Optimal and Adaptive Multiprocessor Real-Time Scheduling: The Quasi-Partitioning Approach

Ernesto Massa†, George Lima‡, Paul Regnier‡
†State University of Bahia (UNEB),
†Salvador University (UNIFACS/PPGCOMP),
‡Federal University of Bahia (UFBA)
Salvador, Bahia, Brasil
Email: {ernestomassa,gmlima,pregnier}@ufba.br

Greg Levin, Scott Brandt
University of California,
Santa Cruz, USA
Email: {glevin, sbrandt}@soe.ucsc.edu

***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.**

I. 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.,* [1]–[10]. 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 [10].

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 [11] 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.,* [12]–[15]. Task migration control mechanisms must be applied at run-time. Dividing the system into task/processor clusters is another option, *e.g.,* [7]. In both cases, optimality can be achieved by some hybrid approaches by, again, enforcing short execution windows across the system *e.g.,* [7]–[9], [16], causing high run-time overheads [17].

Two recently described global scheduling algorithms, RUN [18] and U-EDF [19], achieve optimality with low preemption and migration overheads. 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. Quasi-Partitioned Scheduling is a new algorithm capable of scheduling any feasible system composed of independent implicit-deadline sporadic tasks on identical processors. 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 EDF (Earliest Deadline First). 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 Figure 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 excess capacity of that processor is used to handle the excess requirements of P_1. The

1068-3070/14 $31.00 © 2014 IEEE
DOI 10.1109/ECRTS.2014.19

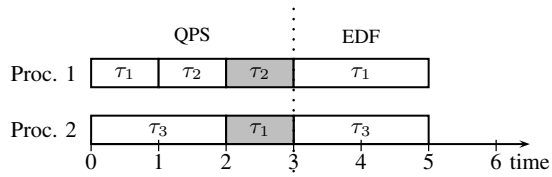


Figure 1. QPS schedule for three tasks. All tasks arrive at time 0; tasks τ_1 and τ_3 are activated with period 3; but the second job of τ_2 is late. Task migration occurs in the period $[0, 3)$; partitioned scheduling is applied during [3,6) due to a late task arrival.

QPS 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 7. 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 Figure 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.

Paper structure. Section II defines the system model and establishes the notation used in the paper. We introduce the generalized *fixed-rate server* in Section III, and present the concept of a *quasi-partition* in Section III-B. Section IV details the QPS scheduling algorithm and Section V proves it correct. Results obtained from simulations are discussed in Section VI. Final comments are given in Section VII.

II. 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 II.1 (Job)**.** *A job $J{:}(c, r, d)$ of a task τ is an instance of τ 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. When a task τ releases a job $J{:}(c, r, d)$, its execution time c equals $\mathrm{R}(\tau)(d - r)$, as usual. However, the active duration $d - r$ is no longer constant over the jobs of τ, and so neither is c. 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 II.2 (Fixed-Rate Task)**.** *A* fixed-rate task *(henceforth simply "task") τ, with rate $R(\tau) \leqslant 1$, 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 $\tau{:}(R(\tau)T, T)$.*

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 after the deadline of its previous job.

III. Fixed-Rate Servers and Quasi-Partitions

A server is a scheduling mechanism used to reserve processor time for a set of tasks or other servers, known as its *clients*. 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, called **quasi-partitions**, and illustrate how servers are used in QPS to schedule a quasi-partitioned set of tasks.

A. Fixed-Rate Servers

Definition III.1 (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 $\mathrm{R}(\sigma) \leqslant 1$ *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 [18]. 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.

B. Quasi-Partition

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

Definition III.2 (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) $|\mathcal{Q}(\Gamma, m)| \leqslant m$
(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 $\mathrm{R}(P) \leqslant 1$) 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 resources required are 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 Section IV-B.

There are numerous possible ways of quasi-partitioning a given server set Γ. Our chosen method begins by running First-Fit Decreasing (FFD) 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.

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 $R(\Gamma) > m$, and an unfeasible system. Further, we note that conditions (i)-(iii) of Definition III.2 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.

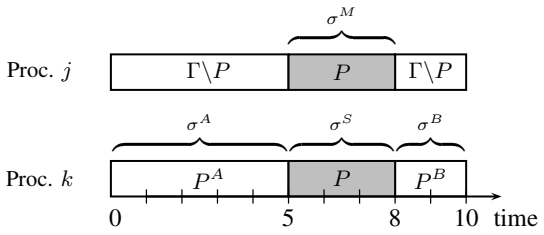


Figure 2. Illustration of a QPS schedule for the tasks in Example III.1. When Processor k schedules σ^M, then σ^S is scheduled on Processor k, and P^A and P^B are executed in parallel. When Processor j is running other tasks, Processor k schedules σ^A or σ^B.

