IMPROVING PARTITIONED SCHEDULING BOUNDS

by

CHARULAKSHMI VIJAYAGOPAL

(Under the direction of Shelby H. Funk)

ABSTRACT

We consider multiprocessor partitioned scheduling of real-time systems of periodic or sporadic tasks. Prior to execution, tasks are partitioned onto the processors using the First-Fit Decreasing algorithm. We demonstrate methods of allocating tasks onto processors even without complete information about the task. We propose two new models to answer two general questions:

1. *Given the description of a task set, how many processors should our system have to ensure that the actual task set can be partitioned onto the system?*

2. *Given the description of a task set and a specific number of processors, how large can our total utilization be?*

Answers to these are useful to system designers who make design decisions before fully developing tasks. A task set is characterized using maximum utilization, u_{max} and utilization binder, γ in our first model and a cumulative utilization function $U_{cum}(i)$ in our second model. Compared to the current method, we reduce the number of processors a task set requires by up to 90% and nearly double utilization bound for a fixed number of processors.

INDEX WORDS: Partitioned scheduling, Multiprocessor scheduling algorithm,
Utilization bounds, Number of processors, Dissertations, Theses (academic)

IMPROVING PARTITIONED SCHEDULING BOUNDS

by

CHARULAKSHMI VIJAYAGOPAL

B. Tech, Anna University, 2005

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

MASTER OF SCIENCE

ATHENS, GEORGIA

2010

IMPROVING PARTITIONED SCHEDULING BOUNDS

by

CHARULAKSHMI VIJAYAGOPAL

Approved:

Major Professor:	Shelby H. Funk
Committee:	Hamid Arabnia
	Maria Hybinette

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2010

ACKNOWLEDGMENTS

I would like to thank University of Georgia for funding my Masters program and my research work with Dr. Shelby Funk.

I would also like to sincerely thank my major professor Dr. Shelby Funk who was an integral part of this research work and thesis.

I want to dedicate this research work to my Grandmother Mrs. Rajeshwari Vishwanathan who means the world to me.

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iv

LIST OF FIGURES vi

LIST OF TABLES vii

CHAPTER

1 INTRODUCTION 1

2 MODEL AND DEFINITIONS 6

3 RELATED WORK 10

4 USING γ TO IMPROVE THE BOUNDS 17

5 USING CUMULATIVE UTILIZATION FUNCTION 31

6 CONCLUSION 42

BIBLIOGRAPHY 43

LIST OF FIGURES

3.1 Representation of β as a function of u_{max} [1] 12

3.2 Worst-case achievable utilization for EDF-FF [1] 14

3.3 Utilization bound for each processor using Lopez method and FFD 14

4.1 Our results compared to Lopez, et al., bounds. 27

4.2 Our results compared to Lopez, et al., for fixed number of tasks 27

4.3 Our results compared to Lopez, et al., for fixed values of alpha and gamma 28

5.1 Hop over the curve to find the number of processors 32

5.2 Number of processors required does not depend on the total utilization 33

5.3 Hop over the curve to find the number of processors 40

5.4 Comparison of bounds using U_{cum} vs. Lopez, et al., bounds for different sizes of task sets 41

5.5 Comparison of bounds using U_{cum} vs. Lopez, et al., bounds for different values of total utilization 41

LIST OF TABLES

2.1 Example of a task set, the cumulative utilization values and corresponding values returned by $U_{cum}(i)$ function 9

2.2 Table containing the symbols and notations used throughout this thesis 9

4.1 Example of an actual task set τ and its corresponding maximal task set Γ 18

CHAPTER 1

INTRODUCTION

Real time systems are different from general operating systems because correct operation of these systems *require* that they are timely, dependable and predictable. All real-time jobs have deadlines which must be met in order to ensure correct operation of the system. Often, real-time systems are used in embedded systems - i.e., computer systems designed to perform a small number of dedicated functions, often with real-time computing constraints. Unlike a normal personal computer which is flexible and meets a wide range of end-user needs, embedded systems are part of a complete device often including hardware and mechanical parts . With the advent of multicore systems, multiprocessors are more prevalent and are used in many embedded systems. The real time community has increasing desire to use these types of systems. Unfortunately, many algorithms that work well on uniprocessors do not work as well in a multiprocessor environment.

Consider, for example, the Earliest Deadline First algorithm (EDF), a scheduling algorithm which gives higher priority to jobs with earlier deadlines. This algorithm can schedule any set of jobs to meet their deadlines on a uniprocessor if it is possible to do so [2]. However, on multiprocessors, EDF is known to require the processors to idle for as much as half the time [3, 4, 5].

Often real-time jobs execute repeatedly and at regular intervals. We call these repeating jobs as periodic tasks, if the jobs arrive at regular intervals, or sporadic tasks if the intervals between jobs may vary. An apt example would be an airplane auto-pilot system which has to regularly receive signals from ground-based Air Traffic Controller (ATC), update its speed, check on its altitude, steer in the right direction and all these tasks have to done periodically and regularly. The task set of an auto-pilot contains the afore mentioned set of tasks. These tasks are executed periodically and

regularly. These tasks periodically generate jobs and these jobs cannot miss their deadlines. To be able to meet their deadlines, they have to be scheduled on processor(s) using efficient algorithms.

Three main scheduling strategies are:

1. **Global Scheduling**: For m processors, there is a single queue of jobs. Processors execute the next available job from the job queue when they are done with executing the current job.

2. **Partitioned Scheduling**: For m processors, there are m job queues. Tasks are allocated to processors before executing the jobs of tasks,

3. **Restricted Scheduling**: Tasks can only migrate at job boundaries. If a job is preempted, it must restart on the same processor. However, the next job of the same task may execute on a different processor.

We consider partitioned scheduling of task sets on multiprocessors. Partitioned scheduling means tasks are partitioned (i.e. allocated) to processors prior to execution and scheduled locally on each processor using a uniprocessor scheduling algorithm. While tasks are executed online, the partitioning of the task sets is done off-line. Allocating or partitioning of tasks is primarily done off-line using the First Fit Decreasing (FFD) the allocation algorithm which tries to find the first processor that can schedule a given task by sequentially checking the available processors in a fixed order.

One important parameter of a task is its utilization (i.e., the average proportion of processor time a task requires to execute). We can use task utilization values to determine how many tasks can safely be allocated to a single processor. For example, when using the EDF algorithm to schedule each processor, we need only ensure that the total utilization of the tasks assigned to any processor never exceeds 1.

The process of partitioning the tasks onto processors is similar to the bin packing problem [6] which is NP-Complete. NP-Complete problems can be verified quickly, but there is no known way to locate a solution efficiently. Hence, the allocation that uses the minimum possible number of processors could take an reasonably long time to determine. Nonetheless, it is desirable to keep the

number of required processors as small as possible. Fortunately we know that allocating tasks using the First Fit Decreasing algorithm will not cause us to exceed the minimum number of processors by more than a factor of $\frac{11}{9}$ [7].

The goal of our research is to aid designers who need to make hardware and software decisions with incomplete knowledge of the total system. In particular, designers may need to specify the required number of processors before all tasks are fully developed. Alternatively, they may need to tailor task functionality to a given processing platform.

In this research, we address the following two questions:

1. *Given incomplete knowledge of tasks, how many processors must we have to ensure successful partitioning of all the tasks of a task set τ using FFD?*

2. *Given incomplete knowledge of tasks and a fixed number of processors m, how large can τ's total utilization be while still ensuring the successful partitioning of τ onto m processors using FFD?*

Keeping the number of processors as small as possible brings the system costs down. Making total utilization to be as large as possible allows extra features to be added to the system. Thus finding accurate answers to these questions could allow designers to add features to a system while still keeping costs down. Our approach to answeringquestions is to find a lower bound on the number of processors and an upper bound on the total utilization. We call these two bounds *System bounds*.

This problem is similar to the bin-packing problem in which objects of different volumes must be packed into a finite number of bins of finite capacity in a way that minimizes the number of bins used. In our case, because the uniprocessor utilization bound is 1, we can place tasks onto processors as long as their total utilization does not exceed 1. In [1], Lopez, et al., modeled task sets using two parameters, u_{max}, and U_{sum}, where u_{max} is the maximum utilization value of all tasks and U_{sum} is the sum of all the utilization values. While the method suggested by Lopez, et al., is easy to use, it is overly pessimistic in calculating the bounds on the number of processors and the processor utilization.

