2012 IEEE 26th International Parallel and Distributed Processing Symposium

Parametric Utilization Bounds for Fixed-Priority Multiprocessor Scheduling

Nan Guan[1,2], Martin Stigge[1], Wang Yi[1,2] and Ge Yu[2]
[1] Uppsala University, Sweden
[2] Northeastern University, China

***Abstract*—Future embedded real-time systems will be deployed on multi-core processors to meet the dramatically increasing high-performance and low-power requirements. This trend appeals to generalize established results on uniprocessor scheduling, particularly the various utilization bounds for schedulability test used in system design, to the multiprocessor setting. Recently, this has been achieved for the famous Liu and Layland utilization bound by applying novel task splitting techniques. However, *parametric* utilization bounds that can guarantee higher utilizations (up to 100%) for common classes of systems are not yet known to be generalizable to multiprocessors as well. In this paper, we solve this problem for most parametric utilization bounds by proposing new task partitioning algorithms based on exact response time analysis. In addition to the worst-case guarantees, as the exact response time analysis is used for task partitioning, our algorithms significantly improve average-case utilization over previous work.**

I. Introduction

It has been widely accepted that future embedded real-time systems will be deployed on multi-core processors, to satisfy the dramatically increasing high-performance and low-power requirements. This trend demands effective and efficient techniques for the design and analysis of real-time systems on multi-cores.

A central problem in the real-time system design is *timing analysis*, which examines whether the system can meet all the specified timing requirements. Timing analysis usually consists of two steps: task-level timing analysis, which for example calculates the worst-case execution time of each task independently, and system-level timing analysis (also called *schedulability analysis*), which determines whether all the tasks can co-exist in the system and still meet all the time requirements.

One of the most commonly used schedulability analysis approach is based on the *utilization bound*, which is a safe threshold of the system's workload: under this threshold the system is guaranteed to meet all the time requirements. The utilization-bound-based schedulability analysis is very efficient, and is especially suitable to embedded system design flow involving iterative design space exploration procedures. A well-known utilization bound is the $N(2^{1/N}-1)$ bound for RMS (Rate Monotonic Scheduling) on uni-processors, discovered by Liu and Layland in the 1970's [25]. Recently, this bound has been generalized to multiprocessors scheduling by a parti-tioning-based algorithm [16].

The Liu and Layland utilization bound ($L\&L$ bound for short) is pessimistic: There are a significant number of task systems that exceed the $L\&L$ bound but are indeed schedulable. This means that system resources would be considerably under-utilized if one only relies on the $L\&L$ bound in system design.

If more information about the task system is available in the design phase, it is possible to derive higher *parametric* utilization bounds regarding known task parameters. A well-known example of parametric utilization bounds is the 100% bound for *harmonic* task sets [26]: If the total utilization of a harmonic task set τ is no greater than 100%, then every task in τ can meet its deadline under RMS on a uni-processor platform. Even if the whole task system is not harmonic, one can still obtain a significantly higher bound by exploring the "harmonic chains" of the task system [21]. In general, during the system design, it is usually possible to employ higher utilization bounds with available task parameter information to better utilize the resources and decrease the system cost. As will be introduced in Section III, quite a number of higher parametric utilization bounds regarding different task parameter information have been derived for uni-processor scheduling.

This naturally raises an interesting question: Can we generalize these higher parametric utilization bounds derived for uni-processor scheduling to multiprocessors? For example, given a harmonic task system, can we guarantee the schedulability of the task system on a multiprocessor platform of M processors, if the utilization sum of all tasks is no larger than M?

In this paper, we will address the above question by proposing new RMS-based partitioned scheduling algorithms (with task splitting). Generalizing the parametric utilization bounds from uni-processors to multiprocessors is challenging, even with the insights from our previous work generalizing the $L\&L$ bound to multiprocessor scheduling. The reason is that task splitting may "create" new tasks that do not comply with the parameter properties of the original task set, and thus invalidate the parametric utilization bound specific to the original task set's parameter properties. Section III presents this problem in detail. The main contribution of this paper is a solution to this problem, which generalizes most of the parametric utilization bounds to multiprocessors.

The approach of this paper is generic in the sense that it works irrespective of the form of the parametric utilization bound in consideration. The only restriction is a threshold on the parametric utilization bound value when some task has a large individual utilization; apart from that, any parametric utilization bound derived for single-processor RMS can be used to guarantee the schedulability of multiprocessors systems via our algorithms. More specifically, we first proposed an algorithm generalizing all known parametric utilization bounds for RMS to multiprocessors, for a class of "light" task sets in which each task's individual utilization is at most $\frac{\Theta(\tau)}{1+\Theta(\tau)}$, where $\Theta(\tau) = N(2^{1/N} - 1)$ is the $L\&L$ bound for task set τ. Then we proposed the second algorithm that works for any task set and all parametric utilization bounds under the threshold $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$[1].

Besides the improved utilization bounds, another advantage of our new algorithms is the significantly improved average-case performance. Although the algorithm in [16] can achieve the $L\&L$ bound, it has the problem that it never utilizes more than the worst-case bound. The new algorithms in this paper use exact analysis, i.e., Response Time Analysis (RTA), instead of the utilization bound threshold as in the algorithm of [16], to determine the maximal workload on each processor. It is well-known that on uni-processors, by exact schedulability analysis, the average breakdown utilization of RMS is around 88% [24], which is much higher than its worst-case utilization bound 69.3%. Similarly, our new algorithm has much better performance than the algorithm in [16].

Related Work: Multiprocessor scheduling is usually categorized into two paradigms [11]: *global scheduling*, where each task can execute on any available processor at run time, and *partitioned scheduling*, where each task is assigned to a processor beforehand, and at run time each task only executes on its assigned processor. Global scheduling on average utilizes the resources better. However, the standard RMS and EDF global scheduling strategies suffer from the Dhall effect [14], which may cause a task system with arbitrarily low utilization to be unschedulable. Although the Dhall effect can be mitigated by, e.g., assigning higher priorities to tasks with higher utilizations as in RM-US [4], the best known utilization bound of global scheduling is still quite low: 38% for fixed-priority scheduling [3] and 50% for EDF-based scheduling [7]. On the other hand, partitioned scheduling suffers from the resource waste similar to the bin-packing problem: the worst-case utilization bound for any partitioned scheduling can not exceed 50%. Although there exist scheduling algorithms like the Pfair family [10], [2], the LLREF family [13], [15] and the EKG family [5], [6], offering utilization bounds upto 100%, these algorithms incur much higher context-switch overhead than priority-driven scheduling, which is unacceptable in many real-life systems.

Recently, a number of works [1], [6], [5], [17], [18], [19], [20], [22], [16] have studied partitioned scheduling with *task splitting*, which can overcome the 50% limit of the strict partitioned scheduling. In this class of scheduling algorithms, while most tasks are assigned to a fixed processor, some tasks may be (sequentially) divided into several parts and each part is assigned and thereby executed on a different (but fixed) processor. In this category, the utilization bound of the state-of-the-art EDF-based algorithm is 65% [17], and our recent work [16] has achieved the $L\&L$ bound (in the worst case 69.3%) for fixed-priority based algorithms.

[1]When N goes to infinity, $\Theta(\tau) \doteq 69.3\%$, $\frac{\Theta(\tau)}{1+\Theta(\tau)} \doteq 40.9\%$ and $\frac{2\Theta(\tau)}{1+\Theta(\tau)} \doteq 81.8\%$

II. Basic Concepts

We consider a multiprocessor platform consisting of M processors $\mathcal{P} = \{P_1, P_2, ...P_M\}$. A task set $\tau = \{\tau_1, \tau_2, ..., \tau_N\}$ complies with the $L\&L$ task model: Each task τ_i is a 2-tuple $\langle C_i, T_i \rangle$, where C_i is the worst-case execution time and T_i is the minimal inter-release separation (also called period). T_i is also τ_i's relative deadline. We use the RMS strategy to assign priorities: tasks with shorter periods have higher priorities. Without loss of generality we sort tasks in non-decreasing period order, and can therefore use the task indices to represent task priorities, i.e., $i < j$ implies that τ_i has higher priority than τ_j. The *utilization* of each task τ_i is defined as $U_i = C_i/T_i$, and the *total utilization* of task set τ is $\mathcal{U}(\tau) = \sum_{i=1}^{N} U_i$. We further define the *normalized utilization* of a task set τ on a multiprocessor platform with M processors:

$$\mathcal{U}_M(\tau) = \sum_{\tau_i \in \tau} U_i / M$$

Note that the subscript M in $\mathcal{U}_M(\tau)$ reminds us that the sum of all tasks' utilizations is divided by the number of processors M.

