CISTER
Research Center in
Real-Time & Embedded
Computing Systems

Conference Paper

Semi-Partitioned Scheduling of Fork-Join Tasks using Work-Stealing

Cláudio Maia
Patrick Meumeu Yomsi
Luís Nogueira and Luis Miguel Pinho

CISTER-TR-151007

Semi-Partitioned Scheduling of Fork-Join Tasks using Work-Stealing

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

CISTER Research Center

Polytechnic Institute of Porto (ISEP-IPP)

Rua Dr. António Bernardino de Almeida, 431

4200-072 Porto

Portugal

Tel.: +351.22.8340509, Fax: +351.22.8321159

E-mail:

http://www.cister.isep.ipp.pt

Abstract

This paper explores the behavior of parallel fork-join tasks on multicore platforms by resorting to a semi-partitioned scheduling model. This model offers a promising framework to embedded systems which are subject to stringent timing constraints as it provides these systems with very interesting properties. The proposed approach consists of two stages—an offline stage and an online stage. During the offline stage, a multi-frame task model is adopted to perform the fork-join task-to-core mapping so as to improve the schedulability and the performance of the system, and during the online stage, work-stealing is exploited among cores to improve the system responsiveness as well as to balance the execution workload. The objective of this work is twofold: (1) to provide an alternative technique that takes advantage of the semi-partitioned scheduling properties by offering the possibility to accommodate fork-join tasks that cannot be scheduled in any pure partitioned environment, and (2) to reduce the migration overhead which has shown to be a traditional major source of non-determinism in global approaches. The simulation results show an improvement of the proposed approach over the state-of-the-art of up to 15% of the average response-time per task set.

Semi-Partitioned Scheduling of Fork-Join Tasks using Work-Stealing

Cláudio Maia, Patrick Meumeu Yomsi, Luís Nogueira and Luis Miguel Pinho
CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto
Email: {crrm, pamyo, lmn, lmp}@isep.ipp.pt

***Abstract*—This paper explores the behavior of parallel fork-join tasks on multicore platforms by resorting to a semi-partitioned scheduling model. This model offers a promising framework to embedded systems which are subject to stringent timing constraints as it provides these systems with very interesting properties. The proposed approach consists of two stages—an offline stage and an online stage. During the offline stage, a multi-frame task model is adopted to perform the fork-join task-to-core mapping so as to improve the schedulability and the performance of the system, and during the online stage, work-stealing is exploited among cores to improve the system responsiveness as well as to balance the execution workload. The objective of this work is twofold: (1) to provide an alternative technique that takes advantage of the semi-partitioned scheduling properties by offering the possibility to accommodate fork-join tasks that cannot be scheduled in any pure partitioned environment, and (2) to reduce the migration overhead which has shown to be a traditional major source of non-determinism in global approaches. The simulation results show an improvement of the proposed approach over the state-of-the-art of up to 15% of the average response-time per task set.**

***Keywords*—Parallel Tasks, Semi-Partitioned Scheduling, Real-time Systems, Work-Stealing**

I. Introduction

Multicore platforms are now very common in the embedded systems domain as they provide more computing power for the execution of complex applications. Recent platforms, such as Tilera [25] and Keystone [13] are few examples of such platforms. They comprise several cores on a single chip and include several mechanisms (e.g., pipelines, caches, etc.) to increase the average performance of the applications they host. While an increase in performance might be a good indicator in several industry domains such as the high performance computing and multimedia, the picture changes completely when it comes to the real-time systems domain. Here, most applications must guarantee stringent timing constraints in addition to their classical functional requirements, thus making predictability of paramount importance.

The boost in performance brought by the emerging multicore platforms also increased substantially the complexity of the scheduling problem of real-time tasks on these platforms. While in uniprocessors the scheduling problem reduces to deciding *when* to schedule each task, a new dimension orthogonal to this one adds to the problem when shifting to multicores: it must also be decided *where* to execute each task. In order to solve this challenge, several real-time scheduling algorithms have been proposed in the literature for multiprocessor systems (see [10] for a comprehensive and up-to-date survey).

Multicore platforms make the simultaneous execution of tasks possible by taking advantage of their parallel structure. To this end, parallelism is extracted at compile time from the loops of the application by using programming frameworks such as OpenMP [21] and Java Fork-Join [17], in addition to the explicit parallelism that may be inherent to the application or the problem to solve. These frameworks resort to the dynamic scheduling properties provided by the work-stealing algorithm proposed by Blumofe and Leiserson [6]. In summary, work-stealing is a scheduling algorithm which allows an idle core[1] at a time instant t, say core A, to steal some workload from another busy core, say core B. This property allows system designers to reduce the average response-time of the executed task set on the target platform. In this process, Core B is usually chosen randomly and is referred to as "victim". While the randomness in the selection of Core B is acceptable in several computing domains, no guarantee can actually be provided regarding the timing behavior of the tasks, unfortunately. This drawback, which represents a huge limitation for the adoption of the original work-stealing algorithm in the real-time systems domain, is due to the possibility of priority inversion as shown by Maia et al. [20]. Hence, it is necessary to modify the original algorithm to circumvent this issue. To this end, using multiple per-core priority deques [20] is just an example. Another property provided by work-stealing is its ability to balance the platform workload at runtime. This property allows for a better control of the platform energy consumption [2], [14].

