Global Scheduling Not Required: Simple, Near-Optimal Multiprocessor Real-Time Scheduling with Semi-Partitioned Reservations

Björn B. Brandenburg Mahircan Gül
Max Planck Institute for Software Systems (*MPI-SWS*)

***Abstract*—Prior work has identified several optimal algorithms for scheduling independent, implicit-deadline sporadic (or periodic) real-time tasks on identical multiprocessors. These algorithms, however, are subject to high conceptual complexity and typically incur considerable runtime overheads. This paper establishes that, *empirically*, near-optimal schedulability can also be achieved with a far simpler approach that combines three well-known techniques (reservations, semi-partitioned scheduling, and period transformation) with some novel task-placement heuristics.**

In large-scale schedulability experiments, the proposed approach is shown to achieve near-optimal hard real-time schedulability (99+% schedulable utilization) across a wide range of processor and task counts. With an implementation in LITMUS$^{\text{RT}}$, the proposed approach is shown to be practical and to incur only low runtime overheads, comparable to a conventional partitioned scheduler. It is further shown that basic slack management techniques can help to avoid more than 50% of all migrations of semi-partitioned reservations if tasks execute on average for less than their provisioned worst-case execution time.

Two main conclusions are drawn: pragmatically speaking, global scheduling is not required to support static workloads of independent, implicit-deadline sporadic (or periodic) tasks; and since such simple workloads are well supported, future research on multiprocessor real-time scheduling should consider more challenging workloads (*e.g.*, adaptive workloads, dynamic task arrivals or mode changes, shared resources, precedence constraints, *etc.*).

I. Introduction

Commonsense dictates that *simplicity* must be a primary goal in the design of critical real-time systems, as complex solutions are more costly to develop, test, and validate, and potentially also less robust. A well-engineered design will hence employ a more complex, more difficult solution only if all simpler alternatives fail to accomplish a given objective.

However, simplicity is not the only concern: to be economically viable, a design must usually also be *efficient*, in the sense that it must fully exploit the available resources to avoid over-provisioning. This is particularly true in the context of embedded real-time systems subject to tight resource constraints.

Simple vs. efficient scheduling. Unfortunately, when it comes to scheduling real-time tasks upon shared-memory multiprocessors, simplicity is traditionally at odds with design-time efficiency. The simplest approach is *partitioned scheduling*, wherein tasks are statically mapped to individual processors and a classic uniprocessor policy such as *earliest-deadline first* (EDF) is used on each processor [27]. However, partitioned scheduling suffers from inherent efficiency limitations due to the underlying bin-packing problem [7]: for certain workloads that require more than 50% of the available processing capacity, a valid task-to-processor mapping simply does not exist. *Semi-partitioned* schedulers [5] partially overcome this limitation by *migrating* tasks that could not be statically assigned among two or more processors at runtime, thus spreading their load dynamically, but have been observed to still struggle with some high-utilization task sets (*i.e.*, 90% utilization or higher) [13, 19, 35].

At the other extreme, in work focused primarily on design-time efficiency (rather than simplicity), many *optimal* multiprocessor real-time scheduling algorithms have been identified (*e.g.*, [8, 11, 32, 39, 42, 44, 48, 51]). These schedulers, most prominently PD^2 [48], LLREF [24], RUN [44], U-EDF [42], and QPS [39], are optimal in the sense that all processors can be *fully* allocated to real-time tasks without risking deadline misses (under certain assumptions, see §II). Ignoring overheads, these schedulers are *perfectly efficient* as no over-provisioning is needed to guarantee a workload's timing requirements.

Unfortunately, this provable efficiency comes with a substantial tradeoff in terms of complexity. At runtime, optimal schedulers tend to require careful coordination among all or many of the processors—they are *global schedulers* in nature—which translates into non-trivial runtime overheads [12, 16, 18, 37]. Optimal schedulers are arguably also much more difficult to understand, to extend and adapt, to implement, and to test in a real OS. One can further hazard a guess that sophisticated scheduling approaches are likely quite challenging to certify.

This paper. We take another look at multiprocessor real-time scheduling, but this time with a focus on simplicity. Motivated by our experience in implementing, evaluating, and maintaining schedulers in LITMUS$^{\text{RT}}$ [2, 16, 21], we ask: *is it really necessary to resort to complex, difficult to understand, difficult to implement, and difficult to extend algorithms to efficiently provision hard real-time workloads on multiprocessors?*

This work establishes that, empirically, the answer is 'no'. Near-optimal hard real-time schedulability—that is, schedulable processor utilizations exceeding 99%, which is arguably "good enough" for all practical purposes—can be achieved with a simple (but novel) combination of three well-known techniques:

1) *processor reservations* [40] with basic slack management,
2) *semi-partitioned scheduling* with the C=D heuristic [19], and
3) *period transformation* (*i.e.*, "slicing" jobs across multiple budgets with a shorter period).

While each technique is well-established by itself, to the best of our knowledge, they have not been studied in conjunction, nor has their surprising joint efficacy been reported in the literature.

Contributions. To achieve the claimed near-optimal schedulability, and to limit the number of job migrations, we introduce several simple, but effective "tweaks" to the underlying techniques (§IV), including two *meta-heuristics* (*i.e.*, heuristics that combine other heuristics) that have not been studied before.

This paper has passed an Artifact Evaluation process. For additional details, please refer to http://2016.rtss.org/artifact-evaluation-for-rtss/.

We empirically substantiate our central claim with schedulability experiments covering a wide range of processor and task counts using two different task-set generators (§V). We first identify under which conditions conventional partitioned scheduling fails, and then assess the efficacy of the proposed semi-partitioning and period-transformation techniques in these cases. *Our results show that, under the idealized, overhead-free conditions assumed in optimality proofs, the proposed approach generally achieves schedulable utilizations exceeding 99%.*

We further show the proposed approach to be practical. To this end, we first establish that it does not require infeasibly small job slices: high schedulability is achieved even if a minimum allocation granularity of hundreds of microseconds is enforced (§VI). We also show that the maximum context-switch rate is usually (far) lower than under optimal schedulers.

Most importantly, in §VII, we report on a robust implementation of the proposed approach in LITMUS$^{\text{RT}}$ [2, 16, 21]. We benchmarked this implementation on a 44-core Intel Xeon platform and observed that the proposed approach incurs overheads as low as a conventional partitioned scheduler (§VII-A). In particular, our implementation exhibited substantially lower overheads than prior implementations [25, 26] of the optimal RUN [44] and QPS [39] schedulers (§VII-B).

Finally, we provide empirical evidence that shows the proposed slack management heuristics to be highly effective at avoiding job migrations (§VII-C), yielding an up to 5x reduction in migration rate if tasks under-run their WCET.

A discussion of limitations and related work is provided in §VIII. We next clarify our assumptions (§II) and provide a brief review of the concepts that comprise our approach (§III).

II. System Model and Assumptions

We seek to schedule a real-time workload τ consisting of n *sporadic* real-time tasks $\tau_1, \ldots, \tau_n$ upon a shared-memory platform consisting of m *identical* processors. As in prior work on optimal multiprocessor real-time scheduling [11, 39, 44, 48], we initially assume a highly idealized setting. We revisit overheads and more general workloads in §VII and §VIII, respectively.

Each task $\tau_i = (C_i, D_i, T_i)$ is characterized by a *period* (or *minimum inter-arrival time*) T_i, a *relative deadline* D_i, and a per-job *worst-case execution cost* (WCET) C_i. As in prior work on optimal multiprocessor real-time scheduling [11, 39, 44, 48], we initially permit only *implicit-deadline* tasks: $D_i = T_i$ for all τ_i, in which case we write $\tau_i = (C_i, T_i)$. If tasks are permitted to have *constrained deadlines* (*i.e.*, if $D_i < T_i$), optimal scheduling generally requires clairvoyance [30]. If relative deadlines exceed periods (*i.e.*, if $D_i > T_i$), they can be trivially truncated.

Each sporadic task τ_i releases a sequence of jobs. Each job requires at most C_i time units of processor service. Any two jobs of τ_i are released *at least* T_i time units apart. As a special case, consecutive jobs of a *periodic* task are separated by *exactly* T_i time units. Jobs and tasks are sequential: a job can use at most one processor at any time, and the next job can commence execution only when the previous job has completed.

Each job of τ_i must finish within D_i time units of its release. The workload is *schedulable* (under a particular scheduler) if all jobs of all tasks are guaranteed to finish on time for all possible job arrival sequences. The ratio $\frac{C_i}{\min(T_i, D_i)}$ denotes τ_i's *density*, the ratio $\frac{C_i}{T_i}$ denotes τ_i's *utilization*, and the sum $U(\tau) = \sum_{i=1}^{n} \frac{C_i}{T_i}$ is the workload's *total utilization*. We say that an algorithm (empirically) achieves a *schedulable utilization of* $X\%$ if all (tested) workloads with $\frac{U(\tau)}{m} \leq \frac{X}{100}$ are schedulable.

Again matching the prior literature on optimal scheduling, all tasks are assumed to be *independent* (*i.e.*, they share no resources besides the processors) and to execute *without self-suspensions* (*i.e.*, a job, once released, is ready for execution until finished). The workload is furthermore *static*: tasks do not join or leave the system, nor do any task parameters change. Finally, WCETs are considered fixed and to *not* be affected by migrations, cache effects, or inter-core interference. In §VIII, we discuss the impact of lifting these unrealistic assumptions.

