Fixed-Priority Multiprocessor Scheduling: Beyond Liu & Layland Utilization Bound

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

Abstract

The increasing interests in multicores raise the question whether utilization bounds for uni-processor scheduling can be generalized to the multiprocessor setting. Recently, this has been shown 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 open problem for most parametric utilization bounds by proposing new partitioning-based scheduling algorithms.*

Besides the improved utilization bounds, another advantage of our new algorithms is the significantly improved average-case performance, since exact analysis, i.e., Response Time Analysis, instead of the utilization bound threshold as in previous work, is used to determine the maximal workload on each processor.

1 Introduction

Liu and Layland discovered the famous utilization bound $N(2^{1/N}-1)$ for fixed-priority scheduling on uni-processors in the 1970's [7]. Recently, we generalized this bound to multiprocessors by a partitioning-based scheduling algorithm [3].

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. System resources would be considerably under-utilized if one only relies on the $L\&L$ bound in the system design.

However, 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 [8]: 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" in the system [4]. Generally, 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 3, quite a few 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 in the system is no larger than M?

In this paper, we will address the above question by proposing a new RMS-based partitioned scheduling algorithms (with task splitting). The new algorithm RM-TS/light generalizes all known parametric utilization bounds for RMS to multiprocessors, for a subclass of "light" task sets in which each task's individual utilization is at most $\frac{\Theta(\tau)}{1+\Theta(\tau)}$, where $\Theta(\tau)$ denotes the $L\&L$ bound for task set τ. Then we present the second algorithm RM-TS that works for any task set, if the parametric utilization bound is under the threshold $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$ [1].

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[2] 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 3 will discuss this problem in detail. In this paper, we use more sophisticated proof techniques to solve this problem, and thereby, generalize the parametric utilization bounds to multiprocessors.

Besides the improved utilization bounds, another advantage of our new algorithms is the significantly improved average-case performance. Although the algorithm in [3] 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 [3], to determine the maximal workload on each processor. Therefore, our new algorithm has much better performance than the algorithm in [3].

2 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.

[1] Note that when $\Theta(\tau) \doteq 69.3\%$, $\frac{\Theta(\tau)}{1+\Theta(\tau)} \doteq 40.9\%$ and $\frac{2\Theta(\tau)}{1+\Theta(\tau)} \doteq 81.8\%$

[2] Task splitting is needed to exceed the 50% utilization bound limitation of conventional partitioned scheduling. Section 2 will introduce task splitting in detail.

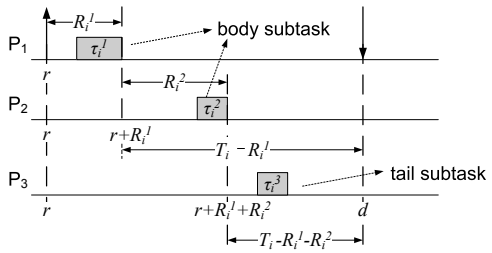

Figure 1. An Illustration of Task Splitting.

With the a partitioned scheduling algorithm (with task splitting), 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

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

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 $\tau(P_q)$:

$$\mathcal{U}(P_q) = \sum_{\tau_i \in \tau(P_q)} U_i$$

3 Deflatable Parametric Utilization Bounds

A *Parametric Utilization Bound* (PUB for short) $\Omega(\tau)$ for a task set τ is the result of applying a function $\Omega(\cdot)$ to τ's task parameters, such that all tasks in τ are guaranteed to meet their deadlines under RMS on a uni-processor if τ's total utilization $\mathcal{U}(\tau) \leq \Omega(\tau)$[3].

There have been quite a few parametric utilization bounds 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$(\tau) = K(2^{1/K} - 1)$ [4] , 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) [6] 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* [6].
- R-Bound(τ) [6] 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: Suppose a PUB $\Omega(\tau)$ is derived from a task set τ's parameters. If we decrease the execution times of some tasks in τ to get a new task set τ', then $\Omega(\tau)$ is still applicable to τ'. We call a PUB holding this property a *deflatable* parametric utilization bound, as formally stated in the following definition:

Definition 1. *A* Deflatable *Parametric Utilization Bound (*D-PUB*)* $\Omega(\tau)$ *is a* PUB *satisfying the following property: We decrease the execution times of some tasks in* τ *to get a new task set* τ'*. If* τ' *satisfies* $\mathcal{U}(\tau') \leq \Omega(\tau)$*, then it is guaranteed to be schedulable by* RMS *on a uni-processor.*

The *deflatable* property is very common for PUBs: In fact all PUBs for RMS on uni-processors we are aware of are deflatable[4]. In the following, we use $\Omega(\tau)$ to denote an arbitrary D-PUB derived from τ's parameters under RMS on uni-processors.

4 The Algorithm for Light Tasks: RM-TS/light

In the following we introduce the first algorithm RM-TS/light, which achieves $\Omega(\tau)$ (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 2. *A task* τ_i *is a* light task *if*

$$U_i \leq \frac{\Theta(\tau)}{1 + \Theta(\tau)} \tag{2}$$

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 tasks.*

4.1 Algorithm Description

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

- 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.
- A task (subtask) can be entirely assigned to the current processor, if all tasks including this one on this processor can meet their deadlines under RMS.
- When a task (subtask) cannot be assigned entirely to the current processor, we split it into two parts[5]. The first part is assigned to the current processor. The splitting is done such

[3] Note that the Hyperbolic Bound [1] is not a PUB.

[4] The PUBs we are aware of include the ones listed above, and the non-closed-form bounds in [2]. We do not exclude the possibility that there might exist (undiscovered) parametric utilization bounds that are *not* deflatable. However, proving the existence of, or finding such a non-deflatable bound is out of the scope of this paper.

