Harmonic-Aware Multi-Core Scheduling for Fixed-Priority Real-Time Systems

Ming Fan, *Student Member, IEEE,* and Gang Quan, *Senior Member, IEEE*

Abstract—This paper presents a new semipartitioned approach to schedule sporadic tasks on multicore platforms based on the *Rate Monotonic Scheduling* policy. To improve the schedulability, our approach exploits the fact that the utilization bound of a task set increases as task periods become closer to harmonic on single processor platforms. The challenge for our approach, however, is how to take advantage of this fact to assign and split appropriate tasks on different processors in the semipartitioned approach, and how to guarantee the schedulability of real-time tasks. We formally prove that our scheduling approach can successfully schedule any task set with a system utilization bounded by *Liu&Layland's bound* for N tasks, that is, $N(2^{1/N}-1)$. Our extensive experimental results demonstrate that the proposed algorithm can significantly improve the scheduling performance compared with the previous work.

Index Terms—Harmonic, real-time semipartitioned scheduling, fixed-priority, rate monotonic scheduling (RMS)

1 Introduction

As embedded applications become more and more complicated, embedded system designers rely more on multi-processor or multi-core platforms to obtain high computing performance [1], [2]. Meanwhile, due to the power/thermal constraints, the memory bottleneck, as well as the limitation of the instructional level parallelism in programs [3], industry is changing its gear toward the multi-core architecture rather than continuing to pursue high performance uniprocessor architecture. Conceivably, most of the future embedded systems will be built upon multi-core architectures. A major issue in developing multi-core computing systems is how to utilize the available computing resources most effectively. This is particularly critical for real-time systems with stringent timing constraints. It is a well known fact that scheduling real-time tasks on multiprocessor platform is NP-hard [4].

Traditionally, the well-known RT scheduling algorithms, such as the *Rate Monotonic Scheduling (RMS)* and *Earliest Deadline First (EDF)* scheduling, have been proven to be optimal for uniprocessor scheduling [5]. However, when the problem comes to multi-core platform, these optimal algorithms are no longer optimal any more [6].

There have been extensive literature published on real-time scheduling for multi-core systems [7], [8], [9], [10]. These scheduling algorithms can be largely categorized into two classes [6], [11]: the *partitioned* approach (e.g., [7]) and the *global (or non-partitioned)* approach (e.g., [8]). In the partitioned scheduling approach, each real-time task is assigned to a dedicated processor. All instances from the same task will be executed solely on that particular processor. In the global scheduling approach, all jobs from different tasks first enter a global queue, and thus each task can be potentially executed on any processor. Both approaches have their own pros and cons, and none of them dominates the other one in terms of schedulability [11].

Recently, a new multi-core scheduling approach, i.e., so called *semi-partitioned* approach [12], [13], [14], [10], [9], [15], [16], has been proposed. In the semi-partitioned scheduling approach, most tasks are assigned to one particular processor, i.e., the same as the partitioned scheduling approach. However, a few of tasks (i.e., no more than $(M-1)$ tasks, where M is the number of processors) are allowed to be split into several subtasks and assigned to different processors. From a different perspective, these tasks can migrate among different processors. The semi-partitioned approach not only outperforms the traditional partitioned approach and global approach theoretically [10], [15], [17], but also has been shown as sound and practical in the real implementation [16]. Furthermore, by implementing the semi-partitioned scheduling method in the Linux operating system, and running experiments on an Intel Core-i7 4-cores computer, Zhang *et al.* [18] showed that the overhead in the task migration can be relatively low, and thus its impact on the schedulability is small.

In this paper, we present a new semi-partitioned strategy and related schedulability analysis for sporadic tasks on multi-core platform based on RMS. Compared with the existing work on semi-partitioning of real-time tasks, we have made a number of novel contributions. First, we take the harmonic relation among tasks into consideration for fixed-priority semi-partitioned scheduling strategy on multi-core platform. As shown in our motivational example, taking advantage of harmonic property in semi-partitioned scheduling is non trivial. Two new semi-partitioned algorithms are developed. The first algorithm, namely *Harmonic Semi-Partition for Light tasks* (*HSP-light*), is

• *The authors are with the Department of Electrical and Computer Engineering, Florida International University, Miami, FL 33174 USA. E-mail: {mfan001, gaquan}@fiu.edu.*

Manuscript received 13 Apr. 2012; revised 10 Jan. 2013; accepted 19 Feb. 2013. Date of publication 18 Mar. 2013; date of current version 16 May 2014. Recommended for acceptance by H. Jiang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2013.71

intended for task sets with utilization factor of each task no more than 0.5. The second one, namely *Harmonic Semi-Partition* (*HSP*), is developed for more general task sets, i.e., the utilization factor of each task is no more than 1. Second, we present new schedulability analysis results for the semi-partitioned scheduling algorithms developed in this paper. Note that, to maximally utilize a processor such that adding more high priority tasks will cause deadline miss does not immediately imply the validity of *Liu&Layland's bound* [5] for semi-partitioned scheduling, since when a task needs to migrate to a different processor, its deadline becomes smaller than its period. We formally prove that the proposed algorithms can guarantee the schedulability for task sets with utilizations no larger than the *Liu&Layland's bound*. Moreover, different from the approach in [15], task sets with utilizations higher than the *Liu&Layland's bound* may also be schedulable with our approaches. Third, we conducted extensive experiments to study the performance of our approach, and our experimental results demonstrate that our proposed algorithms can significantly outperform previous work. A preliminary version of this paper has been published in [19].

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 introduces system models and other background information necessary for this paper. Sections 4 and 5 present two semi-partitioned algorithms we developed. Experiments and results are discussed in Section 6, and we present the conclusions in Section 7.

2 Related Work

In this section, we discuss the related work from two aspects: the work that exploit the harmonic property for periodic tasks and the work on semi-partitioned scheduling.

The property of harmonic tasks, i.e., the tasks with periods being integer multiples of each other, has been widely studied on uniprocessor systems. Compared with the *Liu&Layland's bound*, many researchers have proposed more efficient bound for RMS uniprocessor scheduling. One known result is that if all tasks are harmonic in a task set, the utilization bound can be as high as 1 [20]. Han *et al.* [21] proposed a polynomial-time method to determine the task set schedulability through testing the schedulability of a harmonic task set derived from the original task set. They proved that any task set that can pass the schedulability test by *Liu&Layland's bound* can pass the proposed test. Kuo *et al.* [22] presented another polynomial-time schedulability test method. By combining harmonic tasks into one task, the method can reduce the effective number of tasks and then the *Liu&Layland's bound* can be used to test the schedulability. There are also a number of other researches that study the relationship between system schedulability and task periods under RMS for uniprocessor scheduling [23], [24], [25]. For multiple processor RMS scheduling, Jung [26] *et al.* studied the problem of scheduling harmonic tasks on a uniform multiprocessor platform. Müller [27] adopted the schedulability test by Han *et al.* [21] to minimize the number of processors, and Fan *et al.* [28] proposed a scheduling technique that improves the system schedulability by taking advantage of the harmonic relation among tasks. All these work indicate that system schedulability can be greatly improved if harmonic relations among different tasks can be appropriately exploited for RMS scheduling on both single and multiple core platforms.

Semi-partitioned scheduling, by splitting a few tasks, has been shown as an effective and practical scheduling method to improve the system utilization significantly compared with the traditional global scheduling and partitioned scheduling (e.g., [12], [13], [14], [19], [10], [9], [15], [16].) As an example, the best known utilization bound for either global or partitioned fixed-priority schedule is no more than 50 percent [7], [29], [8], while the utilization bound can reach much higher using semi-partitioned scheduling. For instance, Lakshmanan *et al.* [10] have shown an utilization bound of 65 percent, and Guan *et al.* [15], [30] improved this bound to the traditional *Liu&Layland's bound*, i.e., 69.3 percent as the number of tasks goes to infinite, or any valid utilization bounds (such as the *K-bound* [22] or *R-bound* [31]) established on single processor platforms. Kandhalu *et al.* [32] proposed two semi-partitioned scheduling algorithms. They show that, for task sets with each individual task utilization factor no more than 0.5, the utilization bound can increase with the number of cores and approach 100 percent.