III. Essential Background

Our proposal reuses simple, widely-known concepts. For the sake of completeness, we provide a brief summary here.

A. Reservation-based EDF Scheduling with Slack Reclamation

The idea behind *processor reservations*, a classic concept already employed in RT-Mach [40], is to separate (low-level) process dispatching from the (high-level) real-time policy and bookkeeping. Instead of scheduling processes directly, the scheduler manages a set of processor reservations. Each processor reservation in turn serves one or more processes.

In this simple two-level scheduling scheme, whenever a reservation is selected for service by the top-level scheduler, a reservation-local scheduler picks one of the contained processes for dispatching. When a reservation has exhausted its *budget*, the contained processes are cut off from processor service until the reservation's budget has been *replenished*. The primary advantages of reservation-based scheduling are improved temporal isolation (*i.e.*, WCET overruns and aperiodic activation patterns are contained due to precisely enforced budgets) and the ability to implement *conceptually* sequential real-time "tasks" as collections of cooperating processes or threads.

A rich literature on various reservation types and algorithms exists (see *e.g.* [4, 40, 43, 49]). We require here only the most basic case: each task τ_i is contained in a corresponding, private *polling reservation* with a *maximum budget* Q_i and a *replenishment interval* Δ_i. By default, we simply set $Q_i = C_i$ and $\Delta_i = T_i$. At runtime, each reservation further maintains a *current budget* b_i and a current *scheduling deadline* d_i.

A polling reservation works as follows. Initially, a polling reservation is *inactive*. When the contained task first releases a job at time t, it becomes *active*, b_i is set to Q_i, and d_i is set to $t + \Delta_i$. Whenever the reservation is selected for service, b_i depletes at linear rate. When b_i reaches zero, the reservation is *depleted* and no longer considered for scheduling until, at time $d_i = t + \Delta_i$, the current budget b_i is replenished to Q_i and the scheduling deadline d_i is postponed by Δ_i. When the contained task's job completes, the reservation becomes inactive again.

There are two points to clarify. First, what happens when an active reservation exhausts its budget (*i.e.*, when a job fails to complete within one budget)? Rajkumar *et al.* [43] introduced two types of reservations: a *hard* reservation is indeed cut off from service, as described so far. In contrast, a *soft* reservation is allowed to receive additional service even when it is depleted, however only with background priority (*i.e.*, if the system would otherwise idle). We use both types of reservations.

Second, what happens if a job under-runs its WCET (*i.e.*, when a reservation becomes inactive with a non-zero current budget)? In this case, the reservation generates *dynamic slack* (*i.e.*, allocated, but unneeded processor capacity) that can be *reclaimed* and repurposed by the scheduler. To exploit such slack, Caccamo *et al.* [20] proposed a particularly simple and effective slack management strategy called CASH, which we adopt in our solution (§IV). In a nutshell, CASH awards all slack to the reservation with the next-earliest scheduling deadline, modulo some corner cases that we gloss over here.

The scheduler thus works as follows (on a uniprocessor): the top-level EDF scheduler always services (one of) the active, non-depleted reservation(s) with the earliest scheduling deadline(s), or if no such reservation exists, any active, but depleted soft reservations in a round-robin fashion. Whenever possible, a reservation selected for service consumes any dynamic slack before draining its own budget, as determined by CASH [20].

B. Period Transformation

By *period transformation*, we refer to an obvious "trick" that exploits that reservation parameters and task requirements need not exactly match. For example, a sporadic task τ_i with WCET $C_i = 50ms$ and period $T_i = 100ms$ can also be scheduled by a reservation with budget $Q_i = 25ms$ and period $\Delta_i = 50ms$: in this case, each job of τ_i is "sliced" across (up to) two replenishment intervals, which spreads out the job's execution in time at the expense of causing additional context switches.

In general, ignoring overheads, an implicit-deadline task $\tau_i = (C_i, T_i)$ can be trivially served by any reservation with parameters $Q_i = \frac{C_i}{k_i}$ and $\Delta_i = \frac{T_i}{k_i}$, where $k_i \in \mathbb{N}$. We call this parameter k_i the *period ratio*. Non-integer period ratios and more advanced budgeting are possible, but not relevant here.

C. Partitioned Scheduling

As discussed in §I, partitioned scheduling is a straightforward way to extend any uniprocessor scheme (such as the one in §III-A) to multiprocessors. The scheme is simply instantiated in isolation once per processor, and each task (or reservation) is statically assigned to one of the processors. At runtime, no communication takes place among the individual schedulers.

In the case of implicit-deadline tasks (or reservations) and EDF-scheduled processors, the task assignment problem corresponds exactly to the classic bin-packing problem (where tasks are items, utilizations are item sizes, and processors are bins), and is hence usually solved with common bin-packing heuristics.

Relevant to this work are the *first-fit decreasing* (**FFD**) and *worst-fit decreasing* (**WFD**) heuristics. In both cases, the to-be-assigned tasks are considered in order of decreasing density and placed as follows. Let $P_1, \ldots, P_m$ denote the available processors, and let $\tau(P_i)$ denote the tasks already assigned to P_i. With the FFD heuristic, a task τ_i is assigned to the lowest-indexed processor that satisfies $U(\tau(P_i)) + \frac{C_i}{T_i} \leq 1$ (if any).

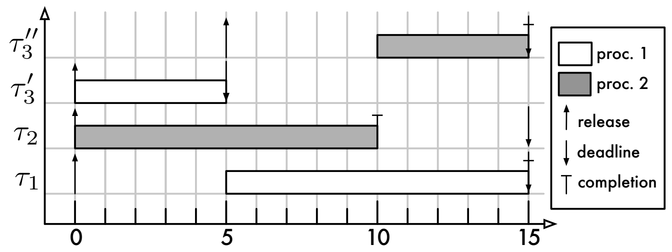


Fig. 1. An example semi-partitioned schedule. Task τ_3 has been split into two separately allocated subtasks. The second subtask τ_3'' is activated at time 5 when the first subtask τ_3' has exhausted its budget of $C_3' = 5$ time units.

The WFD heuristic instead picks the processor P_i such that $U(\tau(P_i)) + \frac{C_i}{T_i}$ is minimized (*i.e.*, it maximizes the remaining gap). As a result, the WFD heuristic tends to spread out tasks roughly evenly among all processors, whereas the FFD heuristic tries to fully load some processors while idling others.

D. Semi-Partitioned Scheduling

The key limitation of partitioned scheduling is that a valid task-to-processor mapping may not exist (or may not be found). For example, consider three identical tasks $\tau_1 = \tau_2 = \tau_3 = (10, 15)$ that are to be scheduled on two processors. As any two of the tasks over-utilize a single processor, no valid mapping exists.

Semi-partitioning strategies attempt to overcome this problem by *splitting* some of the tasks into multiple smaller, constrained-deadline *subtasks*. These subtasks are then assigned to different processors, conceptually subject to a *precedence constraint* to serialize their execution. At runtime, each job of the split task must then *migrate* among the processors to use the processor time allocated to the subtasks on their respective processors.

Returning to our example, suppose that τ_1 and τ_2 have been assigned to processors P_1 and P_2, respectively. In this case, one can split τ_3 into two constrained-deadline subtasks $\tau_3' = (5, 5, 15)$ and $\tau_3'' = (5, 10, 15)$, where $C_3 = C_3' + C_3''$ and $T_3 = D_3 = D_3' + D_3''$. Using Baruah's *processor-demand criterion* (PDC) [10], it can be verified that it is safe to assign τ_3' to processor P_1 and τ_3'' to processor P_2.

An illustrative example schedule is shown in Fig. 1. In this example, all three tasks release a job at time 0. As a result, the subtask τ_3' is also considered to release a job at time 0. Because of its tight deadline, the "job" of τ_3' is serviced immediately on processor P_1, which means that the actual job of task τ_3 begins execution. At time 5, the "job" of τ_3' is "complete" (*i.e.*, has used up its allocation) and τ_3'', the second subtask of τ_3, is activated on processor P_2. However, the newly released "job" of τ_3'' is not serviced until time 10 because it does not have an earlier deadline than the currently scheduled job (of τ_2). At that point, the actual job of τ_3 migrates from P_1 to P_2 to be scheduled until time 15. All jobs complete by their deadline because τ_3's execution has been carefully distributed across both processors.

Generally speaking, the main decisions in a semi-partitioned approach are **(i)** how to identify which tasks to split, **(ii)** how to split them (*i.e.*, selecting the parameters of the subtasks), and **(iii)** where to assign the subtasks. No provably optimal approach is known, but several heuristics have been proposed.

In this work, we build upon Burns *et al.*'s C=D approach [19], which has been empirically shown to be particularly effective [19]. Burns *et al.* introduced the *C=D task-splitting rule* (described below), which they combined with two configurable *task-selection strategies* (or meta-heuristics), called the *continuous* and the *pre-selection* strategies, respectively. The continuous strategy identifies which tasks to split dynamically, whereas the pre-selection strategy first determines which tasks to split, then partitions the remaining tasks, and finally attempts to split the pre-selected tasks. Both strategies can be instantiated with different task ordering and placement heuristics [19].

For our purposes, we require only the continuous approach, which we apply as follows. We start with the FFD heuristic, and whenever a task cannot be assigned to a non-empty partition, we split the task according to these rules:

1) Let τ_i denote the task that did not fit, and let P_k denote the processor to which a task was last added.
2) Identify the largest possible $x < C_i$ such that a subtask $\tau_i' = (x, x, T_i)$ can be feasibly added to $\tau(P_k)$. Note that $C_i' = D_i'$, hence the name of Burns *et al.*'s heuristic.
3) Add the subtask $\tau_i' = (x, x, T_i)$ to P_k, and return the subtask $\tau_i'' = (C_i - x, T_i - x, T_i)$ to the set of unassigned tasks, to be reconsidered according to its density.

