Maia *et al. EURASIP Journal on Embedded Systems* (2017) 2017:31
DOI 10.1186/s13639-017-0079-5

EURASIP Journal on Embedded Systems

RESEARCH Open Access

Real-time semi-partitioned scheduling of fork-join tasks using work-stealing

Cláudio Maia*, Patrick Meumeu Yomsi, Luís Nogueira and Luis Miguel Pinho

Abstract

This paper extends the work presented in Maia et al. (Semi-partitioned scheduling of fork-join tasks using work-stealing, 2015) where we address the semi-partitioned scheduling of real-time fork-join tasks on multicore platforms. The proposed approach consists of two phases: an offline phase where we adopt a multi-frame task model to perform the task-to-core mapping so as to improve the schedulability and the performance of the system and an online phase where we use the work-stealing algorithm to exploit tasks' parallelism among cores with the aim of improving the system responsiveness. The objective of this work is twofold: (1) to provide an alternative scheduling technique that takes advantage of the semi-partitioned properties to accommodate fork-join tasks that cannot be scheduled in any pure partitioned environment and (2) to reduce the migration overheads which has been shown to be a traditional major source of non-determinism for global scheduling approaches. In this paper, we consider different allocation heuristics and we evaluate the behavior of two of them when they are integrated within our approach. The simulation results show an improvement up to 15% of the proposed heuristic over the state-of-the-art in terms of the average response time per task set.

Keywords: Parallel tasks, Semi-partitioned scheduling, Work-stealing, Multicore platforms

1 Introduction

Multicore platforms are now very common in the embedded systems domain as they provide more computing power for the execution of complex applications with stringent timing constraints. This boost in performance increases substantially the complexity of the scheduling problem of real-time tasks that execute upon these platforms. While a uniprocessor scheduling problem reduces to deciding *when* to schedule each task, a new dimension adds to this one when shifting to multicores: it must also be decided *where* to execute each task. In order to solve this rather challenging issue, several scheduling algorithms have been proposed in the literature (see [1] for a comprehensive and up-to-date survey).

Another important feature of these platforms is that they make intra-task parallelism possible by taking advantage of the task structure. At compile time, intra-task parallelism can be extracted from application loops by using programming frameworks such as OpenMP [2]. These frameworks resort to dynamic scheduling strategies in order to schedule application tasks. One of the most common strategies in use is work-stealing [3]. In summary, work-stealing is a load-balancing algorithm which allows an idle core to randomly steal some workload from a busy core, referred to as the "victim", with the objective of reducing the average response time of a task executing on a target platform[1]. While randomness in the selection of a victim is traditionally acceptable in several computing domains, no guarantee can actually be provided regarding the timing behavior of the tasks as there is a possibility of priority inversion among them. A solution to circumvent this limitation consists of using multiple per-core priority *double-ended queues* (known as deques[2]) [4].

In this paper, we consider fork-join real-time tasks (i.e., a special case of parallel real-time tasks) in a semi-partitioned scheduling context so that we can explore the potential parallelism of migrating tasks at runtime by resorting to the load balancing property provided by a variant of the work-stealing algorithm [4]. The goal is to reduce the average response time of the tasks and create additional room in the schedule for less-critical tasks (e.g., aperiodic and best-effort tasks).

*Correspondence: crrm@isep.ipp.pt
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

We recall that semi-partitioned scheduling [5–7] considers two steps: (step 1) a task-to-core mapping is performed at design time where a subset of tasks (the subset of *non-migrating tasks*) is assigned to specific cores and is not allowed to migrate at runtime; (step 2) if a task cannot be assigned to any of the cores without jeopardizing its schedulability, then this task is referred to as a *migrating task* and is scheduled by using a global scheduling approach to seek for a valid schedule.

In the proposed approach, the behavior of each migrating task is further restricted. At runtime, each job activation of a migrating task follows a job-to-core execution pattern elaborated at design time in order to improve both the schedulability of the system and its utilization factor. In addition, we consider a *task-level migration* strategy, i.e., various jobs of a migrating task are allowed to be assigned to different cores, but once a job is assigned to a core, migrations of this job prior to its completion are forbidden. In contrast, *job-level migration* approaches allow each job assigned to a core to migrate to another core prior to its completion. By design, the proposed model limits the number of migrations, which has been recognized as one of the main sources of non-determinism on multicores, by limiting work-stealing to occur between cores that share a copy of a task[3].

Contributions The contribution of this work is fourfold: (1) we present a complete framework that supports the scheduling of fork-join real-time tasks onto a multicore platform together with the associated schedulability analysis. (2) As we assume that cores that share jobs of a migrating task have a local copy of this task, we reduce both the overhead concerning task fetching and the number of task migrations due to the offline job-to-core mapping. (3) As the parallel regions of each fork-join task can execute simultaneously on different cores, we take advantage of the work-stealing mechanism to reduce the average response time of the tasks without jeopardizing the schedulability of the whole system. To the best of our knowledge, we are the first using work-stealing in the context of a semi-partitioned scheduling scheme. (4) We extend the work presented in [8] by comparing different allocation heuristics in terms of their allocation behavior. For two of these heuristics, we evaluate the improvement given by using work-stealing in terms of task average response times. Moreover, we explain how to integrate tasks with a density greater than one into our framework.

Paper organization The rest of this paper is organized as follows. Section 2 presents the related work. Section 3 describes the model of computation used throughout the paper. Section 4 details our proposed approach. Section 5 provides an example of how the framework works. Section 6 explains how decomposition-based models can be used to accommodate tasks with density greater than one. Section 7 presents the schedulability analysis of the proposed approach. Section 8 reports on simulation results from experiments on synthetic task sets. Finally, Section 9 concludes the paper.

2 Related work

Three task models supporting *intra-task parallelism* exist in the real-time systems domain: (1) the fork-join task model, (2) the synchronous task model, and (3) the directed acyclic graph (DAG) task model. From these models, the fork-join task model (see Fig. 1) is the simplest in terms of parallel structure. Specifically, the initial sequential sub-task may fork into several independent sub-tasks which can execute simultaneously in parallel. Upon completion, these parallel sub-tasks join into a sequential sub-task and this behavior may repeat again up until the completion of the task. This model is a special case of the synchronous task model. Indeed, in the fork-join task model as presented in [9], parallel segments

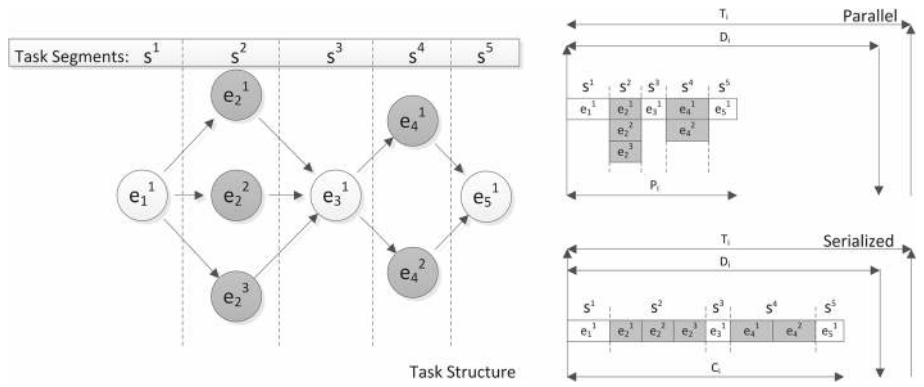

Fig. 1 Fork-join task. In this figure, it is possible to observe the task structure of a fork-join task (left side), its timing properties (upper right side), and its serialized representation (lower right side)

must have the same number of sub-tasks, with a restriction on the number of sub-tasks that each task can fork into (not greater than the number of cores on the platform). This restriction does not apply to the synchronous model [10] nor does it apply to the model proposed in this paper. The DAG task model [11] is the most general one. In this model, each sub-task is represented as a node[4] and an edge connecting two nodes represents a data/precedence dependency between the connected nodes.

Both decomposition-based techniques [9, 10, 12] and non-decomposition-based techniques [11, 13] have been proposed to analyze the schedulability of these three task models. Specifically, resource and capacity augmentation bounds can be used to evaluate the schedulability of all task models while response-time analysis [14, 15] can be used to analyze synchronous parallel tasks.

