Experimental evaluation of optimal schedulers based on partitioned proportionate fairness

Davide Compagnin, Enrico Mezzetti and Tullio Vardanega
University of Padua, Department of Mathematics
Email: {dcompagn, emezzett, tullio.vardanega}@math.unipd.it

***Abstract*—The Quasi-Partitioning Scheduling algorithm optimally solves the problem of scheduling a feasible set of independent implicit-deadline sporadic tasks on a symmetric multiprocessor. It iteratively combines bin-packing solutions to determine a feasible task-to-processor allocation, splitting task loads as needed along the way so that the excess computation on one processor is assigned to a paired processor. Though different in formulation, QPS belongs in the same family of schedulers as RUN, which achieve optimality using a relaxed (partitioned) version of proportionate fairness. Unlike RUN, QPS departs from the dual schedule equivalence, thus yielding a simpler implementation with less use of global data structures. One might therefore expect that QPS should outperform RUN in the general case. Surprisingly instead, our implementation of QPS on $\mathrm{LITMUS}^{\mathrm{RT}}$ invalidates this conjecture, showing that the QPS offline decisions may have an important influence on run-time performance. In this work, we present an extensive comparison between RUN and QPS, looking at both the offline and the online phases, to highlight their relative strengths and weaknesses.**

I. Introduction

Optimality is a very attractive property of a real-time scheduling algorithm, which allows achieving full utilization of the computational capacity of the system. Most of the results concerning the optimal multiprocessor scheduling problem proceed from the well known concept of *proportionate fairness*, first introduced in [1]. Algorithms based on proportionate fairness try to approximate the fluid model, apportioning scheduling time in arbitrarily small intervals called quanta and enforcing tasks to execute at steady rate at every such interval. Unfortunately, these approaches are not practical, as the frequent preemptions and migrations that they incur cause an inordinately high penalty at run time. A generalization of this model [2] arises from the observation that, whenever tasks share the same deadlines at system level, a feasible schedule can easily be found. Scheduling intervals are thus generated every two consecutive task deadlines, determining a coarse-grained approximation of the theoretical fluid model, which slightly reduces the scheduling events and thus the ensuing overhead. Although better than its predecessor, this approach still incurs unacceptable overheads when the intervals are small.

Indeed, seeking optimality by extensive use of preemptions and migrations pays a too high price at run time on real platforms, for disruption of working sets, use of global resources and of heavy-weight kernel services. When that overhead is considered in the equation, non-optimal solutions may be competitive, in spite of their lower utilization bounds. In that spectrum of solutions, the class of *partitioned* approaches relies on a static mapping between tasks and processors, so that each task can only be scheduled on the processor it has been assigned to. These solutions incur lower overhead, as they do away with task migrations and global resources, but require finding an off-line task allocation, which in the general case equates to a NP-hard bin-packing problem. The class of *global* approaches, instead, does not assume any fixed task-to-processor allocation so that tasks may vary processor assignment across preemptions and suspensions. These solutions may achieve higher utilization than their partitioned counterpart, but at the cost of the extra overhead that we mentioned earlier. Arguably therefore, neither class of solutions offers sufficient pros to compensate for their cons [3]. By selectively allowing migrations on a small subset of tasks, the class of *semi-partitioned* algorithms, first introduced in [4] [5], may attenuate the overhead induced by global scheduling as well the hardness of the partitioning phase. As shown in [6], this class of hybrid algorithms could be a plausible middle ground in the general case.

Coarsely attributed to the class of semi-partitioned approaches, the Reduction-to-UNiprocessor (RUN) [7] and the Quasi-Partitioned Scheduling algorithms (QPS) [8] have traits that make them an interesting alternative: (i) they are optimal; (ii) they reduce the preemption and migration interference, thus preserving the space and time execution locality of tasks; and (iii) they have a softer approach to off-line bin-packing. Both RUN and QPS adopt a relaxed notion of proportionate fairness, which their authors term *partitioned proportionate fairness*. Their underlying intuition rests on the concept of server, a scheduling abstraction that captures the scheduling requirements of a group of tasks. A server behaves for scheduling like a task; as such it can be scheduled, together with other servers; internally, it ensures fairness for the tasks it aggregates. The grouping operation, known as packing, in conjunction with the notion of dual scheduling [9] [10], enables RUN to reduce the scheduling problem on a symmetric multiprocessor to an equivalent problem on a (virtual) uniprocessor system. The notion of dual schedule originates from the intuition that the schedule for a set of tasks can be induced by scheduling the idle time instead. When applied to a symmetric multiprocessor system, the RUN reduction, based on duality, produces a structure called *reduction tree* used on line to optimally allocate tasks to processors. The QPS algorithm transposes the concept of reduction to a uniprocessor into the concept of quasi-partitioning that, unlike RUN, does not involve duality. In the off-line phase, QPS features a flexible partitioning scheme that allows grouping tasks even beyond processor capacity. At run time, it forms those groups into hierarchies of tasks that can execute in parallel. Those hierarchies are at one time a specialization and a generalization of the RUN reduction trees: a specialization, in that they rely on predetermined allocation of tasks to processors (which RUN does not use); a generalization, as they exhibit a more flexible arbitrary shape capable of accommodating variable workloads, including sporadic tasks (which RUN cannot).

As RUN and QPS have several traits in common and also several notable differences, the question arises as to whether one should be preferred to the other for some reason or what factors may most influence their respective performance at run time. The simulations reported in [8] show that QPS incurs slightly more preemptions and migrations than RUN under the same set of assumptions. An actual implementation of QPS might thus be presumed to suffer higher system overhead than RUN. On the other hand, one should also expect that the higher preemption and migration overhead in QPS should be compensated by its intrinsically more efficient use of partitioned-based scheduling. This intuition is also sustained by the RUN implementation presented in [11], which shows that RUN's performance is penalized by its use of global data, the reduction tree, which needs constant updates. This prevents RUN from making local scheduling decisions in parallel, with evident loss of performance. This work studies the practical viability of QPS and compares it with RUN for run-time overhead and utilization cap, in relation to the results presented in [11].

The remainder of this paper is organized as follows: Section II survey the state of the art of interest to this work; Section III provides a theoretical background on RUN and QPS; Section IV focuses on our QPS implementation comparing relevant aspects with that of RUN; Section V presents an empirical evaluation of two algorithms; Section VI draws some conclusions and outlines future work.

II. Related Work

Research on multiprocessor scheduling concentrates on the classes of partitioned and global approaches (e.g., P-EDF and G-EDF). The inherent limitations of these approaches have been widely acknowledged [3], [6], [11], and have been recalled in Section I. Recent work [12] shows that the need of global approaches for keeping a consistent system-wide scheduling state is not intrinsically an impediment to scalability. The cited work presents a solution that significantly outperforms other implementations of G-EDF [13] for a large number of processors. Yet, the choice of devoting an entire processor to the execution of scheduling operations, makes this approach ill-fit for systems with a small number of processors. Semi-partitioned EDF-based approaches have been proposed in [4], [14], [15] to address the shortcomings of partitioned and global algorithms. By splitting tasks, those approaches do indeed perform better than partitioning on average, but they are not always able to guarantee a higher utilization bound [3] [16]. Moreover, as observed in [6], their hybrid nature may require complex implementations. Hence, one must also consider the run-time overhead they incur and the implementation complexity they present.

RUN [7] and QPS [8] are more recent solutions which, for the lack of a better taxonomy, are generally included in the class of semi-partitioned algoritms. An extensive evaluation of RUN has been reported in [7], [17], [11]. The first two works show by simulation that RUN significantly outperforms other optimal algorithms in terms of incurred preemptions and migrations. The latter, instead, empirically evaluates the implementations of RUN, P-EDF and G-EDF [13] on top of LITMUS$^{\mathrm{RT}}$. This empirical evaluation shows that RUN exhibits very modest kernel overhead, in some cases comparable to that of P-EDF, and disproves false beliefs on its run-time cost and practical viability. A simulation-based evaluation of QPS has been reported in [8]. Focusing on optimality, this work compares QPS against other optimal algorithms, including RUN, and shows that their performance are comparable when the system is not fully utilized. At full system utilization, in fact, RUN behaves significantly better than QPS with fewer preemptions and migrations, but it is also evident how this scenario is unrealistic in practice.

III. Background

In this section, we recap RUN and QPS from a theoretical standpoint, reviewing the intuition behind the off-line phase and its run-time application.

