Reducing Preemptions and Migrations in EKG

Geoffrey Nelissen\textsuperscript{11}, Shelby Funk\textsuperscript{§}, Joël Goossens\textsuperscript{†}

\textsuperscript{†}PARTS Research Center
Université Libre de Bruxelles (ULB)
Brussels, Belgium

\textsuperscript{§}Department of Computer Science
University of Georgia
Athens, GA, USA

Abstract—EKG is a multiprocessor scheduling algorithm which is optimal for the scheduling of real-time periodic tasks with implicit deadlines. It consists in a semi-partitioned algorithm which adheres to the deadline partitioning fair (DP-Fair) theory. It was shown in recent studies that the division of the time in slices bounded by two successive deadlines and the systematic execution of migratory tasks in each time slice inherent in DP-Fair algorithms, significantly reduce the practicality of EKG. Nevertheless, its semi-partitioned approach allows to bound the number of migrating tasks and increases the locality of the tasks in memories, thereby lowering the time overheads imposed by task preemptions and migrations. Hence, we propose two techniques with the aim of reducing the amount of preemptions and migrations incurred by the system when scheduled with EKG, while maintaining the advantages of its semi-partitioned approach. The first improvement consists in a swapping algorithm which exchanges execution time between tasks and time slices. The second one aims at decreasing the number of time slices needed to ensure that all job deadlines are respected. Both have a strong impact on the number of preemptions and migrations while keeping the optimality of EKG.

I. INTRODUCTION

The scheduling of periodic real-time tasks on uniprocessor platforms has been extensively studied over the years. Nowadays, well mastered and high performing algorithms exist. For instance, the earliest deadline first (EDF) algorithm is a simple optimal scheduling algorithm on uniprocessor platforms [1]. That is, EDF respects all task deadlines for any schedulable task set. Unfortunately, when straightforwardly extended for the multi-processor real-time scheduling, EDF (and all other optimal uniprocessor schedulers) cannot guarantee that all task deadlines will be respected. Nevertheless, the simplicity of EDF on the one hand, and its performances in terms of preemptions on the other hand, encourage the real-time community to keep EDF as a model for the design of new scheduling algorithms [2]–[6].

Over the last two decades, numerous optimal multiprocessor scheduling algorithms for periodic tasks have been proposed [7]–[15]. Each new result attempts to outperform previous algorithms in terms of preemptions, migrations, number of scheduling points or complexity. Indeed, many studies state that the run-time overheads caused by these various factors, dramatically impact the practicality of optimal multiprocessor scheduling algorithms [16]–[18].

The first optimal scheduling algorithms for multiprocessor platforms were based on the notion of proportionate fairness [7], [8] and early-release fairness [9]. With these solutions each task is scheduled at a rate proportional to its weight in the system. Although these algorithms are optimal, they cause an extensive amount of preemptions and migrations during the schedule. Hence, they were rapidly followed by boundary fair or deadline partitioning fair (DP-Fair) algorithms [10]–[13] which perform better in terms of preemptions, migrations and number of scheduling points.

Recently, [11] formalized the DP-Fair theory explaining how the optimality could be reached on multiprocessor platforms by ensuring the fairness for all tasks at the deadlines of jobs released in the system. In this approach, the time is divided in time slices. All tasks are assigned a local execution time in each time slice which is determined so as to ensure that all deadlines will be met.

In 2006, Andersson et al. developed the semi-partitioned algorithm EKG which categorizes tasks as migratory or non-migratory [14]. Migratory tasks are shared between two processors and scheduled according to the DP-Fair approach. On the other hand, non-migratory tasks are statically assigned to one processor and scheduled under an EDF scheduling policy. The EKG algorithm seems very promising in terms of reducing preemptions and migrations [11]. However, recent studies showed that the systematic schedule of migratory tasks in each time slice significantly increases the number of preemptions and migrations and hence has a negative impact on the schedulability of task systems on real computing platforms [18].

Most recent optimal scheduling algorithms such as RUN [6] or U-EDF [5], [19], are not based on the notion fairness but rather extend the mechanisms lying behind the uniprocessor algorithm EDF. Note that these algorithms perform drastically better than EKG in terms of average preemptions and migrations per jobs and can even compete with fully partitioned algorithms [5], [6]. However, unlike EKG for which there is a limited number of tasks that can migrate between two different processors, all tasks can migrate between all processors with RUN and U-EDF. This particularity increases the preemption and migration overheads by lowering the code and data locality in memories. Hence, our goal is to improve EKG so that its average number of preemptions and migrations get close to the results of RUN and U-EDF while keeping the advantages of its semi-
This Research. In this paper, we propose two techniques to address the preemption and migration overheads in EKG. We first present a swapping algorithm which increases (or decreases) the time reserved for migratory tasks in each time slice, thereby suppressing migratory tasks (and associated runtime overheads) from time slices where their execution is not required to keep a correct schedule. Then, we propose a set of rules determining if a time slice can be extended without any risk of missing a task deadline. Hence, by reducing the number of time slices, we are able to decrease the associated preemptions, migrations and scheduling points. Finally, we propose simulation results illustrating the performances of our two improvement techniques of the optimal EKG algorithm.

II. MODEL

We consider the scheduling of a set \( \tau \) composed of \( n \) periodic tasks with implicit deadlines on \( m \) identical processors. Each task \( \tau_i \) in the set \( \tau \) is characterized by a period \( T_i \) and a worst case execution time \( C_i \). That is, each task \( \tau_i \) releases an infinite number of jobs. The arrivals of two successive jobs of \( \tau_i \) are separated by \( T_i \) time units and each job must be executed for \( C_i \) time units before the next job arrival. We denote the utilization of \( \tau_i \) as \( U_i \). Formally, it measures the proportion of time that \( \tau_i \) must execute on average to meet its deadline. Moreover, the system utilization \( U \) is defined as the sum of all task utilizations (i.e., \( U = \sum_{i=1}^{n} U_i \)). This gives the minimum computational capacity that must be provided by the platform to meet all the task deadlines.

We say that a job is active at time \( t \) if it has been released no later than \( t \) and has its deadline after \( t \), i.e., the instant \( t \) lies between the arrival time of the job and its deadline. Since we are working with periodic tasks with implicit deadlines, each task has one (and only one) active job at any time \( t \). We can therefore write without ambiguity that the deadline \( d_i(t) \) of the task \( \tau_i \) at time \( t \) is the deadline of the current active job of \( \tau_i \) at time \( t \). Similarly, we denote the remaining execution time of the current active job of \( \tau_i \) at time \( t \) as \( r_i(t) \).

