Two-level Hierarchical Scheduling Algorithm for Real-time Multiprocessor Systems

Muhammad Khurram Bhatti, Cécile Belleudy, Michel Auguin
LEAT research laboratory, University of Nice-Sophia Antipolis, Nice, France
Email: {bhatti, belleudy, auguin}@unice.fr

Abstract— **The Earliest Deadline First (EDF) scheduling algorithm has the least runtime complexity among job-level fixed-priority algorithms for scheduling tasks on multiprocessor architectures. However, EDF suffers from sub-optimality in multiprocessor systems. This paper proposes a new restricted migration-based scheduling algorithm for multiprocessor real-time systems, called *Two-level Hierarchical Scheduling Algorithm (2L-HiSA)*, to address this sub-optimality. 2L-HiSA algorithm divides the problem of multiprocessor scheduling into a two-level hierarchy of schedulers. This algorithm works in two phases: i- A task-partitioning phase in which, each task from application task set is assigned to a specific processor by following simple bin-packing approach. If a task can not be partitioned on any processor in the platform, it qualifies as migrating task. ii- A processor-grouping phase in which, processors are clustered together such that, per cluster, the unused fragmented computation power equivalent to at most one processor is available. We provide schedulability analysis and experimental evaluation to support our proposition. Moreover, our simulation results show an average difference of 18-folds in the number of preemptions and 10-folds in the number of task migrations under 2L-HiSA and PD^2 algorithm.**

Index Terms— **Real-time Systems, Multiprocessor, Scheduling, EDF, Partitioned scheduling, Global scheduling.**

I. Introduction

Real-time applications have become more sophisticated and complex in their behavior and interaction over the time. Contemporaneously, multiprocessor architectures have emerged to handle these sophisticated applications. Multiprocessor real-time systems, encompassing a range from small-scale embedded devices to large-scale data centers, are already being widely used and the future of embedded systems is in many-core chips [1], [2]. The research on real-time systems has been therefore renewed for these multiprocessor platforms, especially in the context of real-time scheduling. Efficient scheduling of real-time applications is mandatory in order to effectively exploit the parallelism offered by multiprocessor architectures. Real-time scheduling techniques for multiprocessor systems are mainly classified according to the amount of *task migration* the system allows at runtime. A task is said to be *migrating* if its successive jobs (and/or parts of the same job) can execute on different processors. Based on the amount of allowable migration, three types of migration strategies can be considered: no migration, full migration, and restricted migration scheduling [3], [4]. In *no migration* scheduling, also referred as *partitioned* scheduling, each task in a task set is statically assigned to a specific processor for execution and it can not migrate to other processors at runtime as illustrated in figure 1(a). Partitioned scheduling has the virtue of reducing problem of multiprocessor scheduling into a set of single-processor one after tasks are partitioned. In addition, it does not incur runtime overhead since tasks never migrate across processors [5]. Prohibiting migration may cause a system to be *under-utilized* [3] and for that reason, more than enough processing power will be available on some processor when a new job arrives. In fact, authors in [5], [6] report that the worst-case schedulable utilization of partitioning-based scheduling algorithms is not more than $\frac{\beta m+1}{\beta+1}$, where $\beta = \left\lfloor \frac{1}{\alpha} \right\rfloor$ and α is the maximum of individual utilization of tasks. On the other hand, in *full migration* scheduling, also referred as global scheduling, jobs of a task can migrate to other processors at any point in time during their execution. That is, jobs could start executing on one processor and then move to another processor, allowing the spare processing power to be distributed among all the processors. However, a job can only execute on at most one processor at a time -i.e., job parallelism is not permitted (see figure 1(b)). In this class, PFair [7], its heuristic algorithms such as PD^2 [8], LLREF [9], and ASEDZL [10] are known to be optimal algorithms. While full migration strategy is the most flexible, there are overheads associated with allowing migration such as increased context switches, handling of shared resources, and cache-content. Thus, there is a trade-off between scheduling loss due to migration and scheduling loss due to prohibiting migration. For the purpose of finding a balance point between partitioned and global scheduling, a relatively new class of algorithms, called *restricted migration* scheduling, has been made available by some recent research work in [1], [5], [6], [11], [12]. In this class, most tasks are statically assigned to specific processors as in case of partitioned scheduling in order to reduce task-migration, while few tasks may migrate across processors to improve available processor utilization as much as possible. Whenever a new job of a task is released, a global scheduler assigns this job to a particular processor. Once assigned, this job must complete its execution on the processor to which it is assigned as illustrated in figure 1(c). Systems that prohibit

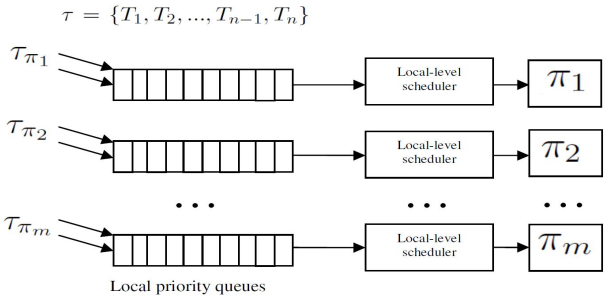
This work was supported in part by the French national project Pherma (grant: ANR-06-ARFU06-003) and the Higher Education Commission (HEC) of Pakistan.

doi:10.4304/jsw.6.11.2308-2320

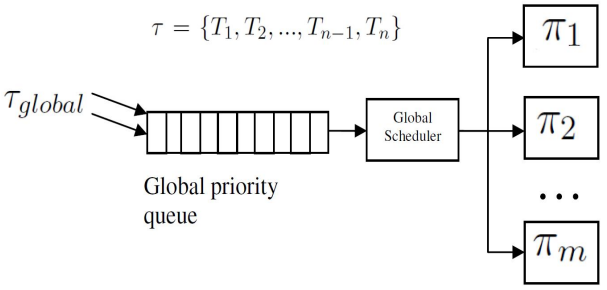
full migration must use either partitioning or restricted migration strategy. Between these two, the partitioning strategy is more commonly used in current systems. However, partitioning can only be used for fixed task sets. If tasks are allowed to dynamically join and leave the system, partitioning is not a viable strategy because a task joining the system may force the system to be repartitioned, thus forcing tasks to move to other processors. Moreover, determining a new partition is analogous to the bin-packing problem, which is considered NP-hard problem [10], [13], [14]. Thus, repartitioning dynamic task sets incurs too much overheads. Restricted migration scheduling strategy is flexible enough to allow dynamic task sets and it does not incur large migration overheads. This strategy is particularly useful when consecutive jobs of a task do not share any data since all data, which need to be passed to subsequent jobs would have to be migrated at job boundaries. In this paper, we advocate the use of restricted migration scheduling strategies for multiprocessor systems with large real-time tasks and propose new algorithm based on restricted migration scheduling, called *Two-level Hierarchical Scheduling Algorithm (2L-HiSA)*. This algorithm uses *Earliest Deadline First (EDF)* scheduling algorithm, which is a single-processor optimal algorithm, in a hierarchical fashion –i.e., an instance of EDF algorithm at top-level and local-level schedulers on each processor. Rest of this paper is organized as follows. Section II provides some related work. In section III, our system model is presented. Section IV presents in detail the 2L-HiSA scheduling algorithm. Section V provides experiments and simulation results, and section VI concludes this paper.

II. Background and Related Work

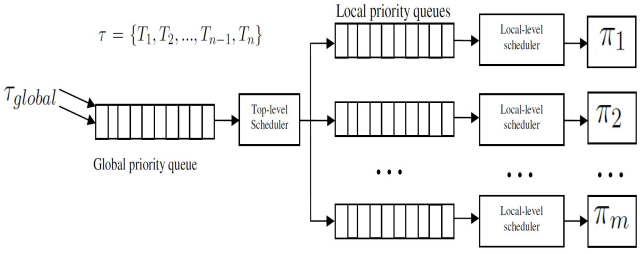
Some novel and promising techniques in the category of restricted migration scheduling have been proposed recently such as [1], [5], [6], [11], [12]. Kato et al. in [5] have presented a *semi-partitioned* scheduling algorithm for sporadic tasks with arbitrary deadlines on identical multiprocessor platforms. In this research work, authors propose to qualify a task as *migrating* task *only if* it is not possible to partition them on any processor of the platform. Thus, there are mostly partitioned tasks and few migrating tasks which are allowed to migrate from one processor to another *only once* per period. The main idea of this algorithm consists in using a *job-splitting* strategy for migrating tasks. In terms of utilization share, a migrating task is *split* into more than one processor. A task is split in such a way that a processor is filled to capacity by the portion of the task assigned to that processor. However, only the last processor to which the portion is assigned may not be filled to capacity. Figure 2 illustrates how a migrating task is executed exclusively among processors by splitting the deadline of each migrating task into the same number of windows as the processors across which the task is qualified to migrate. In figure 2, a migrating task T_k is split across the three processors. Task T_k is presumed to be executed



(a) No migration scheduling



(b) Full migration scheduling



(c) Restricted migration scheduling

Figure 1. Various scheduling approaches for real-time task set τ. (a) Scheduling with no migration. (b) Scheduling with full migration. (c) Scheduling with restricted migration.