In [8], Mohan, et al., state the importance of making system architecture decisions in the design phase rather than integration phase. They explain how a coarse approximation of required results may be wasteful and how an overly optimistic approximation of required resources may lead to undesirable surprises. To this end they have developed a method of determining a fairly accurate description of the worst case behavior of the tasks during the design time. The research presented in this thesis exploits the more accurate task descriptions.

Mohan, et al., in [8] provide a tool for finding how more information about a task set for use during the design time. Using this work in [8], we know we can get more detailed information about a task set than just u_{max} and U_{sum} even before they are fully developed. We show that restricting our understanding to u_{max} and U_{sum} gives overly pessimistic understanding of the system. Correct analysis using such restricted information must assume that all the tasks of a task set have the same large utilization value, namely u_{max}. This assumption increases the system costs by requiring an excessive number of processors and degrades the utility of the system by imposing a small system load (i.e., reduced functionality).

In short, designers who need to make software and hardware decisions must base their decisions on incomplete information of the task set. In [1], Lopez, et al., approach the problem by modeling task sets using just their maximum utilization, u_{max}, and total utilization U_{sum}. On the other hand, in [8], Mohan, et al., claim that we will have more information than just u_{max} and U_{sum}. In reality, we can have a close estimation of the worst-case execution time(WCET) and the utilization values of tasks. In this thesis, we reduce the pessimism of the approach suggested by Lopez, et al., and present two new task models which are more accurate than only using u_{max} and U_{sum}:

1. **The Utilization Binder** γ: For a task set τ, the utilization binder γ can be used to find the upper bound on utilization value of each task. The utilization binder is used in conjunction with u_{max}.

2. **The Cumulative Utilization Function,** $U_{cum}(i)$: A function that provides an upper bound for the sum of utilization values of first the i tasks where the tasks are sorted by their utilization values in descending order.

Using the methods described by Mohan, et al., in [8] and Christian, et al., in [9], we can describe our task sets using either of these models. We show that each of these models can greatly improve the system bounds. The bounds on the number of processors required to schedule a given task set can be reduced up to 65% using the utilization binder and 50% using the cumulative utilization function. The bounds on the total utilization can be improved up to 98% using the utilization binder and 80% using the cumulative utilization function.

The remainder of this thesis is organized as follows: Chapter 2 presents the models and definitions, Chapter 3 presents results that are related to this work, Chapter 4 provides a detailed analysis of our utilization binder model, Chapter 5 provides detailed analysis of the cumulative utilization function. Finally Chapter 6 concludes the thesis.

CHAPTER 2

MODEL AND DEFINITIONS

This Chapter defines all important terms and provides a more detailed description of our assumed system model. This research considers partitioning n periodic or sporadic tasks onto m processors, denoted $\rho_0, \rho_1 \ldots \rho_{m-1}$. Let $\tau = \{T_0, T_1, \ldots, T_{n-1}\}$ denote a set of n periodic or sporadic tasks sorted in the non-increasing order by their utilization values. Each task T_i in τ is described using the 2-tuple $T_i = (p_i, e_i)$, where p_i is its period, and e_i is its worst case execution time. Each task T_i generates a sequence of jobs $J_{i,0}, J_{i,1}, \ldots, J_{i,k}, \ldots$. If T_i is a periodic task, then each job $J_{i,k}$ arrives at time $a_{i,k} = k \times p_i$. If T_i is a sporadic task then $J_{i,0}$ can arrive at any time $t \geq 0$ and all jobs must arrive at least p_i time units apart – i.e., $a_{i,k} \geq a_{i,k-1} + p_i$ for every $k > 0$. Each job $J_{i,k}$ has deadline $d_{i,k} = a_{i,k} + p_i$. Any correct schedule must execute $J_{i,k}$ for e_i time units during the interval $[a_{i,k}, d_{i,k})$.

The utilization, u_i, of task T_i is the proportion of processing time the task requires and is equal to e_i/p_i. The maximum and total utilization of a task set are denoted

$$u_{max} = \max_{0 \leq i < n} \{u_i\} \text{ and } U_{sum} = \sum_{i=0}^{n-1} u_i.$$

Utilization can be a very useful tool in analyzing task sets. For example, if a periodic or sporadic task set τ has utilization $U_{sum} \leq 1$, then EDF will schedule τ to meet all deadlines on a uniprocessor [2]. This result is important for our research because we assume the local scheduler on each processor is EDF. Therefore we can assign tasks $T_i, T_{i+1}, \ldots$ to a processor ρ as long as the total utilization of the tasks assigned to each processor does not exceed 1. Similarly we can add β tasks $\{T_{i+j} | j = 0, 1, \ldots \beta - 1\}$ to a processor whenever

$$\sum_{j=i}^{i+\beta-1} u_j \leq 1.$$

Because τ is sorted by utilization, we know

$$\sum_{j=i}^{i+\beta-1} \leq \beta u_i.$$

Therefore, if $\sum_{j=i}^{i+\beta+1} \leq 1$ then

$$\frac{1}{\beta} \leq u_i.$$

Because $\beta \in \mathbb{Z}$, we can conclude

$$\beta \geq \left\lfloor \frac{1}{u_i} \right\rfloor.$$

Thus the minimum number of tasks that can be assigned to a processor before the processor fills up is a function of the utilization of the first task assigned to that processor. For each processor ρ_j, we let U_{β_j} denote the total utilization of tasks assigned to processor ρ_j is U_{ρ_j} While adding tasks to a processor, we say the *gap* on the processor ρ_j is

$$gap(\rho_j) = 1 - U_{\rho_j} = 1 - \sum_{T_i \in A(\rho_j)} u_i$$

where, $A(\rho_j)$ is the set of tasks already assigned to ρ_j. Thus, after assigning the tasks in $A(\rho_j)$ to ρ_j , no additional task assigned to ρ_j can have utilization larger than $gap(\rho_j)$. If a task T_i has $u_i \leq gap(\rho_j)$, we say T_i *fits* onto ρ_j.

We use two methods in our research to model a task set, namely:

1. The utilization binder, γ, and the maximum utilization, u_{max}.

2. The cumulative utilization function $U_{cum}(i)$.

Given a task set τ composed of n tasks, if $u_i \leq u_{max} \times \gamma^i$ for $i = 0, 1, \ldots n-1$, then we can use u_{max} and γ to model τ. Value of γ can be derived easily if fair estimates of Worst Case Execution Times (WCET) of tasks are available. As discussed in Chapter 1, methods suggested in [9] and [8] can provide reasonable estimates of WCETs of tasks.

The example below shows how u_{max} and γ can be used effectively to characterize a task set.

Example 1. *Assume we know our task set will have five tasks with utilization in the following ranges:*

- *1 task with* $u_i \epsilon [0.3, 0.4]$ $(u_{max} \geq 0.4)$
- *2 tasks with* $u_i \epsilon [0.25, 0.3]$ $(u_{max} \times \gamma^i \geq 0.3 \; for \; i = 1, 2)$
- *2 tasks with* $u_i \leq 0.2$ $(u_{max} \times \gamma^i \geq 0.2 \; for \; i = 3, 4)$

Thus with partial information about τ*, we can select appropriate values for* γ *and* u_{max}*. If we let* $u_{max} = 0.4$*, we need to find* γ *such that* $\gamma \geq (0.3/0.4)^{frac12}$ *and* $\gamma \geq (0.2/0.4)^{1/4}$ *– i.e.,* $\gamma \geq max\{0.87, 0.84\}$*. Letting* $u_{max} = 0.4$ *and* $\gamma = 0.87$ *for* $i = 0, 1, 2, 3, 4$*, we have* $u_{max} \cdot \gamma^i = 0.4, 0.348, 0.3027, 0.2633, 0.2291$ *which satisfies our assumptions about the task set.*

For this model, we define $\Psi(u_{max}, \gamma)$ and $\Gamma u_{max,\gamma}$ as follows.