We consider a system $\mathcal{T}$ composed of m identical processors and n independent real-time tasks, each generating an infinite sequence of jobs. Both RUN and QPS rely on the fixed-rate task model [7], which easily allows to define tasks and groups thereof. Under this model, a task τ_i is a pair (μ_i, D_i) where $\mu_i \leq 1$ is its constant execution rate, that is the fraction of a processor it requires, and D_i an infinite set of deadlines. Accordingly, a job J released by $\tau_i(\mu_i, D_i)$ at time $J.r$ with deadline $J.d$ will execute for $J.c = \mu_i(J.d - J.r)$ time units. In addition, it holds that $J.r \in \{0 \cup D_i\}$ and $J.d = min_d\{d \in D_i, d > J.r\}$. This representation also applies to group of tasks, whose aggregate rate and deadlines are computed respectively as the sum and the union of the individual counterparts. The concept of fixed-rate is fundamental to the formulation of both RUN and QPS as it eases combining execution requirements of multiple tasks when they are grouped together into a server. A *server* S is a scheduling abstraction, serving as a proxy for a collection of "client" tasks from which it inherits the set of deadlines and rates. Equivalently to a fixed-rate task, a server S is a pair (μ_S, D_S) where $\mu_S = \sum_{\tau_i \in \mathcal{T}} \mu_i$, $\mu_S \leq 1$ and $D_S = \bigcup_{\tau_i \in \mathcal{T}} D_i$. In the following, we consider only servers that schedules its clients according to the EDF policy.

A. Review of RUN

The scheduling principle of RUN consists in an off-line transformation of the multiprocessor system to an equivalent uniprocessor system through which a schedule for the original system can be computed on line. The transformation process builds on the notion of *dual scheduling*, first introduced in [18], according to which a schedule for a given task set $\mathcal{T} \ni \{\tau_i\}_{i \in \{1,n\}}$ can be induced by the schedule of its *dual* task set $\mathcal{T}^*$, comprising tasks τ_i^* with exactly the same period and deadline of τ_i, but complementary utilization $\mu_i^* = 1 - \mu_i$. The central idea of dual scheduling is to schedule the task idle time, in place of the task itself, as long as the scheduling problem can be reduced to an equivalent problem, feasible on a smaller number of processors. As the total utilization U^* of the dual task set $\mathcal{T}^*$ is equal to $n - U$ [7], it can be feasibly scheduled on a reduced number of $\lceil U^* \rceil$ processors, whenever $U^* = n - U < m$. Hence, enforcing $n - U < m$ by packing multiple light tasks into heavy servers, guarantees n to be small. By alternately packing tasks into servers and reformulating the scheduling problem into its dual, the number of processors is progressively reduced to one [7]. Each reduction step corresponds to a reduction level in the path starting from an original tasks ending up in the final *unit* server. This process produces the so-called *reduction tree*

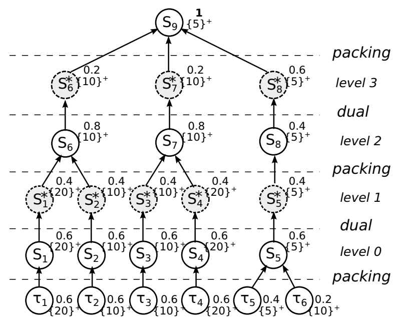


Fig. 1: Example of reduction tree.

whose nodes are servers (packed and dual) determined by reduction. Following the nodes in the reduction tree allows correct scheduling decisions to be derived at run time, as described below.

Example III.1. Consider a set of six periodic tasks $\{\tau_1, \ldots, \tau_6\}$, where τ_1, τ_4 are in the form $(0.6, \{20\}^+)$; τ_2, τ_3 are in the form $(0.6, \{10\}^+)$; τ_5, τ_6 are defined as $(0.4, \{5\}^+)$ and $(0.2, \{10\}^+)$ respectively.

Figure 1 illustrates the reduction carried out by RUN for Example III.1. The notation $\{D\}^+$ indicates that a set of deadlines D is infinitely replicated over $\mathbb{N}$. Firstly, the tasks at the bottom of the tree are packed together into servers as long as the cumulative utilization of each server does not exceed 1. For instance, τ_5 and τ_6 are packed into a level-0 server S_5 of rate 0.6 and deadlines $\{5\}^+$ while the other tasks cannot be involved in any packing at this level. The resulting servers become in turn scheduling entities that inherit their characteristics from the aggregated tasks and generate *virtual jobs*. At this point, each server is associated to its dual S_i^* characterized by a complementary utilization but with the same deadline set: for example, S_5 is associated to S_5^* with rate $1 - \mu_{S_5} = 0.4$ and deadlines $\{5\}^+$. Hence, the problem is switched into the dual representation. Since no server with unitary utilization are produced yet, this sequence of packing and dual must be repeated as long as servers S_6^*, S_7^*, S_8^* pack to 1 and fit for single-processor optimal scheduling. The unit server S_9 represents the root of the reduction tree, whose leaves regroup the original task set. Notably, when applied to a fully-utilized system, the reduction process always converges to a set of unit servers.

The on-line phase consists of scheduling servers at each level of the tree as virtual uniprocessor systems in a top-down fashion. Doing this, the information in the tree is dynamically updated at each release and completion events of virtual and real jobs. On a tree update, tasks are selected by applying two simple rules at each level l of the tree, starting from the root: (i) each primal node at level l, selected for execution at time t, selects for execution its dual child node at level $l-1$ with the earliest deadline; (ii) each node at level $l-1$, which is not executing in the dual schedule, is selected for execution in the primal level $l-2$ and vice-versa. Applying these rules downwards the tree ends with the selection of the tasks that must execute at time t.

B. Review of QPS

Similarly to RUN, QPS operates in two phases. Yet, instead of resorting to duality in the off-line reduction, it uses an original approach, named quasi-partitioning. Quasi-partitioning builds on the notion of quasi-partition: a task-to-processor assignment performed according to some bin-packing heuristic, in which processor capacity can be exceeded when a proper partition is not found. The quasi-partitioning approach profoundly differs from the RUN reduction. RUN only constructs unitary bins in a quantity that may possibly exceed the total number of available processors, but it is reduced by duality. QPS instead fixes the maximum number of allowable bins to the number of processors and allows some bin to exceed the unitary load. In QPS, a processor whose capacity is exceeded is said to be *overpacked* and the group of tasks assigned to it forms a *major* execution set. Conversely, a group of assigned tasks that does not overspill forms a *minor* execution set. A quasi-partition requires that: (i) the accumulated rate of tasks in a major execution set must be lower than 2 to guarantee that each major execution set can be feasibly scheduled on a processor pair; (ii) the exceeding fraction be lower than the rate of every other task in the same partition (i.e., the pack assigned to that processor). Accordingly, a system whose processors are all overpacked is infeasible, whereas a system without overpacked processors is fully partitioned. With QPS, therefore, a system is partitioned into *minor* and *major* execution sets depending on whether they require up to one or more than one processor to execute. As scheduling a minor execution set is straightforward, the on-line part of the QPS algorithm concentrates on how to schedule a major execution set over a processor pair.

Once a quasi-partition of the system is determined, the off-line phase completes by splitting each major execution set into two parts and assigning them by iterating over the available processors. The unitary part of a major execution set is entirely assigned to a *dedicated* processor, whereas the exceeding part is placed in a task pool together with other unallocated tasks, for a deferred allocation. A new quasi-partition of these tasks is then computed and the procedure is repeated until no major execution sets need to be created. Every time a major execution set is split, the exceeding part is associated to a *master* server and then allocated to a *shared* processor. The relationship between master servers and shared processors induced by task splitting determines a *processor hierarchy* that, similarly to the RUN reduction tree, guides scheduling decisions at run time. This hierarchy may involve several processors depending on the allocation of each major execution set generated in the successive iterations of the off-line phase. In fact, when the exceeding part of a major execution set is pooled with execution sets originated by later iterations, the processor assigned to it may be shared and dedicated at the same time. Notably, however, not all processors in the system are necessarily connected to the same processor hierarchy. Instead, multiple processor hierarchies can be formed, each of which involving only a small group of processors. In summary, the assignment is computed iterating over three main steps: (i) computing a quasi-partition of the unallocated tasks/servers; (ii) assigning each major execution set entirely to a dedicated processor; and (iii) associating each exceeding part of the major execution set to an external master server to reserve capacity on a shared processor. In turn, the external master servers are grouped with the other yet unassigned tasks, and a new iteration is performed until no further major execution sets are originated. Finally, each minor execution set is assigned to the remaining