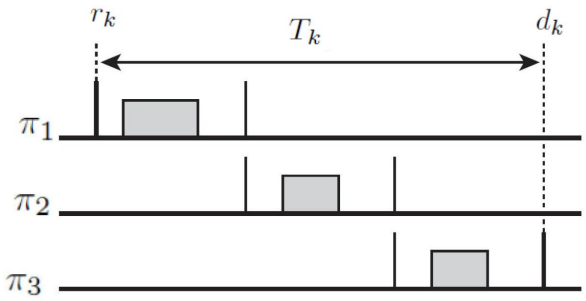


Figure 2. Job-splitting of a migrating task over three processors.

within these *fixed* windows with *pseudo-deadlines* which are smaller than the actual deadline of task. Fixing such pseudo-deadlines with limited allowable migration makes system much less flexible as the migrating tasks must execute within these fixed time slots. Systems with fixed time windows can not take full advantage of early completion of real-time tasks and consequently, cannot apply aggressive energy management techniques. Moreover, the job-splitting may still lead to prohibitive runtime

overheads for the system. Authors in [11] have presented the *Earliest Deadline Deferrable Portion (EDDP)* algorithm, which is also based on the portioned scheduling technique. Migrating tasks in this case are permitted to migrate between any *two* specific processors. In order to curb the cost of task migrations, EDDP makes at most $(m-1)$ migrating tasks on *m*-processors. Authors in this work claim that no tasks ever miss deadlines, if the system utilization does not exceed 65% using EDDP. The approach of limiting the migration of tasks to at most two processors is used earlier as well by authors in [15] who have proposed a scheduling algorithm which considers the trade-off between system utilization and number of preemptions for recurring task systems. The migration overhead is relaxed in this approach compared to the other optimal multiprocessor algorithms by limiting the number of migrating tasks. The algorithm trades an achievable system utilization with the cost of preemptions by adjusting a parameter k, where $2 \leq k \leq m$. For $k < m$, the achievable utilization is claimed to be $k/(k+1)$. For $k = m$, on the other hand, it is 100%, thereby the proposed algorithm performs optimally. Based on the work of [5], [11], [15], authors in [12] have also propose a semi-partitioned hard real-time scheduling approach for sporadic deadline-constrained tasks upon identical multiprocessor platforms. In this work, migration of jobs is prohibited except that *two subsequent jobs* of a task can be assigned to different processors by applying a *periodic strategy*. This technique comprises two steps: an *assigning phase* and a *scheduling phase*. The assigning phase is somewhat similar to that of [5]. That is, if it is not possible to partition a task without violating schedulability guarantees then the concerned task is classified as *migrating* task. Authors propose to distribute jobs of migrating task among several processors using a *multi-frame tasking* approach with a predefined periodic sequence of the occurrence of jobs on various processors. This predefined sequence of jobs repeats itself cyclically at runtime upon the selected processors. The limitation of this approach is the assumption that the number of frames of each migrating task over multiple processors must be available *beforehand* to provide schedulability analysis. Moreover, in [11] and [12], the schedulability bound is 65%, which is still not considerably large as compared to previously proposed partitioned scheduling algorithms that offer 50% utilization bound in worst-case. Calandrino et al. in [1] have proposed a *hybrid scheduling* approach for soft real-time tasks on large-scale multiprocessor platforms with hierarchical shared caches. In this approach, a multiprocessor platform is partitioned into *clusters*, tasks are statically assigned to these clusters (rather than individual processors), and scheduled within each cluster using the preemptive global EDF scheduling algorithm. All tasks are allowed to migrate *within* a cluster but not *across* clusters. Authors in this work demonstrate that, by partitioning the system into clusters instead of individual cores, bin-packing limitations can be alleviated by effectively increasing *bin-sizes* in comparison to *item-sizes*. However, this work still uses a common global scheduler at cluster-level which is equivalent to breaking a larger multiprocessor scheduling problem into *multiple smaller* multiprocessor scheduling problems. Moreover, the solution is limited only to soft real-time applications. In contrast to [1], authors in [16] have proposed a *two-level* scheduling scheme, which uses the idea of *sporadic servers*. In this approach, first an application is partitioned into parallel tasks as much as possible. Then the parallel tasks are dispatched to different processors, so as to execute in parallel. On each processor, real-time tasks run concurrently with non real-time tasks. At the top level, a sporadic server is assigned to each scheduling policy while at the bottom level, a Rate-Monotonic (RM) scheduler is adopted to maintain and schedule the top-level sporadic servers. While this research work uses a two-level hierarchy of schedulers, only soft real-time applications are considered for scheduling. In 2L-HiSA algorithm, most tasks are statically partitions onto processors, which is similar to that of [11]. However, neither the number of *migrating* tasks nor the number of *migrations* per migrating task is limited in 2L-HiSA, which is contrary to that of [1], [11], and [12]. Moreover, unlike in [11], 2L-HiSA does not *fix* time slots for migrating tasks. Rather it reserves a portion of processor time on each processor (in proportion to its under-utilization) for migrating tasks and this portion of time can be dynamically relocated by local-level scheduler within a specified period to allow the execution of statically partitioned tasks. This dynamic relocation of reserved time for migrating tasks improves system flexibility both at design-time and runtime. Section IV provides the 2L-HiSA algorithm in detail.

III. System Model & Notations

we will characterize every *job* $T_{i,j}$ of a real-time *task* $T_i, (\forall i, 1 \leq i \leq n)$ by a quadruplet (r_i, C_i, d_i, P_i): an arrival or release time r_i, a worst-case execution requirement C_i, a relative deadline d_i, and a period P_i. The interpretation of these parameters is that job $T_{i,j}$ arrives r_i time units after system start-time (assumed to be zero in generalized system model) and must execute for C_i time units over the time interval $[r_i, r_i + di)$. r_i is assumed to be a non-negative real number while both C_i and d_i are positive real numbers. The interval $[r_i, r_i + di)$ is referred to as $T_{i,j}$'s scheduling window. A job $T_{i,j}$ is said to be active at time instance *t* if $t \in [r_i, r_i + di)$ and $T_{i,j}$ has unfinished execution. A real-time task set τ has a finite collection of *n* tasks, $\tau = \{T_1, T_2, ..., T_{n-1}, T_n\}$ and a real-time task T_i is composed of an infinite collection of jobs $J = \{T_{i,1}, T_{i,2}, ...\}$. Individual utilization of a task T_i is given by $u_i = C_i/P_i$ and the aggregate utilization of task set τ by $U_{sum}(\tau) \stackrel{def}{=} \sum_{i=1}^{n} u_i$. We consider an identical multiprocessor platform Π comprised of *m* identical processors –i.e., $\Pi = \{\pi_1, \pi_2, ..., \pi_m\}$ where π_m refers to m^{th} processor.

IV. TWO-LEVEL HIERARCHICAL SCHEDULING ALGORITHM

The 2L-HiSA scheduling algorithm uses *multiple instances* of single-processor optimal EDF scheduling algorithm in a hierarchical fashion at two levels: an instance at top-level scheduler and an instance at local-level scheduler on every processor of the platform. Since EDF is an optimal single-processor scheduling algorithm, therefore, in order to determine whether the given task set is *EDF-schedulable*, it suffices to determine whether this task set is *feasible* on the single-processor systems. Unfortunately, it has been shown in [17], [18] that there are no optimal *job-level fixed-priority* scheduling algorithms for *multiprocessors*. Since EDF falls in this category, therefore, determining whether a given task set is feasible on a multiprocessor system will not tell us whether the same task set is EDF-schedulable on the same system as well. Baruah, et al. proved in [19] that there exists a *job-level dynamic-priority* scheduling algorithm, referred as PFair, which is optimal for periodic task sets on multiprocessors. Srinivasan and Anderson later showed in [20] that this algorithm can be modified to be optimal for sporadic task sets as well. However, these results do not apply on EDF because they use a job-level dynamic-priority algorithm. On the issue of determining EDF-schedulability, authors in [6] have provided schedulable utilization bounds for job-level fixed-priority scheduling algorithms for full-migration, restricted-migration, and partitioned scheduling strategies. EDF, being a job-level fixed-priority algorithm, has schedulable utilization bounds of $\frac{m^2}{2m-1} \leq U_{sum} \leq \frac{m+1}{2}$ for full-migration strategies, $U_{sum} = \frac{\beta m+1}{\beta+1}$ ($\beta = \lfloor \frac{1}{\alpha} \rfloor$) for no-migration strategies, and $m-\alpha(m-1) \leq U_{sum} \leq \frac{m+1}{2}$ or otherwise for restricted-migration strategies, respectively. Here, the term α represents a cap on individual task utilizations. Note that, if such a cap is not exploited, then the upper bound on schedulable utilization is approximately $\frac{m}{2}$ or lower. Authors in [21] state that, for a periodic task set with implicit deadlines, the schedulable utilization under EDF or any other static-priority multiprocessor scheduling algorithm –partitioned or global– can not be higher than $\frac{m+1}{2}$ for *m* processors. Clearly, under this schedulability bound, a multiprocessor platform suffers heavily from under-utilization (i.e., by a factor of $\frac{m-1}{2}$). For instance, in a system composed of three processors ($m = 3$), platform resource equivalent to at least one processor $\left(\frac{m-1}{2} = 1\right)$ is wasted. 2L-HiSA, instead of using global EDF scheduling algorithm, proposes a hierarchical scheduling approach using multiple single-processor optimal EDF instances. Section IV-A provides the basic concept of 2L-HiSA.