Unfortunately, very few techniques exist in the literature for the analysis of semi-partitioned scheduling of parallel tasks. Bado et al. [16] proposed a semi-partitioned approach with job-level migration for fork-join tasks, which is similar to the one in [9], but due to the assignment methods proposed in their paper for the offsets and local deadlines, they did not provide any guarantee on the fact that sub-tasks actually execute in parallel. While their work is similar to ours w.r.t. the adopted class of schedulers (semi-partitioned), we differ in that we relax the constraint of restricting the task parallelism and we use task-level migration instead of job-level migration, thus further reducing the number of migrations at runtime.

3 System model

Task specifications We consider a set $\tau \stackrel{\text{def}}{=} \{\tau_1, \ldots, \tau_n\}$ composed of n sporadic fork-join tasks. Each sporadic fork-join task $\tau_i \stackrel{\text{def}}{=} \langle S_i, D_i, T_i \rangle$, $1 \leq i \leq n$, is characterized by a finite *sequence of segments* $S_i \stackrel{\text{def}}{=} \left[s_i^1, s_i^2, \ldots, s_i^{n_i}\right]$, with $n_i \in \mathbb{N}$, a *relative deadline* D_i and a period T_i. These parameters are given with the following interpretation: at runtime, each task τ_i generates a potentially infinite number of successive jobs $\tau_{i,j}$. Each job $\tau_{i,j}$ has a finite sequence of segments S_i, arrives at time $a_{i,j}$ such that $a_{i,j+1} - a_{i,j} \geq T_i$, and must be completed within $[a_{i,j}, d_{i,j})$ where $d_{i,j} \stackrel{\text{def}}{=} a_{i,j} + D_i$ is its absolute deadline. Each segment $s_i^k \in S_i$ (with $1 \leq k \leq n_i$) is composed of a set of *independent sub-tasks*[5] $t_{s_i^k} \stackrel{\text{def}}{=} \left\{t_{s_i^k}^1, \ldots, t_{s_i^k}^{v_k}\right\}$, where v_k denotes the number of sub-tasks belonging to segment s_i^k, and the sequence S_i represents dependencies between segments. That is, for all $s_i^\ell, s_i^r \in S_i$ such that $\ell < r$, the sub-tasks belonging to s_i^r cannot start executing unless those of s_i^ℓ have completed. The execution requirement of sub-tasks $t_{s_i^k}^q$ (with $1 \leq q \leq v_k$) is denoted by $e_{s_i^k}^q$. The *total execution requirement* of task τ_i, denoted by C_i, is the *sum* of the execution requirements of all the sub-tasks in S_i, i.e., $C_i \stackrel{\text{def}}{=} \sum_{k=1}^{n_i} \sum_{q=1}^{v_k} e_{s_i^k}^q$. Every sub-task is assumed to execute on *at most* one core at any time instant and can be interrupted prior to its completion by another sub-task with a higher priority. A preempted sub-task is assumed to resume its execution on the same core as the one on which it was executing prior to preemption. We assume that each preemption is performed at no cost or penalty. The *minimum execution requirement* of task τ_i, denoted as P_i, is defined as the time that τ_i takes to execute when it is assigned to an infinite number of cores[6], *i.e.,* $P_i = \sum_{k=1}^{n_i} c_{s_i^k}$, where $c_{s_i^k}$ denotes the worst-case execution time among the sub-tasks of segment k. The *utilization factor* of τ_i is $U_i = \frac{C_i}{T_i}$ and its *density* is $\lambda_i = \frac{C_i}{\min(D_i, T_i)}$. The *total utilization factor* of τ is $U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_i$ and its *total density* is $\lambda_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} \lambda_i$. For each task τ_i, we assume $D_i \leq T_i$, which is commonly referred to as the constrained-deadline task model. The task set τ is said to be $\mathcal{A}$*-schedulable* if algorithm $\mathcal{A}$ can schedule τ such that all the jobs of every task $\tau_i \in \tau$ meet their deadline D_i.

The left side of Fig. 1 illustrates a fork-join task τ_i with $n_i = 5$ segments, three are sequential segments ($s^1, s^3$and s^5) with one sub-task each and two are parallel segments: s^2 containing three sub-tasks and s^4 containing two sub-tasks. All the sub-tasks in the parallel segments are independent from each other and therefore can execute in parallel. On the upper right side of the figure, it is possible to observe the task structure framed according to the timing properties of the task (P, D, T), and on the bottom right side, it is possible to observe the task's serialized representation (i.e., task execution without parallelism).

Each *migrating task* is modeled as a *multiframe task*. The multiframe task model (as presented by Mok and Chen [17] and later generalized by Baruah et al. [18]) allows system designers to model a task by using a *static* and finite list of execution requirements, corresponding to successive jobs (or frames as they are named in this model). Specifically, by repeating this list (possibly *ad infinitum*), a periodic sequence of execution requirements is generated such that the execution time of each frame is bounded from above by the corresponding value in the periodic sequence.

Platform and scheduler specifications We consider a platform $\pi \stackrel{\text{def}}{=} \{\pi_1, \pi_2, \ldots, \pi_m\}$ comprising m homogeneous cores, i.e., all the cores have the same computing capabilities and are interchangeable. Each core runs a fully preemptive Earliest Deadline First (EDF) scheduler. EDF scheduling policy dictates that the smaller the absolute

deadline of a job, the higher its priority. The schedulability of a task set scheduled by following the EDF scheduler upon a uniprocessor platform can be evaluated by using the Demand Bound Function (DBF) [19]. The DBF of task τ_i at any time instant $t \geq 0$ is defined as $\mathrm{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \left(\left\lfloor \frac{t-D_i}{T_i} \right\rfloor + 1 \right) \cdot C_i$ and the DBF of task set τ is derived as $\mathrm{DBF}(\tau, t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \mathrm{DBF}(\tau_i, t)$. The notations used throughout the paper are summarized in Appendix: Table 1.

We allow work-stealing only among the cores that execute a migrating task. Jobs of migrating tasks execute on selected cores according to an execution pattern that is determined offline. By allowing work-stealing only among these cores, a reduction of the average response-time of each migrating task is possible, thus contributing to the reduction of the overall system responsiveness.

Our framework assumes a shared-memory model with similar properties (multi-threaded, shared address space, etc.) than the parallel frameworks that integrate work-stealing (such as OpenMP).

4 Proposed approach

We propose a semi-partitioned model of execution with work-stealing for fork-join tasks. The proposed approach consists of three phases referred to as (*i*) *task assignment*, (*ii*) *offline scheduling*, and (*iii*) *online scheduling*.

4.1 Task assignment phase

In [8], a variant of the first-fit decreasing (FFD) heuristic, hereafter referred to as FFDO[7], was selected. FFDO first divides the tasks into two classes: (1) tasks with $\lambda_i \leq 0.5$ (light tasks) and (2) tasks with $\lambda_i > 0.5$ (heavy tasks)[8]. The next step is to apply the classical FFD to light sequential tasks first and then to heavy sequential tasks. After this step completes, FFDO selects the light parallel tasks and then the heavy parallel tasks, again using FFD as the packing heuristic. Intuitively, by assigning sequential tasks first followed by the parallel tasks, the probability of having parallel tasks unallocated after the first phase increases.

All the tasks successfully assigned to the cores are referred to as *non-migrating tasks*, and the remaining tasks, i.e., those that cannot be assigned by the heuristic to any core without jeopardizing its schedulability, are referred to as *candidate migrating tasks*.

At the end of the assignment phase, if all tasks are assigned to cores, then there is no candidate migrating task and therefore no migrating task in the system. In this case, there is no need for parallelization and work-stealing as a fully partitioned assignment of the tasks to the cores has been found. Using work-stealing in this situation would just help in load-balancing the execution workload at the cost of allowing for unnecessary migrations. Due to this observation, work-stealing is forbidden for non-migrating parallel tasks. In the other case, if a task cannot be assigned to any core without jeopardizing its schedulability, then this task is deemed as a candidate migrating task and is treated as a multiframe task. The system is deemed schedulable if and only if an execution pattern is found for each candidate migrating task such that all the timing requirements of the system are met.