C. QPS Servers

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

Definition III.3 (QPS Servers)**.** *Let P be a* major execution set*, as defined in III.2, with accumulated rate $\mathrm{R}(P) = 1 + x$. Given a bi-partition $\{P^A, P^B\}$ of P, we define four QPS servers associated with P as $\sigma^A{:}(\mathrm{R}(P^A) - x, P^A)$, $\sigma^B{:}(\mathrm{R}(P^B) - x, P^B)$, $\sigma^S{:}(x, P)$ 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.*

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 III.1 illustrates how servers are managed in QPS.

Example III.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.

IV. 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 into execution sets and for allocating those sets to processors. The *Manager* activates and deactivates QPS servers (Definition III.3) in response to system load changes. The *Dispatcher* is responsible for scheduling the active servers in the system.

A. 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 III.2 as a subroutine; more precisely, let $\mathcal{Q}(\Gamma, m)$ be the quasi-partition generated for Γ on m processors via FFD bin packing. 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 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 a processor that is entirely dedicated to P (line 7), called its *dedicated processor*. The remaining $\mathrm{R}(P)-1$ processing capacity 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.

Algorithm 1: QPS Partitioner

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

1 $\mathcal{P} \leftarrow \mathcal{Q}(\Gamma_0, m)$; $j \leftarrow 0$
2 **while** $\exists P \in \mathcal{P}, \mathrm{R}(P) > 1$ **do**
3 $\Gamma \leftarrow \emptyset$
4 **foreach** $P \in \mathcal{P}$ *such that* $\mathrm{R}(P) > 1$ **do**
5 $j \leftarrow j + 1$; $x \leftarrow \mathrm{R}(P) - 1$
6 $\Gamma \leftarrow \Gamma \cup \{\sigma_{n+j}{:}(x, P)\}$
7 Allocate dedicated processor j to P
8 **foreach** $P \in \mathcal{P}$ *such that* $\mathrm{R}(P) \leqslant 1$ **do**
9 $\Gamma \leftarrow \Gamma \cup P$
10 $\mathcal{P} \leftarrow \mathcal{Q}(\Gamma, m - j)$
11 **foreach** $P \in \mathcal{P}$ **do**
12 $j \leftarrow j + 1$
13 Allocate dedicated processor j to P

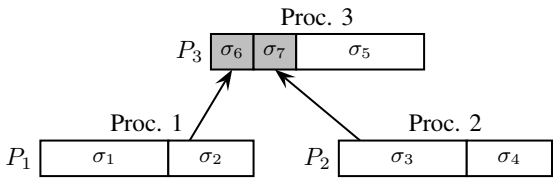


Figure 3. Illustration of the hierarchy of processors defined by Algorithm 1 for Example IV.1. Servers σ_6 and σ_7 are external and define reserves for allocating master servers for major execution sets P_1 and P_2.

Example IV.1. *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 IV.1. 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 IV.1, the two (dedicated) processors to which servers are allocated are linked to the third (shared) processor via external servers. See Figure 3 for illustration.

The Partitioner procedure runs in polynomial time. The worst-case size of $\mathcal{P}$ is $|\mathcal{P}| = \lceil \mathrm{R}(\Gamma) \rceil \leqslant m$, which corresponds to the largest quasi-partition by Definition III.2. 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:

Lemma IV.1. *Any server set* Γ_0 *with* $\lceil \mathrm{R}(\Gamma_0) \rceil \leqslant m$ *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

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 all servers in P are active at t then // QPS mode
3     if QPS servers of P are inactive at t then
4         Let σ ∈ P be a server released at t
5         P^A ← {σ}; P^B ← P\{σ}; x ← R(P) − 1
6         Define and then activate servers σ^M:(x, P);
          σ^S:(x, P); σ^A:(R(P^A) − x, P^A); and
          σ^B:(1 − R(P^A), P^B)
7 else if QPS servers of P are active then // EDF mode
8     Deactivate the QPS servers associated with P
```

$\lfloor \mathrm{R}(\Gamma_0) \rfloor$ 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 $\mathrm{R}(\Gamma) = \mathrm{R}(\Gamma_0) - j \leqslant m - j$.

From Definition III.2(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. ■

B. 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. In such a case, the Manager deactivates the QPS servers (Definition III.3) 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. QPS mode should be used whenever it is detected that more than one processor is necessary to safely schedule P. This can be checked by carrying out an on-line schedulability test for EDF at run-time. However, we take a simple approach by conservatively assuming that whenever all servers are active, QPS mode should be used and EDF mode 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 4-5). Note that, as all servers are considered inactive before $t = 0$, the single server for A may be chosen arbitrarily if all tasks arrive initially. QPS servers are then activated with rates computed as in 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 Section IV-A), the system-wide management operation takes no more than $O(m)$ steps.

C. 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 Section IV-A 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.

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're selected onto
```