A. Basic Concept

The concept of 2L-HiSA algorithm slightly differs from the conventional restricted migration-based scheduling strategies. In restricted-migration scheduling with hierarchical schedulers, all tasks can migrate at *job-boundaries* and they share a common top-level task queue as illustrated in figure 1(c). That is, when a new job of a recurring task is released, the top-level scheduler assigns this job to any processor available in the platform. A released job, once assigned to a particular processor, can execute only on that processor under the control of local-level scheduler. Another job of the same task, however, can be assigned to a different processor. Thus, for every new job of a task, top-level scheduler first decides its assignment to target processor in the platform and then local scheduler executes that job according to its appropriate local priority level. In two-level hierarchical scheduling algorithm, however, local schedulers have certain number of partitioned tasks that do not migrate at all (which is contrary to that of [11]). The 2L-HiSA algorithm is based on the concept of semi-partitioned scheduling, in which most tasks are statically assigned to specific processors, while a few tasks migrate across processors. Once partitioned, these tasks are entirely handled by local-level scheduler and always remain in unique priority space associated only to their respective processor as illustrated in figure 3 by τ_{π_1}, τ_{π_2}, and τ_{π_m}, respectively. A task is qualified to become *migrating* task only if it cannot be partitioned on any processor any more using simple bin-packing approach. Such tasks are *fully* migrating tasks, unlike the migrating tasks in case of [11] and [12], which limit the number of possible migrations per period or per processor. All migrating tasks are placed in a separate subset of tasks (τ_{global}) as illustrated in figure 3. Only subset τ_{global} is handled by the top-level scheduler.

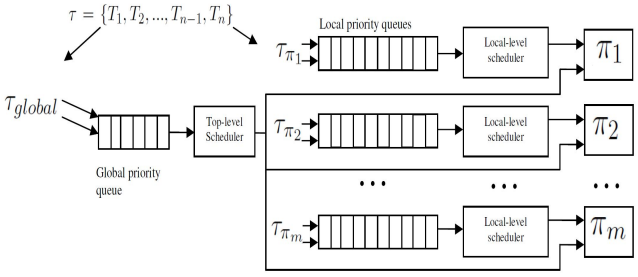


Figure 3. Two-level hierarchical scheduling based on restricted task migration.

Top-level scheduler assigns tasks from τ_{global} to processors at runtime within *suitable* time slots. These suitable time slots are actually the portion of processor's time that is unused and represented by its under-utilization (if any). Section IV-B gives more details on the procedure for determining the size, periodicity, and priority of these time slots. However, it is worth mentioning here that these time slots occur *explicitly* on processors in an *m*-processor platform –i.e., they do not occur in parallel. Moreover, these time slots are not fixed (like in case of [11]) and occur dynamically *within* the specified period. Due to the NP-hardness of the partitioning problem, processors in a multiprocessor platform are often under-utilized with a significant margin in a post-partitioned scenario.

B. Working Principle

In this section, we provide the main steps of our proposed algorithm. 2L-HiSA, as mentioned earlier, is based on the concept of restricted migration scheduling and consists of two phases:

- *The task-partitioning phase*: In this phase, each non-migrating task is assigned to a specific processor by following the bin-packing approach.
- *The processor-grouping phase*: This is a post-partitioning phase in which, processors are grouped together based on their workload characteristics.

1) The task-partitioning phase: Let us consider that a real-time task set τ containing at most *n* tasks such that $\tau = \{T_1, T_i, ...T_{n-1}, T_n\}$, has to be scheduled on an identical multiprocessor platform composed of *m* processors. The task set is considered feasible a priori - i.e., $U_{sum}(\tau) = \sum_{i=1}^{n} u_i \leq m$. In this phase, each task T_i is statically assigned to a particular processor π_k by following the bin-packing approach. Tasks are assigned to processors as long as they do not cause violation of schedulability of tasks being already partitioned upon processor π_k –i.e., $U_{sum}(\tau_{\pi_k}) \stackrel{def}{=}_{(L>0)} \frac{DBF(\tau_{\pi_k}, L)}{L} \leq 1$, where tasks being partitioned on a particular processor π_k are denoted by τ_{π_k}, L refers to interval length, and DBF refers to the classical Demand Bound Function [10], [12]. Algorithm 1 illustrates the task partitioning phase. In the first step, before partitioning any task to processors, the utilization of each processor π_k is initialized to zero – i.e., $U_{\pi_k} = 0$ (lines $1-5$). In the second step, each task is tested for partitioning on *m* processors of the platform according to the condition mentioned earlier (lines $6-14$). If, for any task T_i, $U_{sum}(\tau_{\pi_k}) > 1$ –i.e., it can not be partitioned on π_k ($\forall k, 1 \leq k \leq m$), then this task is classified as *migrating* task and assigned to τ_{glob} subset of tasks (lines $1-18$).

Algorithm 1 Offline task partitioning to processors

```
1: n ← number of tasks in τ
2: m ← number of processors in Π
3: for k = 1...m do
4:     U_{π_k} ← 0;
5: end for
6: for i = 1...n do
7:     for k = 1...m do
8:         if U_{π_k} + u_i ≤ 1 then
9:             assign T_i to π_k;
10:            U_{π_k} = U_{π_k} + u_i;
11:            remove T_i from τ;
12:            break;
13:        end if
14:    end for
15: end for
16: if size(τ) ≠ 0 then
17:    assign all remaining tasks to τ_glob;
18: end if
```

For a feasible task set τ, often it is not possible to partition all tasks due to the NP-hardness of partitioning problem. Thus, in our algorithm, a given task set τ is divided into two subsets of tasks such that $U_{sum}(\tau_{part}) + U_{sum}(\tau_{glob}) = U_{sum}(\tau) \leq m$. In a post-partitioned scenario, we can calculate the aggregate utilization of tasks being partitioned on every processor (π_k) individually using equation 1. Here, np refers to the total number of tasks being partitioned on a particular processor π_k.

$$U_{\pi_k}(\tau_{\pi_k}) = \sum_{i=1}^{np} \frac{C_i}{P_i} \quad (\forall i, 1 \leq i \leq np, \forall T_i \in \tau_{part}) \tag{1}$$

From the complimentary relation of equation 2, we can also compute the under-utilization present on every processor π_k. Let the under-utilization present on any processor π_k be referred as U'_{π_k}.

$$U'_{\pi_k}(\tau_{\pi_k}) = 1 - \sum_{i=1}^{np} \frac{C_i}{P_i} \quad (\forall i, 1 \leq i \leq np, \forall T_i \in \tau_{part}) \tag{2}$$

2) The processor-grouping phase: In the second phase, processors of the platform Π are grouped in such a way that the cumulated under-utilization on all processors *within a group* is not greater than one –i.e., $\sum U'_{\pi_k} \leq 1$. It is mentioned in section IV-A that a portion of processor time is reserved on every processor in proportion to its U'_{π_k} to which, the top-level scheduler could exploit for scheduling tasks from τ_{glob}. Moreover, it is desirable that these portions of processor time must appear explicitly for better exploitation. Now, if the cumulated under-utilization of processors will be more than one ($\sum U'_{\pi_k} \geq 1$), then the computation power equivalent to more than one processor will be *free* within the system. This under-utilization will cause idle time intervals to overlap (appear in parallel) on certain processors. Thus, grouping processors such that the sum of under-utilization on all processors *within a group* is not greater than one allows to have a cumulated (but still fragmented) computation power equivalent to at most one processor *free* within each group. This condition helps avoiding parallelism of the idle time intervals that would occur due to under-utilization. Algorithm 2 illustrates processor-grouping phase. This algorithm outputs the number of possible processor-groups or clusters within the platform that respect above condition. However, limiting the amount of under-utilization per group is not the only condition to ensure explicit occurrence of idle time intervals. These idle intervals would still appear randomly within each group. An issue of concern here is, how to make the idle intervals *explicit* and *periodic* so that migrating tasks could consume them. In the following, we provide a simple illustrative example of how the idle time intervals would appear at runtime in an application's schedule under EDF algorithm and then we should answer the concern related to explicitness and periodicity of idle intervals required for 2L-HiSA.

Example 3.1: Let us consider a periodic task set τ composed of six tasks ($n = 6$) to be scheduled on a multiprocessor platform composed of four identical processors ($m = 4$). Task set τ is scheduled using EDF scheduling algorithm. The value of quadruplet

Algorithm 2 Offline processor-grouping

```
1: m ← number of processors in Π
2: Y ← 0; //number of processor-groups
3: U'_sum ← 0;
4: for k = 1...m do
5:    U'_sum ← U'_sum + U'_πk;
6:    if U'_sum ≥ 1 then
7:       Y ← Y + 1;
8:       U'_sum ← 0;
9:    end if
10: end for
11: output: Y processor-groups are created;
```

of each task is selected such that $U_{sum}(\tau)$ respects sufficient condition bound provided by [21] –i.e., $U_{sum} \leq \frac{m+1}{2} \leq 2.5$. The values of quadruplet are; τ= $T_1(0,3,7,7), T_2(0,7,14,14), T_3(0,5,11,11)$, $T_4(0,4,13,13), T_5(0,4,8,8), T_6(0,3,9,9)$.

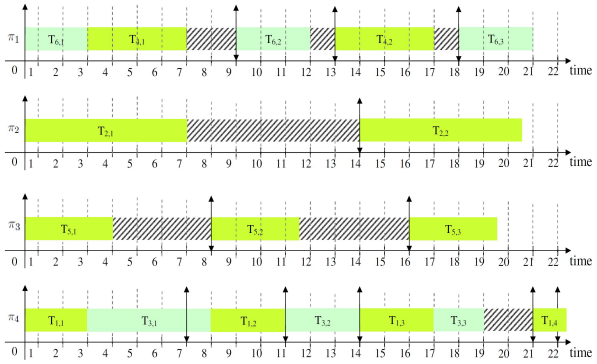


Figure 4. Example schedule of partitioned tasks under EDF scheduling algorithm on SMP architecture (n=6, m=4), illustrating the under-utilization of platform.