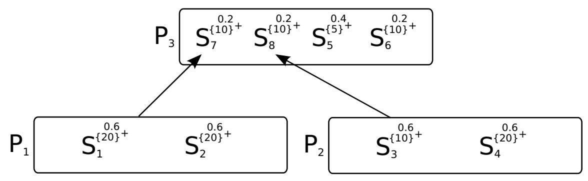


Fig. 2: Example of processor hierarchy.

processors. To illustrate how this procedure works, we apply it to the system in Example III.1, feasible on three processors. Consider the quasi-partition $\{\{S_1, S_2\}\{S_3, S_4\}\{S_5, S_6\}\}$ of the system at the first iteration, where $\{S_1, S_2\}$ and $\{S_3, S_4\}$ are major execution set of rate 1.2, whereas $\{S_5, S_6\}$ is a minor execution set of rate 0.6. Note that, according to the fixed rate task model, we substituted τ_i with S_i. At that point, the first two processors P_1 and P_2 are assigned to the major execution sets $\{S_1, S_2\}$ and $\{S_3, S_4\}$ respectively. P_1 and P_2 thus become their dedicated processors. Master servers $S_7(0.2, D_{S_1,S_2})$ and $S_8(0.2, D_{S_3,S_4})$ are subsequently defined to capture the excess of their respective major execution set, and put in a pool together with S_5 and S_6, which are still unassigned. In the subsequent iteration, the algorithm computes a new quasi-partition $\{S_5, S_6, S_7, S_8\}$ that forms a minor execution set whose cumulative utilization is exactly 1. As there is no major execution set left, the procedure ends, allocating that minor execution set to processor P_3, which becomes the shared processor of $\{S_1, S_2\}$ and $\{S_3, S_4\}$. The processor hierarchy depicted in Figure 2 emerges from the relationship among P_1, P_2 and P_3 induced by the allocation of major execution sets. The scheduling decisions at run time are made after the information contained in this hierarchy.

The idea behind QPS is that a major execution set, defined in terms of parallel execution requirements (the excess), can be feasibly scheduled by opportunely coordinating the execution of its tasks on a processor pair. This coordination at run time must prevent the erroneous situation of simultaneous execution of a job on more than one processor, which would occur if tasks were selected greedily. To this end, QPS associates four servers to each major execution sets, defined as follows. Let Γ be a major execution set with execution rate $\mu_\Gamma > 1$ and excess $x = \mu_\Gamma - 1$, obtained by quasi-partitioning, then consider an arbitrary bi-partition $\Gamma^A \cup \Gamma^B = \Gamma$. Four servers S_M, S_S, S_A and S_B are thus defined such that $S_A(\mu_{\Gamma^A} - x, D_\Gamma)$, $S_B(\mu_{\Gamma^B} - x, D_\Gamma)$, while $S_M(\mu_x, D_\Gamma) = S_S(\mu_x, D_\Gamma)$. It follows that $\mu_{S_S} + \mu_{S_A} + \mu_{S_B} = 1$, then S_S, S_A and S_B can be executed on a single processor. This holds for any bi-partition of Γ such that neither Γ^A nor Γ^B are empty, as long as every task in Γ has a rate higher than the exceeding fraction x. S_A and S_B are called *dedicated* servers and capture the non-parallel execution of the tasks in Γ^A and Γ^B respectively, while S_M and S_S are respectively the *master* and *slave* servers that capture the parallel execution of any tasks in Γ. This means that, at run time, every time the master server S_M is selected for execution on the shared processor, the slave server S_S will simultaneously execute on the dedicated processor. Notably, although servers are associated to major execution sets in the off-line phase, their activation at run time is managed in accord with the fluctuations in system load caused by sporadic releases. In fact, a late sporadic arrival at run time could reduce the demand of the major execution set, for it to become feasible on its dedicated processor, thus needing no activation of the master. The activation and deactivation of servers at run time is also known as *mode change*. For further details, we refer the reader to the original formulation in [8], since this mechanism is out of the scope of this work.

In the on-line phase, QPS uses P-EDF to schedule the ready task with the earliest deadline for each processor, as long as a master server is selected. When that happens, a client is picked among the tasks and servers on each dedicated processor, and the corresponding slave server – if any – is activated in that processor. If another master server is selected, this operation is repeated until an actual task along the processor hierarchy is reached and scheduled on that processor. In summary, QPS: (i) visits each processor in reverse order to that of its allocation and EDF-schedules tasks/servers, with the sole exception of when a master server is selected by a processor already visited, in which case, QPS selects the corresponding slave server; and (ii) selects the client with the earliest deadline, for all selected servers in (i) until this leads to the selection of an actual task. The top-down visiting order of the processor hierarchy effectively determines how tasks may migrate among processors: tasks can only migrate from their dedicated processor to any processors located upwards the hierarchy. Hence, the higher the processor hierarchy the higher the number of possible migrations. Note that, although in the worst case the processor hierarchy levels can be as many as the system processors, our experiments show that on average the quasi-partitioning tends to originate multiple hierarchies with few levels, and possibly spanning many processors. This benefits system isolation and explains why QPS causes few preemptions and migrations.

C. Comparative considerations

Whereas RUN is often classified as a semi-partitioned algorithm, it really is not in the common acceptation of that term. Arguably (and together with QPS as we see below), it belongs to a distinct category of hybrid approaches that may selectively behave similarly to global *and* partitioned algorithms. The packing step makes RUN similar to a partitioned algorithm, except that tasks are not associated to processors but to servers, whose execution is not pinned to any particular processor, much like in global scheduling. Scheduling decisions are thus taken globally, guided by the reduction tree with inevitable impact on the final performance at run time. Thanks to dual scheduling, RUN can aggregate servers under the condition that the sum of the rate of the duals is ≤ 1, which is weaker than the quasi-partitioning constraint. According to [19], the QPS reduction specializes that of RUN, as long as its *virtual cluster* size is fixed to 1. A virtual cluster is a grouping of tasks/servers induced by reduction, which determines how the system processors are shared by tasks at run time. The concept of virtual clustering however, should not be confused with that of clustered algorithms such as [20] and [21], where clusters are defined a priori after some opportunistic criteria. As multiple virtual clusters may share the same processor at run-time, executing tasks may suffer both an intra-cluster interference induced by tasks of the same cluster, and an extra-cluster interference induced by virtual clusters sharing the same processor. Here instead, virtual clusters are a product of the off-line phase that aggregates tasks into servers so that the schedule can preserve optimality while limiting the number of migrations.

For particular task sets, RUN and QPS induce the same virtual clustering. This happens for the task set in Example III.1, where both algorithms yield the virtual clusters $\{\tau_1, \tau_2\}$, $\{\tau_3, \tau_4\}$ and $\{\tau_5, \tau_6\}$, as long as $\mu_1 + \mu_2 = 1 + 1/5$, $\mu_3 + \mu_4 = 1 + 1/5$ and $\mu_5 + \mu_6 = 3/5$. Figure 1 shows that RUN induces these clusters by packing at the dual level: for example, $\{\tau_1, \tau_2\}$ are grouped on server S_6 by duals S_1^* and S_2^*. Note how dual servers S_6^* and S_7^* are, by construction, equivalent to those of QPS S_7 and S_8. A feasible schedule for the system can be thus obtained assigning the first two virtual clusters to two dedicated processors and the third cluster to the third processor, together with the excess of the first two. In this particular example, each cluster size is in the form $k + x$ where k are fully-utilized processors and x the parallel execution required by the virtual cluster. QPS operates such that $k = 1$ so that the corresponding virtual cluster can be pinned to a processor. RUN instead may produce virtual clusters with any k size and therefore cannot pin them to any particular processor. RUN is designed to compute the schedule in virtual servers and map decisions to processors via the reduction tree at run time. Conversely, QPS pins servers to processors and computes the schedule per processor by traversing the processor hierarchy. This latter observation suggests that the off-line phase is key to the run-time performance of QPS, as we see next.

IV. Implementation

This section presents our implementation of QPS. As with RUN, we implemented QPS on top of LITMUS$^{\mathrm{RT}}$ (2013.1): an extension to the Linux kernel maintained at MPI-SWS for the experimental development and evaluation of scheduling algorithms and locking protocols in multiprocessor systems.

A. The QPS implementation

Our first intuition was that the formulation of the QPS algorithm, as recalled in Section III-B, allows dispensing with the global data structure used in RUN, thus favoring a strictly partitioned design. This is also motivated by the fact that each task is notionally assigned to a dedicated processor and migration events occur rarely and guided by the processor hierarchy. As the processor hierarchy involves a small number of processors on average, also the contention in coordinating multiple processor activities should be low. For this reason, our QPS implementation extends that of P-EDF rather than RUN's: our goal was set on simplifying the kernel primitives and reducing run-time interference. We now briefly present our implementation on the LITMUS$^{\mathrm{RT}}$ interface, and discuss the solutions to the main issues we encountered.