We believe that taking advantage of the harmonic relationship among task periods can greatly improve the schedulability of a semi-partitioned algorithm. Some of the existing approaches (such as the ones in [30], [32]) exploit this relationship by using the *R-Bound* [31], i.e., a utilization bound that takes the possible harmonic relationship into consideration. However, employing *R-bound* cannot determine the schedulability of a task set as accurate as the worst case analysis. Moreover, in order to use *R-bound*, all tasks have to go through a period transformation process. After the transformation, Kandhalu *et al.* [32] proposed to allocate the tasks with the smallest periods together. Unfortunately, these tasks do not necessarily form a task set closest to harmonic. In our approach, we developed a metric to quantitatively measure how harmonic a task set is, and based on this metric, to effectively allocate tasks closer to harmonic to the same processor. In addition, we can still employ the worst case analysis to determine the maximal capacity of a processor when adding a task to it and thus has a much better scheduling performance. The proposed scheduling algorithm can guarantee a utilization bound the same as *Liu&Layland's bound*.

3 Preliminary

We are interested in the problem of semi-partitioned scheduling of sporadic tasks on multicore platform based on RMS, which is known as an NP-hard problem [4]. In this section, we first present our system models used in this paper, and then we introduce some pertinent background information and concepts necessarily for our research. We then use an example to motivate our research.

3.1 System Models

The real-time system considered in this paper consists of N sporadic tasks, denoted as $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$, and executed

TABLE 1
Task Set with Five Real-Time Tasks

τ_i	C_i	T_i	u_i
1	2	6	0.33
2	5	10	0.50
3	3	12	0.25
4	4	20	0.20
5	15	25	0.60

on M identical processors, i.e., $\mathcal{P} = \{P_1, P_2, \ldots, P_M\}$. Each task $\tau_i \in \Gamma$, is characterized by a tuple (C_i, T_i), where C_i is the *worst-case execution time* of τ_i, and T_i is the *minimum inter-arrival time* between any two consecutive jobs of τ_i. T_i is also called the *period* of τ_i in this paper. For the sake of simplicity, we use Γ_{P_m} to denote the task set on processor P_m. For the rest of this paper, we make two assumptions: 1) the deadline of each task is equal to its period; 2) Γ is sorted with decreasing priority order, i.e., task τ_i has a higher priority than τ_j if $i < j$.

The *utilization factor* of a task τ_i is denoted as u_i where

$$u_i = \frac{C_i}{T_i}. \tag{1}$$

Based on its utilization factor, a task can be *light* or *heavy*, which we formally defined below:

Definition 1. *Task τ_i is called a light task if $u_i \leq \frac{1}{2}$, or a heavy task otherwise.*

Note that, even though we used the same terminology as that in [15], our definitions of light and heavy tasks are totally different. The *total utilization of a task set* Γ is denoted as $U(\Gamma)$ where

$$U(\Gamma) = \sum_{\tau_i \in \Gamma} u_i. \tag{2}$$

The *system utilization* of task set Γ on a multi-core platform with M processors is denoted as $U_M(\Gamma)$, where

$$U_M(\Gamma) = \frac{U(\Gamma)}{M}. \tag{3}$$

Liu and Layland [5] showed that a task set Γ can be feasibly scheduled by RMS on a uniprocessor if

$$U(\Gamma) \leq \Theta(N) = N\left(2^{1/N} - 1\right). \tag{4}$$

$\Theta(N)$ is also traditionally referred to as the *Liu&Layland's bound*.

3.2 On Semi-Partitioned Scheduling

A semi-partitioned scheduling algorithm consists of two phases: *the partitioning phase* and *the scheduling phase*.

In the *partitioning phase*, most tasks will be assigned to one processor and can be executed only at that particular processor during running time. These tasks are called *non-split tasks* [15]. A few other tasks, so called *split tasks*, are allowed to be split into several subtasks and assigned to different processors with the purpose of maximally utilizing the processor. Let task τ_i be a task that is split into three subtasks, i.e., $\tau_i^{b_1}$, $\tau_i^{b_2}$, and τ_i^t, executed on processor P_1, P_2, and P_3, respectively. The total execution time of $\tau_i^{b_1}$, $\tau_i^{b_2}$, and τ_i^t equals to C_i. Specifically, the last subtask of τ_i, i.e., τ_i^t is called *tail task*, and other subtasks of τ_i, i.e., $\tau_i^{b_1}$ and $\tau_i^{b_2}$, are called *body tasks*. For ease of presentation, we use C_i^B and u_i^B to represent the total execution time and utilization of all body tasks from a split task τ_i, respectively. Note that, once the partitioning phase is done, the assignment of a subtask to a processor is permanent and the subtask can only run on that designated processor.

In the *scheduling phase*, the scheduling strategy for each processor is determined. In our case, all tasks assigned to the same processor are scheduled strictly conforming to RMS policy, i.e., the task with a smaller period always has a higher priority. One complexity, however, is to execute multiple subtasks assigned to different processors according to the original logical order sequentially. Since the scheduler at the operating system level does not necessarily know the nature of a real-time process, to execute multiple subtasks from the same task concurrently may violate the data or control dependency and thus leads to invalid computing results. Therefore, it is vital to make sure that each subtask is executed according to its logical order and without overlapping with other subtasks.

We adopt an existing approach [9], [15], [17] to solve this problem and assume that an appropriate timer is available to monitor the execution of body/tail tasks. Specifically, the scheduler will assign a timer to a split task, e.g., τ_i in the above example. When τ_i arrives, the scheduler dispatches τ_i^{b1} to processor P_1 immediately and sets the timer to C_i^{b1}. After the timer expires, the scheduler then dispatches τ_i^{b2} to processor P_2 and sets the timer to C_i^{b2}. Then if the timer expires again, the scheduler releases τ_i^t to processor P_3. As such, all subtasks split from the same task can only run sequentially following their logical orders to ensure the correctness of program. Therefore, the body/tail tasks from the same task can be viewed as tasks with the same periods but different starting times, and the synchronization problem for split tasks from the same task can be easily resolved in practice. For more details about the semi-partitioned scheduling, readers can refer to [15], [9], [10], [33].

3.3 Motivation Examples

Before we present our approach in detail, we first use an example to motivate our research. Since tasks with harmonic relationship have much higher schedulability on a single processor, an intuitive approach would therefore be the one that groups harmonic tasks together and assigns them to one processor. Unfortunately, such a naive approach may not work in the semi-partitioned approach.

Consider a two-processor platform with a task set shown in Table 1. Since τ_1 and τ_3 are harmonic, we can group τ_1 and τ_3 to one processor, i.e., Processor 1. Similarly, we can group τ_2 and τ_4 to the other processor, i.e., Processor 2. Since no processor can accommodate τ_5 entirely, we have to split τ_5 between these two processors. There are two problems with this assignment. First, as shown in Fig. 1a, the maximum capacity that can be accommodated in Processor 1 is 10. Since the subtasks from τ_5 cannot be executed concurrently on two processors, at most 4 time units from Processor 2 can

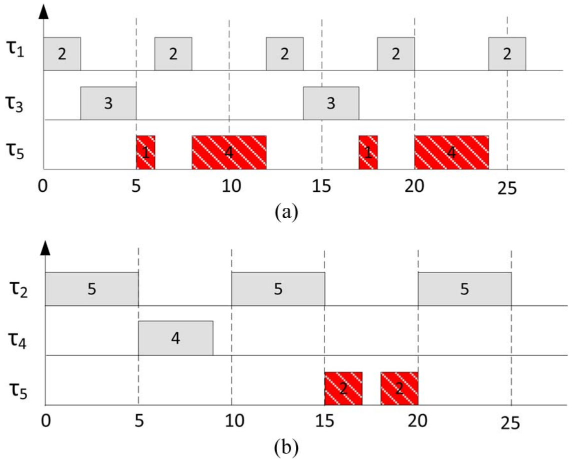


Fig. 1. Allocation fails when simply grouping harmonic tasks and assigning them to the same processor. (a) Processor 1. (b) Processor 2.

be utilized by τ_5 as shown in Fig. 1b. As a result, τ_5 cannot complete before its deadline even if all available time units are used for its execution. Second, in order to use all 4 time units on Processor 2, we need complicated process migration controls and synchronization mechanisms, which increase not only the switching overhead, but also the control complexity among different processors. Note that, if we assign τ_1 and τ_5 to one processor, and the other tasks to another processor, it is not difficult to verify that the schedule is feasible.

As indicated by this example, to take the advantage of harmonic relationship among tasks to improve the schedulability, a critical problem is how to judiciously choose the task to split and to synchronize among different processors. To solve this problem, we present two novel semi-partitioned algorithms, i.e., *HSP-light* and *HSP*, in the following sections.

4 THE HSP-LIGHT ALGORITHM

The *HSP-light* algorithm is a harmonic semi-partitioned algorithm developed for light tasks. When employing the harmonic relationship to improve the scheduling performance, it is not necessary that all tasks in the same task set are strictly harmonic. To this end, we first introduce a metric, namely the *harmonic index*, to quantify the degree of harmonicity for a task set. We then discuss our new algorithm that employs this metric. Finally, we study the schedulability of this algorithm.