In the remainder of this paper, we will assume that, at any instant \( t \), the set \( \tau \) is ordered according to the deadlines of the tasks at time \( t \). That is, for any two tasks \( \tau_x \) and \( \tau_y \), if \( x < y \) then \( d_x(t) \leq d_y(t) \). Furthermore, for theoretical reasons, we assume that the platform is fully utilized, i.e., we have a total utilization \( U = m \). However, if this last assumption should not be respected in the actual application, we simply model the processors idle time with an “idle” task \( \tau_{idle} \) such that \( U_{idle} = m - U \) and \( T_{idle} = +\infty \) (assuming \( \sum_{\tau_i \in \tau} U_i \geq m - 1 \)). That is, we never miss \( \tau_{idle} \)'s deadline and it always has the smallest priority regarding to the EDF algorithm. Since this additional task \( \tau_{idle} \) represents the idle time of the processors, whenever the task \( \tau_{idle} \) is scheduled to be executed on a processor in the theoretical schedule, this means that the processor actually remains idle in the real schedule. Again, \( \tau_{idle} \) is only introduced for theoretical reasons but does not impact the resulting schedule produced by EKG.

III. OVERVIEW OF THE EKG ALGORITHM

EKG, first proposed by Andersson and Tovar in [14], is the short-hand notation for EDF with task splitting and \( k \) processors in a Group. As its name implies, it splits the platform into several clusters through the definition of a parameter \( k \) (with \( k \leq m \)). The platform is divided in \( \lceil \frac{m}{k} \rceil \) clusters containing \( k \) processors and one cluster composed of \( (m - \lceil \frac{m}{k} \rceil \times k \) processors. A bin-packing algorithm\(^2\) is then used to partition the tasks amongst the clusters so that the total utilization on each cluster is not greater than its number of constituting processors. After the partitioning of \( \tau \), EKG works in two different phases. First, the tasks of each cluster are assigned on the processors of this cluster. Then, the tasks are scheduled in accordance with this assignment.

It was proven in [14] that EKG ensures a utilization bound of \( \left( \frac{k}{k + 1} \times m \right) \) when \( k < m \), and is optimal for the scheduling of periodic tasks with implicit deadlines when \( k = m \).

Also, in order to minimize the number of preemptions and migrations, if \( k < m \) then every task \( \tau_i \) with a utilization \( U_i \) not smaller than \( \frac{k}{k + 1} \) is scheduled independently of the other tasks on its own processor.

In the remainder of this paper, we will assume that there is only one cluster in the system. However, if there should be multiple clusters, the same reasoning could be applied on each individual cluster without any variation.

We now present the assignment and scheduling phases that EKG utilizes in each cluster.

\(^2\)In the original version of EKG, a next fit heuristic was used to partition the task set [14]. However, any reasonable bin packing algorithm such as next fit, first fit, worst fit or best fit, does work.
a) **Phase 1:** The assignment follows a slight variation of the next fit heuristic so that the utilization on each processor is exactly 1. That is, tasks are assigned to a processor \( \pi_j \) as long as the capacity \( c_j \) on this processor is not exhausted. It then starts assigning tasks on the next processor \( \pi_{j+1} \). Also, whenever the assignment of a task on \( \pi_j \) would cause the utilization of \( \pi_j \) to exceed 1, the task is “split” between \( \pi_j \) and the next processor \( \pi_{j+1} \).

Formally, let \( u_{i,j} \) denote the proportion of \( \tau_i \)'s utilization assigned on processor \( \pi_j \). We define \( S_j \) as the set of tasks entirely assigned on \( \pi_j \). That is,

\[
S_j = \set{ \tau_i \in \tau \mid u_{i,j} = U_i }.
\]

If we assume that \( \tau_i \) is the next task to assign, and \( \pi_p \) is the first processor with some remaining capacity (i.e., \( c_p > 0 \) and \( c_j = 0 \) for all \( j < p \)), if \( c_p \geq U_i \) then \( S_{p} = S_{p} \cup \set{\tau_i} \).

Otherwise, \( \tau_i \) becomes a migratory task, denoted \( M_{p} \), which is split into two subtasks \( M'_{p} \) with a utilization factor \( U_{M'_{p}} = c_p \), and \( M_{p} \) with a utilization \( U_{M_{p}} = U_{i} - c_p \) (see Fig. 1). \( M'_{p} \) is assigned to the processor \( \pi_p \) and \( M_{p} \) is assigned to \( \pi_{p+1} \). Hence, the utilization of \( \pi_p \) is exactly 1.

For instance, in Fig. 1, \( S_{p-1} = \set{\tau_b, \tau_c, \tau_d} \) and \( \tau_e \) is split in \( M'_{p-1} \) and \( M_{p} \).

Note that at most one migratory task assigned to each couple of processors \( \pi_j \) and \( \pi_{j+1} \), and therefore only two different migratory tasks are executed on each processor.

The set of tasks \( S_j \) is henceforth called the super task of processor \( \pi_j \) and the tasks in \( S_j \) are referred to as the component tasks of \( S_j \) or the non-migratory tasks of \( \pi_j \). The utilization of \( S_j \) is denoted \( U_{S_j} = \sum_{\tau_i \in S_j} U_i \).

b) **Phase 2:** After the assignment, EKG schedules the migratory tasks and super tasks using a Deadline Partitionning Fair (DP-Fair) technique [11]. That is, the time is divided in slices bounded by two successive job deadlines and each migratory task and super task is executed for a time proportional to its utilization within each time slice. Hence, if we let \( b_k \) denote the \( k^{th} \) time instant from the beginning of the schedule where at least one job has its deadline, then we define the \( k^{th} \) time slice \( TS^k \) as the time interval extending from the instant \( b_{k-1} \) to \( b_k \) (with \( b_0 = 0 \)). We say that \( b_{k-1} \) and \( b_k \) are the boundaries of \( TS^k \).

The length of \( TS^k \) is denoted \( L^k \) as \( b_k - b_{k-1} \). In each time slice of length \( L^k \) and on each processor \( \pi_j \), the super task \( S_j \) and the subtasks \( M'_{j} \) and \( M''_{j-1} \) execute for \( t_{S_j}^k = U_{S_j} \times L^k \), \( t_{M'_{j}}^k = U_{M'_{j}} \times L^k \) and \( t_{M''_{j-1}}^k = U_{M''_{j-1}} \times L^k \) time units, respectively. We say that \( t_{S_j}^k \) is the local execution time of the super task \( S_j \) in time slice \( TS^k \) (and similarly for the migratory subtasks \( M'_{j} \) and \( M''_{j-1} \)).