The computation model considered in this paper is a variant of the *semi-partitioned* model of execution with *task-level migration*. We recall that in semi-partitioning scheduling [1], [11], [15], there are two steps: (Step 1) a task-to-core mapping is performed at design time to assign tasks to specific cores with no possible migration for these tasks at runtime. The subset of tasks that have successfully been assigned during this process is referred to as *non-migrating tasks*. If some tasks cannot be assigned, then in (Step 2) the remaining tasks, referred to as *migrating tasks*, are scheduled by using a global scheduling approach to seek for a valid schedule, i.e., each migrating task is allowed to execute on more than one core. Considering the time instant at which a migration occurs, semi-partitioned scheduling can be further classified into two subcategories: (1) *Task-level migration* [11], where various jobs of a migrating task are allowed to be assigned to different cores, but once a job is assigned to a core, migration of this job prior to its completion is forbidden; and (2) *Job-level migration* [15], where various jobs of a migrating task are

[1] An idle core at a time instant t is a core with no pending workload at t.

allowed to be assigned to different cores, and migration of each job prior to its completion is also allowed. In this work, we consider task-level migration and restrict the behavior of each migrating task as follows. The subset of selected cores to execute each migrating task is decided at design time and remains unchanged at runtime in order to improve both the schedulability of the system and its utilization factor. Further, the job activation of each migrating task at runtime is performed relatively to a task-to-core strategy elaborated at design time.

Due to the intrinsic nature of parallel real-time tasks in real world applications, it is common to find some tasks with a density greater than one (i.e., a single core cannot execute such a task entirely while meeting its timing requirements). Thus, new models must be elaborated to accommodate such tasks. To this end, recent works [16], [23], [24] propose *decomposition-based* and *non-decomposition-based* techniques. Decomposition-based techniques require the knowledge of the task structure beforehand. These techniques allow decomposing each parallel task with density greater that one into a set of sequential sub-tasks with density less than or equal to one for which the precedence constraints are guaranteed through the usage of intermediate release times and deadlines. This decomposition is performed with the objective of scheduling the sequential sub-tasks by using well-known multiprocessor scheduling algorithms. Non-decomposition techniques on the other hand do not require any knowledge of the task structures beforehand in order to determine the schedulability of a parallel task.

In this paper, we consider fork-join real-time tasks (i.e., a special case of parallel real-time tasks) and we assume that their structure is known beforehand. Thus, we take advantage of decomposition-based techniques to convert tasks with density greater than one into a set of sub-tasks with density less than or equal to one as long as the original parallel structure of the task is preserved. The potential parallelism of the migrating tasks is explored at runtime by resorting to the load balancing property provided by a variant of the work-stealing algorithm [20]. Our main objective is to provide evidence of the benefits of knowing the structure of parallel tasks and to exploit work-stealing in the real-time systems domain. As such, besides reducing the average response time of the tasks and contributing to the minimization of the energy consumption of the system, we free additional room in the schedule for the scheduling of less-critical tasks (e.g., aperiodic tasks, best-effort tasks). The proposed model limits the number and cost 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[2].

To the best of our knowledge, there is no schedulability test for parallel tasks with a density greater than one on multicore platforms which assumes a partitioned earliest deadline first (P-EDF) scheduler on each core. As soon as a test is developed in this scope, for parallel fork-join tasks with density greater than one, we strongly believe that our model can be easily adapted and applied to parallel tasks with density greater than one.

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

Contributions. The contribution of this work is threefold: (1) we present a complete framework that allows the scheduling of parallel fork-join real-time tasks onto a multicore platform together with the associated schedulability analysis; (2) As it is assumed that cores that share a task have a local copy of this task, not only we reduce the overhead concerning task fetching but also the number of task migrations due to the offline task-to-core mapping; (3) As the regions of each parallel 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 parallel tasks without jeopardizing the schedulability of the whole system. To the best of our knowledge, this work is the first effort in the direction to use work-stealing mechanism in a real-time setting when tasks are scheduled using a semi-partitioned scheduling approach.

Paper organization. The rest of this paper is organized as follows. Section II presents the related work. Section III describes the model of computation used throughout the paper. Section IV details our proposed approach. Section V presents a motivating example for proof of concepts. Section VI presents the schedulability analysis of the proposed approach. Section VII reports on the simulation results from experiments conducted on synthetic task sets. Finally, Section VIII concludes the paper and presents the perspectives.

II. Related Work

There exists three task models which support *intra-task parallelism* in real-time systems: (1) the fork-join model; (2) the synchronous task model; and finally (3) the directed acyclic graph (DAG) model. From these models, the fork-join model (see Figure 1a) is the most simple in terms of parallel structure. The fork-join model allows for fork and join operations to occur infinitely as long as each task starts and ends with a sequential sub-task. Specifically, the initial sequential sub-task may fork into several independent sub-tasks, which can execute simultaneously in parallel. These sub-tasks join in a sequential sub-task upon completion and the process may repeat again. This model is a restricted class of the synchronous task model. Specifically, in the fork-join model presented in [16], segments must have the same number of parallel sub-tasks and there is a restriction on the number of sub-tasks that each task can fork into (not greater than the number of cores on the platform). These restrictions do not apply to the synchronous model [24]. Note that there are precedence constraints among sub-tasks, which are enforced by the order of the segments.

The DAG model [7], [18], [23] is the most general model in which each sub-task is represented as a node and the edges connecting the nodes represent the dependencies between sub-tasks. In this model, there is no restriction on the execution requirement of each node, and the execution times may vary among nodes. In order to perform the schedulability analysis of these three models, decomposition-based techniques [16], [23], [24] and non-decomposition based techniques [7], [18] have been proposed. Resource and capacity augmentation bounds have been derived as a mean to evaluate the schedulability of a task set using various scheduling algorithms, and as an alternative, response-time analysis [9], [19] can also be used to analyze synchronous parallel tasks.