4.1 Quantifying the Harmonicity

Since not all tasks in a given task set are harmonic, it is desirable that we can quantify the *harmonicity* of a task set, i.e., how close a task set is to a harmonic task set. Conceivably, the higher the harmonicity of a task set, the higher the system utilization can be. To achieve this goal, we first introduce the following concept.

Definition 2. *Given a task set $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_N\}$ sorted with decreasing priority order under RMS, where $\tau_i = (C_i, T_i)$, let $\Gamma' = \{\tau'_1, \tau'_2, \ldots, \tau'_N\}$ where $\tau'_i = (C_i, T'_i)$, $T'_i \leq T_i$, and $T'_i | T'_j$ if $i < j$. (Note $a|b$ means "a divides b" or "b is an integer multiple of a".) Then Γ' is called a sub harmonic task set of Γ.* Moreover, for any sub harmonic task set of Γ, let

$$\Delta U' = \begin{cases} U(\Gamma') - U(\Gamma), & \text{if } U(\Gamma') \leq 1, \\ +\infty, & \text{otherwise.} \end{cases} \quad (5)$$

From equation (5), $\Delta U'$ defines the "distance" of a task set to the corresponding sub harmonic task set in terms of its total utilization factor. If the utilization of that sub harmonic task set is greater than 1, then the "distance" is set to be infinity.

Given a task set, there may be more than one sub harmonic task sets. One type of sub harmonic task sets that is of most interest to us, which we call the *primary harmonic task set*, is formally defined as follows.

Definition 3. *Let Γ' be a sub harmonic task set of Γ. Then Γ' is called a primary harmonic task set* of Γ *if there exists no other sub harmonic task set Γ'' such that $T'_i \leq T''_i$ for all $1 \leq i \leq N$.*

We are now ready to define a metric, i.e., the *harmonic index*, to measure the harmonicity of a real-time task set.

Definition 4. *Given a task set Γ, let $\mathcal{G}(\Gamma)$ represent all primary harmonic task sets of Γ. Then the harmonic index of Γ, denoted as $\mathcal{H}(\Gamma)$, is defined as*

$$\mathcal{H}(\Gamma) = \min_{\Gamma' \in \mathcal{G}(\Gamma)} \Delta U'. \quad (6)$$

From equation (6), the harmonic index essentially defines the minimal "distance" of a task set to its primary harmonic task sets in terms of its total utilization factor. If no primary harmonic task set satisfies $U(\Gamma') \leq 1$, then the "distance" is set to infinity. In this paper, we adopt the DCT algorithm [21] to find primary harmonic task sets with a complexity of $O(N^2)$.

For a real-time task set and its primary harmonic task sets, it is not difficult to prove the following theorem [21].

Theorem 1: [21]. *Let Γ' be a primary harmonic task set of Γ. Then Γ is feasible on uniprocessor under RMS if $U(\Gamma') \leq 1$.*

In what follows, we introduce how we develop the HSP-light algorithm based on this index.

4.2 Algorithm Details

HSP-light algorithm assigns tasks to processors from lower priority to higher priority ones. A task is assigned to a processor that can accommodate it and also with the resulting task set having the lowest harmonic index. In other words, a task will be assigned to a feasible processor with the highest harmonic relationship for the resulting task set. The schedulability of the result task set can be guaranteed by performing the exact timing analysis [34] on the corresponding synchronized task set, i.e., assuming all tasks start at the same time. If a task cannot be accommodated entirely by any processor, then split occurs.

To split a task, we adopt a simple heuristic that assigns subtasks to the processor with the highest available capacity. There are two advantages using this splitting strategy: 1) It reduces the total split times by efficiently maximizing the workload for each split subtask. 2) It guarantees the priority of each body task to be the highest one on its host processor. After the split is done, the value to set up the timer for enabling the sub-task is also determined. Algorithm 1 shows the salient aspects of the HSP-light algorithm.

Algorithm 1 HSP-light Algorithm.

Require: $\forall \tau_i \in \Gamma$, $u_i <= 1/2$;
1: **while** $\Gamma \neq \emptyset$ **do**
2: $\tau_i :=$ the task with the lowest priority in Γ;
3: $P_m :=$ the processor with minimum $\mathcal{H}(\Gamma_{P_m} + \tau_i)$ in $\mathcal{P}$;
4: **if** $\Gamma_{P_m} + \tau_i$ is feasible **then**
5: Assign τ_i to processor P_m;
6: Continue;
7: **end if**
8: $P_m :=$ the processor with the maximum capacity (greater than 0) for τ_i;
9: **if** P_m does not exist, **then** break, **end if**
10: **if**{$\Gamma_{P_m} + \tau_i$ is feasible **then**
11: Assign τ_i to processor P_m;
12: **else**
13: Split τ_i into τ_{i1} and τ_{i2} such that $\Gamma_{P_m} + \tau_{i1}$ can maximally utilize P_m;
14: Assign τ_{i1} to processor P_m;
15: Replace τ_i by τ_{i2}, and move τ_i back to Γ;
16: **end if**
17: **end while**
18: **if** $\Gamma = \emptyset$ **then**
19: Return "Success!";
20: **else**
21: Return "Fail!";
22: **end if**

Given a task set Γ and a multiprocessor system $\mathcal{P}$, HSP-light makes the assignment decision for each task through the "while" loop from line 1 to line 17. Among all unassigned tasks left in Γ, the task τ_i with the lowest priority is selected (line 2). τ_i is assigned to the processor with the minimum harmonic index as long as that processor has enough capacity for the task on each processor (from line 4 to line 7). If this assignment fails, we split task τ_i and make the assignment (from line 8 to line 16). We choose the processor with the maximum execution capacity for τ_i. If the corresponding capacity is large enough, then τ_i is assigned entirely. Otherwise, we split τ_i and assign part of τ_i to the processor until it is maximally utilized, i.e., no other higher priority tasks can be assigned to that processor without causing other tasks to miss deadlines. Note that, to check the schedulability of a task set (line 4, line 10) and to calculate the maximum execution capacity available for splitting a task (line 13), we can use the traditional exactly timing analysis method [34] on the corresponding synchronized task set, i.e., tasks with the same starting time. The algorithm succeeds if all tasks are allocated, and fails otherwise. In what follows, we further study the schedulability of Algorithm 1.

4.3 Schedulability Analysis of HSP-Light

In this subsection, we are interested in examining how effective the algorithm HSP-light can be when scheduling real-time tasks on multi-core platforms. From the Algorithm 1, it is easy to conclude the following property.

Lemma 1. *If a task set Γ is successfully partitioned by HSP-light on M processors, then there is at most one body task on each processor; and on all processors, there are at most $(M-1)$ tasks to be split.*

Proof. In HSP-light, splitting occurs only when no processor - can accommodate one task completely. After splitting and assigning a task, the processor that accommodates the body task becomes full for higher priority tasks, and no other higher priority tasks can be assigned to it any more. The body task is the last task assigned to its host processor. Therefore, there is at most one body task on each processor. Since there are M processors, at most $(M-1)$ tasks will be split. □

Lemma 1 constrains the maximum number of tasks that can be split and migrated among different processors, and thus, the extra cost associated with the migrations. From Lemma 1, we can derive the following property.

Lemma 2. *Each body task has the highest priority on its host processor.*

Proof. According to Lemma 1, we know that there is at most one body task on each processor. Moreover, Algorithm 1 guarantees that any body is the last task assigned to its host processor. Since tasks are assigned from the lowest priority to the highest priority, the priority of any body task is higher than any other tasks on its host processor. □

More importantly, if a task set can be successfully allocated by HSP-light, all tasks can satisfy their deadlines. The conclusion is formally formulated in the following theorem.