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

D. Illustration

The behavior of the QPS algorithm is illustrated by Example IV.2, which is a modification of Example III.1 to fully utilize two processors and to take sporadic tasks into account.

Example IV.2. *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 behave 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. 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 Figure 4, 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. 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.

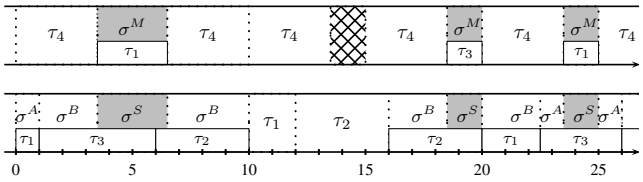


Figure 4. Quasi-partitioned scheduling for Example IV.2 assuming that $\mathcal{Q}(\Gamma) = \{\{\tau_1, \tau_2, \tau_3\}, \{\tau_4\}\}$ and the mode-change strategy. The second job of τ_3 arrives late at t=16.

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, the QPS servers are then activated at time 16 as $\sigma^M{:}(0.3, P_1)$, $\sigma^S{:}(0.3, P_1)$, $\sigma^A{:}(0.2, \{\tau_3\})$ and $\sigma^B{:}(0.5, \{\tau_1, \tau_2\})$. 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.,* $\sigma^M, \sigma^S, \sigma^A$ and σ^B release jobs with execution times $3, 3, 2$, and 5, respectively. The next schedule decisions follow similar reasoning.

V. Proof of Correctness

As described in Section IV, 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 show in Lemma V.1 that QPS servers meet their deadlines so long as the master server of the same execution set also meets its deadlines. It then follows in Lemma V.2

that the elements of a major execution set will also meet their deadlines. Finally, Theorem V.1 completes the proof by inductively extending the lemmas to all execution sets.

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 the complete QPS intervals are precisely the times when all tasks in P are active. Finally, Δ is a *QPS job interval* if some task of P releases a job at t, and $t' = D(\Gamma, t)$ 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.

Lemma V.1. *Let P be a major execution set and consider a complete QPS interval Δ. 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 σ^M, σ^S, σ^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, all their deadlines are met. ■

Lemma V.2. *Let P be a major execution set. 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: 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." As per Algorithm 2, when we partition P into A and B, 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. Our proof of correctness here depends primarily on the observation that processor j is always running uniprocessor EDF on B, sometimes with a minor modification. Even thought the elements of B may change as P goes into and out of QPS mode, these changes conform to standard EDF scheduling.

(A) QPS Mode: P enters QPS mode when all tasks are active, and leaves QPS mode when some task is late arriving. Let $[r, d)$ 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 A or 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 A and σ^S serves B (swapping these may affect migration count, but never execution time allotted to tasks). Thus processor j is scheduling B via EDF whenever σ^A isn't running. Let us consider $B' = B \cup \{\sigma^A\}$, noting that $\mathrm{R}(B') = 1$. Since σ^A releases a job within each QPS job interval $[r, d)$, the time $\mathrm{R}(\sigma^A)(d - r)$ allotted to σ^A within $[r, d)$ is exactly enough to finish this job. If we were to simply run an EDF scheduler for B' on processor j, it would have to schedule σ^A 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 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 B must be met while in QPS mode, as the schedule produced for B is equivalent to a (guaranteed correct) EDF schedule produced for B'.

(B) Migrating task: Recall from Algorithm 2 that $x = \mathrm{R}(P) - 1$, $\mathrm{R}(\sigma^M) = x$, 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 $A = \{\tau\}$, $\mathrm{R}(\sigma^M)$ 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 V.1 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 some task in B arrives late and puts P 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: When a task of P fails to release a new job at its deadline and puts P into EDF mode, EDF execution of 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 B. If it is $\tau \in A$ that fails to arrive, then B continues to execute via EDF as before. If some $\tau' \in B$ fails to arrive, then the remainder of the current job of $\tau \in A$ 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 $B + \tau - \tau'$) 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.