(a) Example of a parallel fork-join task execution

(b) Example of the corresponding serialized fork-join task execution

Figure 1: Illustration of a fork-join task execution

Very few techniques using the same framework as the one developed in this paper exist in the literature. Bado et al. [3] proposed a semi-partitioned approach with job-level migration for fork-join tasks, which is similar to the one in [16] in the sense that the authors limit the task parallelism to the minimum degree possible. However, 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.

III. 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* $S_i \stackrel{\text{def}}{=} [s_i^1, s_i^2, ..., s_i^{n_i}]$, 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 $\tau_{i,j}$, with a finite sequence of segments S_i each, arriving at time $a_{i,j}$ such that $a_{i,j+1} - a_{i,j} \geq T_i$ and which must complete within $[a_{i,j}, d_{i,j})$ where $d_{i,j} \stackrel{\text{def}}{=} a_{i,j} + D_i$ is its absolute deadline. Job $\tau_{i,j}$ is said to be *active* at time t if and only if $a_{i,j} \leq t$ and is not completed yet. More precisely, an active task is said to be *running* at time t if it is being executed. Otherwise the active task is said to be *ready*.

Each segment $s_i^k \in S_i$ (with $1 \leq k \leq n_i$) is composed of a set of *independent sub-tasks*[3] $t_{s_i^k} \stackrel{\text{def}}{=} \{t_{s_i^k}^1, \ldots, t_{s_i^k}^{v_k}\}$, where v_k denotes the number of sub-tasks belonging to segment s_i^k, and the sequence 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., $C_i \stackrel{\text{def}}{=} \sum_{k=1}^{n_i} \sum_{q=1}^{v_k} e_{s_i^k}^q$. The *minimum execution requirement* of task τ_i, denoted as P_i, is defined as the time that τ_i takes to execute when the number of cores is infinite[4], i.e., $P_i = \sum_{k=1}^{n_i} c_{s_i^k}$, where $c_{s_i^k}$ denotes the worst-case execution time of segment k. We assume that all the sub-tasks in a segment have the same worst-case execution time $c_{s_i^k}$. The *utilization factor* of τ_i is $U_i = \frac{C_i}{T_i}$, its *density* is $\lambda_i = \frac{C_i}{\min(D_i, T_i)}$, the *total density* of τ is $\lambda_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} \lambda_i$ and finally, the *total utilization factor* of τ is $U_\tau \stackrel{\text{def}}{=} \sum_{i=1}^{n} U_i$.

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. Figure 1a illustrates a fork-join task τ_i with $n_i = 5$ segments and Figure 1b its serialized representation.

Moreover, we model each *migrating task* as a *multiframe task*. The multiframe task model (as presented by Mok et al. [22] and later generalized by Baruah et al. [4]) allows system designers to model a task by using a *static* and finite list of total execution requirements corresponding to successive jobs (or frames 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. On each core there is a fully preemptive 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 under EDF for uniprocessors can be evaluated by using the Demand Bound Function (DBF) [5]. The DBF of a task τ_i, denoted by $\mathrm{DBF}(\tau_i, t)$, is the maximum cumulative execution requirement of jobs of τ_i in any interval of length t. Formally, $\mathrm{DBF}(\tau_i, t)$ is defined as:

$$\forall t \geq 0,\ \mathrm{DBF}(\tau_i, t) \stackrel{\text{def}}{=} \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1 \right) \cdot C_i \qquad (1)$$

The DBF of a task set τ is derived as $\mathrm{DBF}(\tau, t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \mathrm{DBF}(\tau_i, t)$.

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

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

As each job is composed of sub-tasks, 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 it was executing on prior to preemption. We assume that each preemption is performed at no cost or penalty. Finally, we allow work-stealing among a selected subset of cores at runtime. Our work-stealing approach works as follows. At runtime whenever a core reaches a parallel section of a task and spawns sub-tasks, these are pushed into a *double-ended queue* (in short deque)[5]. A core always executes the sub-tasks by popping them from the local deque. Whenever a core with a copy of a task becomes idle, then it can steal some workload (sub-tasks) from the chosen core (known as the victim) by checking its deque. Note that in this stealing strategy in contrast to the traditional stealing mechanism, the stealing core is no longer randomly chosen. Work-stealing is allowed only among the offline selected cores for the *migrating* tasks. The motivation for this choice is that jobs of the *migrating* tasks execute on selected cores according to an execution pattern determined offline and by allowing work-stealing only among these cores, it is possible to decrease the average response-time of each migrating task, thus contributing to the overall decrease in the system responsiveness.

IV. Proposed Approach

In this paper we propose a semi-partitioned model with work-stealing for parallel tasks. The proposed approach consists of three phases referred to as *task assignment*, *offline scheduling*, and *online scheduling*. The intuitive idea behind each phase is summarized below:

1) *Task assignment.* In this phase, tasks are categorized according to their density. Then, a task-to-core assignment heuristic is applied to determine the non-migrating and the migrating tasks. The heuristic considers the demand of each core after each "new" task is assigned by using the demand bound function (see Equation 1). In this process, sequential tasks are evaluated first so that the capacity of the cores are filled as much as possible and thus, let the work stealing mechanism be exploited by parallel tasks in order to decrease their response times.
2) *Offline scheduling.* In this phase, the execution pattern of each migrating task (i.e., its execution sequence) is determined so as to meet all the timing requirements of the system. This process consists of mapping the jobs (frames) of each migrating task to the cores and form an execution sequence (by using the multiframe model) so that the schedulability on each core can be verified by using uniprocessor schedulability techniques.
3) *Online scheduling.* In this phase, the structure of each parallel task is considered and the work-stealing mechanism is applied among cores that share a copy of a migrating task. Specifically, an idle core with a copy of a migrating task can contribute to the execution of this task by stealing workload from another core and executing the (stolen) workload. Before stealing any workload, an admission control test is performed on the stealing core in order not to jeopardize the schedulability of the tasks already assigned to this core in Phase 2 (Offline Scheduling).