Theorem 2. *If a task set Γ is successfully partitioned by HSP-light on M processors and scheduled according to RMS, then all tasks can meet their deadlines.*

Proof. For each body task, it has the highest priority at its host processor (Lemma 2). Therefore, it can always meet its deadline unless the worst case execution time of the original task is larger than its deadline, which is impossible. For tail tasks or any other regular tasks added to a processor, the schedulability of the entire task set is guaranteed based on the worst case response time analysis for the corresponding synchronous task set as stated above (lines 4, 10, and 13). □

From Theorem 2, HSP-light is not only an allocation method but also can serve as a schedulability test method as well. It is not surprising HSP-light is only a sufficient schedulability test method for multi-core scheduling problem. On the other hand, however, HSP-light is too complex to be used effectively as a schedulability checking

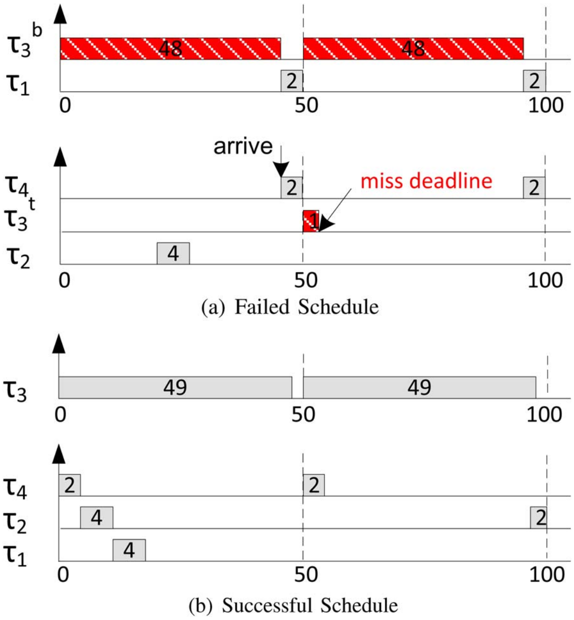


Fig. 2. (a) Task set is failed to be scheduled according to HSP-light. (b) Task set is schedulable if the heavy task τ_2 is pre-assigned.

method. Theorem 3 presents a faster schedulability checking method for our proposed algorithm.

Theorem 3. *Given a light task set* Γ *consisting of N tasks to be scheduled on M processors, if*

$$U_M(\Gamma) \leq \Theta(N), \tag{7}$$

then Γ *is feasible by HSP-light under RMS.*

The proof of Theorem 3 is rather complicated. Interested readers can refer to Appendix A for details. Theorem 3 shows that a light task set with system utilization bounded by the well-known *Liu&Layland's bound* is guaranteed to be feasible using our proposed approach, i.e., Algorithm 1.

It is worth mentioning that Theorem 2 is valid for any general task set, which implies that if a task set can be successfully allocated using HSP-light, all tasks can meet their deadlines. However, Theorem 3 works only for light task sets. In other word, HSP-light cannot guarantee the schedulability of a general task set (which contains heavy tasks), even if its total utilization is less than *Liu&Layland's bound*. In the next section, we introduce a more advanced algorithm, i.e., *HSP*, that can guarantee the schedulability for any task sets with system utilizations no more than the utilization bound.

5 The HSP Algorithm

The reason that HSP-light cannot guarantee the schedulability of an arbitrary task set with utilization lower than the utilization bound is that, if a split task is a heavy task and the tail task is very *light*, the overall system utilization can be very low. We use an example to explain this observation.

Consider a task set with four tasks, $\tau_1 = (4, 100)$, $\tau_2 = (4, 90)$, $\tau_3 = (49, 50)$, $\tau_4 = (2, 50)$, to be scheduled on 2 processors. As shown in Fig. 2a, even though the system utilization is very small, i.e., $(2/50 + 49/50 + 4/90 \ + 4/100)/2 = 0.55 < 0.69$, HSP-light cannot schedule this task set successfully. Note that the tail task from τ_2 can be viewed as a task with worst case execution time of 1 and deadline of 2. Adding any higher priority task with execution time more than 1 will make τ_2 infeasible. On the other hand, if we pre-assign the heavy task τ_2 to a processor, we can see that the task set can be successfully scheduled as shown in Fig. 2b. Therefore, in order to take the advantage of harmonic property to schedule general task sets, a special operation, i.e., the pre-assignment, needs to be performed for heavy tasks.

As discussed before, HSP-light can guarantee all tasks (light or heavy) meet their deadlines if all tasks can be assigned to a processor successfully. At the same time, Fig. 2 implies that heavy task pre-assignment can greatly improve the schedulability of the scheduling algorithm. The question becomes which heavy tasks should be pre-assigned and how other tasks should be assigned accordingly.

In HSP, the pre-assignment for heavy tasks follows the same strategy as introduced in [15]. Specifically, for any heavy task τ_i, let $\mathcal{P}_i^{Emp}$ denote the set of empty processors before τ_i's assignment and $|\mathcal{P}_i^{Emp}|$ denote the number of processors in this set. Then a heavy task τ_i needs to be pre-assigned to an empty processor if

$$\sum_{j>i} u_j \leq \left(\left|\mathcal{P}_i^{Emp}\right| - 1\right) \cdot \Theta(N). \tag{8}$$

The detailed procedure of HSP is shown in Algorithm 2. HSP is very similar to HSP-light, except for two important differences:

- At the beginning of semi-partitioning procedure, heavy tasks are pre-assigned to empty processor set, denoted as $\mathcal{P}^{Pre}$, if they satisfy the criteria as stated in (8) (from line 1 to line 8);
- To ensure that a body task always has the highest priority on a processor, a processor with heavy task pre-assignment may be excluded from the semi-partitioning process. According to Algorithm 2, a task can be assigned to a processor with heavy task assignment only after the heavy task pre-assigned in the processor has a lower priority (from line 12 to line 15).

Similar to Theorem 2, for HSP, the schedulability of tasks are guaranteed as stated in the following theorem.

Theorem 4. *If a task set* Γ *is successfully partitioned by HSP on M processors and scheduled according to RMS, then all tasks can meet their deadlines.*

Moreover, the *Liu&Layland's bound* for single processor can also be applied to HSP for schedulability checking. This conclusion is formally formulated in Theorem 5.

Theorem 5. *Given a task set* Γ *consisting of N tasks to be scheduled on M processors, if*

$$U_M(\Gamma) \leq \Theta(N), \tag{9}$$

then Γ *is feasible by HSP under RMS.*

For the proof of Theorem 5, please refer to Appendix B. Theorem 5 provides a very efficient schedulability checking method for real-time task sets scheduled by HSP. Given any task set Γ, if the total utilization of Γ satisfies Equation (9), then Γ can be successfully scheduled by HSP on M processors. Different from Theorem 3, Theorem 5 works for arbitrary task sets instead of light task sets alone. It is worth mentioning that, based on our proofs in the Appendix, Theorem 3 and Theorem 5 hold true even without the consideration of period relationships, i.e., lines 3-7 of Algorithm HSP-ligh and lines 16-19 of Algorithm HSP. To study if our approach can lead to a better utilization bound is an interesting problem and will be our future study. In what follows, we use experiments to study the potential improvement that can be achieved using our methods.

Algorithm 2 HSP Algorithm.

Require:

1) Task set: $\Gamma = \{\tau_1, \tau_2, \dots \tau_N\}$;
2) Multiprocessor: $\mathcal{P} = \{P_1, P_2, \dots, P_M\}$;

1: // pre-assign heavy tasks;
2: $\mathcal{P}^{Pre} = \emptyset$;
3: **for** $i = 1$ to N **do**
4: **if** $u_i > 1/2$ and $\sum_{j>i} u_j \leq (|\mathcal{P}_i^{Emp}| - 1) \cdot \Theta(N)$ **then**
5: Assign τ_i to processor P_m, where $m = |\mathcal{P}|$;
6: Move P_m from $\mathcal{P}$ to $\mathcal{P}^{Pre}$;
7: **end if**
8: **end for**
9: // assign other tasks;
10: **while** $\Gamma \neq \emptyset$ **do**
11: $\tau_i :=$ the task with the lowest priority in Γ;
12: $\tau_j :=$ the task with the lowest priority in $\Gamma_{\mathcal{P}^{Pre}}$;
13: **if** τ_i has higher priority than τ_j **then**
14: Move $P(\tau_j)$ from $\mathcal{P}^{Pre}$ to $\mathcal{P}$;
15: **end if**
16: $P_m :=$ the processor with minimum $\mathcal{H}(\Gamma_{P_m} + \tau_i)$ in $\mathcal{P}$;
17: **if** $\Gamma_{P_m} + \tau_i$ is feasible **then**
18: Assign τ_i to processor P_m;
19: Continue;
20: **end if**
21: $P_m :=$ the processor with maximum capacity for τ_i in $\mathcal{P}$;
22: **if** P_m does not exist, **then** Break, **end if**
23: **if** $\Gamma_{P_m} + \tau_i$ is feasible **then**
24: Assign τ_i to processor P_m;
25: **else**
26: Split τ_i into τ_{i1} and τ_{i2} such that $\Gamma_{P_m} + \tau_{i1}$ can maximally utilize P_m;
27: Assign τ_{i1} to processor P_m;
28: Replace τ_i by τ_{i2}, and move τ_i back to Γ;
29: **end if**
30: **end while**
31: **if** $\Gamma = \emptyset$ **then**
32: Return ''success'';
33: **else**
34: Return ''fail'';
35: **end if**

6 Experiments and Results

In this section, we investigate the performance of our proposed algorithms with experiments. Five algorithms are implemented in our experiments.