(D) Leaving EDF Mode: Now suppose that the last late task of P arrives at time t, switching P from EDF mode

back into QPS mode. Let's call this task τ, though it may be different from the task in A during the previous QPS execution phase. We now re-partition P so that $A =$ (this new) τ, and $B = P - \tau$, 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 A will meet its deadlines so long as σ^M does the same. ■

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

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

Proof: By Lemma IV.1, given any task set Γ with $\mathrm{R}(\Gamma) \leqslant m$, 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 [20] guarantees that no deadlines are missed in minor execution sets.

For major execution sets, we proceed inductively on processors, last to first. The `while` loop of Algorithm 1 cannot exit while major execution sets remain, so the last processor allocated (say, processor k) must be to 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. The master server σ^M of P 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 V.2, 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. ■

VI. 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 Emberson *et al.*'s procedure [21]. 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 is highly dependent on the hierarchy of processors created by quasi-partitioning, this property is examined in Section VI-A. Then Sections VI-B and VI-C compare QPS against other optimal scheduling algorithms using periodic and sporadic task systems.

A. Processor hierarchy

In the worst case, 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 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 5 summarizes the results found for the considered task sets. 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.

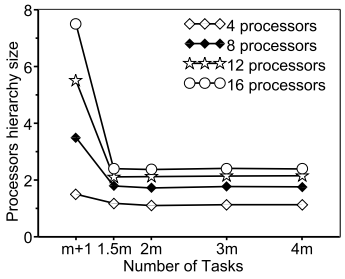
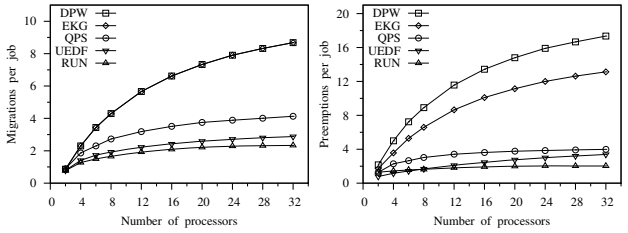


Figure 5. Average processor hierarchy for systems that fully utilize 100% of m processors.

B. Performance for periodic task systems

Figure 6 shows the performance of QPS against other optimal scheduling algorithms on periodic task sets that fully utilize the system processors. Each point in the graphs corresponds to the average of results for $1,000$ task sets. We observe that as DP-Wrap (DPW) [10] and EKG [8] use deadline-equality enforcement to achieve optimality, they tend to perform worse than QPS, RUN [18] and U-EDF [19], 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.



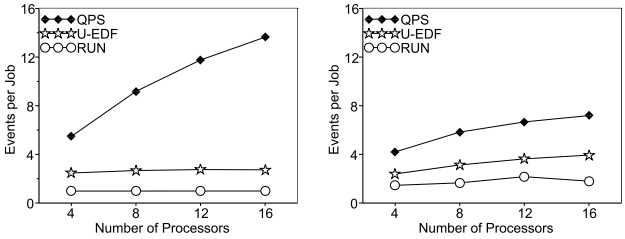
(a) Migrations (b) Preemptions

Figure 6. Average number of preemptions and migrations for periodic systems with $2m$ tasks that fully utilize the processors.

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.

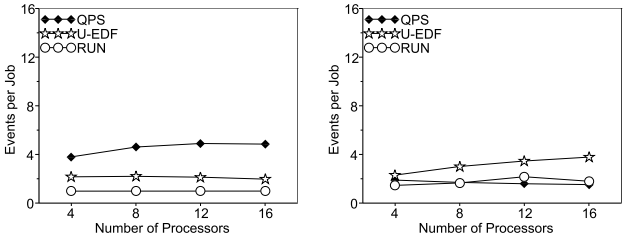
Figure 7 shows how QPS compares against RUN and U-EDF for periodic systems with a varying number of processors, and either $m + 1$ or $2m$ tasks. Again, all systems in the figures require 100% of m processors. As can be seen in the graphs, the performance of QPS for $m + 1$ tasks is not as good as in 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 Section VI-A). 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. It is worth observing that as all servers are always active (periodic tasks), the adaptation strategy used in QPS does not apply for the simulations in this sections. Even so, QPS performs very well.

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 Figure 8, 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 Figures 7 and 8, 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.



(a) Migrations and Preemptions with $m + 1$ tasks (b) Migrations and Preemptions with $2m$ tasks

Figure 7. Average number of migrations and preemptions for periodic task systems which fully utilize m processors.