For Step 2), Burns *et al.* [19] provide an algorithm to identify the largest x that satisfies Baruah *et al.*'s (necessary and sufficient) PDC [10]. After the split, processor P_k is maximally filled and no longer considered by the FFD heuristic.

In the example shown in Fig. 1, task τ_3 was split using the C=D heuristic. In the example, $x = 5$ is the largest possible "chunk" for which $D_3' = C_3' = x$ is feasible on processor P_1.

With all essential concepts in place, we can now introduce how we integrate reservation-based scheduling (§III-A), period transformation (§III-B), and semi-partitioning (§III-D).

IV. Our Approach: Semi-Partitioned Reservations

Our approach is a straightforward combination of the techniques reviewed in the preceding section: we employ reservation-based scheduling with EDF on each processor, encapsulate all tasks in polling reservations, and then apply Burns *et al.*'s C=D approach [19] to the reservations (rather than directly to tasks) to allow for period transformation. On top of this foundation, we introduce several runtime tweaks to lower overheads (§IV-A), and substantially extend the task placement phase to increase the likelihood that a semi-partitioning can be found (§IV-B).

A. Runtime Tweaks

Assuming a valid semi-partitioning has been found, we use three techniques to lower runtime overheads. Compared to a conventional partitioned scheduler, the main additional overhead in a semi-partitioned scheduler stems from job migrations. In the worst case, such migrations are inevitable (otherwise the workload could have been partitioned), but it is possible to dynamically avoid many migrations in common-case situations.

Flip the C=D subtask order. The first step is to *flip* the C=D subtask order: when splitting a task, we still create a subtask τ_i' with $C_i' = D_i'$, but (arbitrarily) declare this to be the *tail* of the job. In other words, when a split task τ_i releases a job at time t, we first activate τ_i'' (with $D_i'' = T_i - D_i'$), and activate

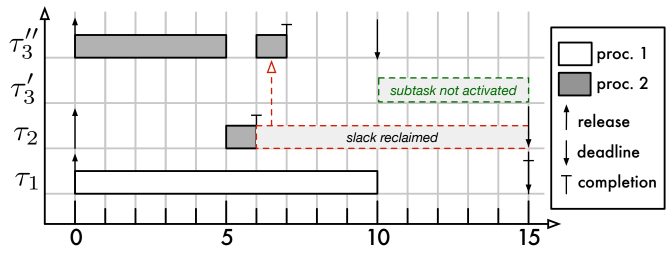


Fig. 2. An example of migration avoidance. As the C=D subtask order is flipped, τ_3'' is activated at time 0. Since τ_2 under-runs its budget, slack reclamation allows τ_3's job to finish before τ_3' must be activated at time 10, thus saving a migration.

τ_i' (and trigger the migration) only at time $t + D_i''$. Fig. 2 shows an example, which we discuss shortly after two more tweaks.

From a theory point of view, the order of the subtasks is irrelevant: each job of a split task τ_i is guaranteed to receive sufficient processor time either way. However, from a practical point of view, flipping the C=D subtask order is a major improvement since it opens the door for the next two tweaks.

Use soft reservations to avoid migrations. Now that jobs of split tasks start executing in the subtask with non-zero laxity (*i.e.*, $C_i'' < D_i''$), we can employ proven techniques for assigning it spare capacity. The hope is that, by giving the first subtask processor time in excess of its reserved amount, the job of the migrating task will be able to finish *before* the migration occurs.

Based on this consideration, we use soft reservations [43] to realize reservations that correspond to subtasks of split tasks. Partitioned tasks can be placed in either hard or soft reservations.

Use slack reclamation. We additionally employ CASH-based slack reclamation [20], as reviewed in §III-A, as it may also benefit migrating reservations. One could further experiment with overriding the CASH heuristic to directly award any slack to migrating reservations (regardless of their current scheduling deadlines). However, in our implementation (§VII), we have for now stuck with the original CASH rules.

An example illustrating the impact of these tweaks is shown in Fig. 2. It shows the same scenario as depicted in Fig. 1, but this time subject to the flipped C=D subtask order and slack reclamation. For the sake of the example, suppose that τ_2 and τ_3's jobs require only 1 and 6 (instead of $C_2 = C_3 = 10$) time units of service, respectively (*i.e.*, they under-run their WCETs).

When τ_3 releases its job at time 0, its subtask τ_3'' on processor P_2 is activated. Since it has an earlier scheduling deadline than τ_2, it is scheduled until its budget is depleted at time 5. At that point τ_2's job commences execution, but since it under-runs its budget, it finishes early at time 6, thus generating 9 time units of slack. This slack is reclaimed by CASH and given to the active, but depleted reservation of τ_3. As a result, τ_3's job finishes at time 7, thereby avoiding the migration at time 10.

In §VII, we report on an empirical evaluation of these migration avoidance techniques in the real system. Next, we present several new strategies for finding valid task assignments.

B. Task Allocation Strategies

The C=D heuristic as instantiated in §III-D bundles three key choices: **(i)** when to split (on FFD failure), **(ii)** which task to split (the task that failed to be assigned), and **(iii)** how to split (the C=D rule). As with all best-effort heuristics, these choices are somewhat arbitrary, but justified by good performance in empirical comparisons [19]. Nonetheless, we obtained some improvements by keeping (iii) while varying both (i) and (ii). In the following, we document the heuristics that, together, achieve near-optimal schedulability, as discussed later in §V.

Run many heuristics. First of all, we observe that most partitioning heuristics are quite cheap to compute, compared to modern hardware capabilities and usual task-set sizes. Hence it is not necessary to limit oneself to just one heuristic—instead, we try many heuristics for each task set, roughly in the order from quickest to slowest to compute, starting with WFD and FFD, until one succeeds. While the success rates of similar heuristics often differ by only a few task sets, the cumulative effect of applying many different heuristics is significant.

Combine WFD with C=D. While Burns *et al.* considered only the FFD heuristic in their experiments [19], we found that the C=D splitting rule combines well also with the WFD heuristic. The reason is that the WFD heuristic leaves roughly equally-sized "gaps" on all processors, which lend themselves to placing large subtasks. We denote this combination as **WFD-C=D**, and call the original heuristic **FFD-C=D** to avoid ambiguity. Under the WFD-C=D heuristic, the C=D splitting rule is triggered for processor P_k and task τ_i when $U(\tau(P_k))$ is minimal among all partitions, but τ_i cannot feasibly be assigned to P_k.

Find the maximal split. The WFD-C=D heuristic can further be improved by observing that the processor that triggered the splitting rule is not necessarily best suited to accommodate a large subtask, that is, it may not yield the maximal possible "chunk-size" x, as determined by the C=D splitting rule. A simple improvement is to compute a "chunk-size" x_k for *each* processor P_k, and to assign the first subtask τ_i' to whichever processor yields the maximal x_k. We denote the resulting placement heuristic **WFD-C=D-MS**.

Since this heuristic is more expensive to compute, and since it is not always better, we run the WFD-C=D heuristic first, and apply WFD-C=D-MS only if plain WFD-C=D fails. We denote this "WFD twice" strategy as **2WFD-C=D**.

Two-phase semi-partitioning. Recall from §III-D that, under Burns *et al.*'s continuous strategy [19], a task is split *immediately* whenever a processor fills up. Sometimes, this is too aggressive.

We add four new two-phase semi-partitioning heuristics that apply first either WFD or FFD, and then 2WFD-C=D or FFD-C=D. For brevity, we call these composite heuristics **WWFD**, **FWFD**, **WFFD**, and **FFFD**, respectively. In the first phase, the WFD or FFD strategy is used *without* task splitting to place all or most tasks. In the second phase, 2WFD-C=D or FFD-C=D is then applied *only* to the remaining, still-unassigned tasks (if any), while respecting all existing first-phase assignments.

In contrast to Burns *et al.*'s pre-selection strategy [19], this two-phase approach avoids having to "guess" which tasks to split. Rather, the set of tasks to be split is discovered automatically as a result of the failure of the heuristic used in the first phase.

Next, we introduce two meta-heuristics that can be applied to any of the heuristics described so far.

Pre-assign failures meta-heuristic. The first meta-heuristic attempts to use the observation that certain tasks failed to be assigned as a signal in a feedback loop: tasks that could not be assigned are likely the "most difficult to assign," and hence should be placed before considering the remaining tasks.

Based on this intuition, the *pre-assign failures* (**PAF**) meta-heuristic proceeds as follows. Given two heuristics h_1 and h_2, it splits the set of tasks τ into two disjoint subsets, $failures$ and $rest$. Initially, $failures = \emptyset$ and $rest = \tau$. It then repeatedly applies h_1 to all tasks in $failures$ to pre-assign any "troublesome" tasks before applying h_2 to $rest$ to place the remaining tasks (while respecting the pre-assignment from h_1). Any task that fails to be placed by h_2 is moved from $rest$ to $failures$. After at most $|\tau|$ iterations, the meta-heuristic either succeeds if h_2 can place all tasks, or fails if h_1 fails to place all tasks in $failures$.