- *SPA*: The *SPA* algorithm [15] assigns the priority of each task by RMS, and splits a task to feed the processor until ''full'' (e.g., utilization equal to the *Liu&Layland's bound*). However, as long as the utilization of a task set exceeds the *Liu&Layland's bound*, it simply aborts.
- *DM_PM*: The *DM_PM* algorithm [9] assigns task priorities by deadline monotonic scheduling (DMS) policy, and splits a task and assigns as large portion of the task as possible to a processor by computing the maximum interference to the task on each processor.
- *PUB*: The *PUB* algorithm [30], similarly to *SPA*, assigns tasks based on a parametric utilization bound, but uses exact timing analysis method for task splitting. In the following experiments, *R-Bound* [31] is applied with this algorithm.
- *pCOMPATS*: The *pCOMPATS* algorithm [32] explores the R-Bound [31] for task partitioning and splitting. R-Bound can only be applied to task sets with ratio of any two periods no smaller than 1 and no larger than 2. In our experiments, we used the same algorithm as that in [32] to scale a general task set.
- *HSP*: Our proposed algorithm. Note that *HSP* is the same as *HSP-light* when the task set is light, and can accommodate task sets containing heavy tasks.

We conducted two groups of experiments to study how performance of each algorithm changes with different numbers of tasks and different system utilizations, respectively. For each group of experiments, we tested on different number of processors, i.e., $M = 4, 8$, and 16. For each testing point in the experiments, we randomly generated 500 task sets as test cases. The utilization of each task set varied from 0.5 to 1 (since task sets with smaller utilizations could be easily schedulable by all approaches). The minimum inter-arrival time of each task was set to have a uniform distribution within [50, 1000]. The scheduling performance for different approaches are compared using the *success ratios*, i.e., the number of feasible tasks over the number of total tasks generated under a specific test point.

6.1 Performance vs. Number of Tasks

In this group of experiments, we varied the number of tasks, i.e., N, in a task set from $2 \times M$ to $10 \times M$ with an increment of M (where M is the number of processors). The success ratios of all five approaches were recorded and plotted in Fig. 3.

From Fig. 3, we can observe that *HSP* can achieve success ratios much better than other four approaches. For example, in Fig. 3a, when the number of tasks is equal to 20, *HSP* can achieve a success ratio of 78 percent, an improvement of 1.7 times of that by *SPA* (45 percent), 1.1 times of that by *DM_PM* (71 percent), 1.2 times of that by *PUB* (64 percent), and 1.1 times of that by *pCOMPATS* (68 percent). The improvement of *HSP* comes from the fact that *HSP* takes the harmonic relationship among tasks

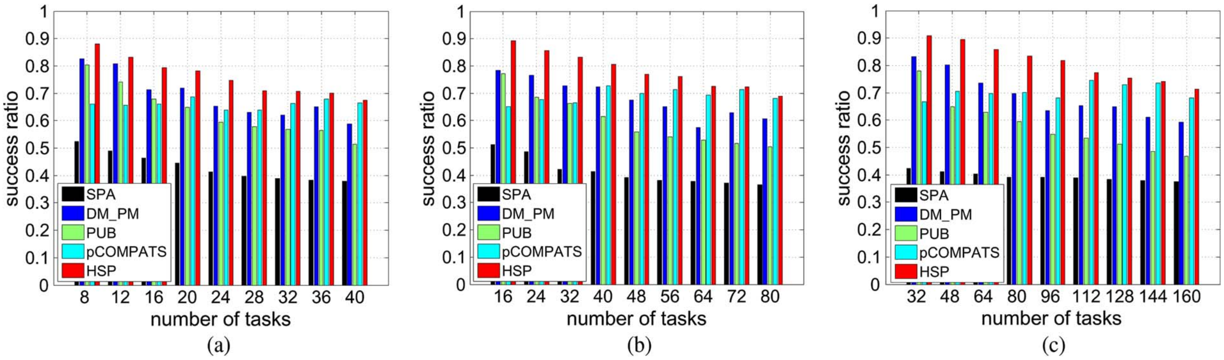


Fig. 3. Experimental results for general task sets by different number of tasks. (a) Number of processors: $M = 4$. (b) Number of processors: $M = 8$. (c) Number of processors: $M = 16$.

aggressively into consideration and tries to allocate tasks closer to harmonic together among multiple processors. The exploitation of harmonicity is limited to that the utilization bounds for different processors may be different depends on how existing tasks are close to harmonic.

From Fig. 3, we can see that, for the same number of processors (M), the success ratio of *HSP* in general decreases with the increase of task numbers (N). For example, in Fig. 3c (as $M = 16$), the success ratio of *HSP* achieves 91 percent when $N = 32$, but it decreases to 71 percent when N increases to 160. The larger the number of task is, the lower the utilization bound can be. As a result, a task set becomes more difficult to be schedulable. From Fig. 3, it is also interesting to see that, if we assume similar average number of tasks for each processor (i.e., assuming N/M as a constant), the success ratio by *HSP* largely increases in general. For example, when $N/M = 5$, the success ratios for $M = 4$, 8, 16 are 78 percent (see Fig. 3a at $N = 20$), 80 percent (see Fig. 3b at $N = 40$) and 83 percent (see Fig. 3c at $N = 80$), respectively. The reason for this is that the more processors are available, there are more opportunities that can be exploited by *HSP* to take advantage of the harmonic property among tasks to improve the processor utilization.

6.2 Performance vs. System Utilizations

To study the performance differences by different scheduling approaches under different system utilizations, we conducted three sub-groups of experiments, for light and general task sets, respectively. In light task sets, the utilization of each task was evenly distributed within [0, 0.5], while in general task sets, the utilization of each task was evenly distributed within [0, 1]. For each experiment, we varied the system utilization from 0.5 to 1.0 with an increment of 0.025. The experimental results for all approaches are collected and shown in Figs. 4 and 5.

Fig. 4 shows our experimental results for task sets containing only light tasks. From Fig. 4, we can observe that *HSP* can achieve success ratios significantly better than other four approaches. Compared with *SPA*, all other four approaches, i.e., *DM_PM*, *PUB*, *pCOMPATS*, and *HSP* can guarantee the schedulability of any task set with utilization below *Liu&Layland's bound*, the same as *SPA*. The success ratio by *SPA* drops sharply when system utilization around 0.7. This is because that while *SPA* can guarantee any task sets with utilizations no more than the *Liu&Layland's bound*, it rejects any task set with system utilization exceeding the *Liu&Layland's bound*. While *DM_PM*, *PUB* and *pCOMPATS* may potentially schedule task sets with utilization higher than the *Liu&Layland's bound*, *HSP* can achieve a much higher performance, especially when the system utilization is high. For example, in Fig. 4a, when the system utilization is around 0.9, *HSP* can still achieve a success ratio up to 30 percent, while that of *DM_PM* is 10 percent, and that of *PUB* and *pCOMPATS* are no more than 5 percent. Similar to our first group of experiments, we can see that the performance improvement by *HSP* tends to increase as the number of processors increases. Under the system utilization of 0.9, *HSP* can achieve a success ratio of 30 percent