Definition 1. *Let* τ *be a task set with maximum utilization* $\leq u_{max}$ *such that* $T_i \in \tau$ *and* $u_i \leq u_{max}\gamma^i$*. Then we say* $\tau \in \Psi(u_{max}, \gamma)$*. We let* $\Gamma u_{max,\gamma} = \{T_{0,\Gamma}, T_{1,\Gamma}, \ldots\}$ *be the "maximal" task set in* $\Psi(u_{max}, \gamma)$ *– i.e., for all* $i \geq 0$*, the utilization of the* i^{th} *task is* $u_i^\Gamma = u_{max}\gamma^i$*. Hence,* $u_i^\Gamma / u_{i-1}^\Gamma = \gamma$ *for all* $i > 0$ *and* $\Gamma(u_{max}, \gamma)$ *is assumed to have an unlimited number of tasks.*

Rather than bounding individual task utilizations, our second method uses cumulative utilization function to model a task set τ. A task set τ can be described using a cumulative utilization function U_{cum}, if $U_{cum}(i) \geq \sum_{k=0}^{i} u_k \forall i = 0, 1, \ldots n-1$. While $\Psi(u_{max}, \gamma)$ bounds every task's utilization, U_{cum} does not, as illustrated in the following example.

Example 2. *Assume a task set* τ *containing tasks with utilization values and cumulative utilization values as shown in Table 2.1. Even though* $U_{cum}(i) \geq \sum_{k=0}^{i} u_k$ *for all* $k = 0, 1, 2, 3, 4$*, we cannot use* U_{cum} *to define individual task utilization values. For example* $u_1 = 0.4$ *¿* $U_{cum}(1) - U_{cum}(0)$*.*

For the purposes of our analysis we often need to know how many tasks are assigned to a processor ρ_j when ρ_j becomes full (i.e, when the first task is unable to fit onto ρ_j). Also we let B_j be the upper bound of tasks assigned to the processors $\rho_0, \rho_1, ..., \rho_j$

We will begin to present our bounds in Chapter 4 and Chapter 5 assuming that tasks are sorted in weakly decreasing order. First, though, we discuss work related to our research.

A table containing all the symbols used in this thesis is given in Table- 2.2

Table 2.1: Example of a task set, the cumulative utilization values and corresponding values returned by $U_{cum}(i)$ function

Task-id (i)	**Task utilization value**	**Cumulative utilization value**	$U_{cum}(i)$
0	0.5	0.5	0.6
1	0.4	0.9	0.95
2	0.4	1.3	1.4
3	0.3	1.6	1.7
4	0.3	1.9	1.9

Table 2.2: Table containing the symbols and notations used throughout this thesis

Symbol	**Meaning**
τ	A task set
n	Number of tasks in τ
m	Number of processors required to schedule a task set τ
ρ_j	The $(j+1)^{th}$ processor where $j = 0$ to $(m-1)$
T_i	The $(i+1)^{th}$ task of a task set τ where $i = 0, 1, ...(n-1)$
T_i^τ	The $(i+1)^{th}$ task of a task set τ where $i = 0, 1, ...(n-1)$
$J_{i,j}$	The $(j+1)^{th}$ job of a task T_i
p_i	The exact (minimum) time period between two consecutive jobs of periodic (sporadic) task T_i
e_i	The worst case execution time (WCET) of each job of task T_i
u_i	The average proportion of CPU time ($\frac{e_i}{p_i}$) that a task T_i of requires
u_{max}	The maximum value of utilization of tasks in τ $u_{max} = max\{u_0, u_1, ..., u_{n-1}\}$
U_{sum}	The maximum total utilization value of all tasks of τ: $U_{sum} = \sum u_i$
γ	Utilization binder such that $u_i = u_{max} \times \gamma^i$ for $i = 0, 1, ..., (n-1)$
$\Psi(u_{max}, \gamma)$	Set of all task sets τ such that $u_i \leq u_{max} \times \gamma^i \; \forall T_i \epsilon \tau$.
$\Gamma u_{max,\gamma}$	Maximal task set of $\Psi(u_{max}, \gamma)$ – i.e., $u_i = u_{max} \times \gamma^i$ for all $i \geq 0$
β_j	The minimum number of tasks on processor ρ_j when the first task is assigned to processor ρ_{i+1}
B_j	Number of tasks on processors $\rho_0, \rho_1, .., \rho_j$
m_L	Number of processors required by the method described by Lopez, et al.
β_j^τ	Number of tasks on processor ρ_j while scheduling tasks from τ
b_j	The j^{th} bin in the bin-packing problem
I_i	The i^{th} item in the bin-packing problem
s_j	The size of j^{th} bin in the bin-packing problem

CHAPTER 3

RELATED WORK

Preemptive multiprocessor scheduling algorithms of periodic or sporadic tasks can be divided into 3 basic categories according to their migration strategies [10].

- **Full Migration:** A preempted task can restart execution on any processor.
- **Restricted Migration:** Tasks can only migrate at job boundaries. If a job is preempted, it must restart on the same processor. However, the next job of the same task may execute on a different processor.
- **Partitioned:** Each task is assigned to a processor. All jobs of a given task can execute only on the processor to which the task is assigned.

This research is concerned with partitioned scheduling on identical multiprocessors, in which all processors are the same – i.e., a task's behavior (in particular its worst case execution time) does not depend on the processor that executes the task. For each of the task models, we will present two bounds – one bound determines the number of processors required, and another bound presents the maximum utilization associated with a given number of processors that guarantees partitioning is possible.

We assume that each processor executes tasks using the Earliest Deadline First (EDF) scheduling algorithm, which always schedules the job with the earliest deadline. Liu and Layland [2] proved that a task set can execute on a uniprocessor if its total utilization is at most 1. Dertouzos and Mok [11] proved the same bound holds for sporadic tasks.

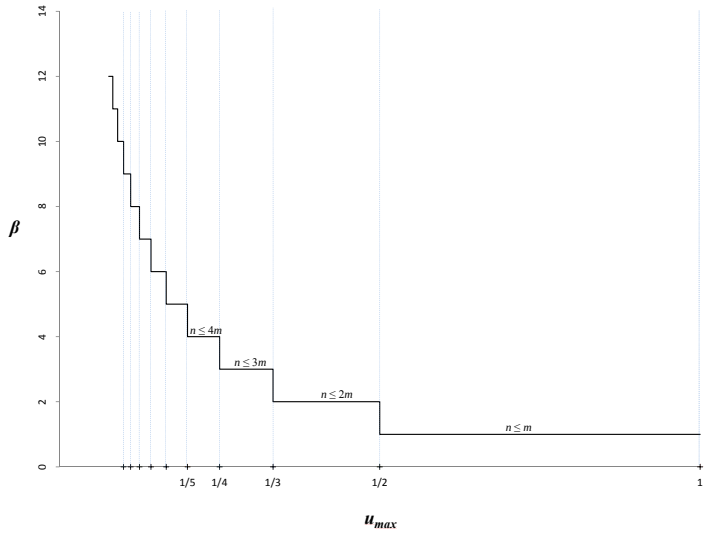
The question of whether a task set can be partitioned on m processors may be restated as follows: *Can we divide the tasks in τ into m subsets $\tau_1, \tau_2, \ldots, \tau_m$ such that each task is in exactly*

one subset and each subset has total utilization at most 1? This is, essentially, the bin packing problem, which considers whether a set of items $I_1, I_2, \ldots, I_n$ with sizes $s_1, s_2, \ldots, s_n$ can be placed into m equally sized bins without any bins overflowing (i.e., the total size of the items placed in each bin must be less than or equal to the size of the bin). Without loss of generality, all sizes are normalized so that the bins all have size 1 and the item sizes must be at most 1.

This problem was studied extensively by Johnson [12, 6], who proved that it is NP-complete. Fortunately, polynomial time approximate solutions to this problem exists. Johnson [7] proved that a number of simple strategies will require less than $\frac{11}{9} \times opt + 4$ where opt is the minimum number of bins required for the given set of items. An algorithm is online if each item is placed in a bin without considering any other items. One of the bin packing algorithms Johnson [12, 6] explored is the First Fit (FF) algorithm. This algorithm numbers the bins $b_1, b_2, \ldots$. Each item I_j is placed in the first bin b_i (i.e., the one with the lowest index) such that $1 - S(b_i) \leq s_j$, where $S(b_i)$ is the total size of all items that have already been assigned to bin b_i. Hence, we expect that any efficient method we devise to determine whether a task set can be partitioned onto m processors will give some "false negatives".