(a) Migrations and Preemptions with $m + 1$ tasks (b) Migrations and Preemptions with $2m$ tasks

Figure 8. Average number of migrations and preemptions for periodic task systems which utilize 98% of m processors.

C. Performance for sporadic task systems

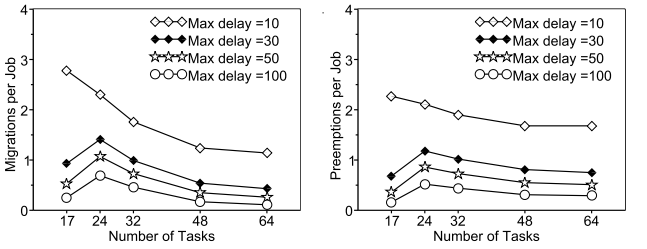
The great advantage of using QPS comes when sporadic task systems are considered, as it will be evident in this section. As RUN does not deal with sporadic tasks, we compare QPS against U-EDF only.

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 $[0, \text{max. delay}]$. If such a task belongs to some major execution set, the Manager deactivates the corresponding QPS servers.

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 Figure 9, which presents the results

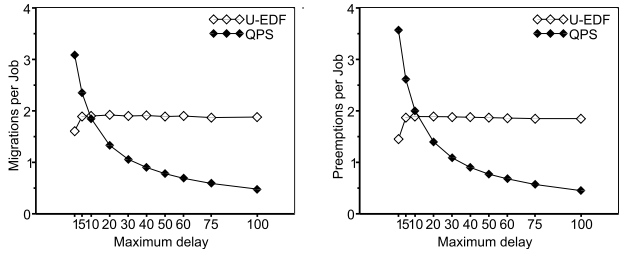
for $m = 16$ processors. Interestingly, systems with $m + 1$ tasks present a lower number of migrations than systems with $1.5m$ tasks, so long as average arrival delay is sufficiently high (generally more than 10 time units). Under the periodic case (or low maximum delay), the large processor hierarchies 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 10 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 [22].



(a) Migrations (b) Preemptions

Figure 9. 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

Figure 10. Average number of migrations and preemptions for sets with 16 sporadic tasks scheduled on 8 processors.

VII. 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. Extensions to other task models, different adaptation strategies, and different quasi-partitioning implementations will be part of future work.

Acknowledgment

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

[1] R. McNaughton, "Scheduling with deadlines and loss functions," *Management Science*, vol. 6, no. 1, pp. 1–12, 1959.

[2] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, 1996.

[3] D. Zhu, D. Mossé, and R. Melhem, "Multiple-resource periodic scheduling problem: how much fairness is necessary?" in *IEEE Real-Time Systems Symposium (RTSS)*, 2003, pp. 142–151.

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

[5] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, 2006, pp. 101–110.

[6] 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.

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

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

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

[10] 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 Systems (ECRTS)*, 2010, pp. 3–13.

[11] G. Koren, A. Amir, and E. Dar, "The power of migration in multiprocessor scheduling of real-time systems," in *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1998, pp. 226–235.

[12] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Proc. 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009, pp. 249–258.

[13] A. Burns, R. Davis, P. Wang, and F. Zhang, "Partitioned EDF Scheduling for Multiprocessors using a C=D Scheme," *Real-Time Systems*, vol. 48, no. 1, pp. 3–33, 2011.

[14] K. Bletsas and B. Andersson, "Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound," *Real-Time Systems*, vol. 47, no. 4, pp. 319–355, 2011.

[15] J. A. M. Santos-Jr, G. Lima, K. Bletsas, and S. Kato, "Multiprocessor Real-Time Scheduling with a Few Migrating Tasks," in *Proc. of the 34th IEEE Real-Time Systems Symposium*, 2013, pp. 170–181.

[16] K. Bletsas and B. Andersson, "Notional processors: An approach for multiprocessor scheduling," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2009, pp. 3–12.

[17] A. Bastoni, B. B. Brandenburg, and J. H. Anderson, "Is semi-partitioned scheduling practical?" in *Proc. 23rd Euromicro Conf. Real-Time Systems*, 2011, pp. 125–135.

[18] 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 Systems Symposium (RTSS)*, 29 2011-dec. 2 2011, pp. 104–115.

[19] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-edf: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *24th Euromicro Conference on Real-Time Systems*. Los Alamitos, CA, USA: IEEE Computer Society, 2012, pp. 13–23.

[20] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *IEEE Real-Time Systems Symposium (RTSS)*, 1990, pp. 182–190.

[21] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2010, pp. 6–11.

[22] G. Nelissen, "Private communication," 2013.