSC research administration and planning. Brandt spent 10 years in industry prior to joining the UCSC faculty in 1999. He holds eight patents and has launched three startups. In 2014, he was elected Fellow of the National Academy of Inventors. Brandt's research focuses on computer systems, specifically storage systems, real-time systems, and system performance management. He is co-founder of the UCSC Storage Systems Research Center and the UCSC/Los Alamos National Lab Institute for Scalable Scientific Data Management. He has published 145 peer-reviewed papers and articles and has received research grants and gifts totaling over 20 million. His work has been incorporated into products and projects by a number of well-known companies and his Ceph high-performance distributed storage system is standard in OpenStack and Red Hat Linux. Brandt received his Ph.D. in Computer Science from the University of Colorado in 1999 and his M.S. in Computer Science and B. Math. from the University of Minnesota in 1993 and 1987.