1) Processor hierarchy: The processor hierarchy is the only data structure around which the scheduler operates at run time. Although at first sight the processor hierarchy could be assimilated to the reduction tree of RUN, its function is converse as also is the design of the algorithm: instead of pushing tasks to processors according to the (global) reduction tree, we let each processor take scheduling decision locally, getting information from other processors as needed.

We encapsulated all the processor scheduling state into a single *processor domain* entity. Each processor domain is used for scheduling an execution set and consists of three different entities: two *local* servers used for scheduling tasks allocated to the dedicated processor and a *remote* server used for scheduling tasks allocated to the shared processor. Tasks in a major execution set are opportunely scheduled by the two local servers according to the result of the bi-partitioning. Our intuition here was that the four notional servers associated to each bi-partition (see Section III-B) are somewhat redundant at run time. Moreover, the fact that the master execution on the shared processor induces that of the slave on the dedicated processor is controlled by sending an inter-processor interrupt (IPI) from the shared processor to the dedicated one. Each local server acts as an EDF server implemented on the `rt_domain` entity of LITMUS$^{\mathrm{RT}}$, an abstraction comprising task queues and timers. Each server (whether remote or local) is also assigned to a budget proportionate to that of the four notional servers defined by QPS to schedule the major execution set. Our QPS implementation works symmetrically for both major and minor execution sets. Hence, in the event that a processor domain is assigned to a minor execution set, we ignore the remote (master) server after simply allocating all of its tasks to one of the two local servers of that processor.

Whereas the local and remote servers act on behalf of the same execution set, the remote server is scheduled on the shared processor as it were a task. More precisely, if the local servers are located in processor P, then the remote server is located on the P's parent as defined by the processor hierarchy computed off line. Each processor domain keeps a link to its remote server, and similarly the remote server keeps a link to its reference processor domain. The way these entities are linked together reflects the fact that the scheduler has to propagate status updates through the processor hierarchy at both the release and the completion events of a task or server. For instance, each release event on a dedicated processor is corresponded by a release of the remote server on the remote processor, which, in turn, could trigger the master server of the higher-level processors along the hierarchy. During this operation, information such as the earliest deadline and the servers budget must be updated from the processor to its parent, in a bottom-up fashion. Similarly, every time a master server is selected for execution, one of its client, located on the reference processor, will execute in place of it. Depending on the hierarchy, the latter may be in turn the master of another processor that will select one of its client until reaching an actual task. In this case, the recursive server selection will lead to a chain of master servers whose execution will be nested on the same processor. The scheduling operation is best understood after a complete description of the data structure.

As reported in [11], one of the main hurdles in the RUN implementation was the inability of collapsing simultaneous events resulting from the scheduling on the reduction tree. For this reason, only one interval timer is associated to a processor domain to keep track of when the server execution budget exhausts. This allows collapsing multiple budget exhaustion events occurring on a processor, attributable to local and remote servers. Hence, every time a processor domain selects a server for execution, it also arms the timer according to the server's spare budget, so that a timer expiration event corresponds to a server budget exhaustion and causes processor rescheduling. From the scheduling standpoint, a budget exhaustion corresponds to a completion of a server job.

The way tasks are packed together to form the processor hierarchy is a central element of the algorithm. Much like in RUN, in fact, different bin-packing heuristics may lead to different processor hierarchies also in QPS, which may in turn result in different preemption and migration patterns at run time. Evaluating the impact of the packing policies with respect to the kernel interference is one of the objectives of this work.

In QPS, the whole hierarchy is statically pinned to the processors as each processor domain is assigned to a processor. Hence, the only information that travels across processors are the tasks. Conversely, RUN servers migrate among processors, as does the reduction tree, while tasks stay pinned to their associated server. In other words, the mechanism that triggers migrations in QPS works exactly the opposite to that of RUN: QPS uses pull-based migration while RUN the push-based one. [6] sees this as an important difference that may lead to different kernel interference.

2) Scheduling on the processor hierarchy: The dispatcher is responsible for taking scheduling decisions on the basis of each processor status. We regarded the solution presented in the original paper as slightly inefficient. We therefore devised an improved version that not only takes into account the order each processor is visited but it also runs in parallel. The intuition behind that was that most of the scheduling decisions take place locally where each processor selects one of its concrete tasks. Only the selection of a master server affects the activity on other processors: starting/stopping the master execution on the shared processor must trigger an activation/deactivation event of the slave on the dedicated processor accordingly. Following this line of thought, the execution of multiple master servers could be virtually nested onto the same processor as long as a master could select another master among its children in some hierarchy where the dedicated processor of the former server coincides with the shared processor of the latter, and so on until a task is selected. In such a case, we make the executing processor responsible for triggering each dedicated processor whose master is executing. Algorithm 1 outlines the revised parallel dispatcher. The aim of Algorithm 1 is twofold. When executed on a processor P, it (i) computes the next task to be executed on P, and (ii) notifies those processors along the hierarchy whose master has been selected in the computation of (i). Computing (i) requires first selecting the correct server among those active in the processor depending on whether a minor or a major execution set is allocated (lines 2-6). Once the server is selected, it could be necessary to follow the hierarchy down to reach the correct task (lines 8-15), giving rise to a chain of servers. Note that, the selection of the slave server (line 12) can only occur after the selection of the corresponding master server. In this case, the dispatcher must consider the bi-partition of the current execution set Γ picking the most urgent task from either Γ^A or Γ^B not yet selected by the master (line 13). It is this requirement that allows QPS to avoid choosing the same task for execution on multiple processors. Interestingly, in the particular case of periodic task sets, the bi-partition can be computed off line for each execution set so that the dispatcher need not compute it. At this point, the computation ends and the task can be executed (line 18). This part of the procedure behaves like the original one, restricting the computation to one processor. Finally, for each dedicated processor whose master is selected during (i), and that was not already selected by the previous execution of this procedure on P (line 14), we send a notification event (line 16) that request the invocation of this procedure on the target processor to release the slave server. Most of the scheduling operations

Algorithm 1 QPS Dispatcher

```
1:  Let $t$ be the current time, $P$ the current processor and $\Pi$
    the set of processors whose master was selected by the
    last execution of the dispatcher on $P$. Also, let $\Gamma^A \cup \Gamma^B$
    be a bi-partition of $\Gamma$
2:  if there is an active server at $t$ on the current processor then
3:      if there is a slave server on the current processor whose
        master was previously selected then
4:          Select the slave server $\sigma$
5:      else
6:          Select a non-slave server $\sigma$
7:      $\Pi' \leftarrow \emptyset$
8:      while $\sigma$ is not a task do
9:          $\Gamma \leftarrow$ the clients of $\sigma$
10:         if $\sigma$ is not a slave server then
11:             Select $\sigma'$ in $\Gamma$ via EDF
12:         else
13:             Select $\sigma'$ in $\Gamma$ via EDF from whichever of $\Gamma^A$
                or $\Gamma^B$ is *not* running on $\Gamma$'s master server
14:         $\Pi' \leftarrow \Pi' \cup \{\pi |$ is the dedicated processor of $\sigma'\}$
15:         $\sigma \leftarrow \sigma'$
16:     Notify all the processors in $(\Pi \cup \Pi') \setminus (\Pi \cap \Pi')$
17:     $\Pi \leftarrow \Pi'$
18:     Execute $\sigma$ on the current processor
```

in QPS are thus executed by this single catch-all scheduling primitive (denoted SCHED in the evaluation section), which gets the scheduler status and selects the next task to execute. This kernel primitive is invoked very frequently indeed, after each job release, job completion, local exhaustion events, notification of the master execution.

On-line correctness is preserved by keeping each server always and consistently up to date with budget and deadline information. Unlike RUN, whose reduction tree can be updated by any processor, the update operation in QPS strictly requires the information to be propagated from the dedicated processor to the shared one along the hierarchy, when major execution sets are involved. Whenever a job release occurs on a dedicated processor, its servers are updated with budget and deadline information. A release on a processor assigned to a major execution set may also cause the release of its master server on the shared processor which, in turn, must be updated. This may lead to a bottom-up chained update of the processor until a minor execution set is reached. To reduce the cost of a chained propagation of release events along the processors hierarchy, we collapse simultaneous release events, in a way similar to what we did in RUN. Nevertheless, a processors domain is updated as long as the release event causes an adjustment of its scheduling interval (i.e., an earlier deadline). This is the most critical and costly operation executed by QPS at run time: a missing or wrong-order update may partially or totally compromise the schedule.