Fig. 1 shows an example of the schedule produced by EKG.

Whenever, a super task \( S_j \) is scheduled on processor \( \pi_j \), its component task with the earliest deadline is actually executed on \( \pi_j \). EKG can therefore be seen as a hierarchical scheduling algorithm. On the one hand, it uses a DP-Fair approach to schedule the super tasks and, on the other hand, it uses EDF to decide which component task to execute on the processor.

Notice that on Fig. 1, in each time slice, the schedule produced by EKG is the “mirror” version of the schedule in the previous time slice. This mirroring technique which was proposed together with the EKG algorithm in [14], reduces the number of preemptions and migrations by keeping running the same task on each processor after a boundary.

Due to this mirroring mechanism, the first task executing on each processor \( \pi_j \) in each time slice \( TS^k \) is not always the same. It alternates between the migratory subtasks \( M''_{j-1} \) and \( M'_{j} \) (see Fig. 1). In the remainder of this paper, we will refer to the first task scheduled to execute in the time slice \( TS^k \) on processor \( \pi_j \) as the migrate-in task (denoted by \( M'^{in}_{j,k} \)). Hence, the migrate-in task \( M'^{in}_{j,k} \) of processor \( \pi_j \) refers to the subtask \( M''_{j-1} \) when we consider odd numbered time slices and refers to \( M'_{j} \) otherwise. Similarly, the last task executed on processor \( \pi_j \) in time slice \( TS^k \) is referred to as the migrate-out task (denoted \( M'^{out}_{j,k} \)) and alternates between \( M'_{j} \) and \( M''_{j-1} \) either.

**IV. REDUCING TASK PREEMPTIONS AND MIGRATIONS**

As shown on Fig. 1, there is one migration per time slice between the processors \( \pi_j \) and \( \pi_{j+1} \) for each migratory task \( M_j \). Furthermore, each migratory task \( M_j \) causes up to two preemptions (one for \( M'_{j} \) and another for \( M''_{j} \)) per time slice. On the other hand, because EDF is used to schedule the component tasks of any supertask \( S_j \), the tasks included in \( S_j \) do not cause any preemption between two successive job arrivals (which correspond to the time slices boundaries) [14]. Additionally, a supertask \( S_j \) never migrates. Hence, the number of preemptions and migrations is proportional to the number of time slices. Moreover, most of the preemptions and all the migrations are caused by migratory tasks. Therefore, the number of preemptions and migrations can be reduced by two means:

1) reduce the number of time slices by skipping some boundaries;
2) remove the execution of the migratory tasks (or subtasks) from as many time slices as possible. This can be done by exchanging execution time between tasks and time slices.

The second objective is formalized in Section IV-A and the first one in Section IV-B. Then, a combination of both solutions is rapidly investigated in Section IV-C. Note that we will skip a deadline or perform a swap of execution time between tasks only if it does not jeopardize the correctness of the schedule previously built by EKG (i.e., all job deadlines are still respected). Hence, our boundary skipping and execution swapping algorithms do not affect the optimality of EKG.

**A. Swapping Execution Times between Tasks**

The swapping algorithm is applied at each time \( t \) corresponding to a boundary \( b_k \) (i.e., at the beginning of each time slice \( TS^{k+1} \)). We consider the current time slice \( TS^t \) as the time interval extending from \( t \) to \( b_t = d_i(t) \) (recall that the tasks are ordered in an increasing deadline order).

The algorithm follows four different rules:
**Rule 1:** Maximize the execution of $M''_j$ on processor $\pi_j$ in the current time slice $TS^c$.

**Rule 2:** Minimize the execution of $M'_j$ on processor $\pi_j$ in the current time slice $TS^c$.

**Rule 3:** Avoid any intra-job parallelism.

**Rule 4:** Ensure that every task will be able to complete its execution before its deadline.

The main idea of Rule 1 is to complete the execution of $M''_j$ as early as possible, while on the other hand, Rule 2 aims at delaying the execution of $M'_j$ as much as possible. The goal is that $M''_j$ only appears in the earlier time slices after each of its new job arrival, and to schedule $M'^{out}$ in the later time slices just prior to each of its job deadlines. Ideally, their executions can be eliminated altogether in some time slices (thereby eliminating the associated preemptions and migrations). Furthermore, reducing $M'_j$’s execution in the current time slice increases the interval in which $M''_j$ can execute in $TS^c$ giving the opportunity to $M''_j$ to complete earlier (and therefore not appear in next time slices anymore).

Assuming that a task set $\tau$ is schedulable under EKG, Rules 3 and 4 make sure that the schedule remains correct after executing the swapping algorithm (i.e., all deadlines are respected without intra-job parallelism). That is, we do not perform a swap if it could impact the correctness of $\tau$’s schedule.

Fig. 2 shows how the schedule might change after swapping. In this example, both the number of preemptions and migrations have been reduced by 2 in the first time slice. By reapplying the swapping algorithm at each boundary $b_k$, the number of preemptions and migrations can be drastically reduced on the entire schedule.

1) **Principles Ensuring Correctness:** Before we formally describe our swapping algorithm in the next section, we first present some principles that we must follow in order to ensure that we never violate Rule 4.

**Principle 1:** For any migratory task $M_j$, we cannot swap execution time between $TS^c$ and any time slice subsequent to the deadline of the current active job of the migratory task (i.e., a time slice $TS^k$ such that $b_k > d_M(t)$). Indeed, the time reserved for $M_j$ after $d_M(t)$ is dedicated to the execution of “future” jobs of $M_j$ that are not yet active in $TS^c$.

**Principle 2:** If we increase (respectively decrease) the local execution time $\ell^c_i$ of a task $\tau_i$ by $\Delta_i$ time units in the current time slice $TS^c$, we have to decrease (respectively increase) the local execution time $\ell^k_i$ by the same quantity $\Delta_i$ in another time slice $TS^k$ such that $k > c$ and $TS^k$ is before $d_i(t)$ (according to Principle 1). That is, the time reserved for the execution of a task must remain constant for the whole schedule.

Principles 1 and 2 give us a straightforward approach for enforcing all migratory tasks’ adherence to Rule 4. However, supertasks require more thought. Indeed, a supertask $S_j$ is a collection of tasks. Hence, each component task $\tau_i$ has its own deadline $d_i(t)$ and its own remaining execution time $r_i(t)$. We must therefore ensure that $S_j$ has enough time reserved on $\pi_j$ to fulfill the execution of all its component tasks before their respective deadlines. Moreover, if $d_i(t)$ is the deadline of the current job of a component task $\tau_i$ of $S_j$, the part of the time reserved for $S_j$ in the time slices subsequent to $d_i(t)$ is dedicated to the execution of “future” jobs of $\tau_i$. These jobs are not yet active within the interval $[t, d_i(t))$. Therefore, we must be sure that we keep enough time available after $d_i(t)$ to execute these future jobs.