A partitioned scheduling algorithm (with task splitting) consists of two parts: the *partitioning algorithm*, which determines how to split and assign each task (or rather each of its parts) to a fixed processor, and the *scheduling algorithm*, which determines how to schedule the tasks assigned to each processor at run time.

With the partitioning algorithm, most tasks are assigned to a processor (and thereby will only execute on this processor at run time). We call these tasks *non-split tasks*. The other tasks are called *split tasks*, since they are split into several *subtasks*. Each subtask of a split task τ_i is assigned to (and thereby executes on) a different processor, and the sum of the execution times of all subtasks equals C_i. For example, in Figure 1 task τ_i is split into three subtasks τ_i^1, τ_i^2 and τ_i^3, executing on processor P_1, P_2 and P_3, respectively.

The subtasks of a task need to be synchronized to execute correctly. For example, in Figure 1, τ_i^2 should not start execution until τ_i^1 is finished. This equals deferring the actual ready time of τ_i^2 by up to R_i^1 (relative to τ_i's original release time), where R_i^1 is τ_i^1's worst-case response time. One can regard this as shortening the actual relative deadline of τ_i^2 by up to R_i^1. Similarly, the actual ready time of τ_i^3 is deferred by up to $R_i^1 + R_i^2$, and τ_i^3's actual relative deadline is shortened by up to $R_i^1 + R_i^2$. We use τ_i^k to denote the k^{th} subtask of a split task τ_i, and define τ_i^k's *synthetic deadline* as

$$\Delta_i^k = T_i - \sum_{l \in [1, k-1]} R_i^l. \qquad (1)$$

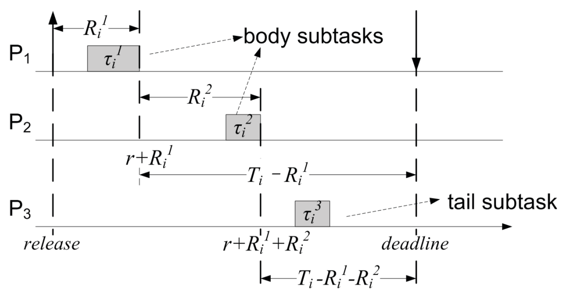

Fig. 1. An Illustration of Task Splitting.

Thus, we represent each subtask τ_i^k by a 3-tuple $\langle C_i^k, T_i, \Delta_i^k \rangle$, in which C_i^k is the execution time of τ_i^k, T_i is the original period and Δ_i^k is the synthetic deadline. For consistency, each non-split task τ_i can be represented by a single subtask τ_i^1 with $C_i^1 = C_i$ and $\Delta_i^1 = T_i$. We use $U_i^k = C_i^k / T_i$ to denote a subtask τ_i^k's utilization.

We call the last subtask of τ_i its *tail subtask*, denoted by τ_i^t and the other subtasks its *body subtasks*, as shown in Figure 1. We use $\tau_i^{b_j}$ to denote the j^{th} body subtask.

We use $\tau(P_q)$ to denote the set of tasks τ_i assigned to processor P_q, and say P_q is the *host processor* of τ_i. We use $\mathcal{U}(P_q)$ to denote the sum of the utilization of all tasks in $\tau(P_q)$. A task set τ is *schedulable* under a partitioned scheduling algorithm $\mathcal{A}$, if (i) each task (subtask) has been assigned to some processor by $\mathcal{A}$'s partitioning algorithm, and (ii) each task (subtask) is guaranteed to meet its deadline under $\mathcal{A}$'s scheduling algorithm.

III. Parametric Utilization Bounds

On uni-processors, a *Parametric Utilization Bound* (PUB for short) $\Lambda(\tau)$ for a task set τ is the result of applying a function $\Lambda(\cdot)$ to τ's task parameters, such that all the tasks in τ are guaranteed to meet their deadlines on a uni-processor if τ's total utilization $\mathcal{U}(\tau) \leq \Lambda(\tau)$. We can overload this concept for multiprocessor scheduling by using τ's normalized utilization $\mathcal{U}_M(\tau)$ instead of $\mathcal{U}(\tau)$.

There have been several PUBs derived for RMS on uni-processors. The following are some examples:

- The famous $L\&L$ bound, denoted by $\Theta(\tau)$, is a PUB regarding the number of tasks N: $\Theta(\tau) = N(2^{1/N} - 1)$
- The harmonic chain bound: HC-Bound$(\tau) = K(2^{1/K} - 1)$ [21] , where K is the number of harmonic chains in the task set. The 100% bound for harmonic task sets is a special case of the harmonic chain bound with $K = 1$.
- T-Bound(τ) [23] is a PUB regarding the number of tasks and the task periods: T-Bound$(\tau) = \sum_{i=1}^{N} \frac{T'_{i+1}}{T'_i} + 2 \cdot \frac{T'_1}{T'_N} - N$, where T'_i is τ_i's *scaled period* [23].
- R-Bound(τ) [23] is similar to T-Bound(τ), but uses a more abstract parameter r, the ratio between the minimum and maximum scaled period of the task set: R-Bound$(\tau) = (N-1)(r^{1/(N-1)} - 1) + 2/r - 1$.

We observe that all the above PUBs have the following property: for any τ' obtained by decreasing the execution times

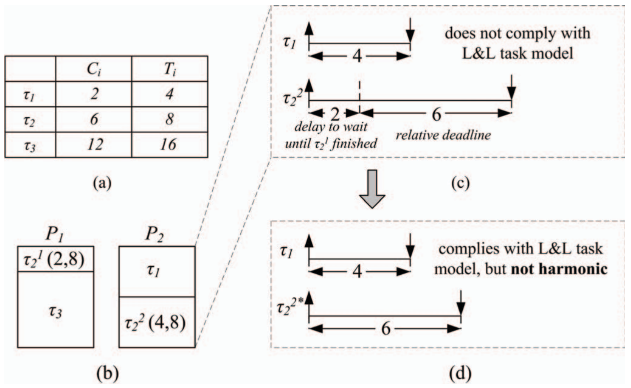

Fig. 2. Partitioning a harmonic task set results in a nonharmonic task set on some processor.

of some tasks of τ, the bound $\Lambda(\tau)$ is still a valid utilization bound to guarantee the schedulability of τ'. We call a PUB holding this property a *deflatable* parametric utilization bound (called D-PUB for short)[2]. We use the following lemma to precisely describe this property:

Lemma 1. *Let $\Lambda(\tau)$ be a* D-PUB *derived from the task set τ. We decrease the execution times of some tasks in τ to get a new task set τ'. If τ' satisfies $\mathcal{U}(\tau') \leq \Lambda(\tau)$, then it is guaranteed to be schedulable by* RMS *on a uni-processor.*

The deflatable property is very common: Actually all the PUBs we are aware of are deflatable, including the ones listed above and the non-closed-form bounds in [12]. The deflatable property is of great relevance in partitioned multiprocessor scheduling, since a task set τ will be partitioned into several subsets and each subset is executed on a processor individually. Further, due to the task splitting, a task could be divided into several subtasks, each of which holds a portion of the execution demand of the original task. So the deflatable property is clearly required to generalize a utilization bound to multiprocessors.

However, the deflatable property by itself is *not* sufficient for the generalization of a PUB $\Lambda(\tau)$ to multiprocessors. For example, suppose the harmonic task set τ in Figure 2-(a) is partitioned as in Figure 2-(b), where τ_2 is split into τ_2^1 and τ_2^2. To correctly execute τ_2, τ_2^1 and τ_2^2 need to be synchronized such that τ_2^2 never starts execution before its predecessor τ_2^1 is finished. This can be viewed as shortening τ_2^2's relative deadline for a certain amount of time from τ_2's original deadline, as shown in Figure 2-(c). In this case, τ_2^2 does not comply with the $L\&L$ task model (which requires the relative deadline to equal the period), so none of the parametric utilization bounds for the $L\&L$ task model are applicable to processor P_2. In [16], this problem is solved by

[2]There is a subtle difference between the *deflatable* property and the *(self-)sustainable* property [9], [8]. The deflatable property does *not* require the original task set τ to satisfy $\mathcal{U}(\tau) \leq \Lambda(\tau)$. $\mathcal{U}(\tau)$ is typically larger than 100% since τ will be scheduled on M processors. $\Lambda(\tau)$ is merely a value obtained by applying the function $\Lambda(\cdot)$ to τ's parameters, and will be used to each individual processor.

representing τ_2^2's period by its relative deadline, as shown in Figure 2-(d). This transforms the task set $\{\tau_1, \tau_2^2\}$ into a $L\&L$ task set $\{\tau_1, \tau_2^{2*}\}$, with which we can apply the $L\&L$ bound. However, this solution does not in general work for other parametric utilization bounds: In our example, we still want to apply the 100% bound which is specific to harmonic task sets. But if we use τ_2^2's deadline 6 to represent its period, the task set $\{\tau_1, \tau_2^{2*}\}$ is not harmonic, so the 100% bound is not applicable. This problem will be solved by our new algorithms and novel proof techniques in the following sections.