Lopez, et al., [1] considered a related problem in which the only information known about a task set τ is its total utilization, U_{sum}, and its maximum utilization u_{max}. Because there is no further information about the utilization values of the tasks in τ, Lopez, et al., must make the worst case assumption that all the tasks of τ have a utilization value of u_{max}. Their work observes that every processor must have at least $\beta = \lfloor 1/u_{max} \rfloor$ tasks assigned to it before adding any additional task would cause the total utilization on that processor to exceed 1. Hence, if β or fewer tasks have been assigned to some processor and the tasks are scheduled using EDF, then those tasks will meet all their deadlines. Moreover, if τ is comprised of $\beta \times m$ or fewer tasks, then we are guaranteed that τ can be partitioned onto m processors. (This conclusion relies on the results of Liu and Layland [2] and Deterzous and Mok [11] mentioned above.)

Figure 3.1 represents β as a function of u_{max}. When u_{max} is larger than $\frac{1}{2}$, we can only guarantee 1 task per processor. As the value of u_{max} increases, the minimum number of tasks per



(a)

Figure 3.1: Representation of β as a function of u_{max} [1]

processor increases. Each range also shows the maximum number of tasks. For example, if u_{max} is in the interval (1/3, 1/2] then β = 2. In this case, the task set τ can be partitioned if it has $2 \times m$ tasks or less.

Using this observation Lopez, et al., proved the following theorem.

Theorem 1. *Let $\tau = \{T_0, T_1 \ldots T_{n-1}\}$ be any task set with maximum utilization u_{max} and total utilization U_{sum}. Let $\beta = \left\lfloor \frac{1}{u_{max}} \right\rfloor$. If either of the following conditions hold*

$$U_{sum} \leq \frac{\beta m + 1}{\beta + 1}, or\ n \leq \beta \cdot m \tag{3.1}$$

then τ can be FFD partitioned onto m processors.

This theorem has the advantage of being quite flexible – as long as u_{max} and U_{sum} do not change, the tasks in τ can be modified without requiring further analysis. Lopez, et al., observed that for most processors ρ_j, when ρ_j is unable to fit some task, its total utilization may be only slightly larger than $\beta/(\beta + 1)$.

The graph in Figure 3.2 shows the normalized worst case utilization (average level of utilization per processor) for FFD partitioning as a function of the number of processors for different

values of β. The worst case achievable utilization is the maximum utilization guaranteeing the tasks can be partitioned onto the given number of processors. If $\beta = 1$, then on each processor there could be up to one task whose utilization may be slightly higher than $\frac{1}{2}$. Therefore, the lower bound on the maximum value of the utilization per processor of the task is $u_i = 0.5 + \epsilon$. However, this analysis assumes all of the task have utilization equal to $0.5 + \epsilon$ which is very pessimistic because the number of processors returned by Lopez method is nearly twice as much as actually needed and nearly all the processors are utilized only 50%

By EDF uniprocessor utilization bound, if there is just one processor, then it can be utilized to the fullest. Thus, while all the other processors have a utilization bound of $\frac{\beta}{\beta+1}$, we give the last processor a utilization bound of 1. If $\beta = \infty$, then utilization value of each task is infinitesimal value. Thus the gap on each processor is also infinitesimal – i.e., $gap(\rho_i) < \epsilon$ or $1 - U_{\rho_i} < \epsilon$. We see that in this case the maximum total utilization per processor is close to 1.

Lopez, et al., [1] also developed a function $m_L(n, U_{sum}, \beta)$ such that any task set τ with at most n tasks with total utilization at most U_{sum} and maximum utilization at most $1/\beta$ is guaranteed to be partitioned successfully onto $m_L(n, U_{sum}, \beta)$ processors where

$$m_L = min \left\{ \left\lceil \frac{n}{\beta} \right\rceil, \left\lceil \frac{(\beta+1)U_{sum} - 1}{\beta} \right\rceil \right\} \tag{3.2}$$

Example 3. *Let us see an example task set with 35 tasks. The first task with a utilization value of 0.6 and all the other tasks of utilization values 0.1. According to Lopez, et al., for such a task set* $\beta = 1$. *The total utilization* $U_{sum} = 4$ *and the number of tasks* $n = 35$. *Substituting the parameters for this task set in Equation 3.2 gives*

$$m_L = min \left\{ \left\lceil \frac{35}{1} \right\rceil \left\lceil \frac{(1+1)4 - 1}{1} \right\rceil \right\}$$

$$\Rrightarrow m_L = 7$$

Thus a designer using Lopez, et al., would require 7 processors.

Figure 3.3(a) illustrates the minimum utilization allocated to each of the 7 processors. Because $\beta = 1$, *nearly all the processors have an utilization bound of 50%. Figure 3.3(b) illustrates that tasks can fully utilize 4 processors.*

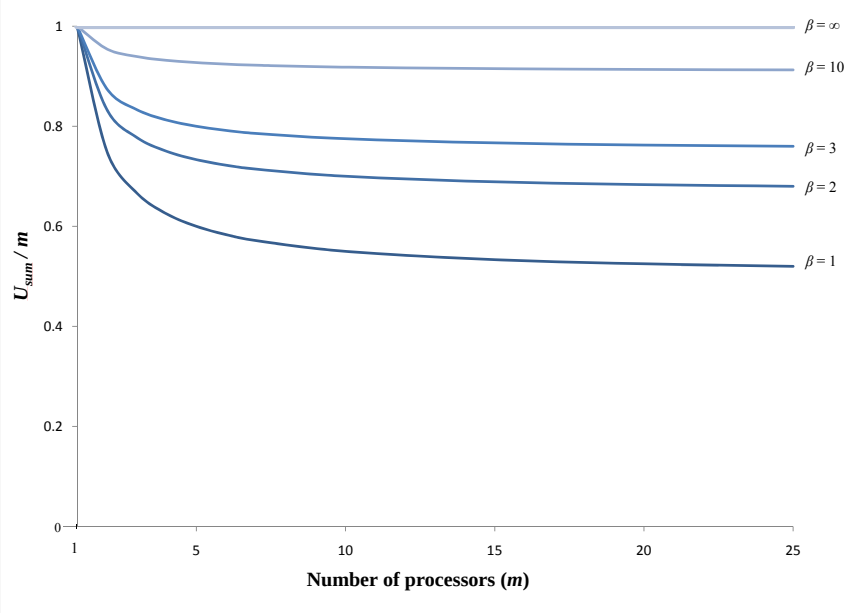
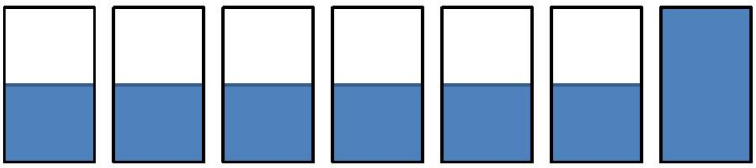
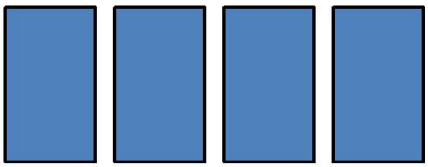


Figure 3.2: Worst-case achievable utilization for EDF-FF [1]



(a) Utilization bound for each processor using Lopez method



(b) Utilization bound for each processor using FFD

Figure 3.3: Utilization bound for each processor using Lopez method and FFD

Because the Lopez method considers only the maximum utilization they derive a very pessimistic bound. If they had taken into account the fact that nearly all the tasks have a much smaller utilization, they would be able to reduce the derived number of processors by almost 50%.

Our work is similar to that of Lopez, et al., but, instead of using U_{sum} and u_{max} to build our tests, we use u_{max} and γ or U_{cum}. When designing a system, there will generally be more information available than just u_{max} and U_{sum}. Methods to find the WCET of tasks, such as those

given in [8] and [9] illustrate that we can get better approximate descriptions of task sets using automatic code generators. Even though it can which can be used to find good approximations for the tasks' WCET, this code needs to be optimized. The code automatically generated using tools are less efficient than the actual code. In [8], Mohan, et al., propose a method to calculate the worst-case execution time for all of the tasks in a task set.

1. Devise a high-level, functional model with a modeling tool like Simulink [13].
2. Automatically generate code using Real-Time Workshop [14] or Real-Time Embedded Workshop [15].
3. Pass the generated code through an analysis suite developed by Mohan, et al.