Let us partition these tasks on four processors[1] such that; τ_{π_1}=$\{T_4, T_6\}$, τ_{π_2}=$\{T_2\}$, τ_{π_3}=$\{T_5\}$, and τ_{π_k}=$\{T_1, T_3\}$. Figure 4 illustrates the EDF-schedule of τ on four processors. In this figure, it can be noticed that (more or less) every processor is under-utilized by a factor of $U'_{\pi_k}(\tau_{\pi_k})$ as stated by equation 2. Moreover, due to this under-utilization, idle intervals appear on processors at random (based on the EDF scheduler's priority mechanism) and in a non-periodic fashion.

2L-HiSA aims at exploiting these random idle intervals to schedule tasks from τ_{glob} under the control of top-level scheduler. The problem, however, is that these idle intervals are not periodic in their occurrence and therefore, can not be used to schedule *periodic* tasks by the top-level scheduler. The intuitive idea behind the 2L-HiSA algorithm is to *force* these idle time slots to appear in a periodic fashion on all those processors which offer positive under-utilization such that the amount of *periodic* idle time should explicitly appear on each processor π_k in proportion to U'_{π_k} offered by that processor. Once idle time slots become periodic, tasks from τ_{glob} can then be placed in these time slots under the control of top-level scheduler. To achieve this objective, a *dummy task* is added on every processor π_k. Let us call this dummy task as T_k^d on processor π_k. Task T_k^d is a periodic task that appears on all processors, which offer $U_{\pi_k} > 0$. In the following, we calculate the parameters of T_k^d such as its period P_k^d and worst-case execution time C_k^d on every processor.

Period of T_k^d is selected as the *minimum period* of all the tasks present in τ ($\tau_{part} \in \tau, \tau_{glob} \in \tau$) as shown in equation 3.

$$P_k^d = \min_{i=1}^{n} \{P_i\} \qquad (\forall k, 1 \leq k \leq m) \tag{3}$$

Note that, apart from being the smallest, the period of T_k^d is the *same* on all processors ($\forall k, 1 \leq k \leq m$). The advantage of having the smallest period for T_k^d on all processors is that the cumulated under-utilization $\sum U'_{\pi_k}$ present in a selected group of processors is proportionately available within the smallest period, hence, available for the most recurring migrating task. The advantage of having the same value for P_k^d on all processors is that it ensures the release of T_k^d at the same time on all processors, which is helpful in managing explicit execution of jobs of T_k^d on different processors. Once the period for T_k^d is determined, its worst-case execution time C_k^d can be calculated on every processor using equation 4, which is proportionate to U'_{π_k} available on each processor.

$$C_k^d = P_k^d \times U'_{\pi_k} \qquad (\forall k, 1 \leq k \leq m) \tag{4}$$

C_k^d refers to the *size* of idle time slots appearing on processor π_k at runtime over a period of P_k^d. Note that T_k^d is an *empty* task used only to reserve C_k^d time units of processor time over the smallest possible period P_k^d. At runtime, top-level scheduler *fills* these C_k^d time slots reserved by T_k^d with tasks from τ_{glob}. Since tasks in τ_{glob} are fully migrating tasks, therefore, they can use C_k^d time units on all processor if T_k^d does not appear in parallel. Thus, one of the design consideration of 2L-HiSA is to make sure that T_k^d is explicit on processors within a group over the interval lengths of P_k^d. Since T_k^d has the same period (P_k^d) on every processor, therefore, it releases at the same time on all processors. Moreover, making P_k^d being the smallest period within the task set τ also makes T_k^d the highest priority task on every processor under EDF scheduling algorithm. Thus, to ensure explicit execution of T_k^d, the 2L-HiSA algorithm performs a priority inversion of T_k^d on all those processors within a group on which T_k^d is not selected for execution at time instant *t*. This priority inversion is non-blocking from the platform resources point of view –i.e., inverting the priority of T_k^d on a processor π_k does not cause processor π_k to become idle or blocked as long as statically partitioned ready tasks exist. We illustrate this concept with an example in the following. Let us consider a multiprocessor platform with three processors belonging to the same group. Each processor has a dummy task T_k^d assigned to it. At time instant $t = 0$, T_k^d is released on all three processors simultaneously. if T_1^d, which is the dummy task on processor π_1, is assigned on π_1 as illustrated in figure 5, then the local schedulers on π_2 and π_3 invert the priority of T_2^d and T_3^d, respectively, to give higher priority to statically partitioned ready task that is having the highest priority (if any). Upon termination of T_1^d on π_1 at time instant $t = 1$, remaining two processors π_2 and π_3 revert the priority of their respective dummy tasks T_2^d and T_3^d, respectively, to allow them to compete for priority at local scheduler's level. Since, at most one local scheduler can assign T_k^d on a processor at any time to ensure explicit execution of jobs of T_k^d, therefore, the other local schedulers invert priority of their respective T_k^d again to allow partitioned tasks to execute. In this example, after T_1^d is terminated on π_1, local scheduler on π_2 assigns T_2^d and local scheduler on π_3 again inverts the priority of T_3^d to let the partitioned tasks run. Finally, at time instant $t = 2$, local scheduler on π_3 assigns T_3^d for execution. At time $t = 3$, T_1^d, T_2^d, and T_3^d are released again and compete for assignment on their respective processors. Note that T_k^d has no specific order of occurrence –i.e., fixed time slot, defined a priori on different

[1]The partitioning of tasks performed in this example may not be the optimal solution.

processors. A newly released job of T_k^d has to, first, compete for the priority among locally partitioned tasks and then compete for priority among T_k^d present on other processors within a group. Failure to obtain highest priority at any of the two levels cause a priority inversion for concerned task. This makes the portion of processors' time reserved for τ_{glob} to appear in a sequential fashion over P_k^d within a group of processors. The priority inversion for T_k^d on the same processor can be performed up to the time instant when laxity of T_k^d becomes zero.

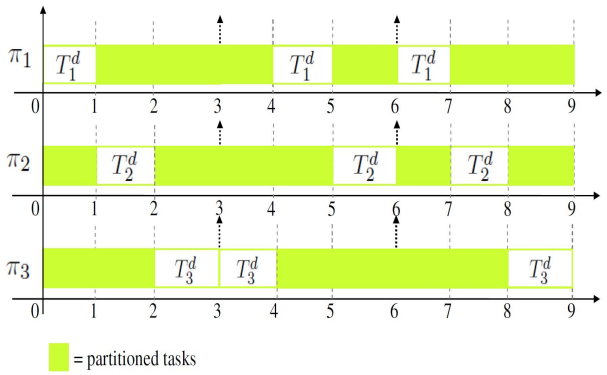


Figure 5. Illustration of T_k^d occurring on different processors with respect to the proportionate under-utilization available on each processor.

Algorithm 3 Local-level scheduler: Online jobs assignment for partitioned tasks present in τ_{π_k}

1: **define** π: processor containing local-level scheduler
2: $Y \leftarrow$ number of processor-groups
3: **for** $i = 1...Y$ **do**
4: $m_i \leftarrow$ number of processors in processor-group i;
5: **for** $k = 1...m_i$ **do**
6: $ReTQ(\tau_{\pi_k}) \leftarrow T_k^d$; {*adds dummy task to local ReTQ of every processor of group* i}
7: **end for**
8: **end for**
9: **for** every scheduling event **do**
10: **sort** $ReTQ(\tau_{\pi_k})$ w.r.t. EDF priority
11: $T \leftarrow$ highest priority ready task from $ReTQ(\tau_{\pi_k})$;
12: **if** $T = T_k^d$ **then**
13: **for** $k = 1...(m_i - 1)$ **do** {*for all processors other than* π}
14: **if** T_k^d is already running on π_k **then**
15: priority of T_k^d is inverted;
16: $T \leftarrow$ subsequent priority task from $ReTQ(\tau_{\pi_k})$;
17: $break$;
18: **end if**
19: **end for**
20: **end if**
21: $\pi \leftarrow T$;
22: **end for**

C. Runtime View of Schedule from Different Levels of Hierarchy

In this section, we provide the reader the view-points of both top-level and local-level schedulers under the 2L-HiSA algorithm.

1) Local-level Scheduler: From the earlier discussion, we know that single-processor optimal EDF scheduling algorithm is used as local scheduler on every processor to schedule statically partitioned tasks. Along with tasks being partitioned on each processor π_k –i.e., τ_{π_k}, there is a dummy task T_k^d assigned on each processor that has an execution requirement of C_k^d, which is exactly equal to the amount of U'_{π_k} available on π_k –i.e., $U'_{\pi_k} = 1 - U_{\pi_k} = \frac{C_k^d}{P_k^d}$. Thus, the worst-case workload of each processor is maximum –i.e., $U_{\tau_{\pi_k}} + U'_{\pi_k} = 1$. Local EDF scheduler on each processor visualizes the problem of scheduling τ_{π_k} along with T_k^d reduced to single-processor scheduling problem for which, EDF is optimal. Algorithm 3 illustrates jobs assignment on processor by local-level scheduler. For each processor-group Y, the number of processors within each group are known a priori and dummy task T_k^d is added to the local priority queue (ReTQ) of each processor (lines $1-8$). Upon the arrival of a scheduling event, this ReTQ is sorted according to EDF priority and highest priority ready task is selected for execution (lines $9-11$). If the selected task is not T_k^d then it is directly assigned to processor π for execution (line 21). Otherwise, if selected task is T_k^d then local scheduler checks if T_k^d is already executing on any other processor in the system. If T_k^d is not assigned on any other processor then local scheduler schedules T_k^d on π. Otherwise, the priority of T_k^d is inverted to allow subsequent higher-priority partitioned task from τ_{π_k} to execute on π.