Note that by allowing work-stealing it is possible to have load-balancing among cores, and consequently a decrease in the average response time of the parallel tasks. Another important aspect of the proposed approach is that the initiative of stealing is always on the idle core, which is advantageous because instead of having a busy core pushing work into the deque of an idle core, the idle core does all this work leaving the busy core executing its tasks. Now, let us discuss the specifics of each phase.

A. Task assignment phase

In this phase tasks are categorized according to their density and then a first-fit decreasing (FFD) heuristic is implemented to partition tasks into cores. The categorization is as follows:

1) Light weight tasks (Class 1) — Tasks with $\lambda_i \leq 0.5$. Class 1 most likely consists of sequential tasks and tasks with a low degree of parallelism for which work-stealing is of little interest or the gain is small.[6]
2) Heavy weight tasks (Class 2) — Tasks with $\lambda_i > 0.5$. Class 2 most likely consists of tasks with a great degree of parallelism for which the gain relative to applying work-stealing is high.

After the task categorization, the FFD heuristic is applied to the sequential tasks in Class 1 and then in Class 2 in order to favor the assignment of sequential tasks to the cores as they cannot benefit from any parallelism. This process is repeated as long as the system is deemed schedulable. Then, the heuristic is applied to parallel tasks in Class 1 and then Class 2 in order to increase the probability of taking advantage of the work-stealing mechanism at runtime. All the tasks successfully assigned to the cores are referred to as *non-migrating tasks* and the remaining tasks, i.e., those that have not been successfully assigned by the FFD heuristic, are referred to as *candidate migrating tasks*. The system is then deemed schedulable if and only if an execution pattern is found for each candidate migrating task so that all the timing requirements are met.

If all tasks are assigned to cores by following the FFD heuristic, 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/or work-stealing as a fully partitioned assignment of the tasks to the available cores has been found. Using work-stealing in this situation would just help load-balancing the execution workload at the cost of allowing for unnecessary migrations among cores. If a task cannot be assigned to any core by following the FFD heuristic without jeopardizing its schedulability, then this task is classified as a candidate migrating task. Now, if a pattern of execution of jobs of this task to cores can be found in such a way that all the timing requirements are met, then this task

[5]A deque is a special type of queue that allows operations on both sides of the queue, i.e., it works as a stack and queue at the same time.

[6]The threshold for classifying tasks into heavy and light varies in the literature, nevertheless a density of 0.5 is usually regarded as a good threshold for classifying tasks.

is treated as a multiframe task and will potentially be subject to work-stealing at runtime if it presents a parallel section or instead is treated as a traditional multiframe task if it has a sequential behavior. A semi-partitioned scheduling approach is used for the migrating tasks which follow an assignment sequence defining the cores responsible for the execution of each of their jobs (frames).

At the end of this procedure if there is a task that has not succesfully been assigned to the cores, i.e., the task is neither a non-migrating nor a migrating task, then the system is deemed unschedulable in the current platform. We recall that for non-migrating tasks, work-stealing is forbidden so as to limit the overhead related to this operation. The intuitive idea behind our assignment policy is to increase the probability of tasks with a high degree of parallelism to be classified as migrating tasks as only these tasks will then benefit from work-stealing among selected cores at runtime and will have their average response time reduced.

B. 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 $\tau^{\pi_j} = \tau_{\mathrm{NM}}^{\pi_j} \cup \tau_{\mathrm{M}}^{\pi_j}$ where $\tau_{\mathrm{NM}}^{\pi_j}$ denotes the subset of non-migrating tasks and $\tau_{\mathrm{M}}^{\pi_j}$ denotes the subset of migrating tasks, assigned to π_j, respectively.

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 the load is less than 1, i.e.,

$$\mathrm{load}(\pi_j) \stackrel{\mathrm{def}}{=} \sup_{t \geq 0} \left\{ \frac{\mathrm{DBF}(\tau_{\mathrm{NM}}^{\pi_j}, t)}{t} \right\} \leq 1, \; \forall \pi_j \in \pi \qquad (2)$$

In Equation 2, $\mathrm{DBF}(\tau_{\mathrm{NM}}^{\pi_j}, t)$ is defined by using Equation 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. Once this operation is completed, uniprocessor schedulability analysis techniques can then be applied as follows.

Definition 1: The number of frames (taken from [11]). The number of frames k_i to consider for each migrating task τ_i is computed as follows:

$$k_i \stackrel{\mathrm{def}}{=} \frac{H}{T_i}, \text{ where } H \stackrel{\mathrm{def}}{=} \mathrm{lcm}_{\tau_j \in \tau}\{T_j\} \qquad (3)$$

In Equation 3, $\mathrm{lcm}_{\tau_j \in \tau}\{T_j\}$ denotes the *least common multiple* of the periods of all the tasks in τ. Goossens et al. [12] proved that this number of frames per migrating task is conservative and safe.