IV. The Algorithm for Light Tasks

In the following we introduce the first algorithm RM-TS/light, which achieves $\Lambda(\tau)$ (any D-PUB derived from τ's parameters), if τ is *light* in the sense of an upper bound on each task's individual utilization as follows.

Definition 1. *A task τ_i is a* light task *if $U_i \leq \frac{\Theta(\tau)}{1+\Theta(\tau)}$, where $\Theta(\tau)$ denotes the $L\&L$ bound. Otherwise, τ_i is a* heavy task. *A task set τ is a* light task set *if all tasks in τ are light. $\frac{\Theta(\tau)}{1+\Theta(\tau)}$ is about 40.9% as the number of tasks in τ grows to infinity.*

For example, we can instantiate this result by the 100% utilization bound for harmonic task sets: Let τ be any *harmonic* task set in which each task's individual utilization is no larger than 40.9%. τ is schedulable by our algorithm RM-TS/light on M processors if its normalized total utilization $\mathcal{U}_M(\tau)$ is no larger than 100%.

A. Algorithm Description

The partitioning algorithm of RM-TS/light is quite simple. We describe it briefly as follows:

1) Tasks are assigned in increasing priority order. We always select the processor on which the total utilization of the tasks that have been assigned so far is *minimal* among all processors.
2) A task (subtask) can be entirely assigned to the current processor, if all tasks (including the one to be assigned) on this processor can meet their deadlines under RMS.
3) When a task (subtask) cannot be assigned entirely to the current processor, we split it into two parts[3]. The first part is assigned to the current processor. The splitting is done such that the portion of the first part is as big as possible, guaranteeing no task on this processor misses its deadline under RMS; the second part is left for the assignment in the step.

Note that the difference between RM-TS/light and the algorithm in [16] is that, RM-TS/light uses the exact response time analysis, instead of the utilization threshold, to determine whether a (sub)task can fit in a processor without causing deadline miss.

Algorithm IV-A and IV-A describe the partitioning algorithm of RM-TS/light in pseudo-code. At the beginning, tasks are sorted (and will therefore be assigned) in increasing priority order, and all processors are marked as *non-full* which means they still can accept more tasks. At each step, we pick the next task in order (the one with the lowest priority), select the processor with the minimal total utilization of tasks that have been assigned so far, and invoke the routine Assign(τ_i^k, P_q) to do the task assignment. Assign(τ_i^k, P_q) first verifies that after assigning the task, all tasks on that processor would still be schedulable under RMS. This is done by applying exact schedulability analysis of calculating the response time R_j^h of each (sub)task τ_j^k on P_q after assigning this new task τ_i^k, and compare R_j^h to its (synthetic) deadline Δ_j^h. If the response time does not exceed the synthetic deadline for any of the tasks on P_q, we can conclude that τ_i^k can safely be assigned to P_q without causing any deadline miss. Note that a subtask's synthetic deadline Δ_j^k may be different from its period T_j. After presenting how the overall partitioning algorithm works, we will show how to calculate Δ_j^k easily.

Algorithm 1 The partitioning algorithm of RM-TS/light.

```
1: Task order $\tau_N^1$, ..., $\tau_1^1$ by increasing priorities
2: Mark all processors as non-full
3: while exists an non-full processor and an unassigned task do
4:     Pick next unassigned task $\tau_i^k$,
5:     Pick non-full processor $P_q$ with minimal $\mathcal{U}(P_q)$
6:     Assign($\tau_i^k, P_q$)
7: end while
8: If there is an unassigned task, the algorithm fails,
   otherwise it succeeds.
```

Algorithm 2 The Assign(τ_i^k, P_q) routine.

```
1: if $\tau(P_q)$ with $\tau_i^k$ is still schedulable then
2:     Add $\tau_i^k$ to $\tau(P_q)$
3: else
4:     Split $\tau_i^k$ via $(\tau_i^k, \tau_i^{k+1}) :=$ MaxSplit($\tau_i^k, P_q$)
5:     Add $\tau_i^k$ to $\tau(P_q)$
6:     Mark $P_q$ as full
7:     $\tau_i^{k+1}$ is the next task to assign
8: end if
```

If τ_i^k cannot be entirely assigned to the currently selected processor P_q, it will be split into two parts using routine MaxSplit(τ_i^k, P_q): the first part that makes maximum use of the selected processor, and a remaining part of that task, which will be subject to assignment in the next iteration. The desired property here is that we want the first part to be as big as possible such that, after assigning it to P_q, all tasks on that processor will still be able to meet their deadlines. In order to state the effect of MaxSplit(τ_i^k, P_q) formally, we introduce the concept of a *bottleneck*:

Definition 2. *A* bottleneck *of processor P_q is a (sub)task that is assigned to P_q, and will become unschedulable if we increase the execution time of the task with the highest priority on P_q by an arbitrarily small positive number.*

[3]In general a task may be split into more than two subtasks. Here we mean at each step the currently selected task (subtask) is split into two parts.

Note that there may be more than one bottleneck on a processor. Further, since RM-TS/light assigns tasks in increasing priority order, MaxSplit always operates on the task that has the highest priority on the processor in question. So we can state:

Definition 3. MaxSplit(τ_i^k, P_q) *is a function that splits τ_i^k into two subtasks τ_i^k and τ_i^{k+1} such that:*

1) *τ_i^k can now be assigned to P_q without making any task in $\tau(P_q)$ unschedulable.*
2) *After assigning τ_i^k, P_q has a bottleneck.*

MaxSplit can be implemented by, for example, performing a binary search over $[0, C_i^k]$ to find out the maximal portion of τ_i^k with which all tasks on P_q can meet their deadlines. A more efficient implementation of MaxSplit was presented in [22], in which one only needs to check a (small) number of possible values in $[0, C_i^k]$. The complexity of this improved implementation is still pseudo-polynomial, but in practice it is very efficient.

The while loop in RM-TS/light terminates as soon as all processors are "full" *or* all tasks have been assigned. If the loop terminates due to the first reason and there are still unassigned tasks left, the algorithm reports a failure of the partitioning, otherwise a success.

Calculating Synthetic Deadlines: Now we show how to calculate each (sub)task τ_i^k's synthetic deadline Δ_i^k, which was left open in the above presentation. If τ_i^k is a non-split task, its synthetic deadline trivially equals its period T_i.

We consider the case that τ_i^k is a split subtask. Since tasks are assigned in increasing order of priorities, and a processor is *full* after a body subtask is assigned to it, we have the following lemma:

Lemma 2. *A body subtask has the highest priority on its host processor.*

A consequence is that, the response time of each body subtask equals its execution time, and one can replace R_i^l by C_i^l in (1) to calculate the synthetic deadline of a subtask. Especially, we are interested in the synthetic deadlines of tail subtasks (we don't need to worry about a body subtask's synthetic deadline since it has the highest priority on its host processor and is schedulable anyway). The calculation is stated in the following lemma.

Lemma 3. *A tail subtask τ_i^t's synthetic deadline Δ_i^t is calculated by*

$$\Delta_i^t = T_i - C_i^{body}$$

where C_i^{body} is the execution time sum of τ_i's body subtasks.

Scheduling at Run Time: At runtime, the tasks will be scheduled according to the RMS priority order on each processor locally, i.e., with their original priorities. The subtasks of a split task respect their precedence relations, i.e., a split subtask τ_i^k is ready for execution when its preceding subtask τ_i^{k-1} on some other processor has finished.

From the presented partitioning and scheduling algorithm of RM-TS/light, it is clear that successful partitioning implies schedulability (remember that for split tasks, the synchronization delays have been counted into the synthetic deadlines, which are the ones used in the response time analysis to determine whether a task is schedulable). We state this in the following lemma:

Lemma 4. *Any task set that has been successfully partitioned by* RM-TS/light *is schedulable.*

B. Utilization Bound

We will now prove that RM-TS/light has the utilization bound of $\Lambda(\tau)$ for *light* task sets, i.e., if a light task set τ is not successfully partitioned by RM-TS/light, then the sum of the assigned utilizations of all processors is *at least*[4] $M \cdot \Lambda(\tau)$.

In order to show this, we assume that the assigned utilization on some processor is *strictly less* than $\Lambda(\tau)$. We prove that this implies there is no bottleneck on that processor. This is a contradiction, because each processor with which MaxSplit has been used must have a bottleneck. We also know that MaxSplit was used for all processors, since the partitioning failed.

In the following, we assume P_q to be a processor with an assigned utilization of $U(P_q) < \Lambda(\tau)$. A task on P_q is either a non-split task, a body subtask or a tail subtask. The main part of the proof consists of showing that P_q cannot have a bottleneck of any type.