Ferdinand, et al., [9], have also developed a detailed method to find the WCET of a task called USES approach. They tested their approach by analyzing the timing behavior of the Motorola ColdFire 5307 processor. Airbus provided the authors with a benchmark system which resembles actual avionics software. The USES estimates for the WCETs of this benchmark were used to design a system at the Airbus Toulouse plant. The initial assessment carried out by the verification specialists at Airbus found that USES reported WCETs that werevery close to the actual WCETs. The steps involved in the USES approach are:

1. The task for which WCET has to be determined is divided into sequential subtasks which can be executed in isolation. Doing this aids in a more precise estimation of the WCET of the task as different methods tailored to the subtask can be used in the analysis.
2. Then the cache analysis, pipeline analysis and value analysis were done using a semantics based method for static programming. Path analysis was done by integer linear programming.
3. While the benchmark was designed for the Motorola ColdFire 5307 processor, generic and generative methods were used so that the WCET can be estimated for any architecture.

The results of Mohan, et al., and of Ferdinand, et al., confirm that we can acquire more information about a task set than just u_{max} and U_{sum}. Though we cannot get the exact information ahead of time, we can get more detailed information about a task set τ. Using the available information, we model task sets more accurately than Lopez, et al. We describe a task set τ using:

1. The utilization binder, γ: For a task set τ, the utilization binder γ is a value such that $u_i \leq u_{max} \times \gamma^i$ for each task T_i when the tasks are sorted in the non-increasing order of their utilization values.

2. The cumulative utilization function, $U_{cum}(i)$: A function that returns an upper bound for cumulative value of utilization values of first the i tasks when the tasks are sorted in the non-increasing order of their utilization values.

The next two chapters illustrate how using this additional information can provide us with much better estimates of design requirements.

CHAPTER 4

USING γ TO IMPROVE THE BOUNDS

This paper presents a new method to find a tighter bound on the number of tasks required to partition the tasks of task set τ. We observed that the bound presented in Equation 3.1 can be very pessimistic if u_{max} is significantly larger than other task utilizations. While we may not have complete information, we will often have more information than just u_{max} and U_{sum}. To this end, we incorporate γ, a term to bound the utilization values of task set, as defined .

Recall $\Psi(u_{max}, \gamma)$ is a set of all task sets τ such that $u_i \leq u_{max} \times \gamma^i \; \forall T_i \epsilon \tau$ and $\Gamma_{u_{max},\gamma}$ is the "Maximal task set" of the set of task sets $\Psi(u_{max}, \gamma)$ i.e. $u_i = u_{max} \times \gamma^i$. For simplicity we use the notation Γ whenever the values of u_{max} and γ are clear. It may seem intuitive that our maximal task set $\Gamma_{u_{max},\gamma}$ will exhibit worst case behavior of any task set $\tau \in \Psi(u_{max}, \gamma)$. The next example illustrates that in some cases FFD may place more tasks of $\Gamma_{u_{max},\gamma}$ than of some $\tau \in \Psi(u_{max}, \gamma)$ onto same processor ρ_i

Example 4. *As stated in chapter 2, we use superscript to distinguish between τ's and Γ's behavior. For example β_j^τ indicates the number of tasks of τ assigned to processor ρ_j. consider a case where $\gamma = 0.75$ and $u_{max} = 0.85$. The tasks in the actual task set and the maximal task set can be as shown in the Table 4.1. Using First Fit Decreasing(FFD) algorithm, The task T_2^τ in the actual task set τ with utilization $u_2^\tau = 0.3$ can be scheduled on processor ρ_0^τ along with T_0^τ. The processor ρ_0^τ can thus have a maximum of two tasks using the actual task set which means $\beta_1^\tau = 2$. As the number of tasks increases and i increases, we are going to have tasks in maximal task set Γ which have really small utilizations like 0.003. The processor ρ_0^Γ will be able to schedule T_0^Γ, T_6^Γ and T_{10}^Γ which means $\beta_1^\Gamma = 3$*

Table 4.1: Example of an actual task set τ and its corresponding maximal task set Γ

Maximal task set Γ	**Actual task set** τ
0.85	0.7
0.6375	0.6
0.478	0.3
0.359	0.05
0.26	0.05
0.195	0.05
0.146	0.05
0.11	0.05
0.08	0.05
0.06	0.04
0.003	0.001

The above example shows that the number of tasks on ρ_i^Γ and ρ_i^τ for any $i \geq 0$, using task set $\tau \epsilon \Psi(u_{max}, \gamma)$ and maximal task set Γ respectively is incomparable when the scheduling is done by FFD. As noted in the discussion regarding Equation 3.1 and illustrated in Figures 3.2 and 3.1, the calculation of utilization bound depends on the number of tasks on a processor ρ_i. In order to ensure an accurate utilization bound, we need to assume $m_i^\tau \geq m_i^\Gamma$, where m_i^τ and m_i^Γ are the number of tasks on the processor ρ_i using the task sets τ and Γ. Hence we use Next Fit Decreasing (NFD) to partition Γ when finding the FFD utilization bound for τ.

Below, we show how to use u_{max} and γ to find a bound on the number of processors required by any $\tau \in \Psi(u_{max}, \gamma)$ and a bound on the total utilization for any such τ, beginning with the processor bound.

4.0.1 Bounding the required number of processors

We begin by presenting a method for finding a bound on the number of processors $m(u_{max}, \gamma, n)$. Any task set $\tau \in \Psi(u_{max}, \gamma)$ can be partitioned onto $m(u_{max}, \gamma, n)$ processors using the First Fit Decreasing (FFD) algorithm. If $\gamma = 1$ then the number of tasks, n must be provided. If $\gamma < 1$, this parameter is optional. We can still find an upper bound on the number of processors by observing

that when $\gamma < 1$,

$$\sum_{i=0}^{\infty} u_{max}\gamma^i = \frac{u_{max}}{1-\gamma} < \infty.$$

Thus, we find $m(u_{max}, \gamma, \infty)$ if n is not known. However, even when $\gamma < 1$, specifying n will allow us to provide a tighter bound on the required number of processors. We show that incorporating γ into the analysis can only improve the bounds – i.e, the bounds developed below cannot be worse than the corresponding bounds presented by Lopez, et al . [1].

We use an iterative method to find $m(u_{max}, \gamma, n)$. Given u_{max} and γ we find the value β_i such that β_i tasks of $\Gamma_(u_{max}, \gamma)$ can fit onto processor ρ_i, but $\beta_i + 1$ tasks cannot. This value varies by processor because the largest task utilization assigned to each processor decreases as the number of processors increases if $\gamma < 1$, and remains constant if $\gamma = 1$.

Lemma 1. *Let the task set $\Gamma_{u_{max},\gamma}$ be the maximal task set for some u_{max} and γ such that $0 < u_{max}, \gamma < 1$. Then*

a) If $u_{max} \leq 1 - \gamma$ then let $\beta_0 = \infty$.

b) If $u_{max} > 1 - \gamma$, let β_0 be defined as follows

$$\beta_0 = \left\lfloor \log_\gamma \left(1 - \frac{1-\gamma}{u_{max}}\right) \right\rfloor. \tag{4.1}$$

Then for all $0 \leq i < \beta_0$, the NFD algorithm will assign task $T_{i,\Gamma}$ to processor ρ_0. Furthermore, when $u_{max} > 1 - \gamma$ task $T_{\beta_0,\Gamma}$ will be assigned to processor ρ_1.

Proof. To prove (*a*), note that

$$\sum_{i=1}^{\infty} u_{max}\gamma^i = \frac{u_{max}}{1-\gamma}.$$

Therefore, if $u_{max} \leq 1 - \gamma$ implies the total utilization of all the tasks in $\Gamma_{u_{max},\gamma}$ is not more than 1. Hence, all the tasks can fit onto a single processor.