Definition 2: The execution pattern (taken from [11]). The job-to-core assignment sequence σ of the migrating task τ_i is defined through k_i sub-sequences as $\sigma \stackrel{\mathrm{def}}{=} (\sigma_1, \sigma_2, \ldots, \sigma_{k_i})$ where the sub-sequence σ_s (with $1 \leq s \leq k_i$) is given in turn by the m-tuple $\sigma_s = (\sigma_s^1, \ldots, \sigma_s^m)$. 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:

$$\sigma_s^j = \left\lceil \frac{s+1}{k_i} \cdot M[i,j] \right\rceil - \left\lceil \frac{s}{k_i} \cdot M[i,j] \right\rceil = 1 \qquad (4)$$

A direct advantage of this job-to-core assignment is its ability to considerably reduce the number of task migrations, as after the assignment each job knows exactly which core is responsible for its execution. For the sake of completeness, let us describe the algorithm that computes the execution pattern of each migrating task. Informally speaking, the algorithm tries to find the largest number of jobs that can be executed on a core until either all jobs are allocated or some jobs cannot be allocated. In this latter case, the task is deemed not schedulable. The algorithm works as follows:

1) In order to track the current job-to-core assignment, a matrix of integers $M[1 \ldots n, 1 \ldots m]$ is used 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 $1 \leq j \leq m$).
2) The matrix $M[i,1]$ is first initialized to k_i, i.e., all jobs of τ_i are assigned to the first core. Obviously, the result of the schedulability test will be not schedulable as it is a migrating task. Otherwise it would be assigned to this core during the FFD task-to-core assignment phase.
3) The number k_i is decremented by one unit (i.e., $M[i,1] := k_i - 1$) and an execution pattern for this number of k_i jobs is computed by applying Equation 4. For each specific execution pattern, the schedulability of the system is checked. The value of $M[i,1]$ is decremented as long as the task is not schedulable. At some point, say when $M[i,1] := k_i - \alpha_{[i,1]}$ (with $1 < \alpha_{[i,1]} < k_i$) and the system becomes schedulable, $M[i,1]$ jobs of task τ_i are assigned to this core, an execution pattern which does not jeopardize the schedulability of the core is found and the algorithm moves on to the next core.
4) The number of jobs just allocated ($M[i,1]$) is removed from k_i and the result is considered as the new value of k_i in Equation 4 for this new core, i.e., $k_i := k_i - M[i,1]$. This process is iterated accross all the cores until all the jobs are assigned to a core. Otherwise, the algorithm keeps reducing the value of k_i in step 3 until a number of jobs (eventually zero) can be accommodated in the current core.

At the end of these steps, if all the jobs of τ_i are not allocated, then τ_i is not schedulable as a migrating task and thus the system is deemed not schedulable.

C. Online scheduling phase

This phase is meant to take advantage of the multicore platform and the execution behavior of the migrating parallel tasks at runtime. This is performed by using the work-stealing algorithm between cores that share a copy of these tasks (referred to as "selected cores") for the execution of the parallel sections of the migrating tasks. We recall that this operation is used to reduce their average response times. Below we state the four necessary rules (R_1 to R_4) that we implemented 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 in the ready state awaiting for execution;

R_2: Idle selected cores are allowed to steal workload (sub-tasks) *only* from the deque of another selected core;

R_3: When stealing workload from the deque of another selected core, the idle core must always steal the highest priority parallel sub-task from the list of deques 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 choosing 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 by only allowing work-stealing to occur between selected cores 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 these cores after the execution of the first job in the core (recall that these cores share migrating tasks and have a local copy of them). Whenever a core performs a steal, data is fetched from the memory of another core which is in a certain extent equivalent to a migration, however in this case the task is already loaded in the core and only input data is fetched. 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 having to migrate between cores as it would happen in a global approach.

V. Example of the approach

This section illustrates the concepts of the proposed approach as discussed in the previous section. 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 2a 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:

(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 phenomenon occurs again at time $t = 7.5$. Figure 2b illustrates the resulting schedule, the system is schedulable.

VI. Schedulability Analysis

In this section, we derive the schedulability analysis of a set of constrained-deadline fork-join tasks onto a homogeneous multicore platform. A modification as explained in Section IV of the semi-partitioned model is adopted 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. The schedulability analysis of the task set must be performed in the three phases of the proposed approach, as follows.

(1) *Task assignment phase:* during this phase, the schedulability of the system is performed by applying the traditional demand bound function based analysis to non-migrating tasks, i.e., by using Equation 2.

(2) *Offline scheduling phase*: during this phase, we should make sure that the additional workload to each core related to the assignment of the migrating tasks (i.e., the workload related to $\tau_{\mathrm{M}}^{\pi_j}$ for core π_j) 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 [11]. 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 $s \stackrel{\text{def}}{=} \lfloor \frac{t}{k_i \cdot T_i} \rfloor$; 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 $nb_i(t) = \lfloor \frac{(t \bmod (k_i \cdot T_i)) - D_i}{T_i} \rfloor + 1$. The corresponding workload is $w_i^j = \max_{c=0}^{k_i - 1} (\sum_{\eta=c}^{c+nb_i(t)-1} C_{i, \eta \bmod k_i})$. It follows that an upper-bound on the total workload associated to the migrating task τ_i on core π_j is computed as:

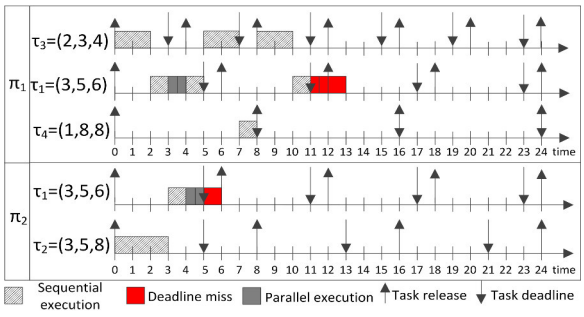
$$\mathrm{DBF}_j(\tau_i, t) \stackrel{\text{def}}{=} s_i \cdot \ell_i^j \cdot C_i + w_i^j \tag{5}$$