Performing scheduling operations on a multiprocessor involves using either local or global locks. Global scheduling operations are inherently more interfering than the local counterpart since they potentially induce more contention on acquiring the lock that protects the global information: no global scheduling operation can occur simultaneously, causing the complete serialization of the scheduler. Neither global nor local locking really fit QPS, as its scheduling events potentially involve a limited set of processors, depending on the hierarchy determined off line. In our implementation we chose a hybrid locking mechanism that benefits of partitioning and is capable of behaving as either global or local, depending on the size of the processor cluster induced by the hierarchy. Each processor hierarchy is associated to a lock that guarantees the coherency of the intra-cluster task execution while preventing interference on extra-cluster task execution.

V. Evaluation

According to [22], an exhaustive evaluation of a scheduling algorithm should also consider its actual implementation to unveil unexpected implementation challenges and actual kernel overheads. For this reason, we performed an empirical comparison of QPS and RUN, to assess their run-time overhead from the standpoint of the interference they cause to the system and to determine under what conditions one performs better than the other. We evaluate the kernel overhead in terms of the number of preemptions and migrations, the cost of each scheduling primitive, and the cumulative overhead generated during execution. We intentionally restrict the analysis to these two algorithms since they share several unique traits that, as discussed in Sections I and II, set them apart in the landscape of optimal scheduling algorithms. Further considerations are also derived by our previous work [11].

A. Experimental setting

The experiments consisted in collecting execution traces obtained by feeding identical task sets to the two algorithms and then comparing the respective execution statistics. The task set features and the overall system workload inevitably result in different impact on the performance of each algorithm. The difficulty in comparatively evaluating RUN and QPS stems from the difference in their off-line phase, as explained in Section III-C, which induces different patterns of preemptions and migrations at run time. Notably, applying the same bin-packing heuristic does not guarantee RUN and QPS to determine the same task set reduction and then the same behaviour at run time. We therefore generated two classes of experiments, one to determine their respective utilization cap, the other to isolate the effects of the off-line phase on execution at run-time.

The former, larger, class of experiments, used randomly generated task sets with overall system utilization increasing between 50% and 100%. We also increased the granularity of the observations at higher system utilization, where we expect most significant variations to occur. We generated task sets to stress both the off-line and the on-line phases: varied task utilization was generated following a uniform distribution (i) over the interval [0.5, 0.9], to incur worst-case packing behavior; and (ii) over the interval [0.1; 0.5], to increase the length of the scheduling queues. In the following, we denote "heavy" the experiments generated according to (i), and "medium" the experiments generated according to (ii).

Besides task utilization, also the distribution of task periods may affect the performance of scheduling algorithms. The harmonic distribution of release events may induce higher scheduling contention and therefore higher overall system overhead. Non-harmonic releases may instead cause considerable fragmentation of scheduling intervals. Accordingly, we performed two sets of experiments, to address harmonic and non-harmonic task sets separately. To keep the experiments manageable in size and complexity, tasks periods were drawn from a uniform distribution in [20ms; 200ms] over an experiment duration of 10s.

To discern the sensitivity of QPS and RUN to different packing strategies, we applied three bin-packing heuristics to the respective off-line reduction phases: First-Fit Decreasing (FFD), Worst-Fit Decreasing (WFD), and Evenly-Fit Decreasing (EFD). The latter aims at filling bins evenly. In practice, it associates the first m tasks, sorted by decreasing size, to the first m bins, and then allocates each remaining task iterating over those bins. Due to space constraints, we only report results from experiments that apply the FFD heuristic to both RUN and QPS. We note however that no significant differences emerged on varying the bin-packing strategy.

The other class of experiments, instead, focused on isolating the effects of the off-line phase on execution at run time, by selecting only those tasks sets whose reduction, in both RUN and QPS, led to similar virtual clustering, hence to an equivalent schedule. In this way, the two algorithms (i) incur the same number of scheduling events and (ii) compute their schedule starting from the same underlying conditions (e.g., equivalent servers and tasks per servers). Hence, the differences that arise in run-time performance can be discerned to descend from the cost of individual scheduling events. Each experiment has been generated in a controlled way for a limited number of specific fully utilized systems. This limited the amount of tests that we could perform in comparison to what could be done using randomly generated sets.

The experiments were run on an AMD Opteron 6370P processor including up to 16 cores running at 2.0 GHz with private L1 and L2 caches of 64K and 2MB respectively and a shared L3 cache of 16MB. Frequency scaling and power management functions were intentionally disabled.

B. Accounting for kernel overheads

Determining the kernel overhead represents not only the objective of our empirical evaluation but also the prerequisite: under some circumstances (e.g., zero-laxity condition and higher system utilizations), disregarding the kernel interference may compromise schedulability. To address this issue we adopted an approach similar to that in [11] where a preliminary execution of the experiments was used to gauge the overhead incurred by each scheduling primitives. These measurements were then used to compute an upper bound to the global scheduling overheads and to account for them in the execution time of each task. In keeping with [6], we obtained an accurate estimation of the cost of each individual scheduling primitive by collecting traces of more than 2000 randomly generated experiments. In the first round of experiments, in the absence of preliminary knowledge on the real system overheads, we reserved a notional 5% of the system utilization for all tracing and scheduling operations and therefore excluded all the task sets with total utilization larger than 95%.

As noted in [13], a LITMUS^{RT} system may incur several types of overheads: (i) the *release* (REL) of a new job, (ii) a *scheduling* (SCHED) event and (iii) the induced *context switch* (CSW), (iv) the *interrupt latency* (LAT), including inter-processor interrupts (IPI), and (v) the *tick interference* (CLK).

Additionally, RUN requires a primitive (TUP) to perform the run-time *update* of the reduction tree in response to a job release or a budget exhaustion event.

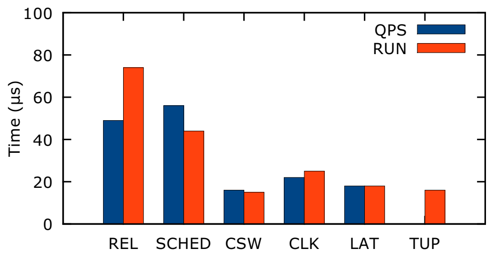


Fig. 3: Maximum observed system overheads.

Figure 3 shows the maximum cost obtained for each individual overhead factor under RUN and QPS from the preliminary round of experiments. In keeping with our expectations, these results show that RUN outperforms QPS with respect to the cost of the scheduling primitive, but not for the cost of release. Most of the scheduling operations of QPS are, in fact, performed by the dispatcher in a *single* scheduling primitive. Unlike RUN, whose scheduling primitive operates just locally, the scheduling decisions in QPS may involve multiple processors (e.g., the dispatcher might have to traverse the processor hierarchy to pick the correct task for execution) inevitably increasing the primitive worst-case cost in a way that depends on the hierarchy size. Conversely, QPS benefits from a leaner release primitive, which behaves like a partitioned algorithm on individual processors, while it collapses simultaneous releases event on a processor hierarchy. RUN instead collapses all simultaneous releases, and updates the global reduction tree accordingly. The tree update primitive (TUP) does not affect QPS, while for the other primitives whose implementation does not pertain the LITMUS$^{\text{RT}}$ plugin, the algorithms exhibit similar behaviour. Interestingly, the worst-case behaviour reported in Figure 3 arises with harmonic tasks sets, where many simultaneous events are handled within the same kernel primitive. We appreciate that there are further indirect sources of interference as, for example, *preemption* (PRE) and *migration* (MIG), mainly determined by cache-related interference [6] [23], that may affect the execution context of a job. However, our setting[1] incurs negligible cache effects as our tasks are synthetically equivalent and feature very small working sets.

Common practice to ensuring system feasibility requires considering scheduling overhead by apportioning to the WCET of individual tasks [24], [25]. We used the following formula to determine an upper-bound on the overhead a job may incur in RUN during its life span:

$$OH^{Job}_{RUN} = REL + \widehat{SCHED} + CLK + \\ + k \times (UPT + \widehat{SCHED} + max(PRE, MIG))$$

where $\widehat{SCHED} = SCHED + CSW + LAT$ and k is the bound on the preemptions a job can suffer at most. This formula combines the cost in handling both the job release and all the potential preemptions a job can suffer during execution. The release event includes the cost of the release when the job gets ready plus the cost of when it gets dispatched which, in turn, includes the schedule, the IPI latency and the context switch overhead. The dispatching cost is also attributed to a preemption which additionally comprises the cost in updating the reduction tree and a PRE or MIG overhead. As shown in [7], the number of preemptions suffered by a job in RUN depends on the reduction tree and, on average, it is at most $k = \lceil (3p + 1)/2 \rceil$ where p is the tree height.