As the first step, we show this for non-split tasks and body subtasks (Lemma 5), after which we deal with the more difficult case of tail subtasks (Lemma 7).

Lemma 5. *Suppose task set τ is not schedulable by* RM-TS/light*, and after the partitioning phase it holds for a processor P_q that*

$$\mathcal{U}(P_q) < \Lambda(\tau) \tag{2}$$

Then a bottleneck of P_q is neither a non-split task nor a body subtask.

Proof: By Lemma 2 we know that the body subtask has the highest priority on P_q, so it can never be a bottleneck.

For the case of non-split tasks, we will show that Condition (2) is sufficient for their deadlines to be met. The key observation is that although some split tasks on this processor may have a shorter deadline than period, this does not change the scheduling behavior of RMS, so $\Lambda(\tau)$ is still sufficient to guarantee the schedulability of a non-split task. For a more precise proof, we use Γ to denote the set of tasks on P_q, and construct a new task set Γ^* corresponding to Γ such that each non-split task τ_i in Γ has a counterpart in Γ^* that is exactly the same as τ_i, and each split subtask in Γ has a counterpart in Γ^* with deadline changed to equal its period. It's easy to see that Γ^* can be obtained by decreasing some tasks' execution times in the original task set τ (a task in τ but not Γ^* can be

[4]By this, the normalized utilization of τ *strictly exceeds* $\Lambda(\tau)$, since there are (sub)tasks not assigned to any of the processors after a failed partitioning.

considered as the case that we decrease its execution time to 0). By Lemma 1 and Condition (2) we know, the deflatable utilization bound $\Lambda(\tau)$ guarantee Γ^*'s schedulability. Thus, if the execution time of the highest-priority task on P_q is increased by an arbitrarily small amount ε such that the total utilization still does not exceed $\Lambda(\tau)$, Γ^* will still be schedulable. Recall that the only difference between Γ and Γ^* is the subtasks' deadlines, and since the scheduling behavior of RMS does not depend on task deadlines (remember that at this moment we only want to guarantee the schedulability of non-split tasks), we can conclude that each non-split task in Γ is also schedulable, which is still the true after increasing ε to the highest priority task on P_q. ■

In the following we prove that in a light task set, a bottleneck on a processor with utilization lower than $\Lambda(\tau)$ is not a tail subtask either. The proof goes in two steps: We first derive in Lemma 6 a general condition guaranteeing that a tail subtask can not be a bottleneck; then we conclude in Lemma 7 that a bottleneck on a processor with utilization lower than $\Lambda(\tau)$ is not a tail subtask, by showing that the condition in Lemma 6 holds for each of these tail subtasks.

We use the following notation: Let τ_i be a task split into B body subtasks $\tau_i^{b_1}...\tau_i^{b_B}$, assigned to processors $P_{b_1}...P_{b_B}$ respectively, and a tail subtask τ_i^t assigned to processor P_t. The utilization of the tail subtask τ_i^t is $U_i^t = \frac{C_i^t}{T_i}$, and the utilization of a body subtask $\tau_i^{b_j}$ is $U_i^{b_j} = \frac{C_i^{b_j}}{T_i}$. We use U_i^{body} to denote the total utilization of τ_i's all body subtasks:

$$U_i^{body} = \sum_{j\in[1,B]} U_i^{b_j} = U_i - U_i^t$$

For the tail subtask τ_i^t, let X_t denote the total utilization of all (sub)tasks assigned to P_t with *lower* priority than τ_i^t, and Y_t the total utilization of all (sub)tasks assigned to P_t with *higher* priority than τ_i^t.

For each body subtask $\tau_i^{b_j}$, let X_{b_j} denote the total utilization of all (sub)tasks assigned to P_{b_j} with *lower* priority than $\tau_i^{b_j}$. (We do not need Y_{b_j}, since by Lemma 2 we know no task on P_{b_j} has higher priority than τ_i.)

We start with the general condition identifying non-bottleneck tail subtasks.

Lemma 6. *Suppose a tail subtask τ_i^t is assigned to processor P_t and $\Theta(\tau)$ is the L&L bound. If*

$$Y_t + U_i^t < \Theta(\tau) \cdot (1 - U_i^{body}) \qquad (3)$$

then τ_i^t is not *a bottleneck of processor P_t.*

Proof: The lemma is proved by showing τ_i^t is still schedulable after increasing the utilization of the task with the highest priority on P_t by a small number ϵ such that : $(Y_t + \epsilon) + U_i^t < \Theta(\tau) \cdot (1 - U_i^{body})$ (note that one can always find such an ϵ). By the definition of U_i^{body} and Δ_i^t, this equals

$$((Y_t + \epsilon) + U_i^t) \cdot T_i / \Delta_i^t < \Theta(\tau) \qquad (4)$$

The key of the proof is to show that Condition (4) still guarantees that τ_i^t can meet its deadline. Note that one can not directly apply the $L\&L$ bound $\Theta(\tau)$ to the task set Γ consisting of τ_i^t and the tasks contributing to Y_t, since τ_i^t's deadline is shorter than its period, i.e., Γ does not comply with the $L\&L$ task model. In our proof, this problem is solved by the "period shrinking" technique [16]: we transform Γ into a $L\&L$ task set Γ^* by reducing some of the task periods, and prove that the total utilization of Γ^* is bounded by the LHS of (4), and thereby bounded by $\Theta(\tau)$. On the other hand, the construction of Γ^* guarantees that the schedulability of Γ^* implies the schedulability of τ_i^t. See [16] for details about the "period shrinking" technique. ■

Note that in Condition (3) of Lemma 6, the $L\&L$ bound $\Theta(\tau)$ is involved. This is because in its proof we need to use the $L\&L$ bound $\Theta(\tau)$, rather than the higher parametric bound $\Lambda(\tau)$, to guarantee the schedulability of the constructed task set Γ^* where some task periods are decreased. For example, suppose the original task set is harmonic, the constructed set Γ^* may not be harmonic since some of task periods are shortened to Δ_i^t, which is not necessarily harmonic with other periods. So the 100% bound of harmonic task sets does not apply to Γ^*. However, $\Theta(\tau)$ is still applicable, since it only depends on, and is monotonically decreasing with respect to the task number.

Having this lemma, we now show that a tail subtask τ_i^t cannot be a bottleneck either, if its host processor's utilization is less than $\Lambda(\tau)$, by proving Condition (3) for τ_i^t

Lemma 7. *Let τ be a* light *task set unschedulable by* RM-TS/light*, and let τ_i be a split task whose tail subtask τ_i^t is assigned to processor P_t. If*

$$\mathcal{U}(P_t) < \Lambda(\tau) \qquad (5)$$

then τ_i^t is not a bottleneck of P_t.

Proof: The proof is by contradiction. We assume the lemma does *not* hold for one or more tasks, and let τ_i be the lowest-priority one among these tasks, i.e., τ_i^t is a bottleneck of its host processor P_t, and all tail subtasks with lower priorities are either not a bottleneck or on a processor with assigned utilization at least $\Lambda(\tau)$.

Recall that $\{\tau_i^{b_j}\}_{j\in[1,B]}$ are the B body subtasks of τ_i, and P_t and $\{P_{b_j}\}_{j\in[1,B]}$ are processors hosting the corresponding tail and body subtasks. Since a body task has the highest priority on its host processor (Lemma 3) and tasks are assigned in increasing priority order, all tail subtasks on processors $\{P_{b_j}\}_{j\in[1,B]}$ have lower priorities than τ_i.