If none of the basic heuristics succeeds for a given task set τ, we try six instantiations of the PAF meta-heuristic with $h_2 =$ 2WFD-C=D and h_1 being one of the heuristics FFD-C=D, 2WFD-C=D, WWFD, FWFD, WFFD, or FFFD.

Reduce-periods meta-heuristic. The final strategy period-transforms tasks with periods above a configurable threshold, based on the following rationale.

We observed that the C=D splitting rule is not "scale-invariant," in the sense that, all other things being equal, it does not work nearly as well for splitting a task $\tau_i = (100, 1000)$ as it does for splitting a task $\tau_j = (2, 20)$, even though the two tasks have the same utilization. This is because the maximum zero-laxity "chunk size" x that can be placed on a given processor P_k is determined by the parameters of the tasks *already placed* on P_k, and not by the utilization of the task that is to be split. Obviously, splitting off a subtask with a budget of, say, $x = 1$ has a much larger effect on τ_j than it does on τ_i.

The reduce-periods (**RP**) meta-heuristic caters to this effect by repeatedly period-transforming both **(i)** all tasks with periods above a certain threshold (which is lowered from iteration to iteration) and **(ii)** any tasks that failed to be assigned in previous iterations (as in the PAF meta-heuristic). When transforming a task, the PAF meta-heuristic picks the smallest period ratio k_i that renders the transformed period smaller than the current threshold. The PAF meta-heuristic terminates either when the underlying heuristic succeeds in finding a valid semi-partitioning, or when a minimum threshold has been reached. In our experiments, we impose a minimum threshold of $4ms$.

In a nutshell, C=D works best if WCET magnitudes are small in relation to the relative deadlines of other tasks, and the RP meta-heuristic breaks down large WCETs. We use it on top of the 2WFD-C=D, FWFD, and WWFD heuristics.

All (meta-)heuristics are available in the open-source library SchedCAT [3]. Illustrative pseudocode is provided online [1].

V. Schedulability Evaluation

We conducted schedulability experiments to evaluate our semi-partitioned reservations approach under the same idealized conditions as assumed in the proofs of optimality of PD^2, LLREF, RUN, QPS, *etc.*, and as spelled out in §II.

Setup. We explored a diverse parameter space and considered platforms with $m \in \{2, 4, 8, 16, 24, 32, 64\}$ processors. For each m, we tested workloads with task counts in the range $n \in [m+1, 4m]$. For a given m and n, we varied the total utilization U across $U \in [1, m]$. Finally, for each (m, n, U), we generated more than 1,100 task sets with Emberson *et al.*'s unbiased task set generator [28], which yields uniformly distributed task utilizations. Task periods were drawn uniformly at random from the set {1, 2, 4, 5, 8, 10, 20, 25, 40, 50, 100, 125, 200, 250, 500, 1000} (in milliseconds), which we chose both to reflect realistic timing constraints and to ensure a short hyperperiod.

We applied all strategies given in §IV-B to all task sets and determined the fraction of task sets for which a valid semi-partitioning could be found. Fig. 3 shows an excerpt of our results for $m = 8$, which we discuss in detail in the following.

Partitioning. Much of the research into global and optimal multiprocessor real-time scheduling is in large parts motivated by the observation that the (far simpler) partitioned approach cannot support certain feasible workloads. However, empirically speaking, when exactly does partitioning actually fail?

Fig. 3(a) shows the fraction of task sets for which either the WFD or FFD heuristic could find a valid mapping, as a function of U for $m = 8$ and different n. It is immediately apparent that the number of tasks has a large impact on schedulability, which intuitively makes sense: bin-packing many smaller items is much simpler than packing fewer, larger ones.

What is surprising, though, is that even for relatively modest task counts, partitioning is very effective up to high utilizations. For example, for $n = 3m = 24$, virtually all tested task sets with a utilization of at most 95% could be successfully partitioned, a trend that we observed also with all other considered processor counts. *We conclude that multiprocessor real-time scheduling (with implicit deadlines) is difficult only if, roughly, $n < 3m$.*

However, we also observe that partitioning fails to support workloads consisting of few high-utilization tasks. In our experiments, task sets with $n = m + 1$ or $n = m + 2$ generally proved to be the most difficult workloads. For example, in Fig. 3(a), the schedulability of workloads with $n = m + 1 = 9$ and $n = m+2 = 10$ tasks starts to degrade already at utilizations as low as 77%-80%. How effective is semi-partitioning at overcoming these limits?

Semi-partitioning. Fig. 3(b) shows the fraction of task sets for which any of the heuristics (but not meta-heuristics) in §IV-B succeeded at finding a valid semi-partitioning. First, note that Fig. 3(b) uses a different X-axis scale: even the most difficult to schedule workloads ($n = m + 1$, $n = m + 2$) pose no problems up to 90% utilization. In fact, for the simpler case of $n = 3m = 24$, close to 99% schedulable utilization is reached. *We observe that basic semi-partitioning techniques, paired with simple heuristics, can reach at least 90% schedulable utilization.*

Still, for workloads with few tasks, noticeable performance degradation becomes apparent around 92%–96% utilization. (However, for context, one may also recall that, on uniprocessors, fixed-priority scheduling with a schedulable utilization below 90% is considered acceptable for most practical purposes.)

Meta-heuristics. Finally, we consider the impact of applying the PAF and RP meta-heuristics. The results for semi-partitioning with (only) the PAF meta-heuristic are shown in Fig. 3(c). As is clearly apparent, the PAF meta-heuristic is highly effective: even in the most difficult case ($n = m + 2 = 10$), 98% schedulable utilization of is reached. All other curves already approach 99% schedulable utilization (or better). And should this not suffice, then the final "gap" can be closed with the RP meta-heuristic: *if both meta-heuristics are enabled, then schedulable utilizations in excess of 99% are reached for all task-set sizes.* No graph is shown for the RP meta-heuristic since all curves overlap, the results being limited by the experiment's sampling resolution.[1]

Varying m. Importantly, these results are not at all specific to $m = 8$ processors. Fig. 4 shows the schedulability achieved with partitioning heuristics, semi-partitioning heuristics, semi-partitioning with only the PAF meta-heuristic, and semi-partitioning with both meta-heuristics for $m = 4$, $m = 16$, and $m = 24$, and $n = 2m$. First, note that, in all three insets, the top-most curve corresponding to the use of both meta-heuristics reaches *almost* 100% schedulable utilization, and well in excess of 99% schedulable utilization, as just discussed. Second, as can be inferred from the similar shapes of the curves in all insets, the reported trends do not strongly depend on the absolute value of m, but rather are a function of relative utilization. Curiously, partitioning and semi-partitioning become slightly *more* effective with increasing core counts.

Task-set composition. In addition to the experiments reported here, we further obtained schedulability results with a different, independently developed task generator used in prior LITMUS^{RT} studies (*e.g.*, see [16, 18]). Due to space constraints, we omit a detailed discussion, but note that the additional results fully confirm the trends reported herein. This shows that the proposed strategies are not tied to Emberson *et al.*'s generator [28].

We conclude: *under the same, highly idealized assumptions used to establish formal claims of optimality, the proposed semi-partitioned reservation scheme (with meta-heuristics) achieves 99% schedulable utilization—empirically, it is near-optimal.*

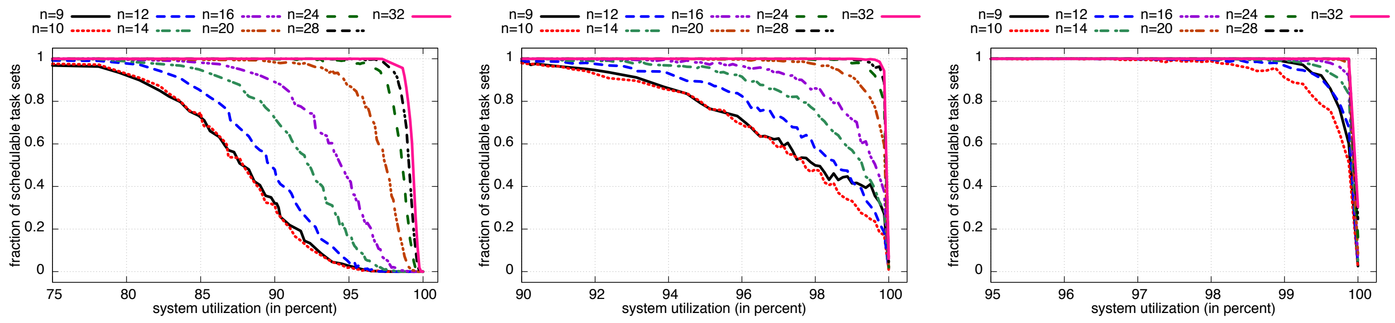
VI. Towards Practice

Clearly, many of the assumptions in §II are not realistic. Of particular concern is the stipulated absence of scheduling overheads, as it potentially allows to schedule tasks in infinitesimally small allocations, with arbitrarily high context-switch rates, until the solution approaches a fluid schedule. We therefore next take a closer look at allocation sizes and context-switch rates.

Allocation granularity. While we do enforce a (somewhat arbitrarily chosen) minimum period of $4ms$ in the RP meta-heuristic, which prevents infinitesimal budgets and infinite context-switch rates, the basic C=D splitting rule can still produce implausibly small budgets: if a processor is *almost* fully packed, the C=D splitting rule could determine a minuscule subtask budget x, below the limits of what can be feasibly enforced (in software) on contemporary commodity platforms.