The goal of this assignment behavior is to increase the possibility of benefiting from parallelism in the third phase of the approach as a way to reduce the response-time of the tasks. For instance, some parallel tasks may not fit into the cores in this first phase, and if this is the case, such tasks can be re-checked in the second phase of the approach by treating them as multiframe tasks. If an execution pattern is found for the multiframe task, then these tasks can benefit from work-stealing in the third phase.

4.2 Offline scheduling phase

After the task assignment phase, let τ^{π_j} denote the set of tasks assigned to core π_j (with $1 \leq j \leq m$). It follows that $\tau^{\pi_j} = \tau^{\pi_j}_{\mathrm{NM}} \cup \tau^{\pi_j}_{\mathrm{M}}$ where $\tau^{\pi_j}_{\mathrm{NM}}$ denotes the subset of non-migrating tasks and $\tau^{\pi_j}_{\mathrm{M}}$ denotes the subset of migrating tasks assigned to π_j.

We remind the reader that each core runs an EDF scheduler, so the schedulability of the non-migrating tasks on each core is guaranteed as long as its load is less than 1. Concerning the migrating tasks, their jobs are distributed among the cores by following an execution pattern that does not jeopardize the schedulability of each individual core. To compute this pattern, the number of frames of each migrating task is computed as follows.

Definition 1 (Number of frames (taken from [7])) *The number of frames k_i to consider for each migrating task τ_i is computed as:*

$$k_i \stackrel{\text{def}}{=} \frac{H}{T_i}, \text{ where } H \stackrel{\text{def}}{=} \mathrm{lcm}_{\tau_j \in \tau}\{T_j\} \tag{1}$$

In Eq. 1, $\mathrm{lcm}_{\tau_j \in \tau}\{T_j\}$ denotes the *least common multiple* of the periods of all the tasks in τ. Goossens et al. [20] proved that this number of frames per migrating task is conservative and safe.

Definition 2 (Execution pattern (taken from [7])) *The job-to-core assignment sequence σ of each migrating task τ_i is defined through k_i sub-sequences as $\sigma \stackrel{\text{def}}{=} (\sigma_1, \sigma_2, \ldots, \sigma_{k_i})$ where the sub-sequence σ_s (with $1 \leq s \leq k_i$) is given in turn by the m-tuple $\sigma_s = \left(\sigma_s^1, \ldots, \sigma_s^m\right)$. By following a uniform job-to-core assignment, the s^{th} job of task τ_i is assigned to core π_j if and only if:*

$$\sigma_s^j = \left\lceil \frac{s+1}{k_i} \cdot M[i,j] \right\rceil - \left\lceil \frac{s}{k_i} \cdot M[i,j] \right\rceil = 1 \tag{2}$$

In Eq. 2, $M[i,j]$ is a matrix of integers $M[1 \ldots n, 1 \ldots m]$ that tracks the current job-to-core assignment where $M[i,j] = x$ means that x jobs of task τ_i out of k_i will execute on core π_j ($1 \leq i \leq n$ and $1 \leq j \leq m$).

To the best of our knowledge, the uniform assignment given by Eq. 2 is the best result found in the literature for finding execution patterns for migrating tasks. An alternative approach is the generation of patterns via enumeration. Equation 2 is part of a set of algorithms that were proposed in [7] for the finding of patterns for multiframe tasks. The intuitive idea of these algorithms is to find the largest number of frames (jobs) that can be assigned to each core such that the migrating task is deemed schedulable. The result in [7] was integrated into our approach.

4.3 Online scheduling phase

This phase takes advantage of the multicore platform and the execution pattern of migrating parallel tasks in order to reduce their average response-time at runtime and consequently that of other tasks assigned to the intervening cores. This is achieved by allowing work-stealing to occur among cores that share a copy of a migrating task during the execution of their parallel regions[9]. Below, we recall the four necessary rules (R_1 to R_4) for an efficient usage of the work-stealing algorithm:

- R_1: At least one selected core must be idle when there are parallel sub-tasks awaiting for execution.
- R_2: Idle selected cores are allowed to steal sub-tasks from the deque of another selected core.
- R_3: When stealing workload, the idle core must always steal the highest priority parallel sub-task from the list of deques (as proposed in [4]) in order to avoid priority inversions (this situation occurs when the number of migrating tasks is greater than 1 and the tasks have different priorities).
- R_4: After selecting a parallel sub-task to steal, say from core A to core B, an admission test must be performed on core B to guarantee that its schedulability is not jeopardized by this additional workload.

We recall that we avoid the overhead of fetching the code of the task from the main memory as the code of the migrating task is already loaded on the selected cores after the execution of the first job in a selected core. Whenever a core performs a steal, data is fetched from the memory of another core, which is to a certain extent equivalent to a migration. However, only input data is fetched in this case. Moreover, the number of migrations is limited by the task-to-core mapping (performed offline), which forces a job to execute in the pre-assigned cores instead of migrating between an arbitrary number of cores as it would happen in a global approach.

5 Example of the approach

This section illustrates the proposed approach. We consider the task set $\tau = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ with the following parameters ($\tau_i = \{C_i, D_i, T_i\}$): $\tau_1 = \{3,5,6\}, \tau_2 = \{3,5,8\}, \tau_3 = \{2,3,4\}, \tau_4 = \{1,8,8\}$. We assume that all the tasks have a sequential behavior except τ_1 for which the execution consists of three regions: (*i*) a sequential region of one time unit, then (*ii*) a parallel region of two sub-tasks of 0.5 time units each, and finally, (*iii*) a sequential region of one time unit. We assume that tasks in τ are released synchronously and scheduled on the homogeneous platform $\pi = \{\pi_1, \pi_2\}$. Finally, we assume that an EDF scheduler is running on each core.

During the assignment phase, let us assume that tasks τ_3 and τ_4 are assigned to π_1 and τ_2 is assigned to π_2 as they cannot benefit from any parallelism. Then, task τ_1 can neither be assigned to π_1 nor to π_2 without jeopardizing the schedulability of the corresponding core. Figure 2 (left side) illustrates the schedules in which τ_1 is tentatively assigned to π_1 (there is a deadline miss at time $t = 11$) and to π_2 (there is a deadline miss at time $t = 5$).

Now, let us apply our proposed methodology to this task set. There is a single parallel task in the system:

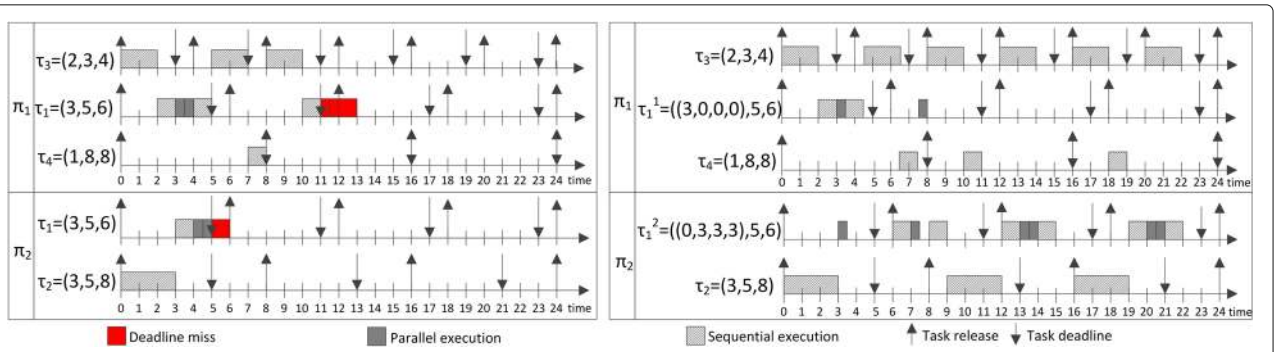

Fig. 2 Illustrative example of the proposed approach. On the left side of this figure, it is possible to observe a schedule under fully partitioned EDF (with deadline miss) and on the right side a schedule with the proposed approach

(1) *Task assignment phase:* during this phase, τ_3 and τ_4 are assigned to π_1 and τ_2 is assigned to π_2. For the same reasons as in the previous case, task τ_1 can neither be assigned to π_1 nor to π_2, so it is considered as a candidate migrating task.