Consequently,

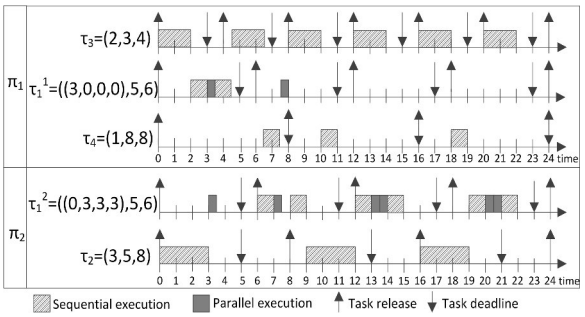
$$\mathrm{DBF}(\tau_{\mathrm{M}}^{\pi_j}, t) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau_{\mathrm{M}}^{\pi_j}} \mathrm{DBF}_j(\tau_i, t) \tag{6}$$

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

$$\mathrm{load}(\pi_j) \stackrel{\text{def}}{=} \sup_{t \geq 0} \left\{ \frac{\mathrm{DBF}(\tau_{\mathrm{NM}}^{\pi_j}, t) + \mathrm{DBF}(\tau_{\mathrm{M}}^{\pi_j}, t)}{t} \right\} \leq 1, \ \forall \pi_j \in \pi \tag{7}$$



(a) Schedule under fully partitioned EDF



(b) Schedule using WS-based EDF

Figure 2: Illustrative example of the proposed approach.

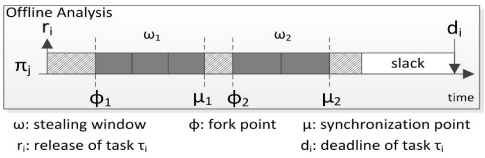


Figure 3: Result after the offline analysis

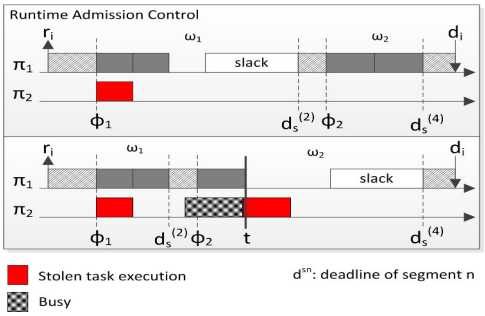


Figure 4: Example of work-stealing and intermediate deadline computation

(3) *Online scheduling phase*: In this phase the schedulability analysis obtained in phase 2 must be extended in order to take into account the potential extra workload related to the work-stealing mechanism allowed among cores sharing copies of migrating tasks. Figure 3 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) and synchronization points (μ_1 and μ_2). The job offers a slack[7] as the offline scheduling phase has been performed without jeopardizing the schedulability of the core. The intuitive idea behind this phase is to 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 (parallel sub-task from the deque of core A) before a time instant which may affect the scheduling decisions initially taken on core B. These time instants are actually the synchronization points of the parallel segments (μ_1 and μ_2 in the example). Such a time instant is referred to as an *intermediate deadline* for the stolen sub-task. To compute this intermediate deadline for each stealing window, we can take advantage of the slack available for each job, i.e., the intermediate deadline of the n^{th} parallel segment can be computed as follows:

$$d_s^{(n)} \stackrel{\text{def}}{=} \phi_n + m_s * c_{s_i^{(n)}} + \text{slack}(\phi_n) \quad (8)$$

In Equation 8, 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 this segment, $c_{s_i^{(n)}}$ denotes its worst-case execution time, and $\text{slack}(\phi_n)$ represents the slack of the job at time ϕ_n. Figure 4 illustrates the computation of the intermediate deadlines for the stealing windows using Equation 8. In this figure, core π_2 can steal sub-tasks from core π_1 in the first stealing window ω_1 and in the second stealing window ω_2. The intermediate deadline for the sub-tasks that may be stolen in ω_1 is computed by using Equation 8 and the result is $d_s^{(2)}$. As the sub-task execution is less than the intermediate deadline, the stealing operation may occur safely. In the same manner, the intermediate deadline for the sub-task that may be stolen in ω_2 is computed and the result is $d_s^{(4)}$. For the same reasons as for the first sub-task, the stealing operation may occur safely.

Now we have everything we need to decide on the actual occurrence of each stealing operation. Before core B can steal a sub-task from core A, an admission control test has to be performed on core B. As mentioned previously, there should be no pending execution workload on core B (see rule R_1). If this is the case, then two possible scenarios can occur during the stealing operation of 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 in this stealing window provided that the execution of the stolen sub-task meets its intermediate deadline; 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; (2.2) some releases have their deadline after $d_s^{(n)}$: In this case, no

[7]The slack of a job is the maximum amount of time that the remaining execution of the job can be delayed on its activation to complete within its deadline [8].

guarantees can be provided on the schedulability of the system as the stolen job may modify the scheduling decisions initially taken on core B. In this latter case we decide not to perform the stealing operation.

VII. Simulation results

This section reports on the results of the simulation of our proposed approach on a set of synthetic and randomly generated task sets. To this end achievement, we conducted an extensive set of experiments on systems with implicit deadline tasks (i.e., $D_i = T_i$ for every task τ_i). As the main focus of our work is the improvement obtained on the average response time of each task, we strongly believe that this assumption is not restrictive and does not affect the overall behavior of the proposed approach (i.e., for constrained deadline tasks). The simulation environment is described as follows.