To prove (*b*), we observe that if $T_{\beta_0,\Gamma}$ is the first task to be assigned to the second processor then β_0 satisfies the following inequalities.

$$\sum_{i=0}^{\beta_0-1} u_{max}\gamma^i \leq 1 < \sum_{i=0}^{\beta_0} u_{max}\gamma^i.$$

Equivalently,

$$\frac{u_{max}(1-\gamma^{\beta_0})}{1-\gamma} \leq 1 < \frac{u_{max}(1-\gamma^{\beta_0+1})}{1-\gamma}.$$

Multiplying by $(1-\gamma)/u_{max}$ gives

$$1-\gamma^{\beta_0} \leq \frac{1-\gamma}{u_{max}} < 1-\gamma^{\beta_0+1}.$$

Isolating the γ terms gives

$$\gamma^{\beta_0} \geq 1-\frac{1-\gamma}{u_{max}} > \gamma^{\beta_0+1}.$$

Taking the log base γ gives

$$\beta_0 \leq \log_\gamma\left(1-\frac{1-\gamma}{u_{max}}\right) < \beta_0+1.$$

Thus, Equation 4.1 holds because $\beta_0 \in \mathbb{Z}$. □

Note that once the first β_0 tasks have been assigned to processor ρ_0, we know that the tasks $T_{0,\Gamma}$ through $T_{\beta_0-1,\Gamma}$ have already been assigned to a processor. Hence, the problem of determining how many processors the *remaining* tasks require can use the same approach using a smaller value for u_{max}, namely, $u_{max} \cdot \gamma^{\beta_0}$. Thus, we can find β_1 using Equation 4.1, but replacing u_{max} with $u_{max} \cdot \gamma^{\beta_0}$, viz.

$$\beta_1 = \left\lfloor \log_\gamma\left(1-\frac{1-\gamma}{u_{max}\gamma^{\beta_0}}\right)\right\rfloor. \tag{4.2}$$

After assigning tasks to the first 2 processors, we have accounted for $\beta_0+\beta_1$ tasks. Hence, we can find the number of tasks that can fit on processor ρ_2 using Equation 4.1 replacing u_{max} with $u_{max} \cdot \gamma^{\beta_0+\beta_1}$. More generally, for $i \geq 1$, if B_{i-1} tasks are assigned to processors ρ_0 through ρ_{i-1} and

$$\beta_i = \left\lfloor \log_\gamma\left(1-\frac{1-\gamma}{u_{max} \cdot \gamma^{B_{i-1}}}\right)\right\rfloor \tag{4.3}$$

we assign tasks $T_{B_{i-1},\Gamma}$ through $T_{B_{i-1}+\beta_i-1,\Gamma}$ to processor ρ_i, thereby letting $B_i = B_{i-1}+\beta_i$.

The value β_0 is used in a similar manner as the value β in [1]. In that work, Lopez, et al., set β to $\lfloor 1/u_{max} \rfloor$. Assuming all tasks have utilization u_{max}, β is the number of tasks that can fit onto a single processor before the next task won't fit. Thus, the value of β used by Lopez, et al. provides

Algorithm 1 Find-$m(u_{max}, \gamma, n)$

Require: $0 < u_{max}, \gamma \leq 1$ and $1 \leq n \leq \infty$. If $\gamma = 1$, then n must be finite.

if $\gamma < 1$ **then**
  $U_{sum} \leftarrow \frac{u_{max}}{1-\gamma}$
else
  $U_{sum} \leftarrow \infty$
$U_{rem} \leftarrow U_{sum}$
$m \leftarrow B_0 \leftarrow 0$
while $U_{rem} > 1$ and $B < n$ **do**
  $m \leftarrow m + 1$
  $B_m \leftarrow B_m + \beta_m$
  if $\gamma < 1$ **then**
    $\beta_m \leftarrow \lceil \log_\gamma(1 - (1-\gamma)/(u_{max}\gamma^B)) \rceil$
    $U_{rem} \leftarrow U_{rem} - (u_{max}\gamma^\beta)/(1-\gamma)$
  else
    $\beta \leftarrow \lfloor 1/u_{max} \rfloor$
    $U_{rem} \leftarrow U_{rem} - \beta/(\beta+1)$
if $B_m, < n$ **then**
  $m \leftarrow m + 1$
return m

the same measurement as the β_0 using our method. The use of γ, however, allows us to put more tasks onto the *next* processor. Furthermore, it is easy to see that β_0 cannot be smaller than β. Once all tasks T_i^Γ with utilization u_i have been assigned to processors, the number of tasks per processor increases.

Algorithm FIND-m (Algorithm 1) illustrates our iterative strategy to find $m(u_{max}, \gamma, n)$. The algorithm iteratively assigns β_i tasks to each processor. On each iteration, the value of β is determined using a smaller value for u_{max}.

The algorithm continues to iterate until the remaining utilization is less than 1 or the number of tasks B has exceeded the input n. If the remaining utilization is less than or equal to 1, all the remaining tasks can fit onto a single processor, so the algorithm increments m and returns. If $B \geq n$, then all tasks have been assigned to processors, so the algorithm returns.

While Algorithm FIND-m finds the number of processors required for $\Gamma u_{max},\gamma$ we need to assure ourselves that any τ in $\Psi(u_{max}, \gamma)$ will use no more than the number of processors returned

by this algorithm when partitioned using FFD. We begin by showing that for each processor ρ_j, $\Gamma u_{max,\gamma}$ allocates its first task no later than τ does.

Lemma 2. *Let τ be any task set in $\Psi(u_{max}, \gamma)$. Assume T_{ω_j} is the first task assigned to processor ρ_i when τ is partitioned using FFD. Let B_j be the value of B (i.e., the number tasks assigned to processors $\rho_0, \ldots rho_{j-1}$) when Algorithm 1 enters the loop for the j^{th} time. Then $\omega_j \geq B_j$.*

Proof. The proof is by induction on the number of processors. Both τ and $\Gamma u_{max,\gamma}$ place their first task on processor ρ_0. Hence $\omega_0 \geq \Omega_0$.

Assume $\omega_j \geq B_j$ for all $j \leq k$ and consider processor ρ_{k+1}. Because $\omega_k \geq B_k$, we know that $u_{\omega_k} \leq u_{B_k} \leq u_{B_k,\Gamma}$. Moreover, we know that $u_{\omega_k+i} \leq u_{B_k+i,\Gamma}$ for all $i \geq 0$. Algorithm 1 determines β_k, the number of tasks starting with $T_{B_k,\Gamma}$ that can be assigned to processor ρ_k and then moves on to processor ρ_{k+1}. Because the β_k tasks of $\Gamma u_{max,\gamma}$ starting with $T_{B_k,\Gamma}$ have total utilization at most 1 and $u_{\omega_k+x} \leq u_{B_k,\Gamma+x}$ for $0 \leq x \leq \beta_k$, we know that the total utilization of the β_k tasks of τ starting with T_{ω_k} cannot exceed 1. While we do not know which processors these tasks will be assigned to, we can conclude that NFD will not need to start a new processor before task $T_{\omega_k+\beta_k}$. □

Theorem 2. *Let τ be any task set in $\Psi(u_{max}, \gamma)$ and let $m(u_{max}, \gamma, n)$ be the value returned by Algorithm 1. Then when τ is partitioned using FFD, no more than $m(u_{max}, \gamma, n)$ processors will be required.*

Proof. By Lemma 2, the first task of τ assigned to any processor using FFD has an index no smaller than the first task of $\Gamma u_{max,\gamma}$ assigned to the same processor by Algorithm 1. Therefore, if $n < \infty$, the Theorem must hold. If $n = \infty$ then Algorithm 1 assigns tasks whose total utilization is $\sum_{i=0}^{\infty} u_{max}\gamma^i$. Hence, any number of tasks in τ will have corresponding tasks accounted for in the derivation of $m(u_{max}, \gamma, n)$. Thus, τ will be successfully partitioned onto these $m(u_{max}, \gamma, n)$ processors and any task set $\tau \epsilon \Psi(u_{max}, \gamma)$ will use at most $m(u_{max}, \gamma, n)$ processors when partitioned using FFD. □

Algorithm 2 Find-$U_{bound}(u_{max}, \gamma, m_{max})$

Require: $0 < u_{max}, \gamma \leq 1$ and $1 \leq m \leq \infty$. If $\gamma = 1$, then n must be finite.

if $\gamma < 1$ **then**
  if $m_{max} > u_{max}/(1-\gamma)$ **then**
    return $u_{max}/(1-\gamma)$
$B \leftarrow U_b \leftarrow 0$
$m \leftarrow 1$
while $m < m_{max}$ **do**
  if $\gamma < 1$ **then**
    $\beta \leftarrow \lceil \log_\gamma(1-(1-\gamma)/(u_{max}\gamma^B)) \rceil$
    $U_{bound} \leftarrow U_{bound} + (1-\gamma^\beta)/(1-\gamma^{\beta+1})$
  else
    $\beta \leftarrow \lfloor 1/u_{max} \rfloor$
    $U_{bound} \leftarrow U_{bound} + \beta/(\beta+1)$
  $m \leftarrow m+1$
  $B \leftarrow B + \beta$
$U_{bound} \leftarrow U_{bound} + 1$
return U_{bound}

4.0.2 BOUNDING THE TOTAL UTILIZATION

We now present a method for finding a utilization bound $U_{bound}(u_{max}, \gamma, m)$. Any task set $\tau \in \Psi(u_{max}, \gamma)$ with total utilization $U_{sum} \leq U_{bound}$, then τ is guaranteed to be partitioned onto m processors using FFD.