(2) *Offline scheduling phase*: during this phase, an execution pattern which does not jeopardize the schedulability of the cores for the migrating task τ_1 is found. Task τ_1 is then treated as a multiframe task on each core with the following characteristics as $k_i = 24/6 = 4$: $\tau_1^1 = ((3,0,0,0),5,6)$ and $\tau_1^2 = ((0,3,3,3),5,6)$. This is given with the interpretation that the first job of τ_1 executes in core 1 and the remaining 3 jobs execute in core 2.

(3) *Online scheduling phase*: during this phase, task τ_1 takes advantage of the work-stealing mechanism in order to reduce its average response time. Indeed, at time instant $t = 3$, core π_1 is executing the parallel region of task τ_1 and core π_2 is idle with sufficient resources, so it can steal one parallel sub-task from the deque of π_1. The same situation occurs again at time $t = 7.5$. Figure 2 (right side) illustrates the resulting schedule; the system is schedulable.

6 Tasks with density greater than 1

In [8], we considered a model that only supports tasks with density no greater than one ($\lambda_i \leq 1$). Nevertheless, it is possible to overcome this limitation by recurring to decomposition-based techniques. This section provides an example of task decomposition using the technique proposed in [9] and discusses the implications of combining such an approach with work-stealing.

Decomposition-based techniques ([9, 10, 12]) traditionally convert tasks with density greater than one into a set of constrained-deadline sequential sub-tasks, each of which with density no greater than one. These approaches try to avoid parallel structures by serializing parallel tasks as much as possible so that they can take advantage of schedulability techniques developed for sequential tasks.

In [9], the authors propose the *task stretch transform* algorithm, which uses the available slack[10] of the task to proportionally *stretch* (i.e., serialize) parallel sub-tasks or parts of them in what is called a *master string*. The master string is assigned to a core and has an execution time length equal to $D_i = T_i$. The remaining parallel sub-tasks that cannot be combined in the *master string* are assigned intermediate releases and deadlines so that they become constrained-deadline tasks.

Figure 3 illustrates an example of such a task decomposition. In this example, the task consists of two sequential sub-tasks and three parallel sub-tasks, $C_i = 11$, $D_i = 10$, and therefore $\lambda_i = 1.1$. In order to stretch the task, we compute its slack (assuming an infinite number of cores and no interference from other tasks), which in the example equals $D_i - P_i = 10 - 5 = 5$ time units. The slack is then proportionally assigned to the parallel sub-tasks so that they execute sequentially in the master string. The sub-tasks that cannot be completely assigned to the master string have to be either parallelized or partly executed in two cores. In our example, one of the sub-tasks executes partly in core π_2 for one time unit and partly in core π_1 for two time units. Note that the parallel sub-tasks must have intermediate release offsets and deadlines in order to guarantee execution consistency, for instance the sub-task that executes partially in π_2 must complete before it is migrated to π_1. Therefore, it has an intermediate deadline of 6 time units after being released.

By treating a parallel task as a set of constrained-deadline sub-tasks, each of the sub-tasks can be used as input to an allocation heuristic to test the schedulability of a task set. Figure 4 illustrates an example where the above task, let us name it τ_0, is integrated into a task set of three tasks: τ_1, τ_2, and τ_3, all with implicit deadlines and τ_1 is sequential and τ_2 and τ_3 are parallel. By applying the presented decomposition approach, task τ_0 has a "stretched" task and a parallel sub-task. The stretched task is assigned to a core and the parallel sub-task can execute in any other core. Then, τ_1 is selected next and is allocated to core π_2. As for the remaining parallel tasks, τ_3 is assigned next and τ_2 is deemed a migrating task and a pattern is computed so that it is possible to schedule it upon the platform.

Some important aspects should be highlighted considering the application of decomposition-based approaches,

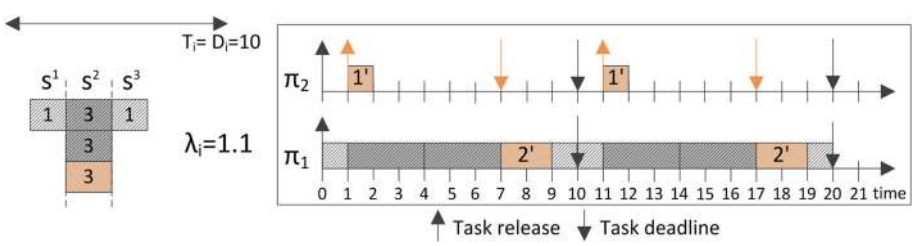

Fig. 3 Example of task decomposition. This figure depicts a task with a density of $\lambda_i = 1.1$ composed of two sequential sub-tasks and three parallel sub-tasks and its respective decomposition according to the approach proposed in [9]

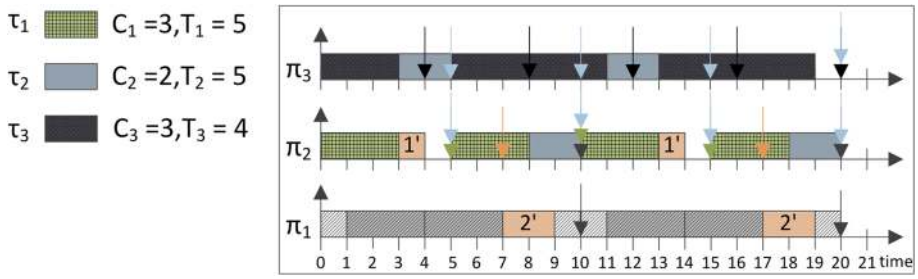

Fig. 4 Example of a task set that uses task decomposition. This figure provides an example of how one can integrate task decomposition with the proposed framework. A task set with four tasks is used where one of the tasks is decomposed into a set of constrained-deadline sub-tasks

specially regarding work-stealing. Decomposition is useful as it allows one to know if a certain task set is schedulable offline. If a task with density greater than one is identified as a migrating task, then it may be subject to stealing. When stealing a sub-task from such a task and its release offset and intermediate deadline are kept, then this task will not benefit from the stealing operation as its response time will not decrease due to the precedence constraints imposed by the master string. However, the offered idle time can be used to execute lower priority tasks or even steal work from other cores. Another option is to handle the offset constraints carefully during runtime so that intermediate deadlines are guaranteed while ensuring that no deadlines are shifted.

7 Schedulability analysis

This section derives the schedulability analysis of a set of constrained-deadline fork-join tasks onto a homogeneous multicore platform. A modification of the semi-partitioned model is adopted (see Section 4), and we assume that each core runs an EDF scheduler, while allowing work-stealing among the "selected cores", i.e., cores that share a copy of a migrating task. A schedulability analysis is performed in each phase of the proposed approach and works as follows.

(1) *Task assignment phase:* during this phase, the schedulability of the system is performed by applying the traditional DBF-based analysis [19] to non-migrating tasks, as explained in Section 3.

(2) *Offline scheduling phase*: during this phase, we make sure that the additional workload added to each core concerning the assignment of the migrating tasks does not jeopardize the schedulability of the core. Specifically, for each migrating task, say τ_i, we use a modified DBF-based schedulability test as presented in [7]. In this test, the execution pattern of each migrating task τ_i is taken into account. More precisely, the number of intervals of length $(k_i \cdot T_i)$ occurring in any interval of length $t \geq 0$ is computed as $s \stackrel{\text{def}}{=} \left\lfloor \frac{t}{k_i \cdot T_i} \right\rfloor$; since $[0,t) = [0, s \cdot k_i \cdot T_i) \cup [s \cdot k_i \cdot T_i, t)$, then the number of frames that contribute to the additional workload on core π_j consists of two terms: (*i*) the number of non-zero frames in the interval $[0, s \cdot k_i \cdot T_i]$ denoted as $s \cdot \ell_i^j$ (where ℓ_i^j is the number of frames out of k_i that were successfully assigned to π_j). The corresponding workload is $s \cdot \ell_i^j \cdot C_i$, and (*ii*) an upper-bound on the number of non-zero frames in the interval $[s \cdot k_i \cdot T_i, t)$ denoted as $nb_i(t) = \left\lfloor \frac{(t \bmod (k_i \cdot T_i)) - D_i}{T_i} \right\rfloor + 1$. The corresponding workload is $w_i^j = \max_{c=0}^{k_i-1} \left(\sum_{\eta=c}^{c+nb_i(t)-1} C_{i,\eta \bmod k_i} \right)$. It follows that an upper-bound on the total workload associated to task τ_i on core π_j is computed as $\text{DBF}_j(\tau_i, t) \stackrel{\text{def}}{=} s_i \cdot \ell_i^j \cdot C_i + w_i^j$. Consequently, $\text{DBF}\left(\tau_{\text{M}}^{\pi_j}, t\right) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_{\text{M}}^{\pi_j}} \text{DBF}_j(\tau_i, t)$.