Unfortunately, the RUN preemption bound does not apply to QPS whose scheduling decisions are determined by the processor hierarchy. The interference a task can cause to the systems depends on whether the tasks belongs to a minor or a major execution set. In a minor execution set, whose tasks are scheduled using uniprocessor EDF, a job can cause at most one preemption accountable to its release event. In a major execution set, instead, a job release event could activate the master server whose job, released at the same instant, could preempt the task executing on the shared processor. In this case no preemption occurs on the dedicated processor since the manager would associate the new job to the master, therefore, it would immediately be selected for execution. However, the dedicated processor can be preempted by a job completion event of the master. In fact, the client executing on the remote processor could not be completed within the master budget hence its execution must be carried forward on the dedicated processor. This is equivalent to saying that the client migrates from the shared processor to the dedicated one. Extending to the general case of a processor hierarchy of k levels, we say that a task at level k causes up to one preemption per job. A task at level $k-1$, instead, could cause up to two preemptions: one accountable to the job release event, whereas the other accountable to the job completion event of the master server at level k. For the same reason, a task at level 0 could cause up to k preemptions. Therefore, each task at level l could cause up to $k - l$ preemptions per job. Although, the number of levels in the processor hierarchy is bounded by m, the number of processor in the system, the number of levels rarely exceeds $\lceil m/2 \rceil$ in practice. This is also confirmed by our empirical experiments. Therefore we used $k = \lceil m/2 \rceil$ as a reference to derive a realistic upper bound on the number of preemptions per job. Therefore, the QPS overhead was determined as follows:

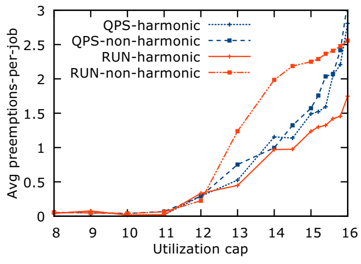
$$OH^{Job}_{QPS} = REL + \widehat{SCHED} + CLK + \\ + k \times (\widehat{SCHED} + max(PRE, MIG))$$

Although OH^{Job}_{RUN} and OH^{Job}_{QPS} likely differ in magnitude, we opted to always use the maximum among them ($max(OH^{Job}_{RUN}, OH^{Job}_{QPS})$) to warrant a fair comparison in the second round of experiments.

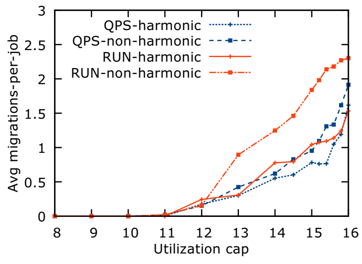
C. Results

As mentioned above, our empirical evaluation of QPS and RUN focuses not only on a set of randomly generated experiments, where the two algorithms are unlikely to incur the same number scheduling events, but also on a set of specific experiments in which instead they do, independently of the off-line phase. We want to determine whether, under those assumptions, QPS could really benefit of a simpler end efficient implementation, also at full-system utilization. Figure 4, 5 and 6 show the results obtained on the wider set of randomly generated experiments. The horizontal axis

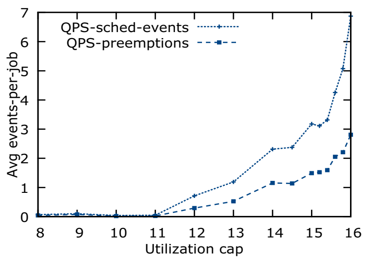
[1]Tasks are based on the dummy real-time task skeleton (`rt_spin`) of LITMUS$^{\text{RT}}$.



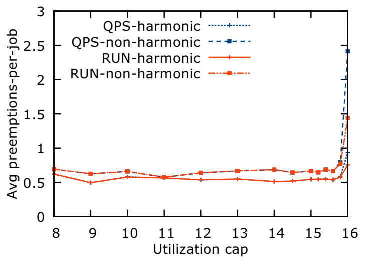
(a) Observed preemptions (heavy tasks).



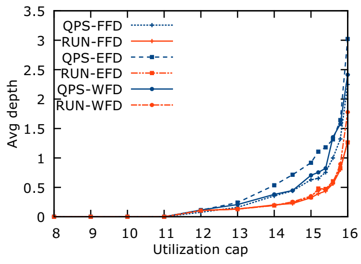
(b) Observed migrations (heavy tasks).



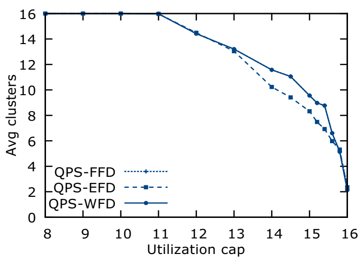
(c) Observed scheduling events (heavy tasks/harmonic periods).



(d) Observed preemptions (medium tasks).



(e) Average depth of processor hierarchy and reduction tree under different bin-packing heuristics (heavy tasks/harmonic periods).



(f) Average number of processors clusters (heavy tasks/harmonic periods).

Fig. 4: Kernel interference.

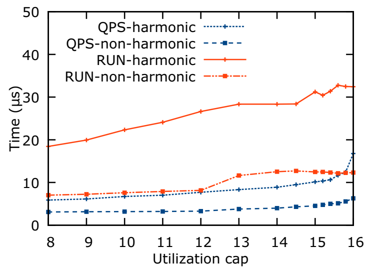
represents the overall system utilization: since our experiments were executed on a 16-core target, the extreme values on the axis stand for 50% and 100% utilization respectively. For each utilization value we collected results performing 20 different task sets. Figure 7 refers instead to the other set of experiments.

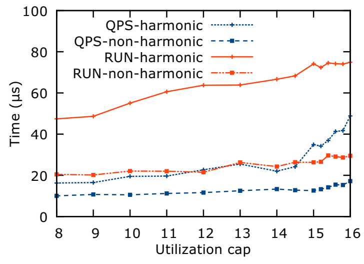
1) Preemptions and migrations: Figures 4a and 4d report the average number of preemptions per job we observed for task sets, generated following the heavy and medium distributions of the tasks utilization respectively. Not surprisingly, applying the same FFD packing heuristic on both the algorithms we observed that they incur exactly the same number of preemptions as when the task set is fully partitioned. While medium tasks ease packing even under extremely high system utilization, heavy tasks can easily cause the packing to fail at relatively low utilization, as confirmed by the average number of migrations per-job reported in Figure 4b. At higher system utilization, QPS incurs slightly more preemptions than RUN and almost the same number of migrations. In other cases, RUN with non harmonic tasks dominates the number of preemptions and migrations per job suffered. This is due to the RUN incapability of collapsing simultaneous, or very close, scheduling events triggered by the reduction tree. Consistently with the results in [8], this is explained by the benefits that QPS draws from the slack reserved for the system overhead.

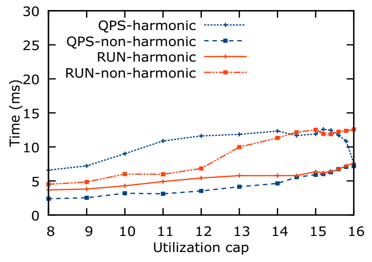
Limiting the observation to the number of preemptions/migrations, we might think that QPS could significantly outperform RUN, on account of its more efficient implementation. Yet, the mechanism used to govern parallel execution between the master and the slave processors considerably increases the number of scheduling events suffered by QPS: the term scheduling event is used here to indicate any invocation of the schedule primitive (dispatcher) that interferes with task execution, without necessarily producing an actual job preemption. Figure 4c relates the average number scheduling events per-job to actual preemptions. At increasingly higher utilization, the number of scheduling events rapidly increases and always exceeds the number of preemptions. This behaviour is directly induced by the processor hierarchy: the higher the processor hierarchy, the higher the number of incurred scheduling events.