Figure 5 illustrates how local scheduler on every processor schedules τ_{π_k} along with T_k^d. Being the highest priority tasks at time instant $t = 0$ on all processors, T_k^d qualifies to execute on all three processors simultaneously. However, once T_1^d starts its execution on π_1, priorities of T_2^d and T_3^d are inverted. Note that, for second job of T_k^d at time instant $t = 3$, T_3^d starts first on π_3 instead of T_1^d on π_1. This dynamic relocation of T_k^d comes from the priority order assigned by local scheduler. For instance, when T_1^d has lower priority than any of the locally partitioned tasks on π_1, it cannot compete for priority among T_2^d and T_3^d present on π_2 and π_3, respectively, and therefore, the order in which T_k^d appears will change.

2) Top-level Scheduler: Top-level scheduler also uses an instance of single-processor optimal EDF scheduling algorithm for migrating sub-set of tasks –i.e., τ_{glob}. Recall that the overall task set τ is considered a priori feasible and the tasks present in τ_{glob} are the tasks that were impossible to be statically partitioned. Thus, the platform resource requirement of τ_{glob} is not more than the under-utilization available in the system. Top-level EDF scheduler visualizes the fragmented amount of computation power available on different processors, which is accessible in a *sequential* manner. Algorithm 4 illustrates jobs assignment on processors by top-level scheduler. If global ready task queue ($ReTQ(\tau_{glob})$) is not empty then at most Y tasks (here, Y refers to the number of processor-groups in the system) are selected for execution (lines $1-7$) such that each selected task executes over each processor-group. Within each group, the top-level scheduler looks for T_k^d task. If T_k^d is running on any of the processors then selected task from $ReTQ(\tau_{glob})$ for that group is assigned on the processor for at most C_k^d units of time (lines $8-14$). Otherwise, if T_k^d is not running on any of the processors of the selected group then task from $ReTQ(\tau_{glob})$ remains suspended until T_k^d starts running.

Figure 6 illustrates that when T_k^d starts executing on a processor, top-level EDF scheduler fills its empty C_k^d with the execution requirement of highest priority task available in τ_{glob} (recall that T_k^d is an empty task). As soon as T_k^d finishes on one processor, top-level scheduler preempts the running tasks from $ReTQ(\tau_{glob})$ and migrates it to the next processor that runs T_k^d within the same processor-group.

D. Schedulability Analysis

In this section, we provide the reader the schedulability analysis of two-level hierarchical scheduling algorithm. We use *demand bound analysis* for this purpose [5], [10]. Demand

Algorithm 4 Top-level scheduler: Online jobs assignment for migrating tasks present in τ_{glob}

```
1: Y ← number of processor-groups
2: sort ReTQ(τ_glob) w.r.t. EDF priority
3: for i = 1...Y do
4:    m_i ← number of processors in processor-group i;
5:    if size(ReTQ(τ_glob)) ≠ 0 then
6:       T_i ← highest priority ready task among τ_glob;
7:    end if
8:    for k = 1...m_i do
9:       if T_k^d is running then
10:         π_k ← T_i; // T_i executes for C_k^d time units on π_k
11:         break;
12:      end if
13:   end for
14: end for
```

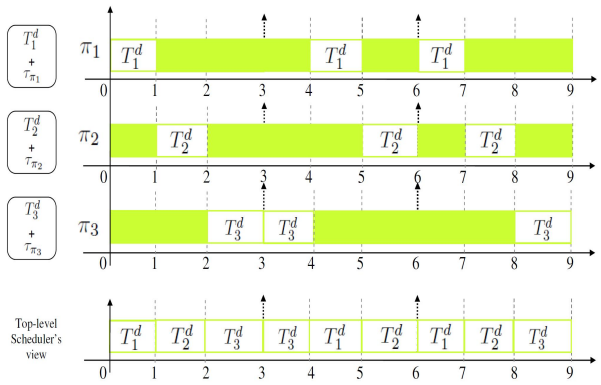


Figure 6. View of runtime schedule by top-level and local-level schedulers under 2L-HiSA on an SMP architecture.

bound analysis is a general methodology for schedulability analysis of EDF scheduling algorithm in single-processor systems. Demand bound analysis uses the concept of *demand function (df)*. Demand function computes the maximum amount of time, so-called *processor demand*, consumed by all jobs of a task T_i that have both release times and deadlines within an interval $[t_1, t_2]$. Demand function for a task T_i can be given by equation 5.

$$df_i(t_1, t_2) = \sum_{r_{ij} \geq t_1, d_{ij} \leq t_2} C_{ij} \tag{5}$$

Similarly, for the entire task set, demand function is simply a sum of individual demand functions of tasks over the same time interval as given by equation 6.

$$df(t_1, t_2) = \sum_{i=1}^{n} df_i(t_1, t_2) \tag{6}$$

It has been shown in [5], [22] that the EDF-schedulability of arbitrarily-deadline task systems can be tested by the demand function: all tasks are guaranteed to meet deadlines by EDF on single processors, if and only if the condition in equation 7 holds for $\forall L > 0$, where L=$t_2 - t_1$. On a single-processor system, this is a necessary and sufficient condition for EDF-schedulability.

$$df(t_1, t_2) \leq (t_2 - t_1) \qquad \forall t_1, t_2 \tag{7}$$

We divide the schedulability analysis of 2L-HiSA into two parts. In the first part, we analyze the EDF-schedulability of migrating tasks and in second part, we analyze EDF-schedulability of partitioned tasks.

1) Schedulability of migrating tasks: As discussed earlier in section IV-B, subset of migrating tasks can not have an aggregate utilization (τ_{glob}) more than the under-utilization available in the system ($U_{sum}(\tau_{glob}) \leq \sum_{k=1}^{m} U'_{\pi_k}$). We have illustrated in figure 6 that this under-utilization is proportionately fragmented over different processors of the system and the computation power not more than the equivalent of one processor is freely available within each group. Top-level EDF scheduler, thus, has this fragmented computation power (more than or equal to the cumulated execution requirement of migrating tasks) available in the system to which, it can access in a sequential manner thanks to the explicit occurrence of T_k^d (see figure 6). Moreover, T_k^d is the most frequently occurring task on every processor (i.e., it recurs over the smallest period). Thus, migrating tasks always find the portion of processor time reserved for them, which is sufficient w.r.t. their execution requirement. Partitioned tasks, on the other hand, find the remaining non-reserved time units to execute.

2) Schedulability of partitioned tasks in the absence of T_k^d: In a multiprocessor system with fully partitioned task set, the problem of schedulability analysis is reduced to multiple *single*-processor systems. Therefore, it is sufficient to prove that all tasks that are partitioned on a processor $\pi_k (\forall k, 1 \leq k \leq m)$ respect their deadlines. We consider the EDF-schedulability on every processor individually. First, let us consider that only statically partitioned tasks are present on every processor and T_k^d does not exist. We assume that the complementary relation of equation 8 and equation 9 holds on any processor π_k due to NP-hardness of partitioning problem.

$$U_{\pi_k}(\tau_{\pi_k}) = \sum_{i=1}^{np} \frac{C_i}{P_i} < 1 \qquad (\forall i, 1 \leq i \leq np, \forall T_i \in \tau_{part}) \tag{8}$$

$$U'_{\pi_k}(\tau_{\pi_k}) = 1 - \sum_{i=1}^{np} \frac{C_i}{P_i} > 0 \qquad (\forall i, 1 \leq i \leq np, \forall T_i \in \tau_{part}) \tag{9}$$

For synchronous task system, demand function changes values only at discrete time instants corresponding to arrival times and deadlines of a task. Therefore, the demand function needs to be verified only for those values of time intervals that are aligned with deadlines of jobs. Moreover, the worst case demand is found for intervals starting at 0 due to synchronized release instants of all tasks. We believe that the hyper-period (i.e., least-common-multiple of task periods) is a safe interval length to analyze demand function. Thus, we consider that the worst-case demand interval on every processor π_k is defined from 0 to the hyper-period (let us say H) of partitioned tasks -i.e., $[t_1, t_2]$=$[0, H]$. As long as $U_{\pi_k}(\tau_{\pi_k}) < 1$ –i.e., the aggregate utilization of partitioned tasks is less than the computation power of a single processor, the demand function of all partitioned tasks on processor π_k is strictly less than the amount of time available in the time interval $[0, H]$ as given by equation 10. Hence, all partitioned tasks respect the necessary and sufficient schedulability condition of EDF scheduling in the absence of T_k^d on every processor independently. Equation 10 also holds for any sub-interval of time $[0, t]$ $(\forall t, 0 < t \leq H)$.

$$\forall t_1, t_2 \qquad df(t_1, t_2) \leq (H - 0) \tag{10}$$

3) Schedulability of partitioned tasks in the presence of T_k^d: In this section, we consider the EDF-schedulability of partitioned tasks in the presence of T_k^d on every processor individually. In order to be EDF-schedulable, a single-processor

system must satisfy the inequality presented by equation 11 in the presence of T_k^d.

$$\sum_{T_i \in \tau_{\pi_k}} df_i\,(T_i, H) + df(T_k^d, H) \leq H \tag{11}$$

The first addend refers to demand function of partitioned tasks and second addend refers to the demand function of T_k^d on processor π_k, respectively. Recalling from section IV-B, the size of time slot reserved by T_k^d –i.e., C_k^d, on any processor π_k is in proportion to $U_{\pi_k}^{'}$ available on π_k. Moreover, T_k^d competes for priority at runtime at local scheduler's level, thus, T_k^d is treated as any other partitioned task by the local scheduler.