Finally, the schedulability at the end of this phase is guaranteed if:

$$\text{load}(\pi_j) \stackrel{\text{def}}{=} \sup_{t \geq 0} \left\{ \frac{\text{DBF}\left(\tau_{\text{NM}}^{\pi_j}, t\right) + \text{DBF}\left(\tau_{\text{M}}^{\pi_j}, t\right)}{t} \right\} \leq 1, \ \forall \pi_j \in \pi \quad (3)$$

In Eq. 3, $\text{DBF}\left(\tau_{\text{NM}}^{\pi_j}, t\right)$ represents the demand for the non-migrating tasks assigned to π_j in the task assignment phase.

(3) *Online scheduling phase*: In this phase the schedulability analysis obtained in phase 2 is extended to consider the potential extra workload related to work-stealing. Figure 5 illustrates an example of the schedule of a job of a task, say τ_i, on a core, say π_j, after the offline scheduling phase. In this figure, we can see a fork-join task with its fork points (ϕ_1 and ϕ_2), synchronization points (μ_1 and μ_2), and its slack time. In this phase, we exploit the stealing windows (ω_1 and ω_2 in the example) and the available slack of each job to accommodate the stolen workload.

A work-stealing operation is feasible from one core, say core A, to another core, say core B, if core B can execute the stolen workload (i.e., a parallel sub-task from

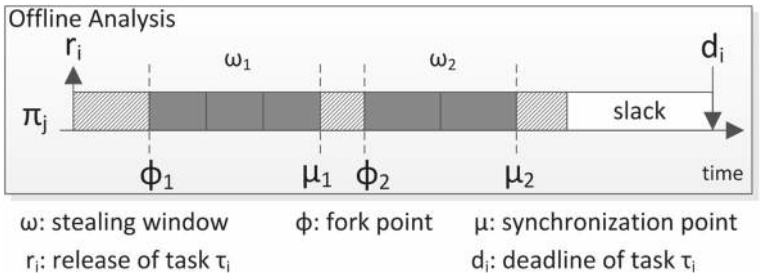

Fig. 5 Result after the offline analysis. This figure depicts a fork-join task with its fork points (ϕ_1 and ϕ_2), synchronization points (μ_1 and μ_2), and its slack time

the deque of core A) before the end of each stealing window (μ_1 and μ_2 in the example). Such time instants are denoted as the intermediate deadlines for the stolen sub-task. To compute the intermediate deadline for each stealing window, we can take advantage of the slack available for each job. Thus, the intermediate deadline of the n^{th} parallel segment can be computed as: $d_s^{(n)} \stackrel{\text{def}}{=} \phi_n + m_s * c_{s_i^{(n)}} + \text{slack}(\phi_n)$. In this equation, ϕ_n denotes the time instant at which the n^{th} parallel segment spawns the sub-tasks, m_s denotes the number of sub-tasks spawned in segment n, $c_{s_i^{(n)}}$ denotes the worst-case execution time among the tasks in segment n, and $\text{slack}(\phi_n)$ represents the slack of the job at time ϕ_n.

Figure 6 illustrates the computation of the intermediate deadlines for the stealing windows using this equation. In this figure, core π_2 can steal sub-tasks from core π_1 in stealing windows ω_1 and ω_2. The intermediate deadline for the sub-tasks that may be stolen in ω_1 is computed and the result is $d_s^{(1)}$. As the sub-task execution is less than the intermediate deadline, the stealing operation is successful. Similarly, the intermediate deadline for the sub-tasks in ω_2 is computed and the result is $d_s^{(2)}$. For the same reasons, the stealing operation is also successful in ω_2.

Before core B can steal a sub-task from core A, an admission control test has to be performed on core B. Two possible scenarios can occur when stealing a sub-task in the n^{th} parallel region of task τ_i: (1) *no release occurs in core B between* ϕ_n *and* $d_s^{(n)}$. In this case, core B can safely steal a sub-task from core A provided that the execution of the stolen sub-task meets its intermediate deadline (case 1 in Fig. 7); or (2) *at least a release occurs in core B in this stealing window*. In this case, we can distinguish between two sub-cases. (2.1) Some releases have their deadline before $d_s^{(n)}$: in this sub-case, we should update the idle time interval in the stealing window by subtracting the interference related to the corresponding new job releases from the size of the stealing window (case 2.1 in Fig. 7. In the figure, task τ_i and τ_j have releases and deadlines within ω_1). (2.2) Some releases have their deadline after $d_s^{(n)}$: in this case, no guarantees can be provided on the schedulability of the system as the stolen job may modify the scheduling decisions initially taken on core B. Therefore, no stealing occurs (case 2.2 in Fig. 7. In the figure, task τ_k has a release in ω_1 but deadline outside of the window).

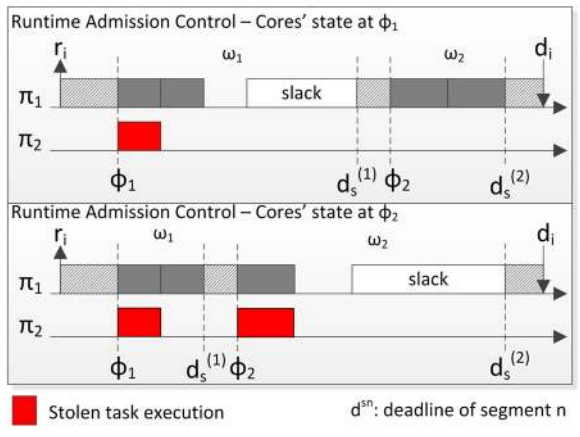

Fig. 6 Example of work-stealing and intermediate deadline computation. This figure illustrates the computation of the intermediate deadlines in the stealing windows ω_1 and ω_2

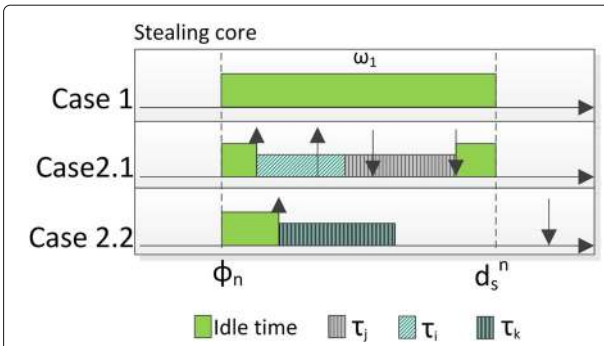

Fig. 7 Possible cases for the admission control test. This figure illustrates the possible cases for the admission control test

8 Simulation results

This section presents the results of simulating our approach on a set of synthetic and randomly generated task sets. The simulation environment is described next.

Considered platform We consider a platform consisting of two or four homogeneous cores.

Task generation Each task τ_i can be sequential or parallel. The number of each type of tasks depends on the generation itself and is not controlled beforehand. Tasks are created until the total utilization of the task set does not exceed the total platform capacity (i.e., $U_\tau \leq m$).