We will first show that all processors $\{P_{b_j}\}_{j\in[1,B]}$ have an individual assigned utilization at least $\Lambda(\tau)$. We do this by contradiction: Assume there is a P_{b_j} with $\mathcal{U}(P_{b_j}) < \Lambda(\tau)$. Since tasks are assigned in increasing priority order, we know any tail subtask on P_{b_j} has lower priority than τ_i. And since τ_i is the lowest-priority task violating the lemma and $\mathcal{U}(P_{b_j}) < \Lambda(\tau)$, we know any tail subtask on P_{b_j} is not a bottleneck. At the same time, $\mathcal{U}(P_{b_j}) < \Lambda(\tau)$ also implies the non-split tasks and body subtasks on P_{b_j} are not bottlenecks either. (by Lemma 5). So we can conclude that there is no bottleneck on P_{b_j} which contradicts the fact there is at least one bottleneck on each processor. So the assumption of P_{b_j}'s assigned utilization being lower than $\Lambda(\tau)$ must be false, by which we can conclude that all processors hosting τ_i^t's body tasks have assigned utilization at least $\Lambda(\tau)$. Thus we have:

$$\sum_{j\in[1,B]} \underbrace{(U_i^{b_j} + X_{b_j})}_{\mathcal{U}(P_{b_j})} \geq B \cdot \Lambda(\tau) \tag{6}$$

Further, the assumption from Condition (5) can be rewritten as:

$$X_t + Y_t + U_i^t < \Lambda(\tau) \tag{7}$$

We combine (6) and (7) into:

$$X_t + Y_t + U_i^t < \frac{1}{B} \sum_{j\in[1,B]} (U_i^{b_j} + X_{b_j})$$

Since the partitioning algorithm selects at each step the processor on which the so-far assigned utilization is minimal, we have $\forall j \in [1, B] : X_{b_j} \leq X_t$. Thus, the inequality can be relaxed to:

$$Y_t + U_i^t < \frac{1}{B} \sum_{j\in[1,B]} U_i^{b_j}$$

We also have $B \geq 1$ and $U_i^{body} = \sum_{j\in[1,B]} U_i^{b_j}$, so:

$$Y_t + U_i^t < U_i^{body}$$

Now, in order to get to Condition (3), which implies τ_i^t is not a bottleneck (Lemma 6), we need to show that the RHS of this inequality is bounded by the RHS of Condition (3), i.e., that:

$$U_i^{body} \leq \Theta(\tau)(1 - U_i^{body})$$

It is easy to see that this is equivalent to the following, which holds since τ_i is by assumption a light task:

$$U_i^{body} \leq \frac{\Theta(\tau)}{1 + \Theta(\tau)}$$

By now we have proved Condition (3) for τ_i^t and by Lemma 6 we know τ_i^t is not a bottleneck on P_t, which contradicts to our assumption. ■

We are ready to present RM-TS/light's utilization bound.

Theorem 8. *$\Lambda(\tau)$ is a utilization bound of* RM-TS/light *for light task sets, i.e., any light task set τ with*

$$\mathcal{U}_M(\tau) \leq \Lambda(\tau)$$

is schedulable by RM-TS/light.

Proof: Assume a light task set τ with $\mathcal{U}_M(\tau) \leq \Lambda(\tau)$ is not schedulable by RM-TS/light, i.e., there are tasks not assigned to any of the processors after the partitioning procedure with τ. By this we know the sum of the assigned utilization of all processors after the partitioning is *strictly less* than $M \cdot \Lambda(\tau)$, so there is at least one processor P_q with a utilization *strictly less* than $\Lambda(\tau)$. By Lemma 5 we know the bottleneck of this processor is neither a non-split task nor a body subtask, and by Lemma 7 we know the bottleneck is not a tail subtask either, so there is no bottleneck on this processor. This contradicts the property of the partitioning algorithm, that all processors to which no more task can be assigned must have a bottleneck. ■

V. The Algorithm for Any Task Set

In this section, we introduce RM-TS, which removes the restriction to light task sets in RM-TS/light. We will show that RM-TS can achieve a D-PUB $\Lambda(\tau)$ for any task set τ, if $\Lambda(\tau)$ does not exceed $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$. In other words, if one can derive a D-PUB $\Lambda'(\tau)$ from τ's parameters under uniprocessor RMS, RM-TS can achieve the utilization bound of $\Lambda(\tau) = \min(\Lambda'(\tau), \frac{2\Theta(\tau)}{1+\Theta(\tau)})$. Note that $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$ is decreasing respect to N, and it is around 81.8% when N goes to infinity. For example, we can instantiate our result by the harmonic chain bound $K(2^{1/K} - 1)$:

- $K = 3$. Since $3(2^{1/3} - 1) \approx 77.9\% < 81.8\%$, we know that *any* task set τ in which there are at most 3 harmonic chains is schedulable by our algorithm RM-TS on M processors if its normalized utilization $\mathcal{U}_M(\tau)$ is no larger than 77.9%.
- $K = 2$. Since $2(2^{1/2} - 1) \approx 82.8\% > 81.8\%$, we know 81.8% can be used as the utilization bound in this case: *any* task set τ in which there are at most 2 harmonic chains is schedulable by our algorithm RM-TS on M processors if its normalized utilization $\mathcal{U}_M(\tau)$ is no larger than 81.8%.

So we can see that despite an upper bound on $\Lambda(\tau)$, RM-TS still provides significant room for higher utilization bounds.

For simplicity of presentation, we assume each task's utilization is bounded by $\Lambda(\tau)$. Note that this assumption does not invalidate the utilization bound of our algorithm for task sets which have some individual task's utilization above $\Lambda(\tau)$[5].

RM-TS adds a pre-assignment mechanism to handle the heavy tasks. In the pre-assignment, we first identify the heavy tasks whose tail subtasks would have low priority if they were split, and pre-assign these tasks to one processor each, which avoids the split. The identification is checked by a simple test condition, called *pre-assign condition*. Those heavy tasks that do not satisfy this condition will be assigned (and possibly split) later, together with the light tasks. Note that the number of tasks need to be pre-assigned is at most the number of processors. This will be clear in the algorithm description.

We introduce some notations. If a heavy task τ_i is pre-assigned to a processor P_q, we call τ_i a *pre-assigned task* and P_q a *pre-assigned processor*, otherwise τ_i a *normal task* and P_q a *normal processor*.

A. Algorithm Description

The partitioning algorithm of RM-TS contains three phases:

[5]One can let tasks with a utilization more than $\Lambda(\tau)$ execute exclusively on a dedicated processor each. If we can prove that the utilization bound of all the other tasks on all the other processors is $\Lambda(\tau)$, then the utilization bound of the overall system is also at least $\Lambda(\tau)$.

1) We first pre-assign the heavy tasks that satisfy the *pre-assign condition* to one processor each, in decreasing priority order.
2) We do task partitioning with the remaining (i.e. normal) tasks and remaining (i.e. normal) processors similar to RM-TS/light until all the normal processors are full.
3) The remaining tasks are assigned to the pre-assigned processors in increasing priority order; the assignment selects the processor hosting the lowest-priority pre-assigned task, to assign as many tasks as possible until it is full, then selects the next processor.

The pseudo-code of RM-TS is given in Algorithm V-A. At the beginning of the algorithm, all the processors are marked as *normal* and *non-full*. In the first phase, we visit all the tasks in decreasing priority order, and for each *heavy* task we determine whether we should pre-assign it or not, by checking the *pre-assign condition*:

$$\sum_{i<j} U_j \leq (|\mathcal{P}^{\triangleright}(\tau_i)| - 1) \cdot \Lambda(\tau) \tag{8}$$

where $|\mathcal{P}^{\triangleright}(\tau_i)|$ is the number of processors marked as *normal* at the moment we are checking for τ_i. If this condition is satisfied, we pre-assign this heavy task to the current selected processor, which is the one with the minimal index among all normal processors, and mark this processor as *pre-assigned*. Otherwise, we do not pre-assign this heavy task, and leave it to the following phases. The intuition of the pre-assign condition (8) is: We pre-assign a heavy task τ_i if the total utilization of lower-priority tasks is relatively small, since otherwise its tail subtask may end up with a low priority on the corresponding processor. Note that, no matter how many heavy tasks are there in the system, the number of pre-assigned tasks is at most the number of processors: after $|\mathcal{P}^{\triangleright}(\tau_i)|$ reaching 0, the pre-assign condition never holds, and no more heavy task will be pre-assigned.

In the second phase we assign the remaining tasks to *normal* processors only. Note that the remaining tasks are either light tasks or the heavy tasks that do not satisfy the pre-assign condition. The assignment policy in this phase is the same as for RM-TS/light: We sort tasks in increasing priority order, and at each step select the normal processor P_q with the minimal assigned utilization. Then we do the task assignment: we either add τ_i^k to $\tau(P_q)$ if τ_i^k can be entirely assigned to P_q, or split τ_i^k and assigns a maximized portion of it to P_q otherwise.

In the third phase we continue to assign the remaining tasks to *pre-assigned* processors. There is an important difference between the second phase and the third phase: In the second phase tasks are assigned by a "worst-fit" strategy, i.e., the utilization of all processors are increased "evenly", while in the third phase tasks are now assigned by a "first-fit" strategy. More precisely, we select the pre-assigned processor which hosts the lowest-priority pre-assigned task of all non-full processors. We assign as much workload as possible to it, until it is full, and then move to the next processor. This strategy is one of the key points to facilitate the induction-based proof of the utilization bound in the next subsection.