Note that because $\Gamma_{u_{max},\gamma}$ is the *maximal* task set in $\Psi(u_{max}, \gamma)$, its utilization does not provide a lower bound on the utilization of $\tau \in \Psi(u_{max}, \gamma)$. Consider the following example.

While $\Gamma_{u_{max},\gamma}$'s utilization does not provide us with the information we need, the values of β_i can be used to derive a utilization bound. Lemma 1 above observed that β_i is the minimum number of tasks NFD assigns to ρ_i for any $\tau \in \Psi(u_{max}, \gamma)$. We use this observation to find a bound on the total utilization of any such τ. Specifically, we want to determine what the smallest utilization can be if β_i tasks are assigned to a processor and some task cannot fit into the remaining gap.

Theorem 3. *Let τ be any task set in $\Psi(u_{max}, \gamma)$. Assume FFD has assigned β_j tasks to processor ρ_j. Let U_{ρ_j} be the total utilization of these tasks. If the next task $T_i \in \tau$ that FFD attempts to assign to ρ_j that will not fit then*

a) If $\gamma = 1$ *then*

$$U_{\rho_j} > \frac{\beta_j}{\beta_j + 1}.$$

b) If $\gamma < 1$ *then*

$$U_{\rho_j} > \frac{1 - \gamma^{\beta_j}}{1 - \gamma^{\beta_j + 1}}.$$

Proof. Let T_i be the next task that FFD attempts to assign to ρ_j. Observe that the worst case occurs when ρ_j has a gap only slightly smaller than the task's utilization, viz. $U_{\rho_j} = 1 - u_i + \epsilon$. Hence, our goal is to find the smallest value of U_{ρ_j} such that $U_{\rho_j} = 1 - u_i$ for some task u_i. Because we assume FFD partitioning, we know $u_i \leq u_k$ for all T_k assigned to ρ_j. Let $T_{k_1}, \ldots, T_{k_{\beta_j}}$ be the tasks assigned to ρ_j and assume that U_{ρ_j} is as small as it can be while still having some unassigned task T_i satisfy $u_i = 1 - U_{\rho_j}$. We consider the two cases separately.

Case 1: $\gamma = 1$.

We wish to minimize $U_{\rho_j} = \sum_{h=1}^{\beta_j} u_{k_h}$ with the restriction that $u_{k_1} \geq u_{k_2} \geq \ldots u_{k_{\beta_j}} \geq u_i$ and $1 - U_{\rho_j} > u_i$. Because $1 - U_{\rho_j} > u_i$, the value of U_{ρ_j} is minimized when the value at u_i is maximized i.e. when $u_i = u_{k_{\beta_j}}$. Also because $U_{\rho_j} = \sum_{h=1}^{\beta_j} u_{k_h}$, the value of U_{ρ_j} is minimized when the values of u_{k_h} are minimized - i.e. when $u_{k_h} = U_{k_{h+1}}$ for $h = 1, 2, ...\beta_j - 1$. Thus U_{ρ_j} is minimized when $u_{k_1} = u_{k_2} = ... = u_{k_{\beta_j}} = u_i$ Thus,

$$U_{\rho_j} = \sum_{h=1}^{\beta_j} u_{k_h} = \beta_j \times u_i \tag{4.4}$$

also $u_i > 1 - U_{\rho_j} = 1 - \beta_j \times u_i$ Therefore,

$$u_i > \frac{1}{\beta_j + 1} \tag{4.5}$$

Plugging Equation- 4.5 into Equation- 4.4 gives $U_{\rho_j} > \frac{\beta_j}{\beta_j + 1}$

Case 2: $\gamma < 1$.

We wish to minimize $U_{\rho_j} = \sum_{h=1}^{\beta_j} u_{k_h}$ with the restriction that $u_{k_1} \geq u_{k_2} \geq \ldots u_{k_{\beta_j}} \geq u_i$ where $u_{k_x} \leq u_{k_1} \times \gamma$ and $1 - U_{\rho_j} > u_i$. Because $1 - U_{\rho_j} > u_i$, the value of U_{ρ_j} is minimized when the value at u_i is maximized i.e. when $u_i = u_{k_{\beta_j}} \times \gamma$. Also because $U_{\rho_j} = \sum_{h=1}^{\beta_j} u_{k_h} =$

$\sum_{h=1}^{\beta_j} u_{k_1} \times \gamma^{h-1}$, the value of U_{ρ_j} is minimized when the values of u_{k_h} are minimized - i.e. when $u_{k_h} \times \gamma = U_{k_{h+1}}$ for $h = 1, 2, ...\beta_j - 1$. Thus U_{ρ_j} is minimized when $u_{k_1} \times \gamma = u_{k_2}$, $u_{k_2} \times \gamma = u_{k_2}, ..., u_{k_{\beta_j}} \times \gamma = u_i$ Thus,

$$U_{\rho_j} = \sum_{h=1}^{\beta_j} u_{k_1} \gamma^{h-1} = u_{k_1} \times \gamma_j^{\beta} \tag{4.6}$$

$$\implies 1 - \frac{u_{k_1}(1 - \gamma^{\beta_j})}{1 - \gamma} = u_{k_1} \gamma^{\beta_j}$$

$$\Rightarrow 1 - \frac{u_{k_1}(1 - \gamma^{\beta_j})}{1 - \gamma} = u_{k_1} \gamma^{\beta_j}$$

$$\Rightarrow u_{k_1} \left(\frac{1 - \gamma^{\beta_j}}{1 - \gamma} + \gamma^{\beta_j} \right) = 1$$

$$\Rightarrow u_{k_1} = \frac{1 - \gamma}{1 - \gamma^{\beta_j + 1}}$$

Therefore,

$$\begin{aligned} U_{\rho_j} &= \sum_{h=1}^{\beta_j} u_{k_1} \gamma^h \\ &= \frac{1 - \gamma}{1 - \gamma^{\beta_j + 1}} \cdot \frac{1 - \gamma^{\beta_j}}{1 - \gamma} \end{aligned}$$

This proves the second case. □

Algorithm FIND-U_{bound} (Algorithm 2) illustrates our iterative method for finding the utilization bound for any $\tau \in \Psi(u_{max}, \gamma)$. We initially check if $m > u_{max}/(1-\gamma)$. If so, that is the utilization bound by Lemma 1. Otherwise, we iteratively find how many tasks $\Gamma u_{max,\gamma}$ would place on the next processor and adjust U_{bound} accordingly. We iterate $m-1$ time and add 1 to our bound because the final processor, being a uniprocessor, has a utilization bound of 1.

Note that the utilization bound presented by Lopez, et al., follows directly from the above result, as the following corollary demonstrates.[1]

[1]The proof of Corollary 1 below is an alternate and (much simpler) proof of the bound presented by Lopez, et al.

Corollary 1. *Let τ be any task set with maximum task utilization equal to u_{max}. If $\gamma = 1$ or if γ is unknown, then the utilization bound for m processors determined by Theorem 3 is*

$$\frac{\beta \cdot m + 1}{\beta + 1},$$

where $\beta = \lfloor 1/u_{max} \rfloor$.