Tasks are created by randomly selecting a number of segments $k \in [1, 3, 5, 7]$. When $k = 1$, the task is sequential; otherwise, it is parallel. In case of a parallel task, the number of sub-tasks is $n_{\text{subtsk}} \in [k, 10]$. The worst-case execution time per sub-task ($\text{C}_{\text{i,subtsk}}$) in each task varies in the range [1, max_Ci_subtsk] where max_Ci_subtsk = 2 for performance reasons. We compute the worst-case execution time of each task as $C_i = \sum_{\forall\ \text{subtsk} \in \tau_i} \text{C}_{\text{i,subtsk}}$[11]. Then, we derive the remaining parameters: the period T_i and utilization U_i. The period T_i is uniformly generated in the interval $[C_i, n_{\text{subtsk}} * \text{max\_Ci\_subtsk} * 2]$. This interval allows us to have a task utilization $\left(\text{recall that } U_i = \frac{C_i}{T_i}\right)$ that falls in the interval [0.50, 1] if all nodes are assigned max_Ci_subtsk or [0.25, 1] if all nodes are assigned the minimum value for $\text{C}_{\text{i,subtsk}}$[12]. To generate execution patterns for the migrating tasks, we use Eq. 2 first and if no pattern is found we follow an enumeration approach. In our experiments, $D_i = T_i$. This procedure is repeated until 1000 task sets with migrating tasks are generated for two and four cores.

Selected heuristics In order to evaluate the performance of FFDO, we have conducted benchmarks against other well-known bin-packing heuristics, namely the standard first-fit decreasing (FFD), best-fit decreasing (BFD), and worst-fit decreasing (WFD). FFD assigns each task to the first core from the set of cores with sufficient idle time to accommodate it; BFD assigns each task into the core which after the assignment minimizes the idle time among all cores; and WFD assigns the task to the core which after the assignment maximizes the idle time among all cores. All the heuristics, except FFDO, group the tasks into sequential and parallel tasks and sort each group in a decreasing order of task utilization.

In order to compare the heuristics, we measured the percentage of unallocated tasks over a large number of task sets (to this end, we generated one million task sets in this experiment) to decide which heuristics have a higher number of candidate migrating tasks. Figure 8 depicts the results. We clearly observe that FFDO and WFD are the heuristics that present a higher number of unallocated tasks, while BFD and FFD allocate nearly the same amount of tasks and present a lower value of unallocated tasks when compared to FFDO and WFD. These results indicate that our initial heuristic is a good candidate for our approach as it allows the approach to try to re-allocate a high number of tasks in the second phase as migrating tasks. Due to this result, we selected both FFDO and WFD for a direct comparison in terms of the number of schedulable task sets.

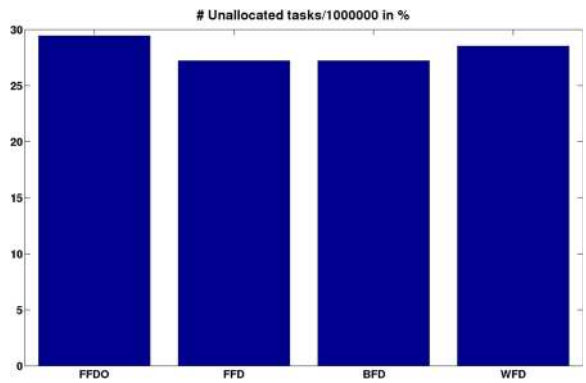

Fig. 8 Percentage of unallocated tasks per heuristic. This figure presents the percentage of unallocated tasks per heuristic over one million task sets

To compare these two heuristics, we followed a procedure where a number of task sets are randomly generated in order to obtain 100 schedulable task sets with FFDO, and then, for all the generated task sets, we evaluate how many of them are schedulable using WFD. Figure 9 depicts the results of this comparison.

The task sets schedulable by using WFD can be divided into four groups: 26.85% of these task sets are schedulable by using both heuristics; 24.51% are not schedulable by using FFDO due to k_i[13]; 43.19% are not schedulable by using FFDO with a k_i value in the range of valid values; and finally, 5.45% of the task sets are deemed not schedulable with FDDO after applying the heuristic. Overall, in a two-core setting, the total number of task sets that are schedulable by using WFD is 257, which represents an increase of 157% over FFDO for the same input. From the diagram, the majority of the task sets that are schedulable by using WFD fit in a potential feasible region for FFDO heuristic (43.19%) — here, all task sets have migrating tasks and k_i values that fit in the range of valid values but no feasible pattern is found. These results still hold for four cores but to a less extent as only 17.9% more task sets were schedulable by using WFD over FFDO.

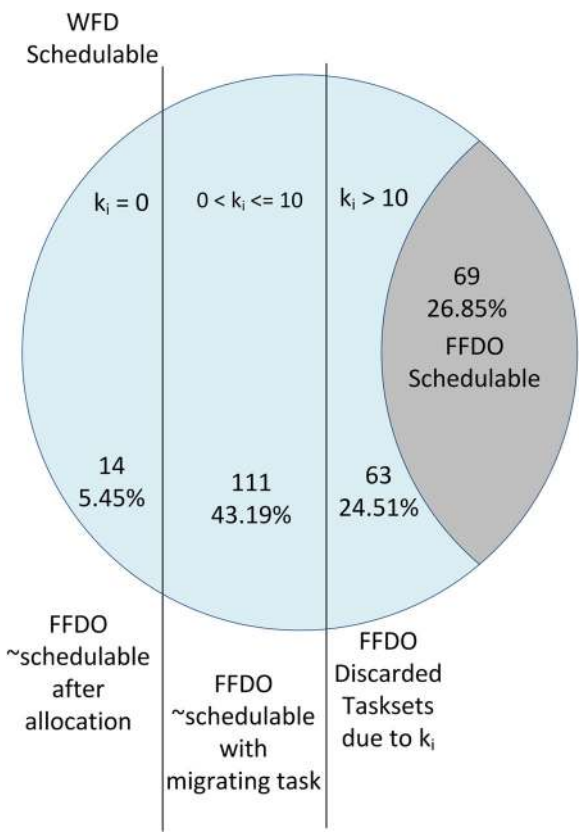

Fig. 9 Comparison between FFDO and WFD. This figure presents how the data is categorized for WFD when a fixed number of task sets is used as input. For the same number of input task sets used to obtain 100 schedulable task sets for FFDO, we have obtained 257 for WFD

We conjecture that WFD behaves better than FFDO (even though FFDO has a higher percentage of unassigned tasks as shown in Fig. 8) for smaller number of cores because of the task-to-core assignment. Depending on the granularity of the utilization of the task sets, more empty space may be available globally in the cores when performing the task allocation for a small number of cores. These idle slots make it possible for our pattern-finding procedure to find enough room to fit a job of a task when computing the execution pattern for a migrating task. However, as the number of cores increases, WFD naturally balances the workload through the cores, whereas FFDO assigns the workload in the initial cores leaving more room in later cores. For this reason, we envision that WFD will have the tendency to behave either equally to or even worse than FFDO with the increase in the number of cores.

Considered metrics In order to evaluate the proposed approach, we measure the gain obtained in terms of the average worst-case response time for each schedulable task set. Specifically, for each task set, we generate the complete schedule for the two approaches: the approach that schedules migrating tasks without applying the work-stealing mechanism among the selected cores, denoted as Approach-NS; and the approach that applies the work-stealing mechanism among the selected cores, denoted as Approach-S. After generating both schedules for each task set, we compute the average response-time of the jobs of each task throughout the hyperperiod by adding the response time of each individual job and by dividing the obtained result by the number of jobs in one hyperperiod. This process is applied to both approaches. The improvement, i.e., the gain of Approach-S over Approach-NS is computed by applying the following formula for each task τ_i: $AV_{\tau_i} = \frac{AV^{NS}_{\tau_i} - AV^{S}_{\tau_i}}{AV^{NS}_{\tau_i}} \cdot 100$, where $AV^{NS}_{\tau_i}$ denotes the average response-time for task τ_i in Approach-NS and $AV^{S}_{\tau_i}$ denotes its average response-time in Approach-S. It follows that the average gain for each task in the task set is computed by dividing AV_τ: $AV_\tau = \frac{1}{|\tau|} \cdot \sum_{\tau_i \in \tau} AV_{\tau_i}$.

Figure 10 illustrates the average gain for two and four cores, respectively, for the selected heuristics, namely FFDO and WFD.

Interpretation of the results The improvement in terms of average response time per task (in %) is grouped by utilization—see Fig. 10—when using Approach-S over Approach-NS. For each sub-figure, the distribution of data is depicted in the form of box plot. In the plot, for each utilization value, it is possible to see the minimum and maximum values of gain per task, the median and the mean (in the form of a diamond shape), the first and third quartiles, and finally, the outliers in the shape of a cross. The line in red depicts a linear regression on the data (the mean value was used to compute the regression) in order to depict the pattern of prediction of the gain per task.