Let $F^k_j$ be the tasks in $S_j$ with a deadline no later than $b_{k-1}$ (i.e., $F^k_j \triangleq \{ \tau_i \in S_j \mid d_i(t) \leq b_{k-1} \}$). The time reserved in $TS^k$ (i.e., the time slice extending from $b_{k-1}$ to $b_k$) is dedicated to future jobs of $\tau_i$ (because $\tau_i$’s current job has its deadline prior to $b_k$). Hence, the local execution time $\ell^k_j$ allocated to the supertask $S_j$ within the time slice $TS^k$ can be separated into two parts – namely, the time reserved for the future jobs (i.e., jobs of tasks in $F^k_j$), denoted $f^k_j$, and the time allocated to active jobs at time $t$ (i.e., jobs of tasks in $S_j \setminus F^k_j$), denoted $a^k_j$. That is, we have $\ell^k_j = a^k_j + f^k_j$. Since, by definition, future jobs have not yet arrived at time $t$, we cannot swap execution time with these jobs. Therefore, the value of $f^k_j$ can never be changed. When using a DP-Fair approach such as EKG, $f^k_j$ is equal to the utilization of the tasks in $F^k_j$ multiplied by the length of the time slice $[11],[14]$. That is,

$$f^k_j \triangleq L^k \times \sum_{\tau_i \in F^k_j} U_i$$
We must therefore respect the following two principles:

**Principle 3**: In any time slice $TS^k$, we cannot decrease the local execution time of a supertask $S_j$ by more than $a_{S_j}^k = \ell_{S_j}^k - f_{S_j}^k$.

**Principle 4**: Assuming that $b_q$ is a boundary subsequent to $t$ (i.e. $b_q \geq b_h$), then, to respect Rule 4 for a supertask $S_j$, we must ensure that the remaining execution at time $t$ of the active jobs in $S_j$ with a deadline before or at $b_q$, never exceeds the time reserved for those jobs in the interval $[t, b_q]$. Specifically, for each $q \geq c$,

$$
\sum_{i=\ell}^{q} a_{S_j}^i \geq \sum_{d_i(t) \leq b_q} r_i(t).
$$

(1)

Note that Equation 1 can be rewritten as

$$
a_{S_j}^v \geq \sum_{d_i(t) \leq b_q} r_i(t) - \sum_{i=\ell}^{q} a_{S_j}^i.
$$

Hence, after a swap between $TS^c$ and $TS^k$, Principle 4 requires that the local execution time $\ell_{S_j}^c$ of $S_j$ in $TS^c$ to be large enough to ensure, for each $TS^q (q < k)$, that there is enough time allocated during $TS^c$ to $TS^k$ to meet the demand. That is,

$$
\ell_{S_j}^c \geq \bar{a}_{S_j}^c \equiv \max_{e \leq q < k} \left( \sum_{d_i(t) \leq \bar{a}_{S_j}^c} r_i(t) - \sum_{i=\ell}^{q} a_{S_j}^i \right)
$$

(2)

where $\bar{a}_{S_j}^c$ is the minimum time $S_j$ must execute in $TS^c$ to respect Expression 1.

2) **Algorithm Description**: The swapping algorithm works as follows; initially, EKG is used to assign the tasks amongst the processors and compute the local execution time of every migratory task and supertask in each time slice bounded by the job deadlines. Then, at $t = 0$ and at any time $t$ corresponding to a time slice boundary, our swapping algorithm is executed on a depth of $\eta$ time slices as shown in Algorithm 1. That is, Algorithm 1 browses the $\eta$ time slices $TS^k$ subsequent to $b_q$ and updates the local execution times by performing swaps (if possible) between $TS^k$ and the current time slice $TS^c$ in accordance with Rules 1 to 4. The value of $\eta$ is chosen at design time and must be greater than 0. The simulation results presented in Section VI show the impact of this parameter on the number of preemptions and migrations. The swapped quantities are computed starting with processor $\pi_m$ and ending with processor $\pi_1$. On each processor $\pi_j$, we first perform the swap for the subtask $M_j$ and then for the subtask $M_j''$ using Lemmas 1 and 2 presented later (lines 2 to 5 in Algorithm 1). However, since the swapping decision on processor $\pi_j$ is taken without complete knowledge of the swaps that will be performed on processors $\pi_1 \to \pi_{j-1}$, Rule 3 is partially ignored during this first swapping pass and some intra-job parallelism could arise. Therefore, with a second pass, we must correct the values of the local execution times on each processor to remove the parallelism and hence respect Rule 3 (lines 7 to 9). Finally, the local execution time of the supertask $S_j$ is updated (line 10).

We now derive the maximum execution time that can be swapped between $TS^c$ and $TS^k$ for migratory tasks on processor $\pi_j$. According to Rules 1 and 2, we will increase the local execution time $\ell_{M_j''}^{\pi_j}$ of $M_j''$ by $\Delta_{M_j''}^k$ in $TS^c$ and decrease $\ell_{M_j'}^{\pi_j}$ by $\Delta_{M_j'}^k$. Moreover, as stated in Principle 2, in order to respect Rule 4, we must correspondingly decrease $\ell_{M_j'}^{\pi_j}$ by $\Delta_{M_j'}^k$ and increase $\ell_{M_j''}^{\pi_j}$ by $\Delta_{M_j''}^k$ in $TS^k$.

**Lemma 1**: Let $a_{M_j''}^{\pi_j} = \ell_{M_j''}^{\pi_j}$ if $d_{M_j''}(t) \geq b_k$ and $a_{M_j''}^{\pi_j} = 0$ otherwise. The maximum execution time of the migratory task $M_j'$ that can be swapped from time slice $TS^c$ to time slice $TS^k$ is

$$
\Delta_{M_j'}^k = \min \left\{ \ell_{M_j'}^{\pi_j}, a_{M_j''}^{\pi_j}, L^k - \ell_{M_j'}^{\pi_j} - \ell_{M_j''}^{\pi_j} \right\}
$$

if $d_{M_j''}(t) \geq b_k$. Otherwise, $\Delta_{M_j'}^k = 0$.