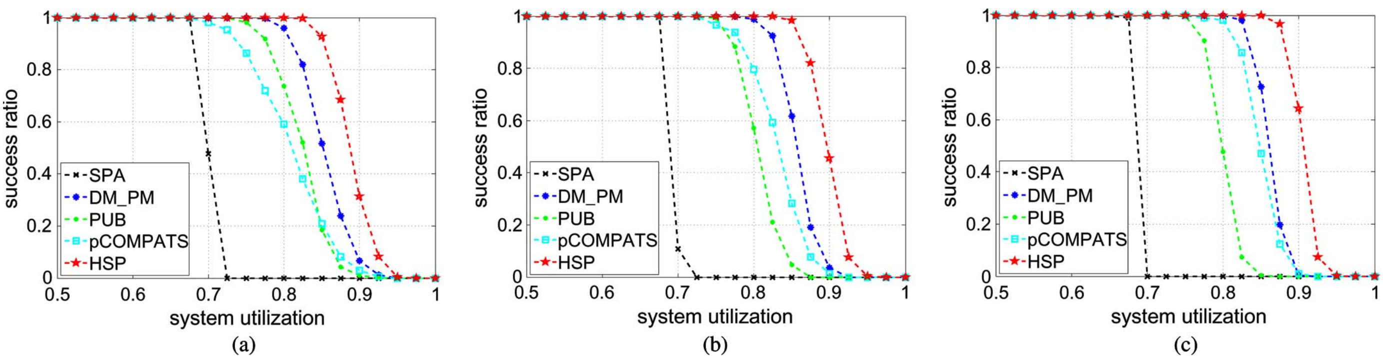
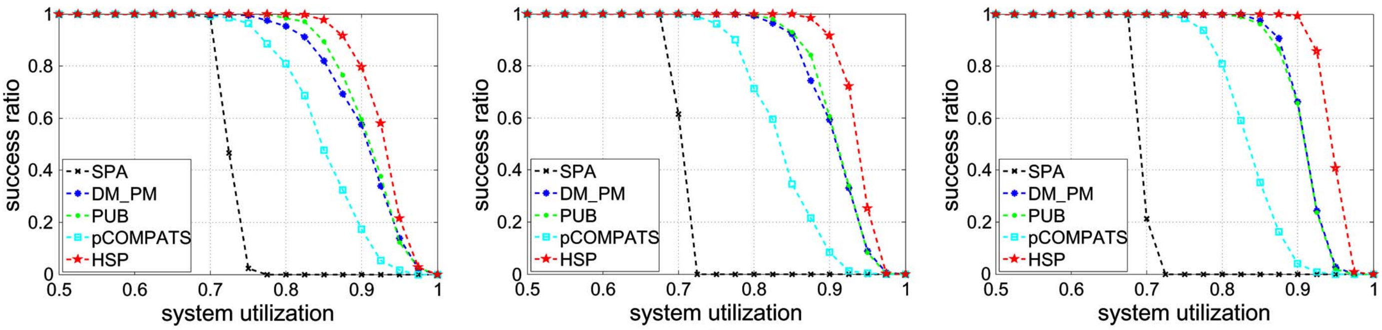


Fig. 4. Experimental results for light task sets, $u \in [0, 0.5]$. (a) Number of processors: $M = 4$. (b) Number of processors: $M = 8$. (c) Number of processors: $M = 16$.



(a) (b) (c)

Fig. 5. Experimental results for general task sets, $u \in [0, 1]$. (a) Number of processors: $M = 4$. (b) Number of processors: $M = 8$. (c) Number of processors: $M = 16$.

with 4 processors, 40 percent with 8 processors, and increased up to 60 percent with 16 processors.

Fig. 5 shows our experimental results for general task sets containing both heavy and light tasks. From Fig. 5, we can also observe that *HSP* performs significantly better than other four approaches. In Fig. 5c, *HSP* can achieve a success ratio four times of that by *DM-PM* and *PUB* when the system utilization is around 0.925.

Our experimental results clearly show, by exploiting the harmonic relationship among tasks more aggressively, HSP can significantly improve the schedulability of semi-partitioned scheduling compared with the existing algorithms.

7 Conclusion

In this paper, we have presented a new semi-partitioned approach for scheduling real-time sporadic tasks on multi-core platform under RMS. Our approach can take advantage of the harmonic relations among task periods and improve the schedulability. To achieve this goal, we introduced a metric to quantify how close a task set is to a harmonic task set. Two algorithms, i.e., HSP-light and HSP, were presented to schedule light and general task sets, respectively. We have formally analyzed the schedulability for both algorithms, and presented a simple schedulability test method for each one. Specifically, we formally proved that our scheduling algorithms can successfully schedule any task set with a system utilization bounded by the *Liu&Layland's bound*. The experimental results demonstrated that the proposed algorithm can significantly improve the scheduling performance compared with previous work.

Appendix A
Proof of Theorem 3

Before we introduce the theorem formally, we first study the schedulability of a task set containing a *critical task*, with its formal definition presented in Definition 5.

Definition 5. *Let $\Gamma = \{\tau_1, \ldots, \tau_i, \ldots, \tau_N\}$ be a task set that is schedulable by RMS on a single processor. τ_i is called the critical task if when increasing the execution time of the highest priority task, τ_i is the first task to miss its deadline.*

In addition, for ease of presentation, we introduce the following definition.

Definition 6. *A processor is called to be maximally utilized by a task set if any increase of the execution time for its highest priority task will cause at least one task on the same processor to miss its deadline.*

In a semi-partitioned system, after partitioning, we divide the tasks into three types: non-split task, body task and tail task. According to Lemma 2, a body task always has the highest priority on its host processor. Thus, from Definition 5, no body task can be a critical task.

Lemma 3. *The critical task on each processor can only be a non-split task or a tail task.*

In what follows, we want to study the schedulability characteristics for processors containing non-split or tail tasks that are critical tasks. We assume that a split task τ_i is split into B_i body tasks and one tail task, denoted as $\tau_i^{b_j}$ ($j \in [1, B_i]$) and τ_i^t, respectively.

For two different types of critical tasks, i.e., non-split tasks and tail tasks, we introduce two important properties, which are formulated in the following lemmas.

Lemma 4. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP-light. If the critical task is a non-split task and P_m is maximally utilized* by Γ_{P_m}, then $U(\Gamma_{P_m}) > \Theta(N)$.

Proof. By contradiction. Assume that processor P_m is *maximally utilized* by Γ_{P_m} but

$$U(\Gamma_{P_m}) \leq \Theta(N). \tag{10}$$

Let N_m denote the number of tasks on P_m, and let a non-split task τ_j be the critical task on P_m. Then we know that $N_m < N$. Since $\Theta(N)$ is a monotonically decreasing function with respect to N, we have $\Theta(N) < \Theta(N_m)$. According to our assumption in equation (10), we get

$$U(\Gamma_{P_m}) \leq \Theta(N) < \Theta(N_m).$$

Note that Γ_{P_m} may contain some tail tasks with deadlines less than their periods. Given Γ_{P_m}, we can always construct another Γ'_{P_m} such that any tail task in Γ'_{P_m} has its deadline equal to its original period. As such, we have

$$U\left(\Gamma'_{P_m}\right) = U(\Gamma_{P_m}) \leq \Theta(N) < \Theta(N_m).$$

Also, since τ_j is a non-split critical task, processor P_m is also *maximally utilized* by Γ'_{P_m}.

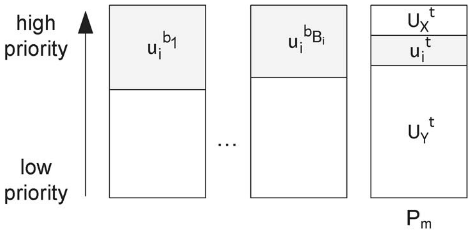


Fig. 6. Illustration of U_X^t and U_Y^t.

Now consider the critical task τ_j. Let us keep its period (T_j) the same, but increase its execution time such that

$$\Delta u_j = \min\big(\Theta(N_m) - U(\Gamma_{P_m}), 1 - u_j\big).$$

After the above transformation, the new utilization on P_m, denoted as $U(\Gamma''_{P_m})$ still satisfies that $U(\Gamma''_{P_m}) \leq \Theta(N_m)$, which implies that Γ''_{P_m} is feasible by RMS on processor P_m even though τ_j's execution time increases. This contradicts that P_m has been *maximally utilized* by Γ'_{P_m} and τ_j is the critical task. □

Lemma 5. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP-light. If the critical task is a tail task and P_m is maximally utilized* by Γ_{P_m}*, then $U(\Gamma_{P_m}) > \Theta(N)$.*

Proof. Let τ_i^t be the critical tail task on P_m. To simplify the description below, let U_X^t (U_Y^t) denote the total utilization of tasks with priorities higher (lower) than τ_i on P_m (see Fig. 6). From HSP-light, the processor containing the first body task $\tau_i^{b_1}$ of τ_i has the largest capacity to accommodate τ_i. Thus, we have

$$U_Y^t + u_i^{b_1} \geq \Theta(N).$$

Otherwise, $\tau_i^{b_1}$ would be assigned to P_m instead. Moreover, since τ_i is a light task, we have that $u_i^{b_1} < u_i \leq 1/2$, from the above inequality we can derive that

$$U_Y^t > \Theta(N) - \frac{1}{2}. \tag{11}$$

On the other hand, for processor P_m, since τ_i^t is the critical task, there will be no idle time within interval $[0, T_i - C_i^B]$, where C_i^B is the total execution time of τ_i's body tasks. Therefore, for τ_i^t and all higher priority tasks on P_m, we have

$$\sum_{j<i} C_j \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil + C_i^t \geq T_i - C_i^B. \tag{12}$$

Divide $(T_i - C_i^B)$ on both side of the above, we can get that

$$\sum_{j<i} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} + u_i^t \cdot \frac{T_i}{T_i - C_i^B} \geq 1.$$