From equation 11, we can deduce that, by design, the demand function of partitioned tasks on processor π_k is always less than or equal to $(H-0) \times U_{\pi_k}$ as shown by equation 12. Similarly, from the complimentary relation of equation 9, we can deduce that the amount of time allocated to T_k^d is less than or equal to $(H-0) \times U_{\pi_k}^{'}$ as shown by equation 13

$$\sum_{T_i \in \tau_{\pi_k}} df_i\,(T_i, H) = (H-0) \times U_{\pi_k} \tag{12}$$

$$df(T_k^d, H) = (H-0) \times U_{\pi_k}^{'} \tag{13}$$

By substitution, the inequalities of equation 11 results in equation 14.

$$\sum_{T_i \in \tau_{\pi_k}} df_i\,(T_i, H) + df(T_k^d, H) \leq H \times \left[\sum_{i=1}^{np} \frac{C_i}{P_i} + 1 - \sum_{i=1}^{np} \frac{C_i}{P_i}\right] \tag{14}$$

$$\sum_{T_i \in \tau_{\pi_k}} df_i\,(T_i, H) + df(T_k^d, H) \leq H \tag{15}$$

Equation 15 illustrates that the overall demand function of partitioned tasks together with T_k^d is still less than or equal to the amount of time available in hyper-period (H). Therefore, necessary and sufficient conditions of EDF-schedulability holds at local scheduler-level as well. The bound on schedulable utilization of tasks under the 2L-HiSA scheduling algorithm depends on the following condition.

Condition-I: *Subset τ_{part} shall be partitioned on* m*-processors of the platform in such a way that the under-utilization per group of processors is less than or equal to 1.*

The assignment of tasks to processors is often seen as analogous to bin-packing problem, which is considered a strong NP-hard problem [13]. The NP-hardness of partitioning problem can often be a limiting factor for our proposed algorithm. However, the fact that 2L-HiSA makes clusters of identical processors such that, per cluster, the unused fragmented computation power equivalent to at most one processor is available, improves on the schedulable utilization bound of EDF for multiprocessor systems. Clustering of processors instead of considering individual processors, helps in alleviating bin-packing limitations by effectively increasing *bin* sizes in comparison to *item* sizes. With a cluster of processors, it is much easier to obtain the under-utilization per cluster less than or equal to the processing power equivalent to one processor as compared to finding an optimal partitioning of tasks on a single processor. 2L-HiSA is an optimal algorithm for hard real-time tasks if a subset of tasks can be partitioned such that the under-utilization per cluster of processors remain less than or equal to the processing power equivalent of one processor. General schedulable utilization bound of 2L-HiSA is greater than EDF and less than *m*. However, the exact bound depends on efficient partitioning.

V. Experimental Setup and Results

In this section, we provide the reader the simulation-based evaluation of the 2L-HiSA scheduling algorithm. We have validated that the analytical optimality of 2L-HiSA holds in practice and all real-time tasks of target application respect their timing constraints (deadline guarantees). We have also analyzed scheduling-related overheads compared to existing scheduling strategies. The evaluation is performed using a free-ware Java-based simulation tool called STORM (Simulation TOol for Real-time Multiprocessor Scheduling) [23]. This tool is intended mainly to evaluate, through simulation, various multiprocessor scheduling algorithms for their functionality and performance by abstractly defining both application as well as architecture parameters of an entire multiprocessor system. We have used the parameters of Marvell's XScale® technology-based embedded processor PXA270 [24] in our simulations. Let's consider a set of ten real-time periodic and independent tasks $(n = 10)$, referred as τ to be scheduled on an SMP-type processing platform composed of four processors $(m = 4)$. Table I presents the quadruple values of all these tasks. Note that the task names start from T_5. This is because initial four task names (from T_1 to T_4) are reserved to represent *dummy* task T_k^d (from T_1^d to T_4^d) on processors from π_1 to π_4, respectively. Task set τ has an aggregate utilization $U_{sum}(\tau)$=4.0.

We simulate τ in two steps. In the first step, we simulate τ under global EDF algorithm to highlight its limitations when 100% platform resources are utilized. In the second step, we simulate τ with 2L-HiSA. Obviously, τ remains schedulable with other optimal multiprocessor scheduling algorithms like PFair, LLREF, and ASEDZL as well. Figure 7 provides a snapshot view, generated by STORM simulator, of the simulation traces when task set τ executes on four processors under global EDF scheduling algorithm. One can notice in figure 7 that despite 100% workload ($U_{sum}(\tau)$=4.0), some processors remain idle momentarily as highlighted with dotted line boxes. This is due to the inherent priority mechanism of global EDF scheduling algorithm. As a consequence, some of the lower priority tasks, such as T_{12} and T_{13}, miss their deadlines as illustrated in figure 8. The jobs of a task for which deadline miss occurs is highlighted with oval-shaped red box beneath the simulation trace of each task. This illustration validates theoretically known sub-optimality of global EDF scheduing algorithm.

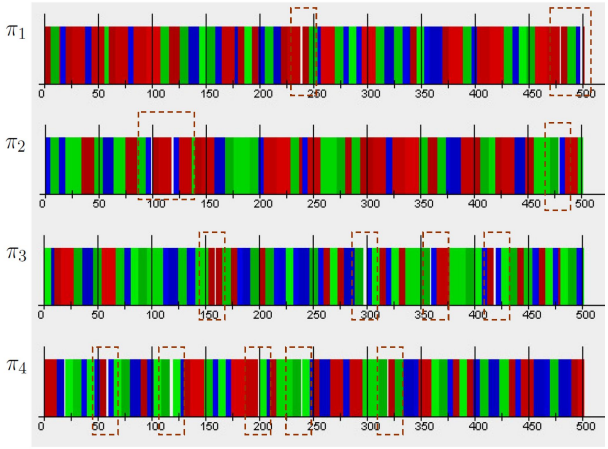


Figure 7. Simulation traces of EDF global scheduling of task set τ on four processors

In the following, we simulate the same application task set with the 2L-HiSA algorithm. In the first phase of 2L-HiSA algorithm –i.e., the task-partitioning phase (see section IV-B), each task is statically assigned to a particu-

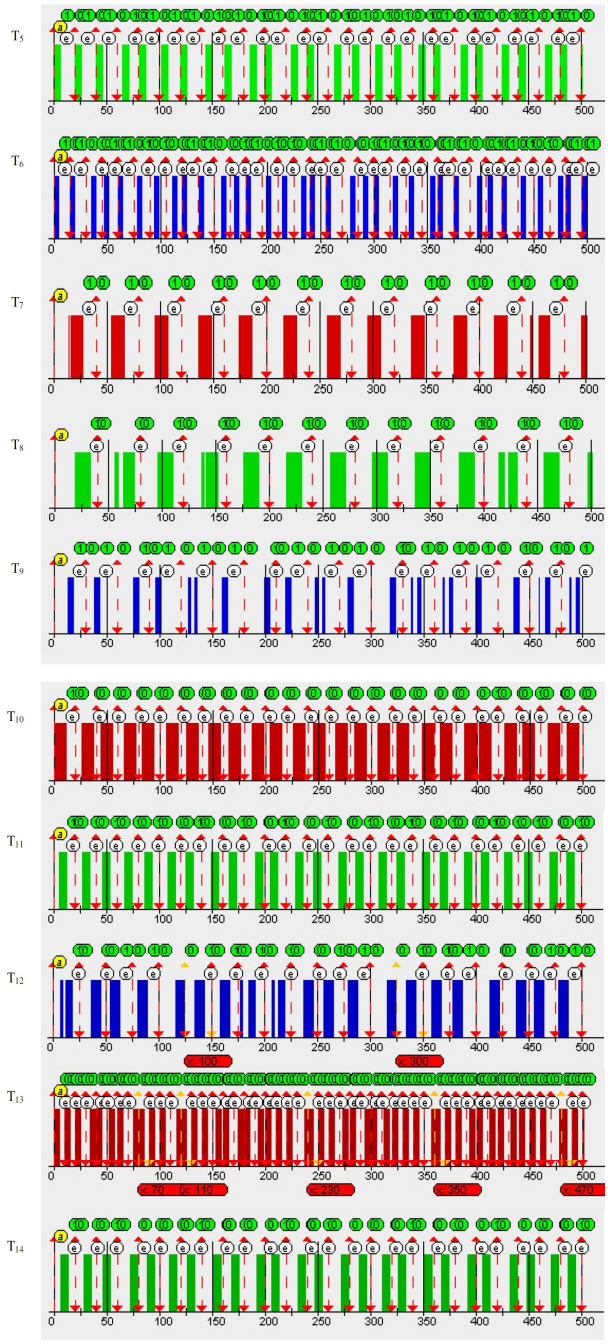


Figure 8. Simulation traces of individual tasks under global EDF scheduling algrithm

lar processor by following the bin-packing approach[2]. We obtain $\tau_{part} = \{T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}, T_{12}\}$ such that τ_{π_1}=$\{T_5, T_6\}$, τ_{π_2}=$\{T_7, T_8\}$, τ_{π_3}=$\{T_9, T_{10}\}$, τ_{π_4}=$\{T_{11}, T_{12}\}$, and τ_{glob}=$\{T_{13}, T_{14}\}$.