**Proof**: We cannot swap more from $TS^c$ than what is allocated to $M_j''$. Hence, $\Delta_{M_j'}^k \leq \ell_{M_j''}^{\pi_j}$. Similarly, increasing $M_j'$s allocation in $TS^k$ will decrease the time allocated to $M_j''$ and $S_j$. Therefore, we cannot swap more into $TS^k$ than what is allocated to $M_j''$ and $S_j$ in $TS^k$. That is, $\Delta_{M_j'}^k \leq \ell_{M_j''}^{\pi_j} + a_{S_j}^k$ (see Fig. 3 (a)). Note that from Principle 1, we cannot swap any time from $M_j''$ to $TS^c$ and a time slice subsequent to $d_{M_j''}(t)$. In this last case we therefore have $\Delta_{M_j'}^k \leq a_{S_j}^k$. Furthermore, from Rule 3, we forbid any intra-job parallelism. Therefore, because $\ell_{M_j'}^{\pi_j}$ has already been computed and will remain fixed, we must have $\Delta_{M_j'}^k \leq L^k - \ell_{M_j'}^{\pi_j} - \ell_{M_j''}^{\pi_j}$ (recall that $M_j'$ and $M_j''$ belong to the same migratory task $M_j$).

Finally, we stated in Principle 1 that we cannot swap any time from $M_j$ between $TS^c$ and a time slice subsequent to $d_{M_j}(t)$. That is, $\Delta_{M_j}^k = 0$ for such time slices.

**Algorithm 1**: Swapping algorithm.

```plaintext
1 forall the \(TS^k : k := c + 1 \to c + \eta \) do
   2   for \(j := m \to 1 \) do
      3      Swap between \(\ell_{M_j'}^{\pi_j} \) and \(\ell_{M_j''}^{\pi_j} \) using Lemma 1;
      4      Swap between \(\ell_{M_j''}^{\pi_j} \) and \(\ell_{M_j'}^{\pi_j} \) using Lemma 2;
   5   end
   6   for \(j := 1 \to m \) do
      7      if There is parallelism between \(M_j' \) and \(M_j'' \) then
         8         Correct \(\ell_{M_j'}^{\pi_j} \) and \(\ell_{M_j''}^{\pi_j} \) values using Eq. 3;
      9      end
   10   end
   11   Update \(\ell_{S_j}^c \) and \(\ell_{S_j}^k \) using Eq. 4;
12 end
```
**Lemma 2:** The maximum execution time of the migratory task $M'_{j-1}$ that can be swapped from time slice $TS^k$ to time slice $TS^c$ is

$$\Delta_{M'_{j-1}} = \min \{ \ell_{M''_{j-1}}, (L^c - \ell_{M''_{j-1}} - \ell_{M'_{j-1}} - \bar{a}_{S_j}) \}$$

if $d_{M_{j-1}}(t) \geq b_k$. Otherwise, $\Delta_{M'_{j-1}} = 0$.

**Proof:** We cannot swap more from $TS^k$ than what is allocated to $M'_{j-1}$. Therefore, $\Delta_{M'_{j-1}} \leq \ell_{M''_{j-1}}$. Similarly, we cannot swap more from $TS^c$ than what must not mandatorily be executed in $TS^c$. Since $M'_{j}$ and $S_j$ must execute for $\ell_{M'_{j}}$ (determined at line 1 of Algorithm 1) and $\bar{a}_{S_j}$ (from Expression 2) in time slice $TS^c$ respectively, and because $M''_{j-1}$ already has $\ell_{S_j}$ time units allocated in $TS^c$, that leaves $\Delta_{M'_{j-1}} \leq (L^c - \ell_{M''_{j-1}} - \ell_{M'_{j-1}} - \bar{a}_{S_j})$ (see Fig. 3 (b)).

Finally, we stated in Principle 1 that we cannot swap any time from $M_{j-1}$ between $TS^c$ and a time slice subsequent to $d_{M_{j-1}}(t)$. That is, $\Delta_{M'_{j-1}} = 0$ for such time slices.

Note that Lemma 2 uses the updated values of the local execution times of $S_j$, $M''_{j-1}$ and $M'_{j}$ in $TS^c$ and $TS^k$ after the application of Lemma 1 at line 3 of Algorithm 1.

In case of intra-job parallelism between $M'_{j}$ and $M''_{j}$ (recall that they both belong to the migratory task $M_{j}$), we can correct the values of the local execution times by adjusting the $\Delta_{M'_{j}}$ and $\Delta_{M''_{j+1}}$ quantities (see Fig. 3 (c)). In this situation, we decrease $\Delta_{M'_{j}}$ so that task $M'_{j}$ starts to execute at the exact moment when $M''_{j}$ finishes on processor $\pi_{j}$ (i.e., $M_j$ executes for exactly $L^c$ time units during $TS^c$). This frees up some time which can be spent either executing $S_{j+1}$ or $M'_{j+1}$. By Rule 2, our preference would be to avoid increasing $M'_{j+1}$. However, Principle 3 states that we cannot decrease the execution of $S_{j+1}$ in $TS^k$ for more than $\Delta_{S_{j+1}}$ time units. Therefore, if $\Delta_{S_{j+1}}$ is too small then we may need to increase $M'_{j+1}$ in $TS^c$.

This gives the following adjustments to $\Delta_{M'_{j}}$ and $\Delta_{M''_{j+1}}$.

$$\begin{align*}
\Delta_{M'_{j}} &\leftarrow \Delta_{M'_{j}} - (\ell_{S_{j-1}} + \ell_{M'_{j}} - L^c) \\
\Delta_{M''_{j+1}} &\leftarrow \Delta_{M''_{j+1}} - \max \{ 0, \ell_{M'_{j}} + \ell_{M''_{j}} - L^c - \Delta_{S_{j+1}} \}
\end{align*}$$

(3)

Whenever we have computed new values for the local execution times of tasks $M''_{j-1}$ and $M'_{j}$ on processor $\pi_{j}$, we must update $\ell_{S_{j-1}}$ and $\ell_{S_{j}}$ so that the total execution time remains constant in each time slice. We therefore get

$$\begin{align*}
\ell_{S_{j-1}} &\leftarrow \ell_{S_{j-1}} + \Delta_{M''_{j-1}} - \Delta_{M''_{j}} \\
\ell_{S_{j}} &\leftarrow \ell_{S_{j}} - \Delta_{M'_{j}} + \Delta_{M''_{j}}
\end{align*}$$

(4)

**B. Skipping Boundaries**

As previously explained, the EKG algorithm causes up to two preemptions and one migration per processor in each time slice. Therefore, a straightforward solution improving the number of preemptions and migrations during the schedule would be to reduce the number of time slices by suppressing some time slice boundaries. Clearly, we cannot remove all such boundaries – it is the imposition of these boundaries that makes EKG an optimal algorithm. In this section, we

\footnote{In fact, at the exception of RUN [6], all the optimal multiprocessor scheduling algorithms use some sort of time slice mechanism to achieve optimality.}

![Fig. 3: Task swapping according to Algorithm 1.](image-url)
explore how to determine which boundaries can be skipped without jeopardizing the schedulability.