Considering two cores: for task sets with a high utilization (over 1.55), there is a clear illustration of the gain of the proposed approach. In the best case, this gain reaches nearly 15% for FFDO and nearly 12% of the average response time per task for WFD, which is non negligible. As the utilization of the task sets increases, the gain per task decreases. This is expected due to the increasing lack of idle time available for stealing. The trend shows that above 1.95 of utilization, the work-stealing mechanism becomes of little interest. This is explained by the fact that the total workload on each core is very high, thus leaving very small room for improvement on the average response time of each migrating task through work-stealing. It is important to note that task sets with utilizations below 1.55 using FFDO and 1.45 using WFD are not included in the plot as they do not contain any migrating task.

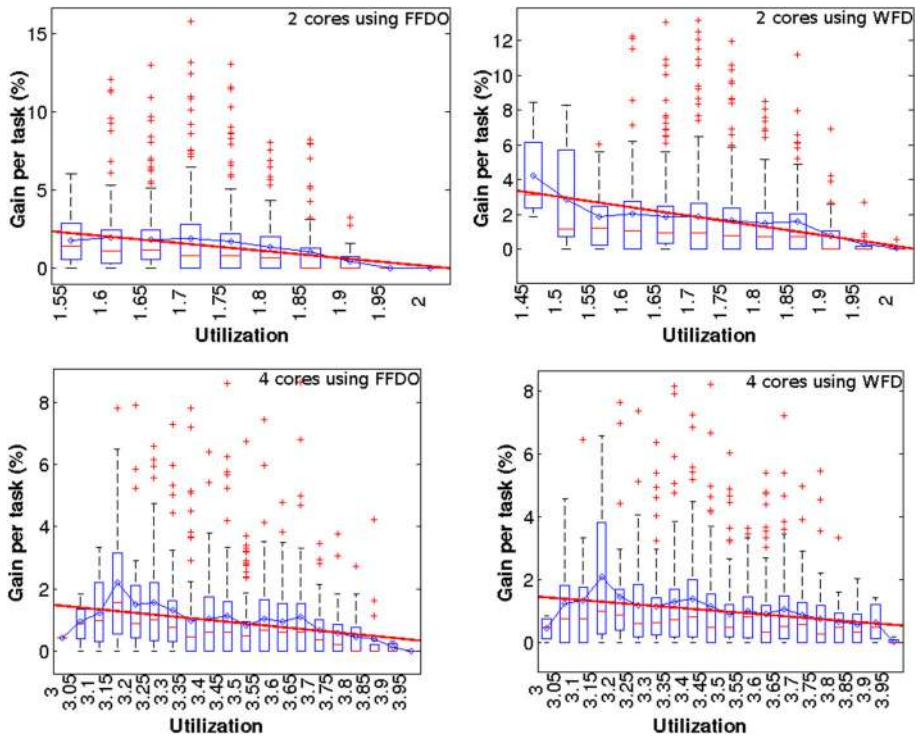

Fig. 10 Simulation results for FFDO and WFD. This figure presents the improvement in the average worst-case response-time when using an approach with work-stealing and an approach without work-stealing. On the top, one can see the results for two cores and on the bottom one can see the results for four cores. On the left side, the results show the improvement for FFDO and on the right the improvement for WFD

Considering four cores: the trend is similar to the one depicted for two cores. This trend is also shown by the linear regression line where it is possible to predict the average gain per task as a function of the utilization of the task set. The regression shows that for lower utilizations in two cores, the expected improvement starts at 2.3% for FFDO and 3.3% for WFD. For four cores, it starts at 1.4% for both heuristics. We can also observe that the expected improvement decreases with an increase in the tasks' utilization. This behavior suggests that work-stealing is useful for task sets with migrating tasks with a utilization that span from the lowest possible utilization for task sets with migrating tasks up to the platform capacity. Closer to this upper limit, the benefits of using work-stealing are limited. From the observed behavior in two and four cores, we conjecture that the proposed approach will behave similarly when the number of cores increases.

Overheads of the approach This work shows that it is possible to decrease the average response time of tasks and use this newly generated free time slots to execute less critical tasks (e.g., aperiodic or best-effort tasks). While such a decrease involves overhead costs, such as the number and cost of migrations or even the impact of online admission control on the overall approach, we did not explicitly measure them. Still, we provide an overview of the existing costs and their possible impact on system performance.

We assume that cores that share a migrating task have a local copy of this task. However, keeping task copies is platform dependent as for some platforms it might not be possible to have copies due to memory constraints. In our approach, local copies are used for migrating tasks which might be subject to stealing, and having a local copy prevents fetching the task code from the main memory. Whenever a stealing operation occurs, a core fetches data from another core's memory in order to help in the execution of the task. While this is not a task migration *per se*, it has some commonalities as data needs to be moved from one core to another. This may cause interference in the execution of other tasks in the system (for instance due to the existence of shared resources). In our approach, this overhead only occurs when stealing occurs and is performed by a core that is idle, so part of the cost is supported by the idle core (which is negligible due to the idleness of the core). Considering the number of data transfers, this number can be bounded in our framework as in the worst-case the number of data

fetches when stealing depends on the number of sub-tasks in each segment and the number of cores that share the task.

Considering the online admission control, our test requires the current time instant and the available slack at a specific time instant. Both of these variables can be easily computed in any given platform either by using the platform timing functions and a cumulative function that computes the slack for the current job. Therefore, we consider that this does not pose any significant overhead in our approach.

9 Conclusions

In this paper, we combined techniques that allow us to schedule fine-grained parallel fork-join tasks onto multicore platforms. By using the proposed technique, we can schedule systems with high utilizations. Moreover, the proposed technique takes advantage of the semi-partitioned scheduling properties by offering the possibility to accommodate parallel tasks that cannot be scheduled in any pure partitioned environment, while reducing the migration overhead which has been shown to be a traditional major source of non-determinism in global approaches. Parallel tasks are heavy in their nature, and therefore, a natural candidate for this model if execution time constraints is present. Our results show that by using work-stealing, it is possible to achieve an average gain on the response times of the parallel tasks between 0 and nearly 15% per task, which may leave extra idle time in the schedule to execute less critical tasks in the platform (i.e., aperiodic, best-effort).

Endnotes

[1] Note that the balance of the platform workload at runtime also allows for a better control of the platform energy consumption [21, 22].

[2] A deque is a special type of queue which also works as a stack.

[3] Two or more cores executing a migrating task share a copy of this task.

[4] There is no restriction on the execution requirement of each node, and the execution time of each node may vary from one node to another.

[5] There is no communication, no precedence constraints and no shared resources (except for the cores) between sub-tasks.

[6] A task which consists of a single sub-task in each of its segments is considered a sequential task.

[7] We have explored alternative heuristics (see Section 8).

[8] The threshold for classifying tasks varies in the literature, nevertheless a density of 0.5 is usually regarded as a good threshold for classifying tasks.

[9] These cores are also referred to as "selected cores".

[10] Slack is the maximum amount of time that the remaining computation time of a job can be delayed at a time instant t (with $a_{i,j} \leq t \leq d_{i,j}$) in order to complete within its deadline.

[11] By considering the worst-case execution time for each sub-task in the experiments we are evaluating the benefits of using work-stealing in the worst possible scenario.

[12] As we evaluate the behavior of each task set in the interval $[0, H]$, where H denotes the least common multiple of the periods of all the tasks in the task set, and as T_i in our generation depends on C_i, the higher the C_i, the higher the T_i and consequently, the higher the hyperperiod of the task set. By limiting $C_{i,subtsk}$ we are also limiting the amount of time we need to generate the schedule.

[13] As explained in [8], we reject task sets that have a number of frames over 10 for performance considerations. In summary, the complexity of the computation of the migrating patterns increases for large k_i, which leads to higher computation times.