Split the sum of the above into two parts, and rewrite as

$$\sum_{j<i, T_j<T_i-C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} + \sum_{j<i, T_j \geq T_i-C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} + u_i^t \cdot \frac{T_i}{T_i - C_i^B} \geq 1. \tag{13}$$

For the first part on the left side of equation (13), since $\lceil \frac{T_i - C_i^B}{T_j} \rceil \leq \frac{T_i - C_i^B}{T_j} + 1$, we can derive that

$$\sum_{j,i, T_j<T_i-C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} \leq \sum_{j<i, T_j<T_i-C_i^B} u_j \cdot \left(1 + \frac{T_j}{T_i - C_i^B}\right).$$

Moreover, in the above, since $T_j < T_i - C_i^B$, we have $\frac{T_j}{T_i - C_i^B} < 1$. Then we can further derive

$$\sum_{j<i, T_j<T_i-C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} \leq \sum_{j<i, T_j<T_i-C_i^B} 2 \cdot u_j. \tag{14}$$

For the second part on the left side of equation (13), since $T_j \geq T_i - C_i^B$, we have $\lceil \frac{T_i - C_i^B}{T_j} \rceil = 1$. Thus we can derive

$$\sum_{j<i, T_j \geq T_i-C_i^B} u_j \cdot \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i - C_i^B} = \sum_{j<i, T_j \geq T_i-C_i^B} u_j \cdot \frac{T_j}{T_i - C_i^B}. \tag{15}$$

And further since $u_i^B < 1/2$, then

$$\frac{T_j}{T_i - C_i^B} \leq \frac{T_i}{T_i - C_i^B} < 2, \text{ if } \mathrm{T_j} \geq \mathrm{T_i} - \mathrm{C_i^B}. \tag{16}$$

Put equation (16) into (15), we can derive

$$\sum_{j<i, T_j \geq T_i-C_i^B} u_j < \sum_{j<i, T_j \geq T_i-C_i^B} 2 \cdot u_j. \tag{17}$$

For the third part on the left side of equation (13), by applying (16), we have

$$u_i^t \cdot \frac{T_i}{T_i - C_i^B} < 2 \cdot u_i^t. \tag{18}$$

Apply equation (14), (17) and (18) into (13), we can get

$$\sum_{j<i, T_j<T_i-C_i^B} u_j + \sum_{j<i, T_j \geq T_i-C_i^B} u_j + u_i^t > \frac{1}{2}$$

or

$$U_X^t + u_i^t > \frac{1}{2}. \tag{19}$$

Finally, sum up equations (11) and (19), and replace $(U_Y^t + U_X^t + u_i^t)$ by $U(\Gamma_{P_m})$, we obtain that

$$U(\Gamma_{P_m}) > \Theta(N).$$

□

Based on Lemma 4 and Lemma 5, we can derive the following property.

Lemma 6. *If all processors in $\mathcal{P}$ are maximally utilized according to HSP-light, then we have*

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N). \tag{20}$$

Proof. Let $\mathcal{P}^A$ denote the processors with critical tasks as non-split tasks, and $\mathcal{P}^B$ denote the processors with critical tasks as tail tasks. According to Lemma 6, we have that $\mathcal{P} = \mathcal{P}^A \bigcup \mathcal{P}^B$ and $\mathcal{P}^A \bigcap \mathcal{P}^B = \emptyset$. Thus, we have

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) = \sum_{P_m \in \mathcal{P}^A} U(\Gamma_{P_m}) + \sum_{P_m \in \mathcal{P}^B} U(\Gamma_{P_m}). \quad (21)$$

Moreover, for any $P_m \in \mathcal{P}^A$ *or* $\mathcal{P}^B$, from Lemma 4 and Lemma 5, we know that $U(\Gamma_{P_m}) > \Theta(N)$. Applying this to the above equation, we get

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}^A| \cdot \Theta(N) + |\mathcal{P}^B| \cdot \Theta(N) \quad (22)$$

or

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N). \quad (23)$$

□

We are now ready to formally prove Theorem 3 in Section 4.3.

Proof (For Theorem 3). By contradiction. Assume that Γ is not feasible by HSP-light, thus we know every processor is *maximally utilized*.

From the given condition (equation (7)) we have that

$$U(\Gamma) \leq M \cdot \Theta(N). \quad (24)$$

On the other hand, since all processors are *maximally utilized*, according to Lemma 6, we know

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N).$$

Since $|\mathcal{P}| = M$, the above can be rewritten as

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > M \cdot \Theta(N). \quad (25)$$

This contradicts equation (24). □

Appendix B
Proof of Theorem 5

Two important observations, similar to that in Lemma 4 and Lemma 5, are also true and formulated in the following two lemmas.

Lemma 7. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP. If the critical task is a non-split task and P_m is maximally utilized, then $U(\Gamma_{P_m}) > \Theta(N)$.*

Lemma 8. *Let Γ_{P_m} be the task set allocated to processor P_m in HSP. If the critical task is a tail task from a light task and P_m is maximally utilized, then $U(\Gamma_{P_m}) > \Theta(N)$.*

Lemma 7 and Lemma 8 can be proved in the same way as that for Lemma 4 and Lemma 5. Moreover, if a tail task from a heavy task is the critical task, we have a very important observation which is formulated in the following lemma.

Lemma 9. *Let Γ_{P_k} be the task set allocated to processor P_k in HSP. If the critical task is a tail task from a heavy task τ_i and P_k is maximally utilized, then*

$$\sum_{P_m \in \mathcal{P}^R} U(\Gamma_{P_m}) > |\mathcal{P}^R| \cdot \Theta(N) \quad (26)$$

where $\mathcal{P}^R = \{P(\tau_j) | j \in [i, N]\}$.

Proof. For all tasks assigned to processors in $\mathcal{P}^R$, we divide them into two groups: 1) tasks with priorities lower than τ_i, denoted as Γ^Y, 2) tasks with priorities equal or higher than τ_i, denoted as Γ^X. Then we have

$$\sum_{P_m \in \mathcal{P}^R} U(\Gamma_{P_m}) = \sum_{\tau_j \in \Gamma^Y} u_j + \sum_{\tau_j \in \Gamma^X} u_j. \quad (27)$$

One one hand, since τ_i is heavy but not pre-assigned, according to equation (8), we have

$$\sum_{\tau_j \in \Gamma^Y} u_j > (|\mathcal{P}^R| - 1) \cdot \Theta(N). \quad (28)$$

Since τ_i^t is the critical task on its host processor P_m, there will be no idle time within interval $[0, T_i - C_i^B]$. Therefore, for τ_i^t and all higher priority tasks on P_m, we have

$$\sum_{j<i, \tau_j \in \Gamma_{P_m}} C_j \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil + C_i^t \geq T_i - C_i^B$$

or

$$\sum_{j<i, \tau_j \in \Gamma_{P_m}} u_j \left\lceil \frac{T_i - C_i^B}{T_j} \right\rceil \cdot \frac{T_j}{T_i} + u_i \geq 1. \quad (29)$$

Note that 1) $T_j \leq T_i$ for $j < i$, 2) and $T_i - C_i^B \leq \frac{1}{2} T_i$, since $u_i^B \geq \frac{1}{2}$. By putting them into the above, we can derive

$$\sum_{j<i, \tau_j \in \Gamma_{P_m}} u_j + u_i \geq 1. \quad (30)$$

Therefore, for all tasks in Γ^X we have

$$\sum_{\tau_j \in \Gamma^X} u_j \geq 1. \quad (31)$$

Finally, apply equation (31) and (28) into (27), since $\Theta(N) \leq 1$, we get

$$\sum_{P_m \in \mathcal{P}^R} U(\Gamma_{P_m}) > |\mathcal{P}^R| \cdot \Theta(N).$$

□

Lemma 10. *If a system is maximally utilized through HSP, then for all processors in $\mathcal{P}$, we have*

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N). \quad (32)$$

Proof. Select the heavy task, i.e., τ_i, that is not pre-assigned and has the highest priority among the ones with its tail task being the critical task on its host processor. Let $\mathcal{P}^A$ denote the processors to which τ_i and other lower priority tasks are assigned. Let $\mathcal{P}^B$ denote the rest of processors besides $\mathcal{P}^A$. From Lemma 9 we know that

$$\sum_{P_m \in \mathcal{P}^A} U(\Gamma_{P_m}) > |\mathcal{P}^A| \cdot \Theta(N). \tag{33}$$

From Lemma 7 and Lemma 8, we have that

$$\sum_{P_m \in \mathcal{P}^B} U(\Gamma_{P_m}) > |\mathcal{P}^B| \cdot \Theta(N). \tag{34}$$

Sum up equations (33) and (34), since $\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) = \sum_{P_m \in \mathcal{P}^A} U(\Gamma_{P_m}) + \sum_{P_m \in \mathcal{P}^B} U(\Gamma_{P_m})$, we can derive

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N). \tag{35}$$

□

With the above conclusions, Theorem 5 in Section 5 can be proved as below.