1) A Set of Rules to Safely Suppress a Boundary: Suppose that we are at time $t = b_{q-1}$ and we remove the boundary $b_q$. Hence, we are scheduling in the interval $[b_{q-1}, b_{q+1})$ rather than $[b_{q-1}, b_q)$ (see Fig. 4). We call this extended time slice $TS^q$. If we are to maintain a correct schedule, we cannot reduce the amount of work the migratory tasks complete during this extended interval. Additionally, we still want to execute $S_j$ between $M_{j,q}^{in}$ and $M_{j,q}^{out}$ on each processor $\pi_j$. Otherwise, suppressing the boundary will not have the benefit of reducing the number of preemptions. Hence, we can view the boundary suppression as the merging of two time slices. While we would normally have $M_{j,q}^{out}$ executing before $b_q$ and $M_{j,q}^{in}$ executing after $b_q$, these execution time shares are shifted so that the migratory tasks execute contiguously within the merged time slice, as illustrated in Fig. 4. We note that because of this shifting, the merged time slice will not have any intra-job parallelism.

Whenever we consider to skip a boundary, we must be concerned by the impact that the shifting of the execution time shares could have on the correctness of the produced schedule. Hence, we must verify that we still respect the two following rules after the boundary suppression:

**Rule 5:** Never relegate the execution of a job after its deadline.

**Rule 6:** Never schedule the execution of a job before its release time.

We first consider what would happen if either $M_{j,q}^{in}$ or $M_{j,q}^{out}$ had a deadline at time $b_q$. Since $T_{j,q}^{q+1}$ is initially scheduled to be executed after $b_q$, it means that this time is reserved for the execution of the next job of $M_{j,q}^{out}$ and because we shift $T_{j,q}^{q+1}$ backward in time when we merge $TS_q$ and $TS_q^{q+1}$ (see Fig. 4), we would end up trying to execute the newly arrived job before it has arrived. Similarly, when we shift $T_{j,q}^{q+1}$ forward in time, this would result in $M_{j,q}^{out}$ missing its deadline. Therefore, both situations break one of the two Rules 5 and 6. Hence, this gives us our first principle regarding the suppression of time slice boundaries:

**Lemma 3:** A time slice boundary $b_q$ cannot be skipped if a job of any migratory task $M_j$ has its deadline at time $b_q$.

Even for non-migratory tasks, we need to be concerned about the same two issues, i.e., we need to ensure that the boundary suppression will not result in a missed deadline or in the attempt to execute a job before it has arrived.

Let $T_{j}^{q+1}$ be the set of non-migratory tasks that have a deadline no later than $b_q$ on processor $\pi_j$. Formally,

$$T_{j}^{q+1} \text{ def } \{ \tau_i \mid \tau_i \in S_j \land d_i(t) \leq b_q \}$$

As for migratory tasks, to ensure that the tasks in $T_{j}^{q+1}$ still finish their executions before their deadlines after the suppression of $b_q$, we could simply impose $r_i(t)$ to be equal to 0 for all tasks $\tau_i \in T_{j}^{q+1}$ on every processor $\pi_j$. In other words, to be authorized to skip a boundary $b_q$, all tasks with a deadline at or before $b_q$ must have finished their execution. This, however, is more restrictive than necessary. Tasks in $T_{j}^{q+1}$ can execute within the merged time slice $TS^q$ provided that each task $\tau_i \in T_{j}^{q+1}$ executes for exactly $r_i(t)$ time units during the interval $[b_{q-1}, b_q]$. Hence, after the boundary suppression, the local execution time $T_{j,q}^{q+1}M_{j,q}^{in}$ allotted to $M_{j,q}^{in}$ cannot be so large that the tasks in $T_{j}^{q+1}$ are unable to finish the execution of their current jobs by $b_q$. This gives the following constraint:

$$\ell_{j,q}^{q+1} + \sum_{\tau_i \in T_{j,q}^{q+1}} r_i(t) \leq b_q - b_{q-1}$$

where $\ell_{j,q}^{q+1} \text{ def } \ell_{j,q}^{q+1} + \ell_{j,q}^{q+1}$. That is, there is enough time to execute $M_{j,q}^{in}$ and all tasks in $T_{j}^{q+1}$ within $[b_{q-1}, b_q]$.

Now, let us assume that we want to skip $s$ successive boundaries. Applying the same reasoning, we obtain the following lemma:

**Lemma 4:** Let $k \text{ def } q + s - 1$. We can skip the $s$ successive boundaries $b_q$ to $b_k$ if only if for all $0 \leq v < s$ and all processors $\pi_j$, the following inequality is respected

$$\sum_{p=0}^{s} \ell_{j,q}^{q+p} + \sum_{\tau_i \in T_{j,q}^{q+v}} r_i(b_{q-1}) \leq b_{q+v} - b_{q-1}$$

Lemma 4 makes certain that Rule 5 is respected. Additionally, a second constraint must ensure that no job arriving after $b_q$ will be scheduled to execute before its arrival time (Rule 6).

Let us assume that a task $\tau_j$ which has its deadline at time $b_q$ (i.e., $d_j(t) = b_q$) is a non-migratory task of processor $\pi_j$. That is, $\tau_j$ is a component task of $S_j$. As explained in Section IV-A, EKG reserved a time equal to $f_{j,q}^{q+1} = U_j \times (b_{q+1} - b_q)$ in the interval $[b_q, b_{q+1})$ for the execution of the next job of $\tau_j$. Because the local execution time $\ell_{j,q}^{q+1}M_{j,q}^{out}$ is shifted forward in the merged time slice $TS^q$ (see Fig. 4), it is possible that $\ell_{j,q}^{q+1}M_{j,q}^{out} + \ell_{j,q}^{q+1}M_{j,q}^{in} + f_{j,q}^{q+1} > (b_{q+1} - b_q)$, thereby implying that the future job of $\tau_j$ should at least partially execute in $[b_{q-1}, b_q]$. However, this execution time is not available before $b_q$ (i.e., the arrival time of the new job), and because we assume $U = m$, there can be no idle time in the schedule. Hence, we must respect the following constraint when suppressing the boundary $b_q$:

$$\forall \pi_j : \ell_{j,q}^{q+1}M_{j,q}^{out} + f_{j,q}^{q+1} \leq (b_{q+1} - b_q)$$

where $\ell_{j,q}^{q+1}M_{j,q}^{out} \text{ def } \ell_{j,q}^{q+1}M_{j,q}^{out}$ and $f_{j,q}^{q+1}$ is the time reserved for future jobs of tasks in $S_j$ with a deadline at or before $b_q$, i.e., $f_{j,q}^{q+1} \text{ def } \sum_{\tau_i \in T_{j,q}^{q+1}} U_j \times (b_{q+1} - b_q)$. 
Before

\[ \begin{align*}
\pi_j & \quad TS^q & \quad b_q & \quad TS^{q+1} \\
\pi_{j+1} & \quad M_{j+1}^m & \quad \cdots & \quad M_{j+1}^m
\end{align*} \]