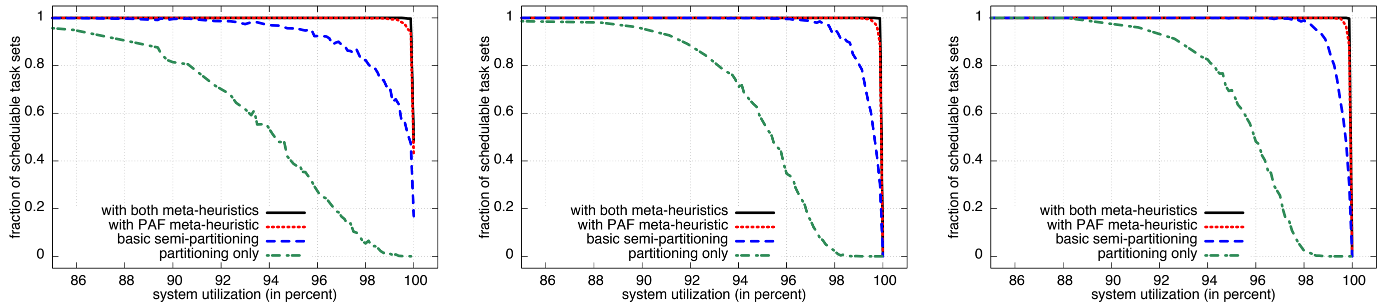
We therefore augmented the C=D splitting rule with a *minimum slice size* parameter ϵ, with the interpretation that a split is accepted only if both resulting subtasks have a budget of

[1]The complete set of results (including all graphs and detailed comparisons of all individual heuristics), all relevant source code, and detailed instructions for reproducing our results are provided online [1].



(a) partitioning only (b) basic semi-partitioning (c) semi-partitioning w/ the PAF meta-heuristic

Fig. 3. Schedulability results for $m = 8$ and $n \in \{m+1, m+2, m+4, m+6, 2m, 2.5m, 3m, 3.5m, 4m\}$. Note the different X-axis ranges.



(a) $m = 4$ (b) $m = 16$ (c) $m = 24$

Fig. 4. Schedulability results for $m \in \{4, 16, 24\}$ and $n = 2m$. The general trends are largely independent of m.

at least ϵ time units. This method eliminates implausibly small budgets, but how does it affect schedulability?

To find out, we reran a subset of the experiments with eight different ϵ values. A representative excerpt for $m = 8$ and $n \in \{10, 16, 24\}$ is shown in Fig. 5. As can be seen in Fig. 5(a) for the most difficult to support workload ($n = m + 2 = 10$), enforcing a non-zero ϵ does have a measurable, but overall minor effect on schedulability. In Fig. 5(a), even with a large threshold of $\epsilon = 500\mu s$, a schedulable utilization of 98% is achieved. A curve for $\epsilon = 2000\mu s$ is also shown, however, this is a much larger threshold than realistically required and is included only for illustration purposes. Insets Fig. 5(b) and Fig. 5(c) show that the impact of ϵ diminishes with larger n as the scheduling problem becomes easier. In Fig. 5(c), where $n = 3m$, even a choice of $\epsilon = 2000\mu s$ still yields almost 99% schedulable utilization. Pragmatically speaking, this is likely more than "good enough."

Context-switch rates. Finally, in a third experiment, we determined for each task set a simple upper bound on the maximum context-switch rate per second, normalized per core. An excerpt of these experiments for $m = 8$ and $n \in \{10, 16, 24\}$ is shown in Fig. 6. Each inset shows five curves, corresponding to partitioning only, basic semi-partitioning, semi-partitioning with all meta-heuristics, and the optimal RUN and QPS algorithms.

Under our approach, each task and subtask causes two context switches per activation: one to switch to the (sub-)job, and one to switch away. A similar bound can be inferred for QPS. A bound for RUN is provided by Regnier *et al.* [44]. Other optimal multiprocessor schedulers are known to preempt and migrate jobs (much) more frequently than RUN or QPS [39, 42, 44].

Consider Fig. 6(a), where $n = m + 2 = 10$, which is one of the most challenging considered workloads. For low utilizations, all curves but the one corresponding to RUN coincide because QPS and our approach reduce to conventional partitioning for workloads that are easy to schedule. The curve corresponding to partitioning first becomes "noisy" and then stops early due to a lack of samples at high utilizations. RUN exhibits a higher maximum context-switch rate at low utilizations, but has the advantage of remaining relatively stable even at extremely high utilizations when the rates under the other schedulers begin to rise. The curves corresponding to basic semi-partitioning and semi-partitioning with meta-heuristics start to rise slowly around 90%, but for the most part remain below the QPS curve, with the exception of the meta-heuristics curve at the very end. This, however, is an outlier due to the small number of tasks. As Figs. 6(b) and 6(c) demonstrate, maximum context-switch rates typically remain far below QPS, with a growing gap as n increases. It is thus *not* the case that the task-splitting or period-transformation techniques induce intolerable context-switch rates. Further note that these are *static* upper bounds on *maximum* context-switch rates that do not yet reflect any positive effects from the efforts to avoid migrations (§IV-A).

VII. Semi-Partitioned Reservations in LITMUS^{RT}

To assess the viability of the proposed approach in a real OS, we developed and evaluated semi-partitioned reservations in LITMUS^{RT} [2, 16, 21]. We focused on three questions:

1) how much additional kernel overhead is incurred compared to a conventional partitioned scheduler (§VII-A),
2) how much lower are kernel overheads compared to the optimal schedulers RUN and QPS (§VII-B), and
3) are the proposed runtime tweaks (§IV-A) effective at lowering migration rates (§VII-C)?

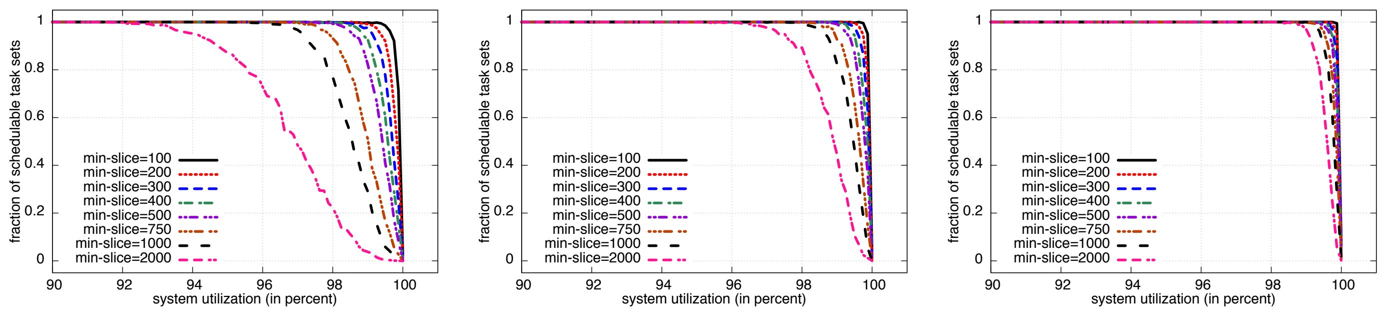


Fig. 5. Schedulability results for $m = 8$, $n \in \{m + 2, 2m, 3m\}$, and minimum slice-size thresholds in the range $\epsilon \in [100\mu s, 2000\mu s]$.

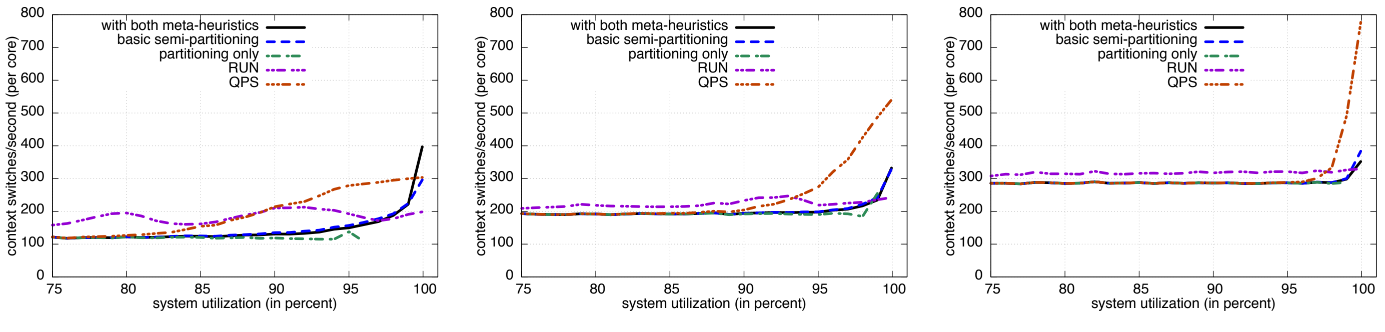


Fig. 6. Bounds on context-switch rates under the optimal QPS scheduler and three task placement strategies for $m = 8$ and $n \in \{m + 2, 2m, 3m\}$.

Implementation. Due to space constraints, we omit a discussion of implementation details. The basic (*i.e.*, partitioning-only) reservation framework, which has its origins in an earlier prototype from 2014 [17], is already part of the latest LITMUS$^{\text{RT}}$ release (version 2016.1). We plan to include our semi-partitioning extensions, which are available online [1], in a future release.

Platform. We ran experiments on a two-socket Intel platform consisting of two 22-core Xeon E5-2699 v4 processors clocked at 2.2 GHz with private 256-KiB per-core L2 caches and shared 55-MiB per-socket L3 caches. Though untypical for (embedded) real-time systems, such a large platform has the advantage that scalability bottlenecks become obvious. In terms of overheads, smaller platforms are easier to support.