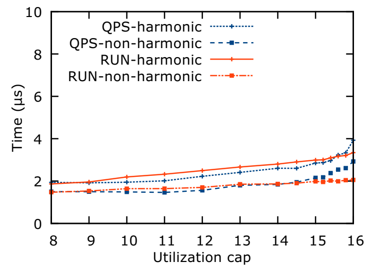
2) The packing heuristic: Figure 4e reports the average size for both the processor hierarchy and the reduction tree under different bin-packing heuristics. As expected, the lower the system utilization, the lower the average size of the processor hierarchy and of the reduction tree. For the processor hierarchy, FFD and WFD are in general better heuristics: the fact that EFD produces slightly higher hierarchies can be intuitively explained by the observation that each bin tends to be equally filled, hence, at higher system utilization, an over-packed bin with a large excess tends to cause another bin to be over-packed, and so forth, thereby increasing the hierarchy size. In our experiments however we never observed a hierarchy deeper than 4 levels. It is important to appreciate that RUN is less prone to the effect of the packing heuristic as it benefits from packing at both the primal and the dual levels. Together with the average size of the hierarchy, Figure 4f reports the average number of clusters determined by QPS for each heuristic. For fully partitioned systems, at low system utilization, the number of cluster equals the number of processors. The higher the system utilization, the higher the average size of the processor cluster determined by the hierarchy. As we show below, there is a strong correlation between the cluster size and the performance of QPS.

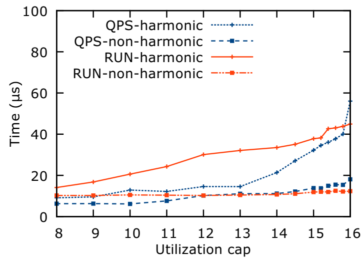
3) Scheduling primitives: Figure 5 focuses on the behaviour of the release (REL) and schedule (SCHED) primitives for harmonic and non-harmonic sets of heavy tasks at increasing overall system utilization. Figures 5a, 5b and 5c report the average, the maximum and the cumulative costs for the release


(a) Average release overhead (heavy tasks).


(b) Maximum release overhead (heavy tasks).


(c) Cumulative release overhead (heavy tasks).


(d) Average schedule overhead (heavy tasks).


(e) Maximum schedule overhead (heavy tasks).

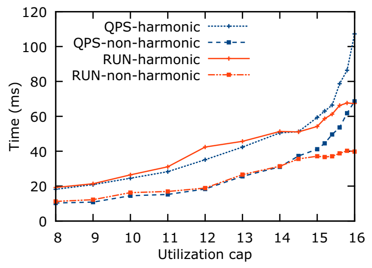
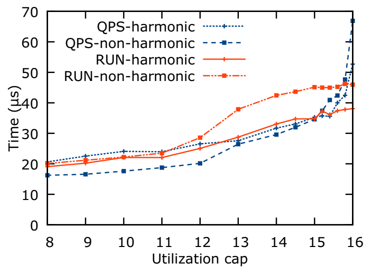

(f) Cumulative schedule overhead (heavy tasks).

Fig. 5: Release and schedule primitives.


(a) Average overhead per job (heavy tasks).

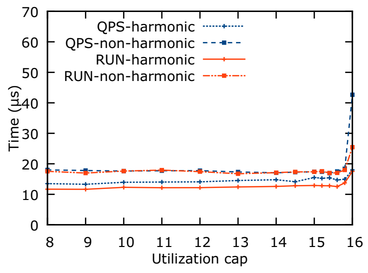
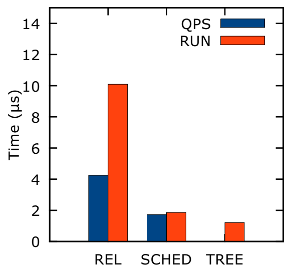

(b) Average overhead per job (medium tasks).

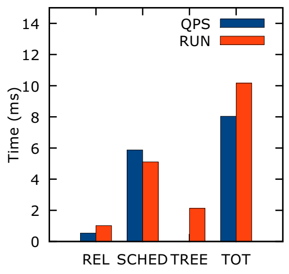
Fig. 6: Per-job scheduling overhead.

primitive, whereas Figure 5d, 5e and 5f report the average, the maximum and the cumulative costs for the schedule primitive. As already observed, both the average and the maximum cost of a release in QPS is lower than in RUN: QPS greatly benefits from partitioning that isolates the processors, so that a release becomes a local event; global releases in RUN, together with the required update of the reduction tree, are quite costly instead. Near full system utilization, where the average cluster size increases, the release in QPS involves multiple processors, thus rapidly increasing its cost. The latter behaviour is more evident on harmonic task sets where simultaneous releases frequently occur. Interestingly, the opposite behaviour can be observed when looking at cumulative costs. For harmonic task sets, RUN greatly benefits from collapsing simultaneous releases, which leads to lower cumulative cost, while QPS does not. More precisely, at low system utilizations, where the system is perfectly partitioned, the cumulative cost of the QPS release is determined by the number of job releases on each processor, as these are independently handled. With an increasing system utilization, QPS collapses most of the releases coming from larger processor clusters. This explains the apparently decreasing behaviour of the cumulative costs while approaching the full utilization. For non-harmonic task sets, QPS is less influenced by the processor cluster size since simultaneous release events are unlikely; still its cumulative overhead is lower than RUN, which also incurs the cost of updating of the reduction tree.

Hierarchy sizes also have a significant impact on the scheduling primitive of QPS. As shown in Figures 5d and 5e, at low system utilization the cost of the primitive in QPS is roughly similar to, and occasionally lower, than that of RUN. This is explained by the fact that both QPS and RUN select their task in EDF mode. Interestingly, the RUN algorithm incurs a higher overhead due to the contention on the global lock that protects the scheduling state. This additional cost emerges particularly for harmonic task sets, where we observe the maximum costs of that primitive (Figure 5e). At higher system utilization, QPS is largely penalized by the size of the processor hierarchy, potentially increasing the probability of nesting the execution of several masters on the same processor, which in turn results in a longer chain. Such a chain of masters increases the number of steps each processor has to make to select the correct task, the cost of the scheduling operation, and the count of scheduling events induced by the execution of masters on the processors, as reported in Figure 4c. Arguably, this also explains the rapid increment of the cumulative costs



(a) Average cost.



(b) Cumulative cost.

Fig. 7: Observed results isolating the packing effect.

of this primitive in QPS, near to the full-system utilization, as reported in Figure 5f. Notably, this holds for both harmonic and non-harmonic tasks sets, with a lower magnitude in the latter case.

4) Per-job overhead: Figure 6 shows the average per-job overhead we observed for both algorithms with harmonic and non-harmonic task sets, inclusive of the cost of all scheduling primitives and other system overheads. With task sets characterized by heavy tasks (Figure 6a), the overhead is quite stable as long as the system is fully partitioned (i.e., $U < 12$): this is explained by the fact that both algorithms reduce to P-EDF and the number of preemptions is only determined by local tasks. For harmonic tasks sets, QPS incurs a slightly higher overhead, mostly due to the major contribution of the release primitive, which is locally handled and thus not collapsed among processors. The same aspect favours QPS in case of non-harmonic tasks sets: at higher system utilization, the contribution of the release and scheduling primitives are combined together with the number of scheduling events induced by the off-line phase. As long as the processor hierarchy size is sufficiently low to cause a reduced number of scheduling events, QPS slightly outperforms RUN for both harmonic and non-harmonic task sets: the light overhead of the scheduling primitives in QPS outweighs the increase in scheduling events. Close to full system utilization, instead, the huge contribution of the cumulative costs of the scheduling primitive dramatically raises the QPS overhead, causing it to exceed that of RUN. Note that non-harmonic tasks sets cause more overhead than harmonic ones, as they induce more scheduling intervals and, in turn, more preemptions and migrations. Similar behaviour can be also observed for experiments with medium tasks: in this case, however, they much ease good partitioning, as it shows from the number of preemptions observed in Figure 4d. *This confirms that the overhead of the two algorithms is largely determined by the off-line phase*[2]. Although more efficient implementations may lead to a lower overhead, we believe that the differences observed in this analysis between RUN and QPS are intrinsic to their formulation.

5) Isolating the packing effect: The results reported so far show that the depth of the processor hierarchy near full system utilization causes a significant increment in the number of scheduling events, that in turn severely penalizes QPS at run time. Clearly, this behavior depends on the way the off-line phase operates, which is less efficient than the RUN one as we observed in Section III-C. In order to isolate the penalizing effect of the off-line phase, we collected results from a small number of experiments, all at full system utilization, where the RUN schedule is identical to that originated by QPS. In all of these experiments we observed results comparable to those reported in Figure 7. That figure reports the average and cumulative cost of the primitives observed for a simple task set comprising 6 tasks of utilization equivalent to that specified in Example III.1, but periods selected randomly over a uniform distribution over the interval [20ms;200ms]. That task set is feasible on 3 processors (modulo the cost accounted for the system overhead) and reduce to the same set of virtual clusters. Also note that in this class of experiments, a single lock protects the scheduling primitives and a single release queue is used for both RUN and QPS.