After

\[ \begin{align*}
\pi_j & \quad TS^q & \quad b_q & \quad TS^q \ast \\
\pi_{j+1} & \quad M_{j+1}^m & \quad \cdots & \quad M_{j+1}^m
\end{align*} \]

\[ \text{Fig. 4: Suppressing a boundary and merging the time slices.} \]

\begin{algorithm}
\caption{Selection of the next boundary that must be taken into account.}
\begin{algorithmic}[1]
\ForAll{the \( b_k > t \)}
\If{Lemma 3 or Lemma 4 or Lemma 5 is not respected}
\State return \( b_k \);
\EndIf
\EndFor
\end{algorithmic}
\end{algorithm}

Now, if we want to skip \( s \) successive boundaries, then applying the same argument for each boundary, we get the following lemma:

**Lemma 5:** Let \( k \overset{\text{def}}{=} q + s - 1 \). We can skip the \( s \) successive boundaries \( b_q \) to \( b_k \) if for all \( 1 \leq v \leq s \) and all processors \( \pi_j \), the following inequality is respected

\[ \sum_{p=0}^{v} f_{M_{j+1}^m}^p + f_{S_j}^{q+v} \leq (b_{q+v} - b_{q+v-1}) \]

2) **Algorithm Description:** In order to reduce the number of time slices, we try to increase the length of the current time slice \( TS^q \) as much as possible. Therefore, we execute Algorithm 2 at each time \( t \) corresponding to the end of a time slice. Algorithm 2 selects the next boundary \( b_k \) we must imperatively take into account to meet all job deadlines occurring within the interval \([t, b_k]\), and still respect Rules 5 and 6. Hence, all boundaries \( b_q \in (t, b_k) \) can be ignored and the current time slice \( TS^q \) therefore extends from \( t \) to \( b_k \).

**C. Skipping and Swapping Altogether**

It is possible to directly skip boundaries after the execution of the swapping algorithm at time \( t \). However, as it will be shown in Section VI, it has a negative impact on the resulting number of preemptions and migrations (see Fig. 6(a) and (d)). This can be explained by the fact that skipping boundaries will cancel the swapping effects by merging the current time slice \( TS^q \) with time slices with which execution time had been previously swapped.

On the other hand, swapping execution time after the execution of Algorithm 2 should improve the results of both techniques. However, we cannot swap execution time after having merged many time slices without taking some precautions. Indeed, through Lemmas 4 and 5, the boundary suppression algorithm imposes constraints on the repartition of the task execution times. These constraints should be added to those already present in Lemmas 1 and 2 which are used by the swapping algorithm. Note that Lemmas 4 and 5 impose their constraints on the migrate-in and migrate-out tasks (i.e., \( M_{j,q}^m \) and \( M_{j,q}^o \)), while Lemmas 1 and 2 refer to the subtasks \( M_j \) and \( M_{j-1} \). Therefore, it must be taken into account that the constraints of the skipping algorithm refer alternatively to the subtasks \( M_j \) and \( M_{j-1} \). Unfortunately, due to space limitation, we cannot enter into the details.

V. IMPLEMENTATION CONSIDERATIONS

a) **Complexity:** By analyzing Algorithms 1 and 2, we can easily show that their run-time complexities are \( O(n) \). These complexities are lower than or equivalent to those of other optimal algorithms such as RUN or U-EDF [6], [19] which are also executed at each job deadline. Hence, their execution should not be more time consuming than RUN or U-EDF while causing less preemptions and migrations in some situations (see simulation results in next section).

Moreover, while EKG basically saves the schedule of 1 time slice at each boundary \( b_k \) (memory complexity of \( O(m) \)), the swapping algorithm needs to save the schedule for \( \eta + 1 \) consecutive time slices to swap execution time between them. Hence, we now need \( O(m \times \eta) \) memory space.

b) **Instantaneous Migrations:** In addition to a reduction of the number of preemptions and migrations, the mirroring technique introduced in Section III also helps to avoid instantaneous migrations of a task from one processor to another. Unfortunately, the swapping algorithm re-introduces such instantaneous migrations since we assumed in the argumentation of Section IV-A that a migratory task can execute for up to the length \( L_k \) of the time slice \( TS^k \) in \( TS^k \). However, by simply modifying this assumption to allow a task to execute for up to \( L_k - \epsilon \) (with \( \epsilon > 0 \)) in \( TS^k \), we overcome the instantaneous migration problem. Expressions 3 and 4 and
Average Preemptions per Job was reached. On the other hand, if the number of tasks $n$ is not imposed, task utilizations were randomly generated using a uniform integer distribution. For the experiments where the number of tasks $n$ is high (as shown in Fig. 6(c) and (f)), we show the average number of preemptions and migrations on a fully utilized platform of 8 processors when the number of tasks varies. We see that the swapping and skipping algorithms decrease the number of preemptions almost at the same level than U-EDF for a number of tasks greater than 50. In addition, the swapping algorithm used with EKG is already competitive with U-EDF when the number of tasks is around 30. Moreover, the skipping algorithm used with EKG has almost the same average number of migrations than RUN (which is better than U-EDF on this point) for 30 tasks and more. The skipping algorithm ($\eta = 8$) even outperforms RUN when there are 20 (or more) tasks.

The last experiment (Fig. 6(c) and (f)) shows that the number of preemptions and migrations produced with EKG used with the swapping or skipping algorithm stay close to the results of U-EDF for a total utilization smaller than or equal to 75%. Furthermore, the number of migrations of the swapping algorithm never deviates from the U-EDF results and can even be better if the number of tasks is high (as shown on Fig. 6(e)).