A. Comparison with Stock LITMUS^{RT} Schedulers

To obtain a flexible benchmark, we generated ten task sets for each combination of $m = 44$, $n \in \{\, m+1, 1.5m, 2m, 4m, 6m, 8m, 10m\}$, and $U \in \{75\%, 80\%, \ldots, 95\%\}$ as described in §V, which yielded 350 task sets in total. The generated workloads simulated CPU-bound, periodic tasks based on the generated task parameters. Each task had a (private) working set matching the L2 cache size. Additionally, a cache-polluting background process was run on each core to generate memory contention.

We ran each task set under four different schedulers for 30 seconds each: with the proposed semi-partitioned reservations (SP-RES), and under LITMUS$^{\text{RT}}$'s stock *global* EDF (G-EDF), *partitioned* EDF (P-EDF), and *partitioned fixed-priority* (P-FP) plugins (the latter with rate-monotonic priorities). We included the three conventional process schedulers to provide a known, well-established baseline. In particular, P-FP and P-EDF are known to incur the lowest overheads in LITMUS$^{\text{RT}}$ [16].

To run task sets that could not be partitioned under P-EDF or P-FP, we simply truncated migrating tasks to the first subtask. While running each workload, we collected overhead samples with the *Feather-Trace* framework built into LITMUS$^{\text{RT}}$. In total, we collected more than 6.6 billion overhead samples (122 GiB) across more than eleven hours of real-time execution.

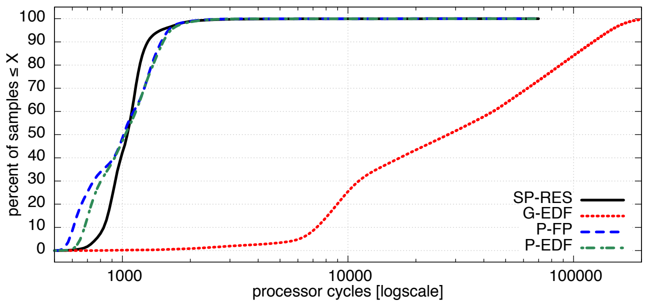
Common overheads. Fig. 7 shows the distributions of three key kernel overheads. Note the log scale in Figs. 7(a) and 7(c).

Scheduling overhead, shown in Fig. 7(a), describes the cost of determining which process to dispatch next; it corresponds to the core scheduling logic plus any synchronization overhead. It is immediately apparent that the scheduling overhead distributions under P-EDF, P-FP, and SP-RES are similar, in contrast to the much larger overheads under G-EDF, which reflect scalability bottlenecks in LITMUS$^{\text{RT}}$'s stock G-EDF scheduler [22]. For example, the observed 99th percentile scheduling overhead is only 2,092 cycles under SP-RES, 2,059 cycles under P-FP, and 2,150 cycles under P-EDF,[2] but 181,934 cycles under G-EDF.[3]

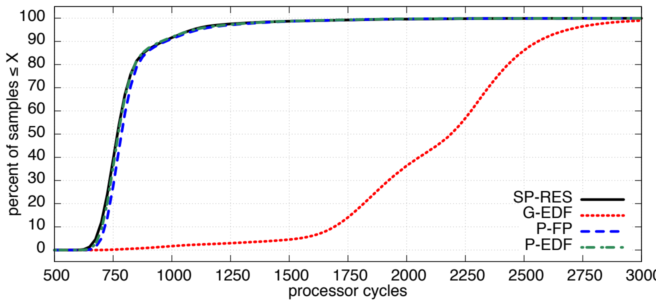
The similarity to partitioned schedulers is even more pronounced in the case of *context-switch* overhead, shown in Fig. 7(b), which describes the cost of actually dispatching a process (*e.g.*, switching address spaces, prefetching the process control block, *etc.*). The distributions under the two partitioned schedulers and SP-RES are virtually identical (the curves overlap almost completely), whereas context-switches are more costly under G-EDF (at the 99th percentile, 2,584 cycles under G-EDF vs. about 990 cycles under the other schedulers). Since

[2]The long tails of the distributions reflect outliers due to unpredictability in the memory hierarchy, which is unavoidable on our commodity platform.

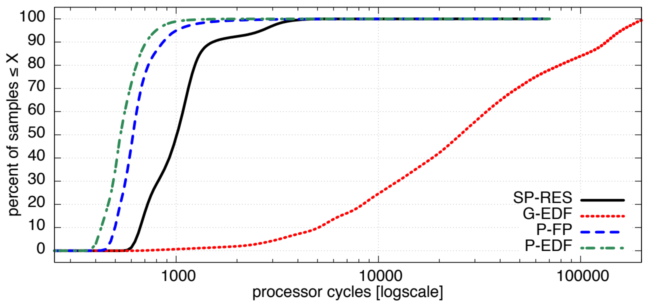
[3]As these measurements are platform- and benchmark-dependent, primarily their relative magnitudes are of interest, not their absolute values.



(a) scheduling overhead



(b) context-switch overhead



(c) release overhead

Fig. 7. Common overheads under three process-based schedulers (G-EDF, P-EDF, P-FP) and the semi-partitioned reservation scheduler (SP-RES). On the experimental platform, $1\mu s$ corresponds to approximately 2200 cycles.

the context-switch path is identical under all schedulers, this difference reflects increased memory contention under G-EDF.

The biggest difference between P-EDF, P-FP, and SP-RES is apparent in Fig. 7(c), which shows *release overhead* (*i.e.*, the cost of activating a task). Here SP-RES incurs somewhat higher overhead than either P-EDF or P-FP because it is optimized for sporadic task activations, whereas P-EDF and P-FP are optimized for periodic workloads (such as the one used in this benchmark). More precisely, release overhead under SP-RES also includes the cost of Linux's wake-up path because tasks suspend between jobs (via $schedule\_hrtimeout()$ in the kernel), whereas periodic tasks remain in Linux's $TASK\_RUNNING$ state under P-EDF and P-FP even while they await their next job release in the release queue. This optimization is not possible for sporadic tasks; for simplicity, our SP-RES implementation omits the special-case handling of periodic tasks.

Release overhead is lowest under P-EDF, which uses binomial heaps to merge newly released jobs of periodic tasks efficiently into the ready queue. Our SP-RES implementation uses a simple linked list instead. Nonetheless, the shape of the curve reveals again that SP-RES is structurally similar to the partitioned schedulers P-EDF and P-FP.

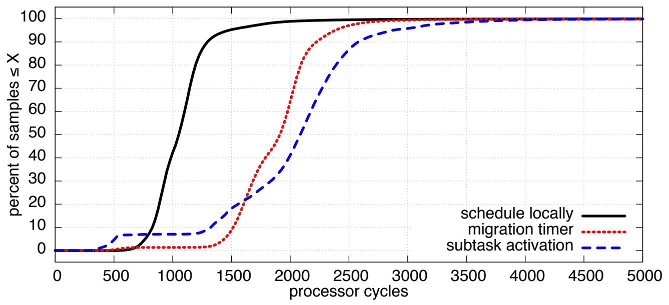


Fig. 8. Scheduling and subtask overheads under SP-RES.

Extra overheads. In addition to the common overheads incurred by all schedulers, SP-RES also incurs new overheads when migrating semi-partitioned reservations among processors. In particular, it incurs the cost of servicing a *migration timer* on the source processor when a migration must be initiated, and the cost of servicing an inter-processor interrupt to *activate the next subtask* on the target processor. The distributions of these two overheads, together with the scheduling overhead from Fig. 7(a) as a point of reference, are depicted in Fig. 8.

While these overheads are more costly than a regular local scheduling decision due to the need to synchronize data structures shared between cores, they are still relatively small in absolute terms (*e.g.*, at the 99th percentile, less than 3,750 cycles $\approx 1.7\mu s$ on our platform). Roughly speaking, we observe a cross-processor subtask activation to cause twice as much overhead as a regular, processor-local job release, which is still two orders of magnitude lower than release overhead under G-EDF.

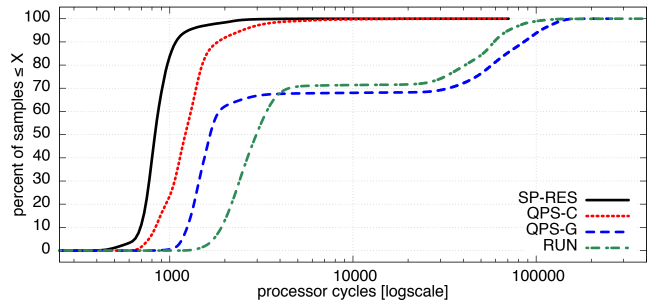
Overall, the comparison with LITMUS^{RT}'s stock plugins confirms that the proposed semi-partitioned reservations approach not only achieves high schedulability in theory, but also very low runtime overheads in practice, which are generally in line with those incurred under a conventional partitioned scheduler.