[5] 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.

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 to the next selected processor.

In the following, we will give a detailed description. Algorithm 1 and 2 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 to do the task assignment. Assign 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^k of each task τ_j^k after assigning the new task τ_i^k to P_q with the well-known fixed-point formula:

$$R_j^k = \sum_{\substack{\tau_h \in \tau(P_q) \\ h<j}} \left\lceil \frac{R_j^k}{T_h} \right\rceil C_h + C_j^k$$

The response time R_j^k obtained for each (sub)task τ_j^k is compared to its (synthetic) deadline Δ_j^k. 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.

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

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

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

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

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): one subtask 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 formally, we introduce the concept of a *bottleneck*.

Definition 3. *A* bottleneck *of processor P_q is a (sub)task that is assigned to P_q, and will become non-schedulable if we increase the execution time of the task with the highest priority on P_q by an arbitrarily small positive number.*

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. Thus, we can state:

Definition 4. 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 $\tau(P_q)$ non-schedulable.*
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 [5], 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 practise it is very efficient.

Calculating Synthetic Deadlines Now we will 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. Now we consider the case that τ_i^k is a subtask of a split task τ_i. Recall that tasks are assigned in increasing order of priorities. Thus, right after a (sub)task is split and assigned to its host processor, the first part of it, which is a body subtask, has the highest priority on that processor. After that the processor will be marked as *full* and consequently no other tasks of higher priority can be assigned to it. So we know:

Lemma 1. *A body subtask has the highest priority on its host processor.*

A consequence of this is, 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 do not 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 explicitly stated in the following lemma.

Lemma 2. *Let τ_i be a task split into B_i body subtasks $\tau_i^{b_1}, \ldots, \tau_i^{b_{B_i}}$, assigned to processors $P_{b_1}, \ldots, P_{b_{B_i}}$ respectively, and the tail subtask τ_i^t assigned to processor P_t. The synthetic deadline Δ_i^t of a tail subtask τ_i^t is calculated by:*

$$\Delta_i^t = T_i - \sum_{j \in [1, B_i]} C_i^{b_j}$$

Scheduling at Run Time At runtime, the tasks will be scheduled using RMS 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, i.e., the guarantee that all deadlines can be met.

Lemma 3. *Any task set that has been successfully partitioned by* RM-TS/light *is schedulable.*

4.2 Utilization Bound

We can prove the utilization bound property for RM-TS/light:

Theorem 4. *$\Omega(\tau)$ is a utilization bound of* RM-TS/light *for light task sets, i.e., any light task set τ with*

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

is schedulable by RM-TS/light.

The proofs are omitted due to space limit.

5 The Algorithm for Any Task Set: RM-TS

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 $\Omega(\tau)$ for any task set τ, if $\Omega(\tau)$ does not exceed $\frac{2\Theta(\tau)}{1+\Theta(\tau)}$. In other words, if one can derive a D-PUB $\Omega'(\tau)$ from τ's parameters under uni-processor RMS, RM-TS can achieve the utilization bound of $\Omega(\tau) = \min(\Omega'(\tau), \frac{2\Theta(\tau)}{1+\Theta(\tau)})$. Note $\frac{2\Theta(\tau)}{1+\Theta(\tau)} = 81.8\%$ when $\Theta(\tau) = 69.3\%$. So we can see that despite an upper bound on $\Omega(\tau)$, RM-TS still provides significant room for higher utilization bounds.

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

    // Phase 1: Pre-assignment
2:  Sort all tasks in τ in decreasing priority order
3:  for each task in τ do
4:      Pick next task τ_i
5:      if DeterminePreAssign(τ_i) then
6:          Pick the normal processor with the minimal index P_q
7:          Add τ_i to τ(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 τ_i
14:     Pick the non-full normal processor P_q with minimal U(P_q)
15:     Assign(τ_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 τ_i
19:     Pick the non-full pre-assigned processor P_q with the largest index
20:     Assign(τ_i^k, P_q)
21: end while

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

Algorithm 3: The partitioning algorithm of RM-TS.

```
1: P^⊳(τ_i) := the set of normal processors at this moment
2: if τ_i is heavy then
3:     if Σ_{j>i} U_j ≤ (|P^⊳(τ_i)| − 1) · Ω(τ) then
4:         return true
5:     end if
6: end if
7: return false
```

Algorithm 4: The DeterminePreAssign(τ_i) routine.

5.1 Algorithm Description

We introduce some notations. If a heavy task τ_i is pre-assigned to a processor P_q in RM-TS, 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*.

The partitioning algorithm of RM-TS is shown in Algorithm 3, which contains three main phases:

1. We first pre-assign the heavy tasks that satisfy the *Pre-assign Condition* (line 4 in Algorithm 4) 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 with the largest index (i.e., the one 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 3.

5.2 Utilization Bound

We can prove the utilization bound of RM-TS:

Theorem 5. *Given a deflatable parametric utilization bound $\Omega(\tau) \leq \frac{2\Theta(\tau)}{1+\Theta(\tau)}$ derived from the task set τ's parameters. If*

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

then τ is schedulable by RM-TS.

The proofs are omitted due to space limit.

6 Conclusions and Future Work

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 deflatable parametric utilization bound for light task sets. The second algorithm RM-TS gets rid of the light restriction and works 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 [3], 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 dependency constraints and task communication.

References

[1] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate monotonic analysis: The hyperbolic bound. *IEEE Transactions on Computers*, 2003.

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

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

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

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

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

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

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