In the first stage, only τ_{part} is executed on Π. Figure 9 illustrates that idle time intervals appear on every processor due to under-utilization. We calculate this under-utilization using equation 9. In this case, processors π_1, π_2, π_3, and π_4 are under-utilized by a factor of 0.3, 0.3, 0.2, and 0.2, respectively. Since

[2]Tasks are partitioned manually to processors. This may not be the best possible partitioning solution for given task set, but it is good enough to illustrate how the 2L-HiSA algorithm functions. However, efficient task partitioning approaches can be used in this phase.

TABLE I.
REAL-TIME PERIODIC TASK SET τ

Task Name	r_i	C_i	d_i	P_i
T_5	0	6	20	20
T_6	0	6	15	15
T_7	0	13	40	40
T_8	0	15	40	40
T_9	0	6	30	30
T_{10}	0	12	20	20
T_{11}	0	8	20	20
T_{12}	0	10	25	25
T_{13}	0	6	10	10
T_{14}	0	8	20	20

$\sum U'_{\pi_k} \leq 1$ for all processors in Π, therefore, all processors can be clustered in the same group.

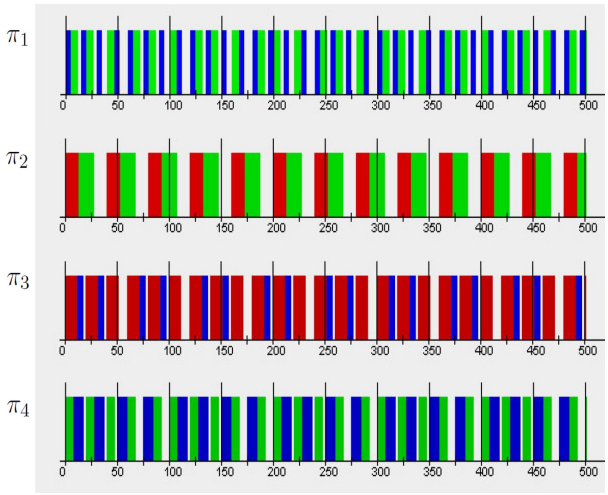


Figure 9. Simulation traces of partitioned tasks under EDF local scheduler on each processor

After task-partitioning and processor-grouping phases are complete, we can add dummy task T_k^d on every processor in proportion to the available under-utilization. The parameters of T_k^d such as, its period P_k^d and worst-case execution time C_k^d are calculated using equation 3 and 4, respectively. From table I, it is straightforward to obtain the smallest period of all tasks, which is equal to 10 –i.e., $P_k^d = 10$. Based on the amount of U'_{π_k} available on each processor over the smallest possible period P_k^d, one can compute the worst-case execution time C_k^d of T_k^d using equation 4. Resulting *dummy* task set is given in table II.

TABLE II.
DUMMY TASKS INTRODUCED IN THE TASK SET τ

Task Name	r_i	C_i	d_i	P_i
T_1^d	0	3	10	10
T_2^d	0	3	10	10
T_3^d	0	2	10	10
T_4^d	0	2	10	10

Figure 10 illustrates simulation traces generated by local-level EDF scheduler on each processor in the presence of T_k^d (note that simulator outputs the task names as PTASK *taskname*). It can be noticed in this figure that T_k^d tasks appear sequentially on processors. In figure 10, task T_1^d (or *PTASK* T_1) appears first on processor π_1 for exactly 3 time units. Since, T_1^d starts its execution first, therefore, T_2^d, T_3^d, and T_4^d on π_2, π_3, and π_4,

respectively, invert their priorities to let the partitioned tasks execute. Once T_1^d finishes its execution, T_2^d starts executing on π_2 for its corresponding worst-case execution time (i.e., $C_2^d = 3$). This process repeats itself for all dummy tasks within each period P_k^d. As mentioned in section IV-B, note that T_k^d has neither a specific order of occurrence nor it is fixed a priori on processors. Rather it can dynamically relocate itself within P_k^d. Every time T_k^d is released, first, it has to compete for the priority among locally partitioned tasks (thanks to the choice of smallest period, T_k^d often has highest priority among locally partitioned tasks) and then compete for priority among T_k^d present on other processors within a group. Failure to obtain highest priority at any of these two levels cause a priority inversion for concerned task itself and corresponding processor can execute locally partitioned ready tasks (if any). However, once any of the T_k^d tasks start executing, no other T_k^d tasks can execute in parallel. The priority inversion for T_k^d on the same processor can be performed until the laxity of T_k^d becomes zero. For instance, T_4^d in figure 10 starts executing at time instant $t = 8$ at which, its laxity becomes zero –i.e., $L_4^d = 10 - (8 + 2) = 0$. After instant $t = 8$, it was no more possible to invert the priority of T_4^d without a deadline miss.

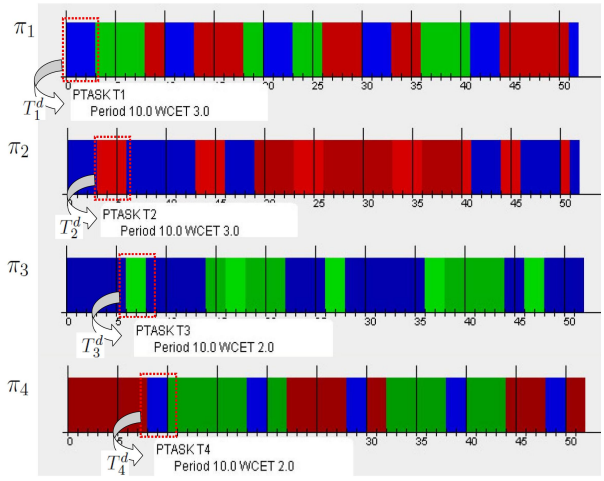


Figure 10. Simulation traces in the presence of T_k^d under EDF local scheduler on each processor

Finally, figure 11 illustrates a complete simulation trace of τ under two-level hierarchical scheduling algorithm along with T_k^d. Figure 11 illustrates that as long as any of the T_k^d task is executing on any of the processors in platform, top-level scheduler can manage to *fill* its C_k^d with the execution time of highest priority migrating task. Moreover, top-level scheduler can preempt and migrate the migrating task(s) to other processor(s) whenever any of the T_k^d task finishes on a processor. In figure 11, blue rectangular boxes on the time scale represent migrating task T_{13} and red rectangular boxes represent migrating task T_{14}. Note that we have carried out experiments with multiple real-time periodic task sets which were randomly generated to verify the working of 2L-HiSA. Providing one example task set here is for the sole purpose of elaborating different steps of the working principle in detail. Moreover, we have also provided a comparative analysis of the performance of 2L-HiSA algorithm w.r.t. the existing optimal multiprocessor scheduling algorithms to demonstrate its robustness and reduced complexity.

A. Performance Evaluation

In this section, we provide analysis of the performance of 2L-HiSA as compared to already existing global optimal scheduling

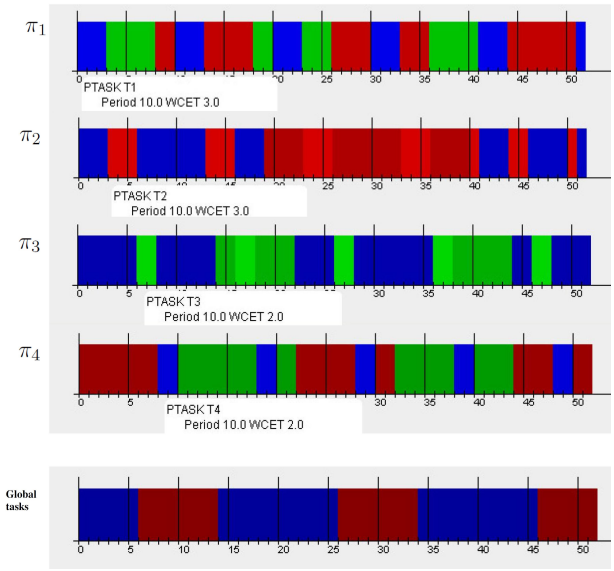


Figure 11. Simulation traces of global and partitioned tasks under EDF local and global schedulers

algorithms such as PFair [7], LLREF [9], and ASEDZL [10] algorithms. PFair and its heuristic algorithms are based on the concept of fluid scheduling mechanism in which, they select tasks to execute at each time instant. Doing so invokes the scheduler at every time instant, which introduces a lot of overhead in terms of increased release instants (r_i), task preemptions, and migrations. PFair is often criticized for its scheduling-related complexity. Unlike PFair, LLREF algorithm[3] is not based on time quanta but it increases preemptions of tasks to a great extent. LLREF schedules all ready tasks between any two release instants. Since all tasks are *active* at all time instants, therefore, context-switching overhead and cache-related preemption delay is significantly large for LLREF. ASEDZL algorithm, contrary to PFair, is not based on time quanta. Execution requirement and time periods of tasks can have any arbitrary value under ASEDZL algorithm. It improves on LLREF algorithm by scheduling minimum number of tasks between any two release instants. However, it still incurs higher number of scheduling events and preemptions than EDF scheduler. 2L-HiSA scheduling algorithm uses multiple instances of single-processor optimal EDF algorithm to schedule tasks both at top-level and local-level schedulers. Since, EDF invokes the scheduler only at job boundaries, therefore, the overhead in terms of release instants and number of preemptions is much less than the techniques discussed earlier. Furthermore, 2L-HiSA has reduced overhead of L1 cache memories due to the limited number of context-switches. Most of the tasks are partitioned under this algorithm, which limits the number of task migrations (only migrating or global sub-set of tasks migrate). Thus, the caches are mostly occupied by partitioned tasks, which helps in reducing the recovery time that a task may suffer from cache-miss and eventually improve performance. We compare the number of task preemptions and task migrations under 2L-HiSA, PD^2 PFair algorithm [8], and ASEDZL [10] algorithms for the task set presented in table I over a simulation time equal to one hyper-period –i.e., 600 time units. Figure 12 illustrates that the number of preemptions under PD^2 PFair algorithm for various number of tasks is the highest. We have estimated an average difference of 15-fold between preemptions under

[3]Although, we have analytically compared the performance of LLREF algorithm with other algorithms, we are unable to provide comparative analysis based on simulations.