B. Comparison with RUN and QPS

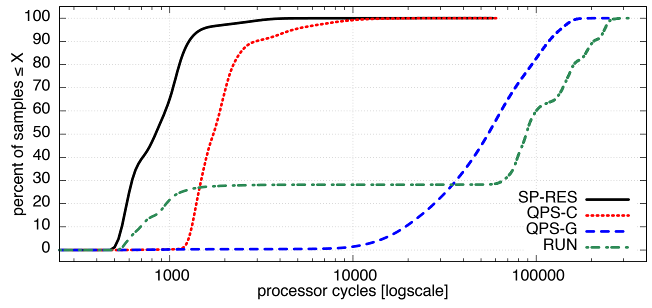
In a second, smaller-scale experiment, we conducted a direct comparison with prior implementations of RUN and QPS in LITMUS^{RT}, which were kindly provided by Compagnin *et al.* [25, 26]. To ensure a common base for comparison, we ported their code to the latest version of LITMUS^{RT} (2016.1). Notably, Compagnin *et al.*'s implementation of QPS supports two configurations: one using a single global spin lock, denoted QPS-G, and one using "per-cluster" spin locks, denoted QPS-C. On our platform, QPS-C actually uses one spin lock per core. The RUN implementation always uses a single global lock.

Due to implementation limitations, we could not execute the same task sets as in the first experiment.[4] We hence instead generated 60 high-utilization task sets using Compagnin *et al.*'s experimental setup [25, 26]. We executed each task set under RUN, QPS-G, QPS-C, and SP-RES for 30 seconds each while

[4]The employed RUN and QPS implementations [25, 26] support only tasks with integral millisecond parameters, whereas our workloads require (at least) microsecond granularity. This is purely an implementation artifact; in the underlying RUN [44] and QPS [39] algorithms, this limitation is not present.



(a) scheduling overhead



(b) release overhead

Fig. 9. Overheads under two process-based optimal schedulers (RUN, QPS) and the semi-partitioned reservation scheduler (SP-RES). For QPS, results for two different configurations are shown: one using a single global lock (QPS-G) and one using "per-cluster" locks (QPS-C). RUN uses a single global lock. The implementations of RUN and QPS are due to Compagnin *et al.* [25, 26].

collecting overheads with Feather-Trace. In total, we collected more than 100 million overhead samples (4.6 GiB).

Due to space constraints, we focus on scheduling and release overheads. In Fig. 9(a), which shows scheduling overhead, it is clearly apparent that RUN and QPS-G suffer from extreme overheads, which are likely due to lock contention, as in the case of G-EDF in Fig. 7(a). QPS-C performs much better, but still incurs noticeably higher overhead than SP-RES. For example, the 99th percentile scheduling overhead under SP-RES is 2,255 cycles, whereas it is more than twice as much under QPS-C (4,993 cycles), and two orders of magnitude higher under QPS-G and RUN (135,994 and 101,294 cycles, respectively).

Fig. 9(b) shows the distributions of release overhead. While the trends are generally similar, the gap between SP-RES and QPS-C is even wider (at the 99th percentile, 3,045 vs. 9,571 cycles, respectively), and RUN incurs even higher overhead than QPS-G (158,397 cycles under QPS-G vs. 253,049 cycles under RUN), which reflects RUN's costly bookkeeping [25, 44].

In conclusion, we observe that our proposal exhibits (much) lower overheads than recent implementations of optimal schedulers, in particular compared to those using global locks.

C. Migration Rates

In our final experiment, we tested whether the dynamic migration avoidance tweaks discussed in §IV-A do have a measurable impact, provided there is some slack that can be reclaimed. To this end, we reran all task sets from the first experiment that contained at least one semi-partitioned reservation while varying a configuration option that causes the simulated real-time tasks to randomly under-run their WCET by, in expectation, the configured amount. We recorded the resulting schedules using LITMUS^{RT}'s built-in *sched-trace* functionality and then counted the number of observed migrations.

TABLE I
OBSERVED TASK MIGRATIONS GIVEN VARYING AMOUNTS OF SLACK

average slack	total migrations	per sec. and core	reduction
0%	412,637	8.23	
10%	346,511	6.91	1.2x
25%	224,713	4.48	1.8x
33%	165,660	3.30	2.5x
50%	82,406	1.64	5.0x

The results are summarized in Table I, which lists the total number of observed migrations (across all task sets), the same data normalized per second and core, and the improvement relative to the zero-slack baseline. The rate of migration drops significantly as the available slack increases due to more frequent and larger under-runs. With just a 25% difference between provisioned WCETs and *average-case execution times* (ACETs), the migration rate is almost halved (8.23 to 4.48), and with 50% slack available on average, a five-fold improvement is observed (8.23 to 1.64). *These results confirm that the proposed tweaks are effective at lowering migration rates if WCETs exceed ACETs.*

VIII. Limitations, Extensions, and Related Work

In the following, we discuss how to lift some of the limitations imposed in §II, speculate about possible future extensions, and highlight some related works and alternatives.

Identical processors. To match prior work, we have assumed identical processors. However, none of the employed heuristics (§IV-B) actually exploit this assumption. Support for uniform or heterogeneous platforms can thus be easily added. Furthermore, this assumption affects only the task-placement phase, as partitioned and semi-partitioned scheduling is oblivious to non-identical platforms at runtime (in contrast to global schedulers, which make placement decisions online). Optimal multiprocessor real-time scheduling on uniform multiprocessors is possible [33], but it is conceptually and implementation-wise no less complicated than on identical multiprocessors.

Implicit deadlines. Since non-clairvoyant optimal multiprocessor schedulers do not support constrained deadlines [30], we have restricted the scope to implicit deadlines. However, our SP-PRES implementation already naturally supports arbitrary (and hence constrained) deadlines (*e.g.*, this is required to realize constrained-deadline subtasks). Introducing constrained deadlines hence affects only the placement phase.

With the exception of the RP meta-heuristic, all proposed (meta-)heuristics are compatible with constrained deadlines. Period transformation is not generally possible for constrained-deadline tasks. However, recall from §V and Fig. 4 that the PAF meta-heuristic, which *is* compatible with constrained deadlines, is responsible for large parts of the observed performance.

A significant practical advantage is that constrained deadlines can be introduced without *any* changes to the runtime system (and without pessimistically truncating periods to deadlines). In contrast, any attempt to add (non-optimal) runtime support for constrained deadlines to optimal schedulers such as QPS, RUN, or PD^2 must overcome non-obvious integration challenges.

Independent tasks. We have not considered task synchronization. In reality, some tasks are most likely going to have mutual-exclusion or precedence constraints. Analytically, precedence constraints can be supported with known uniprocessor techniques (*i.e.*, by accounting for precedence-induced release jitter).

Mutual exclusion requires an appropriate locking protocol. Semi-partitioned reservations as proposed in this paper are conceptually compatible with the MBWI protocol [29]; kernel-level support in LITMUS$^{\mathrm{RT}}$ could be added in future work. Support for spin locks is another option worth exploring [14].

An interesting and largely unexplored research direction is the challenge of finding effective semi-partitioning heuristics that take resource-sharing and precedent constraints into account.

Self suspensions. We have not discussed self-suspensions (*e.g.*, jobs waiting for I/O devices), as prior work on optimal scheduling ignores them, too. Our implementation in LITMUS$^{\mathrm{RT}}$, however, must deal with self-suspensions to function properly.

Fortunately, self-suspensions are simple to deal with since we already employ slack reclamation: jobs in polling reservations generate slack when they suspend, and they regain whatever of their slack is left (if any) when they resume. In fact, from the point of view of the runtime system, a (final) job completion is no different from a (temporary) job suspension: when the job of a periodic task completes, the corresponding process simply suspends until the release of the next job, which is why process wake-up cost factors into SP-RES release overhead in Fig. 7(c).

Furthermore, our implementation already supports *deferrable servers* [49], a reservation type that is ideal for tasks that exhibit long self-suspensions since a deferrable server's budget is independent of suspension length (*i.e.*, there is no need to over-provision the budget). It is easy to integrate *partitioned* deferrable servers into the presented approach (*i.e.*, to account for them when placing or splitting tasks). In future work, it will be interesting to study semi-partitioned deferrable servers.

Cache and bus interference. We have ignored any interference effects due to caches and other shared parts of the memory hierarchy, as such concerns are orthogonal to the main observations of this paper. Prior work on optimal scheduling also ignores these issues. However, integrating existing interference-management techniques (*e.g.*, [34, 36, 38, 50]), which typically assume partitioned scheduling, is likely much easier with a simple approach such as ours, rather than with the quite intricate scheduling and migration rules in optimal schedulers.

Additionally, Sarkar *et al.* [45, 46] and Shekhar *et al.* [47] have developed architectural support for *proactively* migrating (locked) cache lines among cores. Under semi-partitioned scheduling, such an OS-initiated *push migration* of cache lines [45, 46] can ensure that a subtask starts execution with a hot cache [47], which reduces the cost of task migrations [45–47]. Notably, predicting where and when a cache line will be needed next is trivial under the adopted C=D scheme. Under most optimal schedulers, this is considerably more difficult (with QPS [39] being a notable exception).

Power management. We have not considered any resources beside processor time. In practice, managing power and/or energy consumption (*e.g.*, by means of voltage scaling or deep processor sleep states) is often an important requirement in embedded systems. Again, we note that a simple approach such as ours is much easier to extend when it comes to integrating predictable power management. Case in point, most work to date on energy-aware multiprocessor real-time scheduling—Bambagini *et al.* [9] provide a recent survey—actually targets partitioned scheduling [9]. Further, one can simply adopt any of the many known uniprocessor solutions [9] on a per-core basis.

That said, there have been proposals for energy-aware scheduling based on optimal scheduling [23, 31, 41]. However, these (arguably quite involved) proposals have not been implemented or evaluated in real systems. Intuitively speaking, global scheduling may be beneficial for race-to-idle strategies due to its work-conserving nature, especially if some cores cannot sleep (or slow down) while others remain busy (at faster speeds).