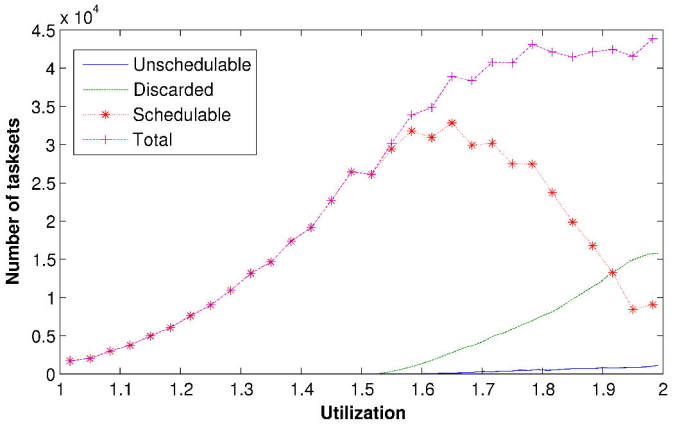


Figure 5: Profile of the task generation for two cores

Considered platform. We consider a platform consisting of two or four homogeneous cores (i.e, all cores have the same computing capabilities and are interchangeable). This allows us to isolate the gain of our approach over a fully partitioned task-to-core assignment.

Tasks generation. Each task τ_i is of type 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 (i.e., $k \in [3, 5, 7]$), 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 interval $[1, \text{max\_Ci\_subtsk}]$ where $\text{max\_Ci\_subtsk} = 2$ for performance reasons. From the generated values we compute the worst-case execution time of each task ($C_i = \sum_{\forall\, \text{subtsk} \in \tau_i} \text{C}_{\text{i,subtsk}}$). 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 (recall that $U_i = \frac{C_i}{T_i}$) 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 $\text{C}_{\text{i,subtsk}}$[8].

The above procedure is repeated until 1000 task sets with migrating tasks are generated for two cores and four cores.

In Figure 5 it is possible to see the total number of task sets generated ($Y - axis$) as a function of the system utilization ($X - axis$). Four lines are represented: in dashed blue the unschedulable task sets; in dashed green the number of discarded task sets; in dashed red the schedulable task sets (with migrating and non-migrating tasks); and finally, in dashed purple the total number of generated task sets. The discarded task sets line presents the number of task sets that were rejected from the analysis due to a high number of k_i frames. Indeed, the number of frames is tightly associated to the pattern of execution of each migrating task. Therefore, this pattern needs to be computed in order to assess the runtime schedule. If this number is too large, then the complexity of the computation of the patterns also increases, which leads to higher computation times. Therefore, we opted to discard task sets with $k_i > 10$. In our opinion this parameter does not compromise the results and their conclusions.

Considered metrics. In order to evaluate our proposed approach, we measure the gain obtained for each generated task set in terms of average worst-case response times by applying work-stealing and without applying work-stealing to the migrating tasks. That is, 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 is denoted as Approach-NS; and the approach that applies the work-stealing mechanism among the selected cores is 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 summing 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:

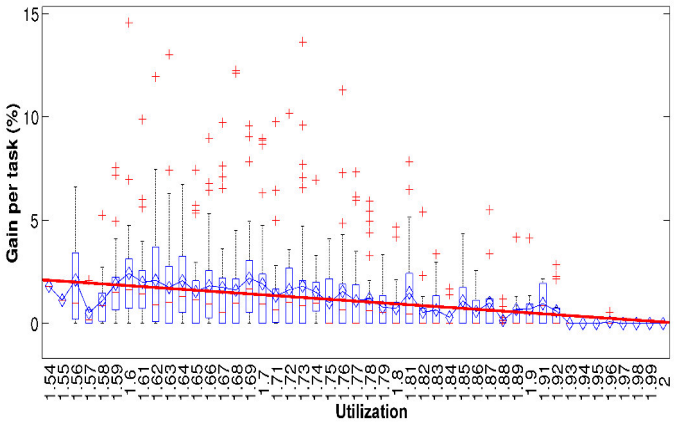
$$AV_{\tau_i} = \frac{AV^{NS}_{\tau_i} - AV^{S}_{\tau_i}}{AV^{NS}_{\tau_i}} \cdot 100 \tag{9}$$

In Equation 9 ($AV^{NS}_{\tau_i}$ 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 as follows.

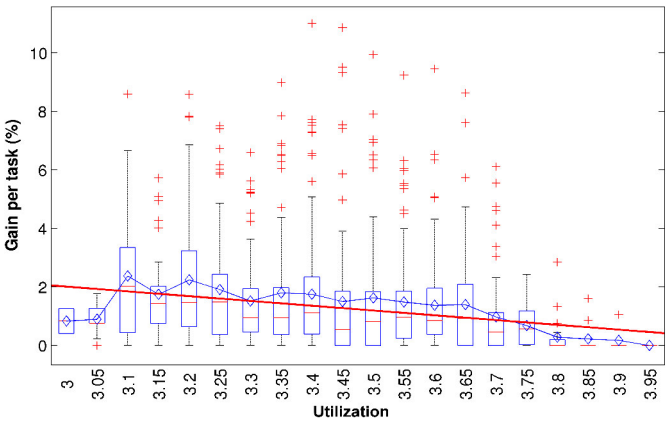
$$AV_\tau = \frac{1}{|\tau|} \cdot \sum_{\tau_i \in \tau} AV_{\tau_i} \tag{10}$$

Figures 6a and 6b illustrate the average gain for two and four cores respectively.

[8] 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. Consequently, the higher the hyperperiod of the task set. By limiting $\text{C}_{\text{i,subtsk}}$ we are also limiting the amount of time we need to generate the schedule.



(a) Average execution time gain for two cores



(b) Average execution time gain for four cores

Figure 6: Simulation results

Interpretation of the results. The improvement in terms of average response-time per task (in %) is grouped by utilization — see Figure 6a and Figure 6b, when using Approach-S over Approach-NS. For each 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. Moreover, 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 high utilizations (above 1.56), there starts to be a clear illustration of the proposed approach. In the best case, this gain reaches nearly 15% of the average response-time per task, which is non negligible. This gain is obtained around 1.60 of utilization. By increasing the utilization of the task sets the gain per task starts to decrease. This is expected due to the increasing lack of idle time available for stealing. The trend shows that above 1.93 of utilization, the work-stealing mechanism becomes of little interest. This is explained by the fact that the total workload on each core in 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 1.54 are not included in the plot as they do not contain any migrating task.