PD^2 PFair and ASEDZL and an average difference of 18-fold between PD^2 PFair and 2L-HiSA. An average difference in the number of preemption between ASEDZL and 2L-HiSA has been estimated up to 1.3-fold. Note that these results take into account the preemptions of tasks under every local-level scheduler as well as top-level scheduler while using 2L-HiSA. Similarly, figure 13 illustrates the number of task migrations for various number of tasks. Still, migration of tasks under PD^2 PFair algorithm is relatively very high. We have estimated an average difference of 4-fold between task migration under PD^2 PFair and ASEDZL and an average difference of 10-fold between PD^2 PFair and 2L-HiSA. An average difference in the number of task migration between ASEDZL and 2L-HiSA has been estimated up to 2.6-fold. These results show that using the 2L-HiSA algorithm can be benevolent from performance point of view.

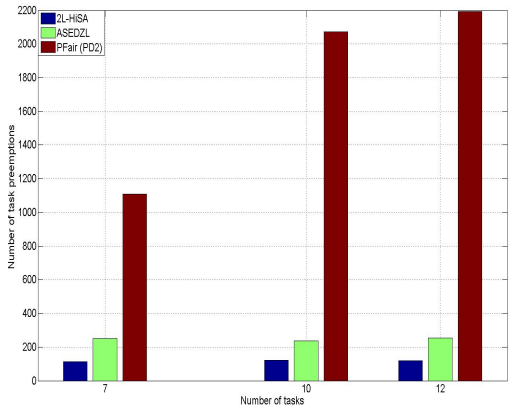


Figure 12. Number of task preemptions under 2L-HiSA, PFair (PD^2), and ASEDZL algorithms.

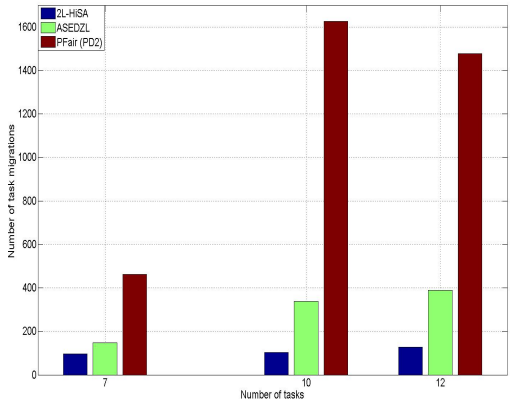


Figure 13. Number of task migrations under 2L-HiSA, PFair (PD^2), and ASEDZL algorithms.

VI. Conclusions

In this paper, we have presented a restricted migration-based multiprocessor scheduling algorithm, called 2L-HiSA. While EDF algorithm has the least runtime complexity among job-level fixed-priority algorithms for scheduling tasks on multiprocessor architecture, it suffers from sub-optimality in multiprocessor systems. 2L-HiSA addresses this sub-optimality of EDF algorithm and divides the problem into a two-level hierarchy of schedulers. We ensure that the basic intrinsic properties of optimal single-processor EDF scheduling algorithm appear both at local-level as well as at top-level scheduler. This algorithm works in two phases: 1) A task-partitioning phase in which, each task from application task set is assigned to a specific processor by following simple bin-packing approach. If a task can not be partitioned on any processor in the platform, it qualifies as migrating task. 2) A processor-grouping phase in which, processors are clustered together such that, per cluster, the unused fragmented computation power equivalent to at most one processor is available. 2L-HiSA improves on the schedulability bound of EDF for multiprocessor systems and it is optimal for independent and periodic hard real-time tasks if a subset of tasks can be partitioned such that the under-utilization per cluster of processors remain less than or equal to the computation power equivalent to at most one processor. The NP-hardness of partitioning problem, however, can often be a limiting factor. By clustering of processors instead of considering individual processors, 2L-HiSA alleviates bin-packing limitations by effectively increasing *bin* sizes in comparison to *item* sizes. With a cluster of processors, it is much easier to obtain the under-utilization per cluster less than or equal to the computation power equivalent to one processor. This paper provides simulation results to support our proposition.

References

[1] J. M. Calandrino, J. H. Anderson, and D. P. Baumberger, "A hybrid real-time scheduling approach for large-scale multicore platforms," in *Proceedings of the ECRTS'07 conference*, 2007, pp. 247–256.

[2] C. Farivar, "Intel Developers Forum roundup: four cores now, 80 cores later." 2006, http://www.engadget.com/2006/09/26/intel-developers-forum-roundup-four-cores-now-80-cores-later/.

[3] S. H. Funk, "Edf scheduling on heterogeneous multiprocessors," in *PhD thesis, department of computer science*. University of North Carolina at Chapel Hill, 2004.

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press LLC, 2003.

[5] S. Kato, N. Yamasaki, and Y. Ishikawa, "Semi-partitioned scheduling of sporadic task systems on multiprocessors," in *Proceedings of the 2009 21st Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 249–258. [Online]. Available: http://portal.acm.org/citation.cfm?id=1581378.1581524

[6] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 199–208. [Online]. Available: http://portal.acm.org/citation.cfm?id=1084012.1084161

[7] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel, "Proportionate progress: a notion of fairness in resource allocation," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, ser. STOC '93. New York, NY, USA: ACM, 1993, pp. 345–354. [Online]. Available: http://doi.acm.org/10.1145/167088.167194

[8] J. H. Anderson and A. Srinivasan, "Early-release fair scheduling," in *In Proceedings of the 12th Euromicro Conference on Real-Time Systems*, 2000, pp. 35–43.

[9] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling algorithm for multiprocessors," in

Proceedings of the 27th IEEE International Real-Time Systems Symposium. Washington, DC, USA: IEEE Computer Society, 2006, pp. 101–110. [Online]. Available: http://portal.acm.org/citation.cfm?id=1193218.1194408

[10] F. Muhammad, "Ordonnancement de tâches efficace et â complexitè maîtrisèe pour des systèmes temps rèel," in *PhD thesis*. University of Nice-Sophia Antipolis, 2009.

[11] S. Kato and N. Yamasaki, "Portioned edf-based scheduling on multiprocessors," in *Proceedings of the 8th ACM international conference on Embedded software*, ser. EMSOFT '08. New York, NY, USA: ACM, 2008, pp. 139–148. [Online]. Available: http://doi.acm.org/10.1145/1450058.1450078

[12] F. Dorin, P. Meumeu Yomsi, J. Goossens, and P. Richard, "Semi-partitioned hard real-time scheduling with restricted migrations upon identical multiprocessor platforms," in *Operating Systems (cs.OS), arXiv:*1006.2637v1 *[cs.OS], Cornell University Library Archives, Technical Report*, June, 2010.

[13] K. Funaoka, S. Kato, and N. Yamasaki, "A context cache replacement algorithm for pfair scheduling," in *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS)*, 2007, pp. 57 – 64.

[14] D. Johnson, "Near-optimal bin packing algorithms," in *PhD thesis*, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[15] B. Andersson and E. Tovar, "Multiprocessor scheduling with few preemptions," in *Proceedings of the 12th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 322–334. [Online]. Available: http://dx.doi.org/10.1109/RTCSA.2006.45

[16] P. Tan, J. Shu, and Z. Wu, "A hybrid real-time scheduling approach on multi-core architectures," in *Journal of software, vol. 5, No. 9, September 2010*. Academy Publisher, 2010, pp. 958–965.

[17] M. Dertouzos and A. Mok, "Multiprocessor scheduling in a hard real-time environment," in *IEEE Transactions on Software Engineering*, 1989, p. 1497 1506.

[18] K. S. Hong and J. Y.-T. Leung, "On-line scheduling of real-time tasks," in *IEEE Real-Time Systems Symposium*, Huntsville,Alabama, 1988, p. 244 250.

[19] S. Baruah, N. Cohen, G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Algorithmica*, 1996, p. 600 625.

[20] A. Srinivasan and J. Anderson, "Fair scheduling of dynamic task systems on multiprocessors," *J. Syst. Softw.*, vol. 77, pp. 67 – 80, July 2005. [Online]. Available: http://dx.doi.org/10.1016/j.jss.2003.12.041

[21] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, ser. RTSS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 93–. [Online]. Available: http://portal.acm.org/citation.cfm?id=882482.883823

[22] S. K. Baruah, A. K. Mok, and L. E. Rosier, "Preemptively scheduling hard-real-time sporadic tasks on one processor," in *In Proceedings of the 11th Real-Time Systems Symposium*. IEEE Computer Society Press, 1990, pp. 182–190.

[23] STORM, "STORM simulation tool," http://storm.rts-software.org.

[24] Marvell, "Marvell's XScale Microarchitecture," http://www.marvell.com/.