Algorithm 3 The partitioning algorithm of RM-TS.

```
1:  Mark all processors as normal and non-full

    // Phase 1: Pre-assignment
2:  Sort all tasks in $\tau$ in decreasing priority order
3:  for each task in $\tau$ do
4:     Pick next task $\tau_i$
5:     if DeterminePreAssign($\tau_i$) then
6:        Pick the normal processor with the minimal index $P_q$
7:        Add $\tau_i$ to $\tau(P_q)$
8:        Mark $P_q$ as pre-assigned
9:     end if
10: end for

    // Phase 2: Assign remaining tasks to normal processors
11: Sort all unassigned tasks in increasing priority order
12: while there is a non-full normal processor
             and an unassigned task do
13:    Pick next unassigned task $\tau_i$
14:    Pick the non-full normal processor $P_q$ with minimal
       $\mathcal{U}(P_q)$
15:    $Assign(\tau_i^k, P_q)$
16: end while

    // Phase 3: Assign remaining tasks to pre-assigned processors
    // Remaining tasks are still in increasing priority order
17: while there is a non-full pre-assigned processor
             and an unassigned task do
18:    Pick next unassigned task $\tau_i$
19:    Pick the non-full pre-assigned processor $P_q$ with the
       largest index
20:    $Assign(\tau_i^k, P_q)$
21: end while

22: If there is an unassigned task, the algorithm fails,
    otherwise it succeeds.
```

Algorithm 4 The DeterminePreAssign(τ_i) routine.

```
1: $\mathcal{P}^{\triangleright}(\tau_i)$ := the set of normal processors at this moment
2: if $\tau_i$ is heavy then
3:    if $\sum_{j>i} U_j \leq (|\mathcal{P}^{\triangleright}(\tau_i)| - 1) \cdot \Lambda(\tau)$ then
4:       return true
5:    end if
6: end if
7: return false
```

After these three phases, the partitioning fails if there still are unassigned tasks left, otherwise it is successful. At run-time, the tasks assigned to each processor are scheduled by RMS with their original priorities, and the subtasks of a split task need to respect their precedence relations, which is the same as in RM-TS/light.

Note that, when Assign calculates the synthetic deadlines and verifies whether the tasks assigned to a processor are schedulable, it assumes that any body subtask has the highest priority on its host processor, which has been proved true for RM-TS/light in Lemma 2. It is easy to see that this assumption also holds for the second phase of RM-TS (the task assignment on normal processors), in which tasks are assigned in exactly the same way as RM-TS/light. But it is not clear for this moment whether this assumption also holds for the third phase or not, since there are pre-assigned tasks already assigned to these pre-assigned processors in the first phase, and there is a risk that a pre-assigned task might have higher priority than the body subtask on that processor. However, as will be shown in the proof of Lemma 14, *a body subtask on a pre-assigned processor has the highest priority on its host processor*, thus routine Assign indeed performs a correct schedulability analysis for task assignment and splitting, by which we know any task set successfully partitioned by RM-TS is guaranteed to meet all deadlines at run-time.

B. Utilization Bound

The proof of the utilization bound $\Lambda(\tau)$ for RM-TS. follows a similar pattern as the proof for RM-TS/light, by assuming a task set τ that can't be completely assigned. The main difficulty is that we now have to deal with heavy tasks as well. Recall that the approach in Section IV was to show an individual utilization of at least $\Lambda(\tau)$ *on each single processor* after an "overflowed" partitioning phase. However, for RM-TS, we will not do that directly. Instead, we will show the appropriate bound for *sets of processors*.

We first introduce some additional notation. Let's assume that $K \geq 0$ heavy tasks are pre-assigned in the first phase of RM-TS. Then $\mathcal{P}$ is partitioned into the set of *pre-assigned* processors:

$$\mathcal{P}^{\mathcal{P}} := \{P_1, \ldots, P_K\}$$

and the set of *normal* processors:

$$\mathcal{P}^{\mathcal{N}} := \{P_{K+1}, \ldots, P_M\}.$$

We also use

$$\mathcal{P}_{\geq q} := \{P_q, \ldots, P_M\}$$

to denote the set of processors with index of at least q.

We want to show that, after a failed partitioning procedure of τ, the total utilization sum of all processors is at least $M \cdot \Lambda(\tau)$. We do this by proving the property

$$\sum_{P_j \in \mathcal{P}_{\geq q}} \mathcal{U}(P_j) \geq |\mathcal{P}_{\geq q}| \cdot \Lambda(\tau)$$

by induction on $\mathcal{P}_{\geq q}$ for all $q \leq K$, starting with base case $q = K$, and using the inductive hypothesis with $q = m + 1$ to derive this property for $q = m$. When $q = 1$, it implies the expected bound $M \cdot \Lambda(\tau)$ for all the M processors.

1) Base Case

The proof strategy of the base case is: We assume that the total assigned utilization of normal processors is below the expected bound, by which we can derive the absence of bottlenecks on some processors in $\mathcal{P}^{\mathcal{N}}$. This contradicts the fact that there is at least one bottleneck on each processor after a failed partitioning procedure.

First, Lemma 5 still holds for normal processors under RM-TS, i.e., a bottleneck on a normal processor with assigned utilization lower than $\Lambda(\tau)$ is neither a non-split task nor a body subtask. This is because the partitioning procedure of RM-TS on normal processors is exactly the same as RM-TS/light and one can reuse the reasoning for Lemma 5 here. In the following, we focus on the difficult case of tail subtasks.

Lemma 9. *Suppose there are remaining tasks after the second phase of* RM-TS*. Let τ_i^t be a tail subtask assigned to P_t. If both the following conditions are satisfied*

$$\sum_{P_q \in \mathcal{P}^{\mathcal{N}}} \mathcal{U}(P_q) < |\mathcal{P}^{\mathcal{N}}| \cdot \Lambda(\tau) \tag{9}$$

$$\mathcal{U}(P_t) < \Lambda(\tau) \tag{10}$$

then τ_i^t is not a bottleneck on P_t.

Proof: We prove by contradiction: We assume the lemma does *not* hold for one or more tasks, and let τ_i be the lowest-priority one among these tasks.

Similar with the proof of its counterpart in RM-TS/light (Lemma 7), we will first show that all processors hosting τ_i's body subtasks have assigned utilization at least $\Lambda(\tau)$. We do this by contradiction. We assume $\mathcal{U}(P_{b_j}) < \Lambda(\tau)$, and by Condition (9) we know the tail subtasks on P_{b_j} are not bottlenecks (the tail subtasks on P_{b_j} all satisfy this lemma, since they all have lower priorities than τ_i, and by assumption τ_i is the lowest-priority task does not satisfy this lemma). By Lemma 5 (which still holds for normal processors as discussed above), we know a bottleneck of P_{b_j} is neither a non-split task nor a body subtask. So we can conclude that there is no bottleneck on P_{b_j}, which is a contradiction. Therefore, we have proved that all processors hosting τ_i's body subtasks have assigned utilization at least $\Lambda(\tau)$. This results will be used later in this proof.

In the following we will prove τ_i^t is not a bottleneck, by deriving Condition (3) and apply Lemma 6 to τ_i^t. τ_i is either light or heavy. For the case τ_i is light, the proof is exactly the same as for Lemma 7, since the second phase of RM-TS works in exactly the same way as RM-TS/light. Note that to prove for the light task case, only Condition (9) is needed (the same as in Lemma 7).

In the following we consider the case that τ_i is heavy. We prove in two cases:

- $U_i^{body} \geq \frac{\Lambda(\tau)-\Theta(\tau)}{1-\Theta(\tau)}$
  Since τ_i is a heavy task but not pre-assigned, it failed the pre-assign condition, satisfying the negation of that

condition:

$$\sum_{j>i} U_j > (|\mathcal{P}^{\triangleright}(\tau_i)| - 1) \cdot \Lambda(\tau) \tag{11}$$

We split the utilization sum of all lower-priority tasks in two parts: $\mathcal{U}^\alpha$, the part contributed by pre-assigned tasks, and $\mathcal{U}^\beta$, the part contributed by normal tasks. By the partitioning algorithm construction, we know the $\mathcal{U}^\beta$ part is on normal processors and the $\mathcal{U}^\alpha$ part is on processors in $\mathcal{P}^{\triangleright}(\tau_i) \setminus \mathcal{P}^{\mathcal{N}}$, We further know that each pre-assigned processor has one pre-assigned task, and each task has a utilization of at most $\Lambda(\tau)$ (our assumption stated in the beginning of Section V). Thus, we have:

$$\mathcal{U}^\beta \leq (|\mathcal{P}^{\triangleright}(\tau_i)| - |\mathcal{P}^{\mathcal{N}}|) \cdot \Lambda(\tau) \tag{12}$$

By replacing $\sum_{j>i} U_j$ by $\mathcal{U}^\alpha + \mathcal{U}^\beta$ in (11) and applying (12), we get:

$$\mathcal{U}^\alpha > (|\mathcal{P}^{\mathcal{N}}| - 1) \cdot \Lambda(\tau) \tag{13}$$

The assigned utilizations on processors in $\mathcal{P}^{\mathcal{N}}$ consists of three parts: (i) the utilization of tasks with lower priority than τ_i, (ii) the utilization of τ_i, and (iii) the utilization of tasks with higher priority than τ_i. We know that part (i) is $\mathcal{U}^\alpha$, part (ii) is U_i, and the part (iii) is at least Y_t. So we have

$$\mathcal{U}^\alpha + U_i + Y_t \leq \sum_{P_q \in \mathcal{P}^{\mathcal{N}}} \mathcal{U}(P_q) \tag{14}$$

By Condition (9), (13) and (14) we get

$$U_i + Y_t \leq \Lambda(\tau)$$

In order to use this to derive Condition (3) of Lemma 6, which indicates τ_i^t is not a bottleneck, we need to prove

$$\Lambda(\tau) - U_i^{body} \leq \Theta(\tau)(1 - U_i^{body})$$
$$\Leftrightarrow U_i^{body} \geq \frac{\Lambda(\tau) - \Theta(\tau)}{1 - \Theta(\tau)} \quad (\text{since} \quad \Theta(\tau) < 1)$$

which is obviously true by the precondition of this case.