Although QPS outperforms RUN with respect to the release primitive (Figure 7a), as it simpler in implementation and does not update the reduction tree (the processor hierarchy), its cumulative cost has an almost negligible impact on the overall system overhead (Figure 7b), which is instead dominated by the schedule primitive. Most of the times, the schedule primitive in QPS operates as in uniprocessor EDF – similarly to RUN on level-0 servers. When a master server is selected, instead, QPS iterates along the hierarchy until a task is found. Hence, the differences observed in the cost of this primitive depend on the depth of the processor hierarchy, albeit small in these particular experiments. Conversely, the cumulative cost of the schedule primitive depends on the frequency of its invocation during execution of the whole experiment. This, in turn, depends on the notification mechanism used to trigger the simultaneous execution of the slave and the master, whic causes an additional scheduling operation on the dedicated processor. As noted in Section IV-A2, the master notification never preempts the dedicated processor and, instead, triggers a kernel invocation that interferes with task execution. Our implementation cannot avoid these interfering events as they stem from arming the local processor timer. This notwithstanding, the last column in the diagram (TOT) shows that QPS incurs a significantly lower overall system overhead than RUN, which pays for the update of the reduction tree at each budget exhaustion event (TREE). The reported value includes the cumulative cost of all kernel primitives (thus all tree updates in RUN) plus the other system overheads. Interestingly, these results appeared only when we isolated the packing effect and *not* for the set of randomly generated experiments.

D. Further considerations

Although accurate, our analysis incurs the significant overhead introduced by the LITMUS$^{\mathrm{RT}}$ stack, as well as Linux timing anomalies [26], which prevent us from measuring a trustworthy absolute upper bound on the execution of each scheduling primitive. For this reason, and in line with our previous work, our analysis focused on the average cost instead of the worst case. The results of this analysis should also be compared to those presented in prior work [11], to observe that QPS behaves exactly as P-EDF, as they share most traits. On a fully partitioned system no additional overhead is introduced by our QPS implementation. Although when approaching full-system utilization, QPS may incur a significant overhead, comparable to that of a global algorithm, it can still sustain much higher workload that partitioning can do.

[2]The reader should note that, while the overhead measurements reported here only refer to task sets reduced with FFD bin-packing heuristic, analogous trends have been obtained with all other heuristics we considered.

Improvements in the implementation of the two algorithms are possible, and we are considering them for future work, particularly for RUN whose implementation cannot take advantage of simultaneous scheduling events originated along the reduction tree. Much like QPS that incurs additional overhead of keeping master/slave synchronization. An approach similar to that presented in [12], adapted to avoid devoting an entire processor to scheduling activities, can also be considered to reduce the kernel interference at run time.

VI. Conclusion

RUN and QPS stand apart in the class of optimal multicore scheduling algorithms, on account of their distinctive use of partitioned proportionate fairness. In spite of numerous commonalities, there are sufficiently many differences between RUN and QPS to warrant a thorough experimental evaluation of their run-time performance. This paper presents a comprehensive implementation and evaluation effort to that very end. The results we obtained for our implementation of both algorithms show that QPS does indeed incur lower run-time overhead than RUN, except near full system utilization, where high sensitivity to packing emerges, which instead leaves RUN unharmed. At high system utilization, a higher depth of the processor hierarchy increases the number of scheduling events that QPS generates at run time and needs to coordinate across that hierarchy. This overhead penalizes QPS over RUN and cannot be fully compensated for the lightweight nature of its scheduling primitives. This trait of QPS obviously pays tribute to its heritage from partitioned approaches. This notwithstanding, the design of QPS allows it to incur an overhead comparable to that of P-EDF when the system is fully partitioned, thus confirming its practical viability. In future work we plan to evaluate the run-time cost of handling sporadic releases in QPS, which we did not consider in this paper for fair comparison with RUN.

Acknowledgment

The work presented in this paper has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 611085 (PROXIMA, www.proxima-project.eu). We are also grateful to the original authors of RUN and QPS for useful comments and advice on the contents of this work.

References

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

[2] G. Levin, S. Funk, C. Sadowski, I. Pye, and S. Brandt, "DP-FAIR: A simple model for understanding optimal multiprocessor scheduling," in *ECRTS*, 2010, pp. 3–13.

[3] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, pp. 35:1–35:44, 2011.

[4] J. Anderson, V. Bud, and U. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *ECRTS*, 2005, pp. 199–208.

[5] J. Anderson, V. Bud, and U. Devi, "An EDF-based restricted-migration scheduling algorithm for multiprocessor soft real-time systems," *Real-Time Systems*, vol. 38, no. 2, pp. 85–131, 2008.

[6] A. Bastoni, B. Brandenburg, and J. Anderson, "Is semi-partitioned scheduling practical?" in *ECRTS*, 2011, pp. 125–135.

[7] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "RUN: Optimal multiprocessor real-time scheduling via reduction to uniprocessor," in *RTSS*, 2011, pp. 104–115.

[8] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt, "OUTSTANDING PAPER: Optimal and adaptive multiprocessor real-time scheduling: The quasi-partitioning approach," in *ECRTS 2014*, 2014, pp. 291–300.

[9] S. Funk, G. Levin, C. Sadowski, I. Pye, and S. Brandt, "DP-Fair: a unifying theory for optimal hard real-time multiprocessor scheduling," *Real-Time Systems*, vol. 47, no. 5, pp. 389–429, 2011.

[10] G. Levin, C. Sadowski, I. Pye, and S. Brandt, "SnS: a simple model for understanding optimal hard real-time multi-processor scheduling," *Univ. of California, Tech. Rep. UCSC-SOE-11-09*, 2009.

[11] D. Compagnin, E. Mezzetti, and T. Vardanega, "Putting RUN into practice: Implementation and evaluation," in *ECRTS 2014*, 2014, pp. 75–84.

[12] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing," in *RTAS*, 2014, pp. 263–274.

[13] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *RTSS 2010*, 2010, pp. 14–24.

[14] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *ECRTS*, 2009, pp. 249–258.

[15] A. Burns, R. Davis, P. Wang, and F. Zhang, "Partitioned EDF scheduling for multiprocessors using a C=D task splitting scheme," *Real-Time Systems*, vol. 48, no. 1, pp. 3–33, 2012.

[16] J. A. Santos-Jr, G. Lima, and K. Bletsas, "On the processor utilisation bound of the C=D scheduling algorithm," in *Proceedings of Real-time systems (Alanfest 2013)*, 2013, pp. 119–132.

[17] P. Regnier, G. Lima, E. Massa, G. Levin, and S. Brandt, "Multiprocessor scheduling by reduction to uniprocessor: an original optimal approach," *Real-Time Systems*, vol. 49, no. 4, pp. 436–474, 2013.

[18] S. Baruah, J. Gehrke, and C. Plaxton, "Fast scheduling of periodic tasks on multiple resources," in *Parallel Processing Symposium, 1995. Proceedings., 9th International*, 1995, pp. 280–288.

[19] E. Massa, G. Lima, and P. Regnier, "Revealing the secrets of RUN and QPS: New trends for optimal real-time multiprocessor scheduling," in *SBESC*, 2014, pp. 150–155.

[20] T. P. Baker and S. K. Baruah, "Schedulability analysis of multiprocessor sporadic task systems," in *Handbook of Realtime and Embedded Systems*. CRC Press, 2007.

[21] J. Calandrino, J. Anderson, and D. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *ECRTS*, 2007, pp. 247–258.

[22] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 8th ed. Wiley Publishing, 2008.

[23] S. Altmeyer and C. Burguiere, "A new notion of useful cache block to improve the bounds of cache-related preemption delay," in *ECRTS*, 2009, pp. 109–118.

[24] N. C. Audsley, I. J. Bate, and A. Burns, "Putting fixed priority scheduling theory into engineering practice for safety critical applications," in *RTAS*, 1996.

[25] L. C. Briand and D. M. Roy, *Meeting Deadlines in Hard Real-Time Systems*. IEEE Computer Society Press, 1997.

[26] B. B. Brandenburg, A. D. Block, J. M. Calandrino, U. Devi, H. Leontyev, and J. H. Anderson, "LITMUS RT: A status report," in *Proceedings of the 9th Real-Time Linux Workshop*, 2007, pp. 107–123.