Considering four cores, it can be seen that the pattern is similar to the one depicted using two cores. This behavior suggests that work-stealing is useful for task sets with migrating tasks with utilizations that span from the lowest possible utilization for task sets with migrating tasks up to the platform capacity. Closer to this upper limit, the benefit of stealing is limited. 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. From this, we conjecture that the proposed approach is scalable with an increase in the number of cores.

Overheads of the approach. The objective of this work is to show 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 neglectable 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 subtasks 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.

VIII. Conclusion and Future Work

In this paper we combined techniques that allow us to schedule fined grained parallel real-time 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, then it reduces the migration overhead which has 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 are 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).

Concerning the future work, we would like to continue pursuing the exploration of this idea of work-stealing in real-time settings as it appears to be very promising. Moreover, we would like to see the variation in terms of average response-times by applying different allocation heuristics, and increasing number of cores as a way to measure the scalability of the proposed approach.

ACKNOWLEDGMENTS

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); also by FCT/MEC and ERDF through COMPETE (Operational Programme 'Thematic Factors of Competitiveness'), within project(s) FCOMP-01-0124-FEDER-020447 (REGAIN); also by FCT/MEC and the EU ARTEMIS JU within project(s) ARTEMIS/0001/2013 - JU grant nr. 621429 (EMC2); also by the European Union under the Seventh Framework Programme (FP7/2007-2013), grant agreement n^{o} 611016 (P-SOCRATES); also by FCT/MEC and the ESF (European Social Fund) through POPH (Portuguese Human Potential Operational Program), under PhD grant SFRH / BD / 88834 / 2012.

REFERENCES

[1] J. H. Anderson, V. Bud, and U. C. Devi. An edf-based scheduling algorithm for multiprocessor soft real-time systems. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, ECRTS '05, pages 199–208, Washington, DC, USA, 2005. IEEE Computer Society.

[2] H. Aydin and Q. Yang. Energy-aware partitioning for multiprocessor real-time systems. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, pages 9 pp.–, April 2003.

[3] B. Bado, L. George, P. Courbin, and J. Goossens. A semi-partitioned approach for parallel real-time scheduling. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS, pages 151–160, New York, NY, USA, 2012. ACM.

[4] S. Baruah, D. Chen, S. Gorinsky, and A. Mok. Generalized multiframe tasks. *Real-Time Syst.*, 17(1):5–22, July 1999.

[5] S. Baruah, A. Mok, and L. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Real-Time Systems Symposium, 1990. Proceedings., 11th*, pages 182–190, Dec 1990.

[6] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, sep 1999.

[7] V. Bonifaci, A. Marchetti-Spaccamela, S. Stiller, and A. Wiese. Feasibility analysis in the sporadic dag task model. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 225–233, July 2013.

[8] G. C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer Publishing Company, Incorporated, 3rd edition, 2011.

[9] H. S. Chwa, J. Lee, K.-M. Phan, A. Easwaran, and I. Shin. Global edf schedulability analysis for synchronous parallel tasks on multicore platforms. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 25–34, July 2013.

[10] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, oct 2011.

[11] F. Dorin, P. M. Yomsi, J. Goossens, and P. Richard. Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms. *CoRR*, abs/1006.2637, 2010.

[12] J. Goossens, P. Richard, M. Lindström, I. I. Lupu, and F. Ridouard. Job partitioning strategies for multiprocessor scheduling of real-time periodic tasks with restricted migrations. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS '12, pages 141–150, New York, NY, USA, 2012. ACM.

[13] T. Instruments. Keystone architecture. www.ti.com/lit/ug/sprugw8c/sprugw8c.pdf, Apr. 2015.

[14] J. Kang and D. Waddington. Load balancing aware real-time task partitioning in multicore systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2012 IEEE 18th International Conference on*, pages 404–407, Aug 2012.

[15] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Real-Time Systems, 2009. ECRTS '09. 21st Euromicro Conference on*, pages 249–258, July 2009.

[16] K. Lakshmanan, S. Kato, and R. R. Rajkumar. Scheduling parallel real-time tasks on multi-core processors. In *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*, RTSS '10, pages 259–268, Washington, DC, USA, 2010. IEEE Computer Society.

[17] D. Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000.

[18] J. Li, K. Agrawal, C. Lu, and C. Gill. Analysis of global edf for parallel tasks. In *Real-Time Systems (ECRTS), 2013 25th Euromicro Conference on*, pages 3–13, July 2013.

[19] C. Maia, M. Bertogna, L. Nogueira, and L. M. Pinho. Response-time analysis of synchronous parallel tasks in multiprocessor systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 3:3–3:12, New York, NY, USA, 2014. ACM.

[20] C. Maia, L. Nogueira, and L. Pinho. Scheduling parallel real-time tasks using a fixed-priority work-stealing algorithm on multiprocessors. In *Industrial Embedded Systems (SIES), 2013 8th IEEE International Symposium on*, pages 89–92, June 2013.

[21] A. Marowka. Parallel computing on any desktop. *Commun. ACM*, 50:74–78, September 2007.

[22] A. Mok and D. Chen. A multiframe model for real-time tasks. *Software Engineering, IEEE Transactions on*, 23(10):635–645, Oct 1997.

[23] M. Qamhieh, L. George, and S. Midonnet. A stretching algorithm for parallel real-time dag tasks on multiprocessor systems. In *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, RTNS '14, pages 13:13–13:22, New York, NY, USA, 2014. ACM.

[24] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 217–226, Nov 2011.

[25] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, Sept. 2007.