- $U_i^{body} < \frac{\Lambda(\tau)-\Theta(\tau)}{1-\Theta(\tau)}$
First, Condition (10) can be rewritten as

$$X_t + Y_t + U_i^t < \Lambda(\tau) \tag{15}$$

Since all processors hosting τ_i's body subtasks have assigned utilization at least $\Lambda(\tau)$ (proved in above), we have

$$\sum_{j \in [1, B_i]} X_{b_j} + U_i^{body} > B_i \cdot \Lambda(\tau)$$

Since at each step of the second phase, RM-TS always selects the processor with the minimal assigned utilization to assign the current (sub)task, we have $X_t \geq X_{b_j}$ for each X_{b_j}. Therefore we have

$$B_i X_t + U_i^{body} \geq B_i \cdot \Lambda(\tau)$$
$$\Rightarrow X_t \geq \Lambda(\tau) - U_i^{body} \quad (\text{since} \quad B_i \geq 1)$$

combining which and (15) we get

$$Y_t + U_i^t < U_i^{body}$$

Now, to prove Condition (3) of Lemma 6, which indicates τ_i^t is not a bottleneck, we only need to show

$$U_i^{body} \leq \Theta(\tau)(1 - U_i^{body})$$
$$\Leftrightarrow U_i^{body} \leq \frac{\Theta(\tau)}{1 + \Theta(\tau)}$$

Due to the precondition of this case $U_i^{body} < \frac{\Lambda(\tau)-\Theta(\tau)}{1-\Theta(\tau)}$, we only need to prove

$$\frac{\Lambda(\tau) - \Theta(\tau)}{1 - \Theta(\tau)} \leq \frac{\Theta(\tau)}{1 + \Theta(\tau)}$$
$$\Leftrightarrow \Lambda(\tau) \leq \frac{2\Theta(\tau)}{1 + \Theta(\tau)}$$

which is true since $\Lambda(\tau)$ is assumed to be at most $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$ in RM-TS.

In summary, we know τ_i^t is not a bottleneck. ■

By the above reasoning, we can establish the base case:

Lemma 10. *Suppose there are remaining tasks after the second phase of* RM-TS *(there exists at least one bottleneck on each normal processor). We have:*

$$\sum_{P_q \in \mathcal{P}^{\mathcal{N}}} \mathcal{U}(P_q) \geq |\mathcal{P}^{\mathcal{N}}| \cdot \Lambda(\tau)$$

2) Inductive Step

We start with a useful property concerning the pre-assigned tasks' local priorities.

Lemma 11. *Suppose P_m is a pre-assigned processor. If*

$$\sum_{P_q \in \mathcal{P}_{\geq m+1}} \mathcal{U}(P_q) \geq |\mathcal{P}_{\geq m+1}| \cdot \Lambda(\tau) \tag{16}$$

then the pre-assigned task on P_m has the lowest *priority among all tasks assigned to P_m.*

Proof: Let τ_i be the pre-assigned task on P_m. Since τ_i is pre-assigned, we know that it satisfies the pre-assign condition:

$$\sum_{j>i} U_j \leq (\underbrace{|\mathcal{P}^{\triangleright}(\tau_i)| - 1}_{|\mathcal{P}_{\geq m+1}|}) \cdot \Lambda(\tau)$$

Using this with (16) we have:

$$\sum_{P_q \in \mathcal{P}_{\geq m+1}} \mathcal{U}(P_q) \geq \sum_{j>i} U_j \tag{17}$$

which means the total capacity of the processors with larger indices is enough to accommodate all lower-priority tasks.

By the partitioning algorithm, we know that no tasks, except τ_i which has been pre-assigned already, will be assigned to P_m before all processors with larger indices are full. So no task with priority lower than τ_i will be assigned to P_m. ■

Now we start the main proof of the inductive step.

Lemma 12. *We use* RM-TS *to partition task set τ. Suppose there are remaining tasks after processor P_m is full (there exists at least one bottleneck on P_m). If*

$$\sum_{P_q \in \mathcal{P}_{\geq m+1}} \mathcal{U}(P_q) \geq |\mathcal{P}_{\geq m+1}| \cdot \Lambda(\tau) \tag{18}$$

then we have

$$\sum_{P_q \in \mathcal{P}_{\geq m}} \mathcal{U}(P_q) \geq |\mathcal{P}_{\geq m}| \cdot \Lambda(\tau)$$

Proof: We prove by contradiction. Assume

$$\sum_{P_q \in \mathcal{P}_{\geq m}} \mathcal{U}(P_q) < |\mathcal{P}_{\geq m}| \cdot \Lambda(\tau) \tag{19}$$

With assumption (18) this implies the bound on P_m's utilization:

$$\mathcal{U}(P_m) < \Lambda(\tau) \tag{20}$$

As before, with (20) we want to prove that a bottleneck on P_m is neither a non-split task, a body subtask nor a tail subtask, which forms a contradiction and completes the proof. In the following we consider each type individually.

We first consider non-split tasks. Again, $\Lambda(\tau)$ is sufficient to guarantee the schedulability of non-split tasks, although the relative deadlines of split subtasks on this processor may change. Thus, (20) implies that a non-split task cannot be a bottleneck of P_m.

Then we consider body subtasks. By Lemma 11 we know the pre-assigned task has the lowest priority on P_m. We also know that all normal tasks on P_m have lower priority than the body subtask, since in the third phase of RM-TS tasks are assigned in increasing priority order. Therefore, we can conclude that the body subtask has the highest priority on P_m, and cannot be a bottleneck.

At last we consider tail subtasks. Let τ_i^t be a tail subtask assigned to P_m. We distinguish the following two cases:

- $U_i^{body} < \frac{\Theta(\tau)}{1+\Theta(\tau)}$
  The inductive hypothesis (18) guarantees with Lemma 11 that the pre-assigned task has the lowest priority on P_m, so X_t contains at least the utilization of this pre-assigned task, which is heavy. So we have:

  $$X_t \geq \frac{\Theta(\tau)}{1+\Theta(\tau)} \tag{21}$$

  We can rewrite (20) as $X_t + Y_t + U_i^t < \Lambda(\tau)$ and apply it to (21) to get:

  $$Y_t + U_i^t < \Lambda(\tau) - \frac{\Theta(\tau)}{1+\Theta(\tau)} \tag{22}$$

  Recall that $\Lambda(\tau)$ is restricted by an upper bound in RM-TS:

  $$\Lambda(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}$$
  $$\Leftrightarrow \Lambda(\tau) - \frac{\Theta(\tau)}{1+\Theta(\tau)} \leq \Theta(\tau)(1 - \frac{\Theta(\tau)}{1+\Theta(\tau)})$$

  By applying $U_i^{body} < \frac{\Theta(\tau)}{1+\Theta(\tau)}$ to above we have

  $$\Lambda(\tau) - \frac{\Theta(\tau)}{1+\Theta(\tau)} < \Theta(\tau)(1 - U_i^{body})$$

  And by (22) we have $Y_t + U_i^t < \Theta(\tau)(1 - U_i^{body})$. By Lemma 6 we know τ_i^t is not a bottleneck.
- $U_i^{body} \geq \frac{\Theta(\tau)}{1+\Theta(\tau)}$
  Since τ_i is a heavy task but not pre-assigned, it failed the pre-assign condition, satisfying the negation of that condition:

  $$\sum_{j>i} U_j > (|\mathcal{P}^{\triangleright}(\tau_i)| - 1) \cdot \Lambda(\tau) \tag{23}$$