Appendix

Table 1 Notation table

Symbols	Description
τ	Set of n tasks
D_i	Relative deadline of task τ_i
T_i	Period of task τ_i
$a_{i,j}$	Arrival time of job j of task τ_i
$d_{i,j}$	Absolute deadline of job j of task τ_i
$S_i = [s_i^1, s_i^2, \ldots, s_i^{n_i}]$	Sequence of n_i segments, $n_i \in \mathbb{N}$
s_i^k	Segment $k \in S_i$
$t_{s_i^k}^q$	Sub-task q belonging to segment s_i^k
v_k	Number of sub-tasks belonging to segment s_i^k
$e_{s_i^k}^q$	Execution time of sub-task $t_{s_i^k}^q$
C_i	Total execution requirement of task τ_i
P_i	Minimum execution requirement of task τ_i
$C_{s_i^k}$	Worst-case execution time among the sub-tasks of segment s_i^k
U_i	Utilization factor of task τ_i
λ_i	Density factor of task τ_i
U_τ	Total utilization factor of the task set τ
λ_τ	Total density of the task set τ

Table 1 Notation table (*Continued*)

π	Set of m homogeneous cores
τ^{π_j}	Set of tasks assigned to core π_j
$\tau_{\text{NM}}^{\pi_j}$	Subset of non-migrating tasks assigned to π_j
$\tau_{\text{M}^{\pi_j}}$	Subset of migrating tasks assigned to π_j
k_i	Number of frames of a migrating task τ_i
H	Least common multiple of the periods of all the tasks in τ
σ	Job-to-core assignment sequence
$M[i,j]$	Matrix of integers of the current job-to-core assignment
s	Number fo intervals of length $k_i \cdot T_i$
ℓ_i^j	Number of frames out of k_i that were successfully assigned to π_j
$nb_i(t)$	Upper-bound on the number of non-zero frames in the interval $[s \cdot k_i \cdot Ti, t]$
w_i^j	Workload in the interval $[s \cdot k_i \cdot Ti, t]$
$dbf_j(\tau_i, t)$	Demand-bound function for task τ_i in π_j in the interval $[0, t]$
ϕ_n	Fork point for segment n
μ_n	Synchronization point segment n
ω_n	Stealing window n
slack (ϕ_n)	Slack of the job at time instant ϕ_n
$d_s^{(n)}$	Intermediate deadline of the n^{th} parallel segment
m_s	Number of sub-tasks spawned in segment n
$c_{s_i^{(n)}}$	Worst-case execution time among the tasks that belong to segment n
AV_{τ_i}	Gain per task of Approach-S over Approach-NS
AV_τ	Average gain per task in the task set

Acknowledgements
This work was partially supported by National Funds through FCT/MEC (Portuguese Foundation for Science and Technology) and co-financed by ERDF (European Regional Development Fund) under the PT2020 Partnership, within project UID/CEC/04234/2013 (CISTER Research Centre); by FCT/MEC and ERDF through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project(s) FCOMP-01-0124-FEDER-020447 (REGAIN); by FCT/MEC and the EU ARTEMIS JU within project(s) ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2); by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement no 611016 (P-SOCRATES); and also by FCT/MEC and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH / BD / 88834 / 2012.

Competing interests
The authors declare that they have no competing interests.

Publisher's Note

Received: 10 February 2016 Accepted: 21 August 2017
Published online: 13 September 2017

References

1. RI Davis, A Burns, A survey of hard real-time scheduling for multiprocessor systems. ACM Comput. Surv. **43**(4), 35–13544 (2011)
2. A Marowka, Parallel computing on any desktop. Commun. ACM. **50**, 74–78 (2007)
3. RD Blumofe, CE Leiserson, Scheduling multithreaded computations by work stealing. J. ACM. **46**(5), 720–748 (1999)
4. C Maia, L Nogueira, LM Pinho, in *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium On*. Scheduling parallel real-time tasks using a fixed-priority work-stealing algorithm on multiprocessors, (2013), pp. 89–92. doi:10.1109/SIES.2013.6601477
5. JH Anderson, V Bud, UC Devi, in *Proceedings of the 17th Euromicro Conference on Real-Time Systems, ECRTS '05*. An edf-based scheduling algorithm for multiprocessor soft real-time systems (IEEE Computer Society, Washington, 2005), pp. 199–208
6. S Kato, N Yamasaki, Y Ishikawa, in *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference On*. Semi-partitioned scheduling of sporadic task systems on multiprocessors, (2009), pp. 249–258
7. F Dorin, PM Yomsi, J Goossens, P Richard, Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. CoRR. **abs/1006.2637** (2010)
8. C Maia, PM Yomsi, L Nogueira, LM Pinho, in *Embedded and Ubiquitous Computing (EUC), 2015 IEEE 13th International Conference On*. Semi-partitioned scheduling of fork-join tasks using work-stealing, (2015), pp. 25–34. doi:10.1109/EUC.2015.30
9. K Lakshmanan, S Kato, RR Rajkumar, in *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium, RTSS '10*. Scheduling Parallel Real-Time Tasks on Multi-core Processors (IEEE Computer Society, Washington, 2010), pp. 259–268. http://dx.doi.org/10.1109/RTSS.2010.42
10. A Saifullah, K Agrawal, C Lu, C Gill, in *Proceedings of the 2011 IEEE 32Nd Real-Time Systems Symposium, RTSS '11*. Multi-core Real-Time Scheduling for Generalized Parallel Task Models (IEEE Computer Society, Washington, 2011), pp. 217–226. http://dx.doi.org/10.1109/RTSS.2011.27
11. V Bonifaci, A Marchetti-Spaccamela, S Stiller, A Wiese, in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference On*. Feasibility analysis in the sporadic dag task model, (2013), pp. 225–233. doi:10.1109/ECRTS.2013.32
12. M Qamhieh, L George, S Midonnet, in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems, RTNS '14*. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems (ACM, New York, 2014), pp. 13–131322. doi:10.1145/2659787.2659818. http://doi.acm.org/10.1145/2659787.2659818
13. J Li, K Agrawal, C Lu, C Gill, in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference On*. Analysis of global edf for parallel tasks, (2013), pp. 3–13. doi:10.1109/ECRTS.2013.12
14. HS Chwa, J Lee, K-M Phan, A Easwaran, I Shin, in *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference On*. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms, (2013), pp. 25–34. doi:10.1109/ECRTS.2013.14
15. C Maia, M Bertogna, L Nogueira, LM Pinho, in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems, RTNS '14*. Response-time analysis of synchronous parallel tasks in multiprocessor systems (ACM, New York, 2014), pp. 3–3312. doi:10.1145/2659787.2659815. http://doi.acm.org/10.1145/2659787.2659815
16. B Bado, L George, P Courbin, J Goossens, in *Proceedings of the 20th International Conference on Real-Time and Network Systems, RTNS*. A semi-partitioned approach for parallel real-time scheduling (ACM, New York, 2012), pp. 151–160. doi:10.1145/2392987.2393006. http://doi.acm.org/10.1145/2392987.2393006
17. AK Mok, D Chen, A multiframe model for real-time tasks. Softw. Eng. IEEE Trans. **23**(10), 635–645 (1997)
18. S Baruah, D Chen, S Gorinsky, A Mok, Generalized multiframe tasks. Real-Time Syst. **17**(1), 5–22 (1999)
19. SK Baruah, AK Mok, LE Rosier, in *Real-Time Systems Symposium, 1990. Proceedings., 11th*. Preemptively scheduling hard-real-time sporadic tasks on one processor, (1990), pp. 182–190
20. J Goossens, P Richard, M Lindström, II Lupu, F Ridouard, in *Proceedings of the 20th International Conference on Real-Time and Network Systems, RTNS '12*. Job partitioning strategies for multiprocessor scheduling of real-time periodic tasks with restricted migrations (ACM, New York, 2012), pp. 141–150
21. J Kang, DG Waddington, in *Proceedings of the 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and*

Applications, RTCSA '12. Load Balancing Aware Real-Time Task Partitioning in Multicore Systems (IEEE Computer Society, Washington, 2012), pp. 404–407. http://dx.doi.org/10.1109/RTCSA.2012.71
22. H Aydin, Q Yang, in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*. Energy-aware partitioning for multiprocessor real-time systems, (2003), p. 9. doi:10.1109/IPDPS.2003.1213225