Dynamic workloads. It's important to acknowledge a key advantage of global scheduling: as there is no explicit task-placement phase, it is much easier to deal with dynamic workloads [15]. That is, when tasks join or leave the system at runtime, or adaptively adjust their parameters, *explicit* load-balancing is required under semi-partitioned scheduling. In contrast, global schedulers can react much more gracefully [15] as they load-balance implicitly. (However, RUN and QPS forgo this advantage, as they also rely on substantial offline phases.)

For similar reasons, some RTOSs designed for versatility default to global scheduling (*e.g.*, Linux and QNX), as it frees users who are not interested in precise timing guarantees from having to reason about task placement. Work-conserving global schedulers can also offer average-case benefits [6, 27], such as lower mean response times and faster recovery from intermittent overload [27]. However, in context of the classic hard real-time correctness criterion—all deadlines must be met, no more and no less—considered here and in prior work on optimal hard real-time scheduling, such average-case considerations are of lesser relevance. Nonetheless, in future work, it will be interesting to explore how to best handle dynamic and adaptive workloads under semi-partitioned scheduling.

Malleable software. As a final remark, we note that this study makes a simplifying assumption regarding task malleability. Our experimental setup assumes—as virtually all prior comparative evaluations of global and (semi-)partitioned scheduling—that tasks are *atomic*, so that they must be placed as a whole.

In settings where it is possible to modify a system's implementation, this assumption may be too restrictive. Specifically, since most software is malleable to some degree, if no *task* partitioning can found, it may still be possible to partition an application at the level of *functionalities*. That is, if (semi-)partitioning fails, it may still be possible to refactor tasks at the code level to move specific functionalities to another core (*e.g.*, in AUTOSAR, by remapping "runnables"). In this view, our study could be seen as biased *against* partitioned and semi-partitioned scheduling.

IX. Conclusion

Near-optimal multiprocessor real-time scheduling does not require sophisticated scheduling approaches. Empirically speaking, a schedulable utilization in excess of 99%, which is arguably "good enough" for practical purposes, can be achieved (§V and §VI) with a simple combination of reservation-based scheduling, semi-partitioning, period transformation, and appropriate task-placement strategies (§IV). Such a simple approach is eminently practical, incurs only low overheads (§VII), and lends itself to extension to deal with other practical concerns (§VIII).

We draw two high-level conclusions from these observations:

- For static workloads consisting of independent, implicit-deadline sporadic (or periodic) tasks, there is—from a practical point of view—no compelling reason to favor global schedulers over (much) simpler semi-partitioned alternatives, such as the one presented herein.
- Future work on multiprocessor real-time systems should move beyond such simple workloads, which are adequately served by the state of the art, and instead target more realistic models and challenges, including precedence constraints, self-suspensions, constrained deadlines, dynamic and adaptive workloads, energy and power constraints, *etc.*

In summary, while there might be situations in which global scheduling is still preferable in practice, to actually demonstrate this, one needs to look beyond the simple workloads primarily considered in the literature on global scheduling to date. As discussed in §VIII, we believe that the proposed simple approach can also be adapted to support many more-demanding workloads. This conjecture provides ample opportunity for future work.

References

[1] Companion page: https://www.mpi-sws.org/~bbb/papers/details/rtss16/.
[2] The LITMUS^{RT} project: http://www.litmus-rt.org.
[3] The SchedCAT project: http://www.mpi-sws.org/~bbb/projects/schedcat.
[4] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS '98*.
[5] J. Anderson, V. Bud, and U. C. Devi, "An EDF-based scheduling algorithm for multiprocessor soft real-time systems," in *ECRTS'05*.
[6] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition," in *RTCSA'00*.
[7] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *ECRTS'01*.
[8] B. Andersson and K. Bletsas, "Sporadic multiprocessor scheduling with few preemptions," in *ECRTS'08*.
[9] M. Bambagini, M. Marinoni, H. Aydin, and G. Buttazzo, "Energy-aware scheduling for real-time systems: A survey," *ACM TECS*, vol. 15, no. 1, 2016.
[10] S. Baruah, A. Mok, and L. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *RTSS'90*.
[11] S. Baruah, N. Cohen, C. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, 1996.
[12] A. Bastoni, B. Brandenburg, and J. Anderson, "An empirical comparison of global, partitioned, and clustered multiprocessor EDF schedulers," in *RTSS'10*.
[13] ——, "Is semi-partitioned scheduling practical?" in *ECRTS'11*.
[14] A. Biondi, G. Buttazzo, and M. Bertogna, "Supporting component-based development in partitioned multiprocessor real-time systems," in *ECRTS'15*.
[15] A. Block, J. Anderson, and G. Bishop, "Fine-grained task reweighting on multiprocessors," *Journal of Embedded Computing*, vol. 4, no. 2, 2010.
[16] B. Brandenburg, "Scheduling and locking in multiprocessor real-time operating systems," Ph.D. dissertation, UNC Chapel Hill, 2011.
[17] ——, "A synchronous IPC protocol for predictable access to shared resources in mixed-criticality systems," in *RTSS'14*.
[18] B. Brandenburg, J. Calandrino, and J. Anderson, "On the scalability of real-time scheduling algorithms on multicore platforms," in *RTSS'08*.
[19] 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, pp. 3–33, 2012.
[20] M. Caccamo, G. Buttazzo, and L. Sha, "Capacity sharing for overrun control," in *RTSS'00*.
[21] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson, "LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers," in *RTSS'06*.
[22] F. Cerqueira, M. Vanga, and B. Brandenburg, "Scaling global scheduling with message passing," in *RTAS'14*.
[23] H. Chishiro, M. Takasu, R. Ueda, and N. Yamasaki, "Optimal multiprocessor real-time scheduling based on RUN with voltage and frequency scaling," in *ISORC'15*.
[24] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in *RTSS'06*.
[25] D. Compagnin, E. Mezzetti, and T. Vardanega, "Putting RUN into practice: implementation and evaluation," in *ECRTS'14*.
[26] ——, "Experimental evaluation of optimal schedulers based on partitioned proportionate fairness," in *ECRTS'15*.
[27] R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 43, no. 4, 2011.
[28] P. Emberson, R. Stafford, and R. Davis, "Techniques for the synthesis of multiprocessor tasksets," *WATERS'10*.
[29] D. Faggioli, G. Lipari, and T. Cucinotta, "The multiprocessor bandwidth inheritance protocol," in *ECRTS'10*.
[30] N. Fisher, J. Goossens, and S. Baruah, "Optimal online multiprocessor scheduling of sporadic real-time tasks is impossible," *Real-Time Systems*, vol. 45, no. 1-2, pp. 26–71, 2010.
[31] S. Funk, V. Berten, C. Ho, and J. Goossens, "A global optimal scheduling algorithm for multiprocessor low-power platforms," in *RTNS'12*.
[32] 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.
[33] S. H. Funk and A. Meka, "U-LLREF: An optimal scheduling algorithm for uniform multiprocessors," in *The 9th Workshop on Models and Algorithms for Planning and Scheduling Problems*, 2009.
[34] Y. Heechul, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms," in *RTAS'13*.
[35] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *ECRTS'09*.
[36] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *ECRTS'13*.
[37] J. Lelli, D. Faggioli, T. Cucinotta, and G. Lipari, "An experimental comparison of different real-time schedulers on multicore systems," *J. Syst. Softw.*, vol. 85, no. 10, pp. 2405–2416, 2012.
[38] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, "Real-time cache management framework for multi-core architectures," in *RTAS'13*.
[39] E. Massa, G. Lima, P. Regnier, G. Levin, and S. Brandt, "Quasi-partitioned scheduling: optimality and adaptation in multiprocessor real-time systems," *Real-Time Systems*, vol. 52, no. 5, pp. 566–597, 2016.
[40] C. Mercer, S. Savage, and H. Tokuda, "Processor capacity reserves: an abstraction for managing processor usage," in *Proc. Fourth Workshop on Workstation Operating Systems*, 1993.
[41] G. A. Moreno and D. De Niz, "An optimal real-time voltage and frequency scaling for uniform multiprocessors," in *RTCSA'12*. IEEE.
[42] G. Nelissen, V. Berten, V. Nelis, J. Goossens, and D. Milojevic, "U-EDF: An unfair but optimal multiprocessor scheduling algorithm for sporadic tasks," in *ECRTS'12*.
[43] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: a resource-centric approach to real-time systems," in *CMCN'98*.
[44] 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.
[45] A. Sarkar, F. Mueller, and H. Ramaprasad, "Predictable task migration for locked caches in multi-core systems," in *LCTES'11*.
[46] A. Sarkar, F. Mueller, H. Ramaprasad, and S. Mohan, "Push-assisted migration of real-time tasks in multi-core processors," in *LCTES'09*.
[47] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, "Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems," in *ECRTS'12*.
[48] A. Srinivasan and J. Anderson, "Optimal rate-based scheduling on multiprocessors," in *STOC'02*.
[49] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *Trans. Comp.*, vol. 44, no. 1, pp. 73–91, 1995.
[50] B. Ward, J. Herman, C. Kenna, and J. Anderson, "Making shared caches more predictable on multicore platforms," in *ECRTS'13*.
[51] D. Zhu, X. Qi, D. Mossé, and R. Melhem, "An optimal boundary fair scheduling algorithm for multiprocessor real-time systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 10, pp. 1411–1425, 2011.