VI. Simulation Results

We evaluated the performances of the swapping and skipping algorithms through an extensive number of simulations. In each experiment, we simulated the scheduling of 1,000 task sets from time 0 to time 100,000. Each task had a period randomly chosen within [5, 100] using a uniform integer distribution. For the experiments where the number of tasks is not imposed, task utilizations were randomly generated between 0.01 and 0.99 until the targeted system utilization was reached. On the other hand, if the number of tasks was imposed, the task utilizations were generated using the procedure proposed in [21].

The first experiment aims at comparing the improvement on the number of preemptions and migrations when the depth of swapping (i.e., the number of successive time slices with which the current time slice $TS^c$ exchanges execution time) increases. The results can be consulted on Fig. 5. We see that for the presented range on the number of processors, it does not benefit to the system to increase the depth of swapping beyond 8. Indeed, the difference in terms of preemptions and migrations per job between a depth of 8 and $n$ (i.e., when we swap with all the time slices after $TS^c$) remains quite small. Hence, for all the following experiments we arbitrarily decided to always use the swapping algorithm with a depth of 8.

The next experiments compare the results of the swapping and skipping algorithms used in conjunction with EKG, with the today best performing online optimal scheduling algorithms – namely U-EDF and RUN.

On Fig. 6(a) and (d), we show the average number of preemptions and migrations per job released in the system for various number of processors when the platform is fully utilized. We can note that EKG used with the swapping algorithm ($\eta = 8$) performs better than EKG used with the skipping algorithm. Furthermore, skipping boundaries after having swapped execution time has a negative impact on the average number of preemptions and migrations.

Fig. 6(b) and (e) present the average number of preemptions and migrations on a fully utilized platform of 8 processors when the number of tasks is high (as shown in Fig. 6(c) and (f)). Indeed, the difference in terms of preemptions and migrations is positive until the targeted system utilization is reached. On the other hand, if the number of tasks is imposed, the task utilizations were generated using the semi-partitioned approach which allows to upper-bound the number of migrating tasks and improves the code and data

VI. Simulation Results

We evaluated the performances of the swapping and skipping algorithms through an extensive number of simulations. In each experiment, we simulated the scheduling of 1,000 task sets from time 0 to time 100,000. Each task had a period randomly chosen within [5, 100] using a uniform integer distribution. For the experiments where the number of tasks is not imposed, task utilizations were randomly generated between 0.01 and 0.99 until the targeted system utilization was reached. On the other hand, if the number of tasks was imposed, the task utilizations were generated using the procedure proposed in [21].

The first experiment aims at comparing the improvement on the number of preemptions and migrations when the depth of swapping (i.e., the number of successive time slices with which the current time slice $TS^c$ exchanges execution time) increases. The results can be consulted on Fig. 5. We see that for the presented range on the number of processors, it does not benefit to the system to increase the depth of swapping beyond 8. Indeed, the difference in terms of preemptions and migrations per job between a depth of 8 and $n$ (i.e., when we swap with all the time slices after $TS^c$) remains quite small. Hence, for all the following experiments we arbitrarily decided to always use the swapping algorithm with a depth of 8.

The next experiments compare the results of the swapping and skipping algorithms used in conjunction with EKG, with the today best performing online optimal scheduling algorithms – namely U-EDF and RUN.

On Fig. 6(a) and (d), we show the average number of preemptions and migrations per job released in the system for various number of processors when the platform is fully utilized. We can note that EKG used with the swapping algorithm ($\eta = 8$) performs better than EKG used with the skipping algorithm. Furthermore, skipping boundaries after having swapped execution time has a negative impact on the average number of preemptions and migrations.

Fig. 6(b) and (e) present the average number of preemptions and migrations on a fully utilized platform of 8 processors when the number of tasks is high (as shown in Fig. 6(c) and (f)). Indeed, the difference in terms of preemptions and migrations is positive until the targeted system utilization is reached. On the other hand, if the number of tasks is imposed, the task utilizations were generated using the semi-partitioned approach which allows to upper-bound the number of migrating tasks and improves the code and data

VI. Simulation Results

We evaluated the performances of the swapping and skipping algorithms through an extensive number of simulations. In each experiment, we simulated the scheduling of 1,000 task sets from time 0 to time 100,000. Each task had a period randomly chosen within [5, 100] using a uniform integer distribution. For the experiments where the number of tasks is not imposed, task utilizations were randomly generated between 0.01 and 0.99 until the targeted system utilization was reached. On the other hand, if the number of tasks was imposed, the task utilizations were generated using the procedure proposed in [21].

The first experiment aims at comparing the improvement on the number of preemptions and migrations when the depth of swapping (i.e., the number of successive time slices with which the current time slice $TS^c$ exchanges execution time) increases. The results can be consulted on Fig. 5. We see that for the presented range on the number of processors, it does not benefit to the system to increase the depth of swapping beyond 8. Indeed, the difference in terms of preemptions and migrations per job between a depth of 8 and $n$ (i.e., when we swap with all the time slices after $TS^c$) remains quite small. Hence, for all the following experiments we arbitrarily decided to always use the swapping algorithm with a depth of 8.

The next experiments compare the results of the swapping and skipping algorithms used in conjunction with EKG, with the today best performing online optimal scheduling algorithms – namely U-EDF and RUN.

On Fig. 6(a) and (d), we show the average number of preemptions and migrations per job released in the system for various number of processors when the platform is fully utilized. We can note that EKG used with the swapping algorithm ($\eta = 8$) performs better than EKG used with the skipping algorithm. Furthermore, skipping boundaries after having swapped execution time has a negative impact on the average number of preemptions and migrations.

Fig. 6(b) and (e) present the average number of preemptions and migrations on a fully utilized platform of 8 processors when the number of tasks is high (as shown in Fig. 6(c) and (f)). Indeed, the difference in terms of preemptions and migrations is positive until the targeted system utilization is reached. On the other hand, if the number of tasks is imposed, the task utilizations were generated using the semi-partitioned approach which allows to upper-bound the number of migrating tasks and improves the code and data
locality in memories. Furthermore, the execution of both the skipping and the swapping algorithm should not be more time consuming than the execution of an algorithm such as RUN or U-EDF. Overall, our new solution should therefore better perform in real applications (this last claim should however be verified through experiments on real hardware platforms).

In future works, we would like to extend the swapping algorithm to the sporadic version of EKG [22] and implement Algorithms 1 and 2 in a real operating systems following the approach proposed in [20]. It can also be interesting to analyze both algorithms to identify the circumstances in which they actually improve the system performances.

REFERENCES


Fig. 6: Simulation Results: for fully utilized processors varying between 2 and 16 (a) and (d); for 8 fully utilized processors with a varying number of tasks (b) and (e); for a varying utilization of 8 processors (c) and (f).