Proof (For Theorem 5). By contradiction. Assume that Γ is not feasible by HSP. With equation(9), we have

$$U(\Gamma) \leq M \cdot \Theta(N). \tag{36}$$

Since all processors are *maximally utilized*, from Lemma 10, we have that

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > |\mathcal{P}| \cdot \Theta(N) \tag{37}$$

or

$$\sum_{P_m \in \mathcal{P}} U(\Gamma_{P_m}) > M \cdot \Theta(N). \tag{38}$$

This contradicts equation (36). □

Acknowledgments

This work was supported in part by NSF under Projects CNS-0969013, CNS-0917021, and CNS-1018108.

References

[1] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, vol. 29, no. 2, pp. 6-16, Mar./Apr. 2009.

[2] W. Wolf, A.A. Jerraya, and G. Martin, "Multiprocessor System-On-Chip (MPSOC) Technology," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 10, pp. 1701-1713, Oct. 2008.

[3] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, and K.A. Yelick, "The Landscape of Parallel Computing Research: A View From Berkeley," Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2006-183, 2006.

[4] K.G. Shin and P. Ramanathan, "Real-Time Computing: A New Discipline of Computer Science and Engineering," *Proc. IEEE*, vol. 82, no. 1, pp. 6-24, Jan. 1994.

[5] C.L. Liu and J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.

[6] S.K. Dhall and C.L. Liu, "On a Real-Time Scheduling Problem," *Oper. Res.*, vol. 26, no. 1, pp. 127-140, Jan./Feb. 1978.

[7] B. Andersson, S. Baruah, and J. Jonsson, "Static-Priority Scheduling on Multiprocessors," in *Proc. IEEE RTSS*, Dec. 2001, pp. 193-202.

[8] B. Andersson, "Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38 percent," in *Proc. ACM Int'l Conf. OPODIS*, 2008, vol. 5401, pp. 73-88.

[9] S. Kato and N. Yamasaki, "Semi-Partitioned Fixed-Priority Scheduling on Multiprocessors," in *Proc. IEEE RTAS*, Apr. 2009, pp. 23-32.

[10] K. Lakshmanan, R. Rajkumar, and J. Lehoczky, "Partitioned Fixed-Priority Preemptive Scheduling for Multi-Core Processors," in *Proc. ECRTS*, July 2009, pp. 239-248.

[11] 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 on Scheduling Algorithms, Methods, Models*. Boca Raton, FL, USA: CRC Press, 2004.

[12] J.H. Anderson, V. Bud, and U.C. Devi, "An EDF-Based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems," in *Proc. ECRTS*, July 2005, pp. 199-208.

[13] S. Kato and N. Yamasaki, "Real-Time Scheduling with Task Splitting on Multiprocessors," in *Proc. IEEE Int'l Conf. Embedded RTCSA*, Aug. 2007, pp. 441-450.

[14] B. Andersson, K. Bletsas, and S. Baruah, "Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors," in *Proc. IEEE RTSS*, Dec. 2008, pp. 385-394.

[15] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-Priority Multiprocessor Scheduling with Liu and Layland's Utilization Bound," in *Proc. IEEE RTAS*, Apr. 2010, pp. 165-174.

[16] A. Bastoni, B.B. Brandenburg, and J.H. Anderson, "Is Semi-Partitioned Scheduling Practical?" in *Proc. ECRTS*, July 2011, pp. 125-135.

[17] N. Guan, M. Stigge, W. Yi, and G. Yu, "Fixed-Priority Multiprocessor Scheduling: Beyond Liu and Layland's Utilization Bound," in *Proc. IEEE RTSS*, Dec. 2010, pp. 165-174.

[18] Y. Zhang, N. Guan, and W. Yi, "Towards the Implementation and Evaluation of Semi-Partitioned Multi-Core Scheduling," in *Proc. DATE Workshop Predictability Perform. Embedded Syst.*, 2011, vol. 18, pp. 42-46.

[19] M. Fan and G. Quan, "Harmonic Semi-Partitioned Scheduling for Fixed-Priority Real-Time Tasks on Multi-Core Platform," in *Proc. DATE Conf. Exhib.*, Mar. 2012, pp. 503-508.

[20] J.W.S. Liu, *Real-Time Systems*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2000.

[21] C.-C. Han and H.-Y. Tyan, "A Better Polynomial-Time Schedulability Test for Real-Time Fixed-Priority Scheduling Algorithms," in *Proc. IEEE RTSS*, Dec. 1997, pp. 36-45.

[22] T.-W. Kuo and A.K. Mok, "Load Adjustment in Adaptive Real-Time Systems," in *Proc. IEEE RTSS*, Dec. 1991, pp. 160-170.

[23] E. Bini, G.C. Buttazzo, and G.M. Buttazzo, "Rate Monotonic Analysis: The Hyperbolic Bound," *IEEE Trans. Comput.*, vol. 52, no. 7, pp. 933-942, July 2003.

[24] S. Lauzac, R. Melhem, and D. Mossé, "An Improved Rate-Monotonic Admission Control and Its Applications," *IEEE Trans. Comput.*, vol. 52, no. 3, pp. 337-350, Mar. 2003.

[25] W.-C. Lu, H.-W. Wei, and K.-J. Lin, "Rate Monotonic Schedulability Conditions Using Relative Period Ratios," in *Proc. IEEE Int'l Conf. Embedded RTCSA*, 2006, pp. 3-9.

[26] M.-J. Jung, Y. R. Seong, and C.-H. Lee, "Optimal RM Scheduling for Simply Periodic Tasks on Uniform Multiprocessors," in *Proc. ACM ICHIT*, Aug. 2009, vol. 321, pp. 383-389.

[27] D. Müller, "Accelerated Simply Periodic Task Sets for RM Scheduling," in *Proc. Embedded Real Time Softw. Syst.*, May 2010, p. 42.

[28] M. Fan and G. Quan, "Harmonic-Fit Partitioned Scheduling for Fixed-Priority Real-Time Tasks on the Multiprocessor Platform," in *Proc. IFIP 9th Int'l Conf. EUC*, Oct. 2011, pp. 27-32.

[29] B. Andersson and J. Jonsson, "The Utilization Bounds of Partitioned and PFAIR Static-Priority Scheduling on Multiprocessors are 50 percent," in *Proc. ECRTS*, July 2003, pp. 33-40.

[30] N. Guan, M. Stigge, W. Yi, and G. Yu, "Parametric Utilization Bounds for Fixed-Prioirity Multiprocessor Scheduling," in *Proc. IEEE IPDPS*, May 2012, pp. 261-272.

[31] S. Lauzac, R. Melhem, and D. Mossé, "An Efficient RMS Admission Control and Its Application to Multiprocessor Scheduling," in *Proc. IPPS/SPDP*, Mar. 1998, pp. 511-518.

[32] A. Kandhalu, K. Lakshmanan, J. Kim, and R. Rajkumar, "pCOMPATS: Period-Compatible Task Allocation and Splitting on Multi-Core Processors," in *Proc. IEEE RTAS*, Apr. 2012, pp. 307-316.

[33] S. Kato and N. Yamasaki, ''Portioned Static-Priority Scheduling on Multiprocessors,'' in *Proc. IEEE IPDPS*, Apr. 2008, pp. 1-12.
[34] J. Lehoczky, L. Sha, and Y. Ding, ''The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behavior,'' in *Proc. RTSS*, Dec. 1989, pp. 166-171.

Ming Fan is a PhD candidate at the Department of Electrical and Computer Engineering at the Florida International University, FL, USA. He received both the BS and MS degrees in Computer Engineering from Bei Hang University, Beijing, China, in 2006 and 2009, respectively. His research interests include real-time systems, power-/thermal-aware computing and fault-tolerant systems. He is a Student Member of the IEEE.

Gang Quan received the PhD degree from the Department of Computer Science and Engineering, University of Notre Dame, USA, the MS degree from the Chinese Academy of Sciences, China, and the BS degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China. He is currently an associate professor at the Department of Electrical and Computing Engineering, Florida International University. Before he joined the department, he was an assistant professor at the Department of Computer Science and Engineering, University of South Carolina. His research interests include real-time systems, embedded system design, power-/thermal-aware computing, advanced computer architecture, reconfigurable computing. He is a Senior Member of the IEEE.