  We split the utilization sum of all lower-priority tasks in two parts: $\mathcal{U}^{\beta}$, the part contributed by tasks on $\mathcal{P}_{\geq m}$, $\mathcal{U}^{\alpha}$, the part contributed by pre-assigned tasks on $\mathcal{P} \backslash \mathcal{P}_{\geq m}$. By the partitioning algorithm construction, we know the $\mathcal{U}^{\alpha}$ part is on processors in $\mathcal{P}^{\triangleright}(\tau_i) \setminus \mathcal{P}_{\geq m}$, We further know that each pre-assigned processor has one pre-assigned task, and each task has a utilization of at most $\Lambda(\tau)$ (our assumption stated in the beginning of Section V). Thus, we have:

  $$\mathcal{U}^{\beta} \leq (|\mathcal{P}^{\triangleright}(\tau_i)| - |\mathcal{P}_{\geq m}|) \cdot \Lambda(\tau) \tag{24}$$

  By replacing $\sum_{j>i} U_j$ by $\mathcal{U}^{\alpha} + \mathcal{U}^{\beta}$ in (11) and applying (12), we get:

  $$\mathcal{U}^{\alpha} > (|\mathcal{P}_{\geq m}| - 1) \cdot \Lambda(\tau) \tag{25}$$

  The assigned utilizations on processors in $\mathcal{P}_{\geq m}$ consists of three parts: (i) the utilization of tasks with lower priority than τ_i, (ii) the utilization of τ_i, and (iii) the utilization of tasks with higher priority than τ_i. We know that part (i) is $\mathcal{U}^{\alpha}$, part (ii) is U_i, and the part (iii) is at least Y_t. So we have

  $$\mathcal{U}^{\alpha} + U_i + Y_t \leq \sum_{P_q \in \mathcal{P}_{\geq m}} \mathcal{U}(P_q) \tag{26}$$

  By (19), (25) and 26 we have:

  $$Y_t + U_i < \Lambda(\tau)$$
  $$\Leftrightarrow Y_t + U_i^t < \Lambda(\tau) - U_i^{body}$$
  $$\Rightarrow Y_t + U_i^t < \frac{2\Theta(\tau)}{1+\Theta(\tau)} - U_i^{body} \quad \left(\Lambda(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}\right)$$

  By the precondition of this case $U_i^{body} \geq \frac{\Theta(\tau)}{1+\Theta(\tau)}$, we have

  $$\frac{2\Theta(\tau)}{1+\Theta(\tau)} - U_i^{body} \leq \Theta(\tau)(1 - U_i^{body})$$

  Applying this to above we get $Y_t + U_i^t < \Theta(\tau)(1 - U_i^{body})$. By Lemma 6 we know τ_i^t is not a bottleneck.

In summary, we have shown that in both cases the tail subtask τ_i^t is not a bottleneck of P_m. So we can conclude that there is no bottleneck on P_m, which results in a contradiction and establishes the proof. ■

3) *Utilization Bound*

Lemma 10 (base case) and Lemma 12 (inductive step) inductively proved that after a *failed* partitioning, the total utilization of all processors is *at least* $M \cdot \Lambda(\tau)$. And since there are (sub)tasks not assigned to any processor after a failed partitioning, τ's normalized utilization $\mathcal{U}_M(\tau)$ is *strictly larger* than $\Lambda(\tau)$. So we can conclude:

Lemma 13. *Given a task set τ and a* D-PUB $\Lambda(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}$. *$\tau$ can be successfully partitioned by* RM-TS *if its normalized utilization $\mathcal{U}_M(\tau)$ is bounded by $\Lambda(\tau)$.*

Now we will show that a task set is guaranteed to be schedulable if it is successfully partitioned by RM-TS.

Lemma 14. *If a task set is successfully partitioned by* RM-TS*, the tasks on each processor are schedulable by* RMS.

Proof: RM-TS uses routine Assign for task assignment and splitting, which assumes a body subtask has the highest priority on its host processor (this has been shown to be true for RM-TS/light in Lemma 2, so in RM-TS/light a successfully partitioning implies the schedulability). In RM-TS, this assumption is clearly true for *normal* processors, on which the task assignment is exactly the same as RM-TS/light. In the following, we will show this assumption is also true for *pre-assigned* processors.

Let P_q be a pre-assigned processor involved in the third phase of RM-TS, and a body subtask $\tau_i^{b_j}$ is assigned to P_q. By Lemma 10 and 12 we can inductively prove that the total utilization of processors in $\mathcal{P}_{\geq q+1}$ is at least $|\mathcal{P}_{\geq q+1}| \cdot \Lambda(\tau)$. So by Lemma 11 we know a pre-assigned task on processors P_q has the lowest priority on that processor, particularly, has lower priority than $\tau_i^{b_j}$. We also know that all other tasks on P_q have lower priority than $\tau_i^{b_j}$, since tasks are assigned in increasing priority order and $\tau_i^{b_j}$ is the last one assigned to P_q.

In summary we know that after partitioned by RM-TS, any body subtask has the highest priority on its host processor. So Assign indeed performs a correct task assignment and splitting, which guarantees that all deadlines can be met at run-time. ■

By now we have proved that any task with total utilization no larger than $\Lambda(\tau)$ can be successfully partitioned by RM-TS, and all tasks can meet deadline if they are scheduled on each processor by RMS. So we can conclude the utilization bound of RM-TS:

Theorem 15. *Given a parametric utilization bound $\Lambda(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}$ derived from the task set τ's parameters. If*

$$\mathcal{U}_M(\tau) \leq \Lambda(\tau)$$

then τ is schedulable by RM-TS.

Proof: Directly follows Lemma 13 and 14. ■

VI. Conclusions

We have developed new fixed-priority multiprocessor scheduling algorithms overstepping the Liu and Layland utilization bound. The first algorithm RM-TS/light can achieve any sustainable parametric utilization bound for light task sets. The second algorithm RM-TS gets rid of the light restriction and work for any task set, if the bound is under a threshold $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$. Further, the new algorithms use exact analysis RTA, instead of the worst-case utilization threshold as in [16], to determine the maximal workload assigned to each processor. Therefore, the average-case performance is significantly improved. As future work, we will extend our algorithms to deal with task graphs specifying task dependencies and communications.

References

[1] J. Anderson, V. Bud, and U.C. Devi. An EDF-based scheduling algorithm for multiprocessor soft real-time systems. In *ECRTS*, 2005.

[2] J. Anderson and A. Srinivasan. Mixed pfair/erfair scheduling of asynchronous periodic tasks. In *Journal of Computer and System Sciences*, 2004.

[3] B. Andersson. Global static priority preemptive multiprocessor scheduling with utilization bound 38%. In *OPODIS*, 2008.

[4] B. Andersson, S. Baruah, and J. Jonsson. Static priority scheduling on multiprocessors. In *RTSS*, 2001.

[5] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems multiprocessors. In *RTSS*, 2008.

[6] B. Andersson and E. Tovar. Multiprocessor scheduling with few preemptions. In *RTCSA*, 2006.

[7] T. Baker. An analysis of EDF schedulability on a multiprocessor. *IEEE Trans. on Parallel and Dist. Sys.*, 2005.

[8] T. P. Baker and S. Baruah. Sustainable multiprocessor scheduling of sporadic task systems. In *ECRTS*, 2009.

[9] S. Baruah and A. Burns. Sustainable schedulability analysis. In *RTSS*, 2006.

[10] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: A notion of fairness in resource allocation. In *Algorithmica*, 1996.

[11] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. *A Categorization of Real-Time Multiprocessor Scheduling Problems and Algorithms*. 2004.

[12] D. Chen, A. K. Mok, and T. W. Kuo. Utilization bound revisited. In *IEEE Transaction on Computers*, 2003.

[13] H. Cho, B. Ravindran, and E. Jensen. An optimal realtime scheduling algorithm for multiprocessors. In *RTSS*, 2006.

[14] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. In *Operations Research, Vol. 26, No. 1, Scheduling*, 1978.

[15] K. Funaoka et. al. Work-conserving optimal real-time scheduling on multiprocessors. In *ECRTS*, 2008.

[16] N. Guan, M. Stigge, W. Yi, and G. Yu. Fixed-priority multiprocessor scheduling with Liu & Layland's utilization bound. In *RTAS*, 2010.

[17] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *EMSOFT*, 2008.

[18] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *IPDPS*, 2008.

[19] S. Kato and N. Yamasaki. Semi-partitioned fixed-priority scheduling on multiprocessors. In *RTAS*, 2009.

[20] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *ECRTS*, 2009.

[21] T. W. Kuo and A. K. Mok. Load adjustment in adaptive real-time systems. In *RTSS*, 1991.

[22] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned fixed-priority preemptive scheduling for multi-core processors. In *ECRTS*, 2009.

[23] S. Lauzac, R. Melhem, and D. Mosse. An efficient rms admission control and its application to multiprocessor scheduling. In *IPPS*, 1998.

[24] J. P. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, 1989.

[25] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *Journal of the ACM*, 1973.

[26] J. W. S. Liu. *Real-time systems*. Prentice Hall, 2000.