Proof. Because we must assume the worst case scenario, we assume the maximum utilization on each processor is u_{max}. Therefore Algorithm 2 will assign total utilization bound of $\frac{\beta}{\beta+1}$ to the first $m - 1$ processors. The last processor is assigned the uniprocessor utilization bound of 1. Hence, the derived bound will be

$$\begin{aligned} & (m-1) \cdot \frac{\beta}{\beta+1} + 1 \\ = \; & \frac{(m-1)\beta + \beta + 1}{\beta} \\ = \; & \frac{\beta \cdot m + 1}{\beta}. \end{aligned}$$

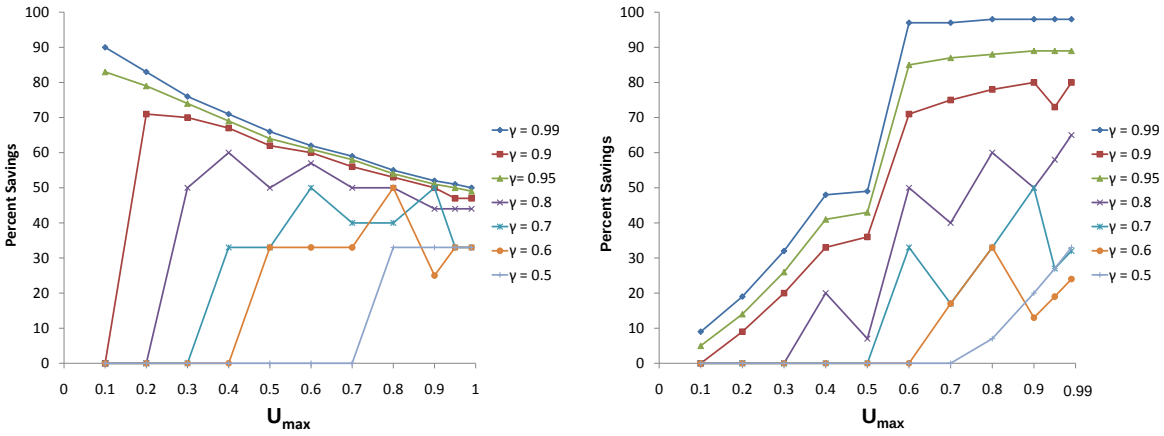
□

The results presented in this chapter can aid designers in building a system when each task's utilization can be bounded. the next chapter relaxes the requirement that each individual task's utilization can be bounded and instead illustrates that similar analysis can still be applied when we are provided with a function that bounds the cumulative utilization of the tasks.

4.0.3 EVALUATION

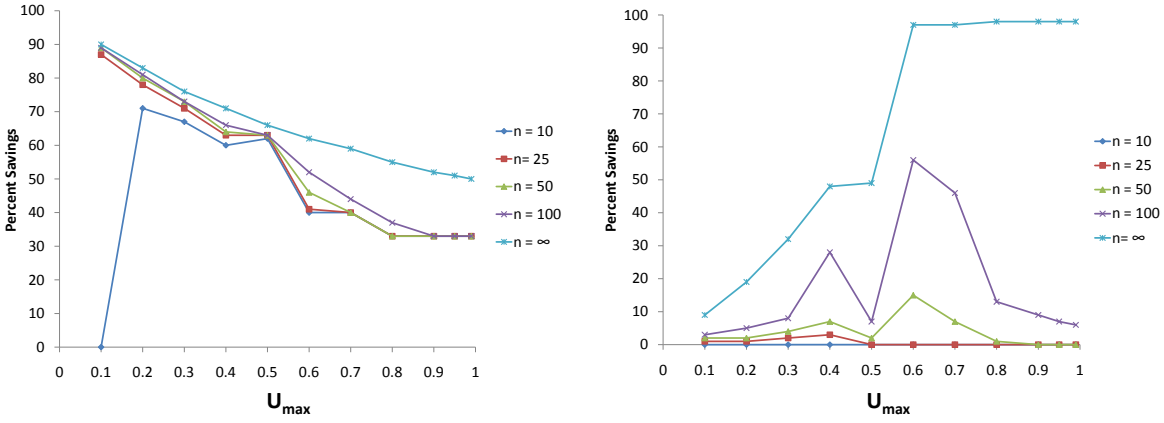
In this section, we compare the bounds determined by our work using the utilization binder, γ, to the corresponding bounds developed by Lopez, et al., [1]. We calculated $m(u_{max}, \gamma, \infty)$ and $U_{bound}(u_{max}, \gamma, m)$ for values of u_{max} and γ ranging from 0.1 to 0.99.

We also determine the utilization bound and the number of processors developed by Lopez, et al., [1]. Our algorithm runs for O(m) while Lopez, et al., runs for O(1). Assume ${u_{max}}^{\gamma}$ denotes the maximum utilization used in our method and ${u_{max}}^{L}$ is the maximum utilization used in the



(a) Reduction in Processors. (b) Increase in U_{bound}.

Figure 4.1: Our results compared to Lopez, et al., bounds.

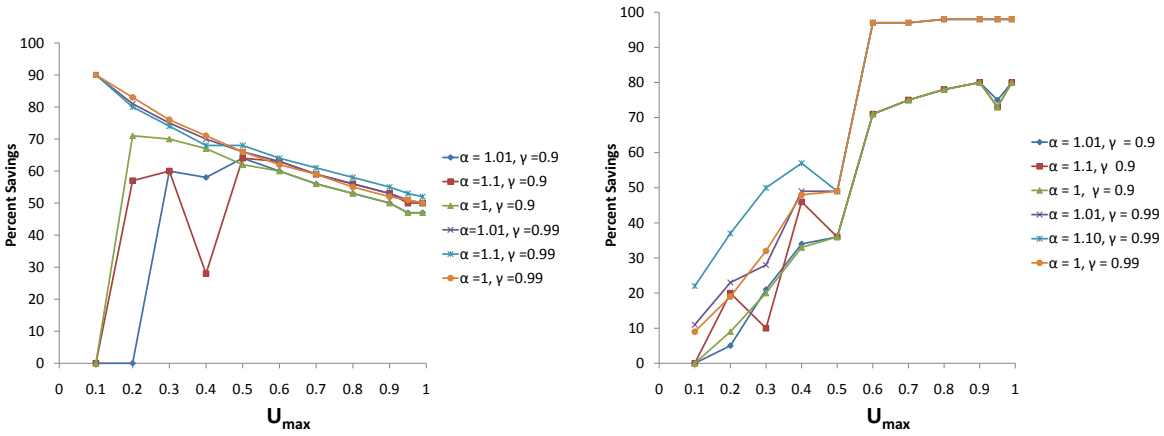


(a) Reduction in Processors. (b) Increase in U_{bound}.

Figure 4.2: Our results compared to Lopez, et al., for fixed number of tasks

Lopez, et al., method. In practical situations where you cannot approximate the task utilization values correctly, increase ${u_{max}}^{\gamma}$ value slightly to bound the utilization values properly. In cases where you want to save more increase ${u_{max}}^{\gamma}$ slightly and decrease γ a little in such a way that utilization calculated using ${u_{max}}^{\gamma}$ and γ still bounds the actual utilization. We will still be able to have tasks that are tighter (i.e. smaller utilization) than the ones Lopez, et al., implicitly uses.

Figure 4.1, Figure 4.2 and Figure 4.3 present the resulting savings. The values of u_{max} and γ presented are 0.6, 0.7, 0.8, 0.9, 0.95 and 0.99. The graphs demonstrate that, even for very large values of u_{max} and γ, our iterative method can significantly reduce the required number of processors. That is, even if the tasks have utilizations nearly equal to each other and γ is almost equal to



(a) Reduction in Processors. (b) Increase in U_{bound}.

Figure 4.3: Our results compared to Lopez, et al., for fixed values of alpha and gamma

1, our method reduces the number of processors up to 65% and increases the utilization bound up to 98%. Thus, even if the approximate value of γ is significantly larger than an actual task set's γ value (e.g., 0.99), we can still reduce the number of processors and increase the utilization quite a bit.

Figure 4.1(a) shows the maximum reduction in the number of processors we get using our method. For each u_{max} and γ we found the maximum percent savings determined as follows.

$$\text{Percent savings} = \frac{m_L - m_\gamma}{m_L},$$

where m_L is the number of processors indicated by Lopez, et al., and m_γ is the number of processors derived by our algorithm. Similarly, the graph in Figure 4.1(b) was drawn with percent savings of the utilization bound determined as follows.

$$\text{Percent savings} = \frac{U_{bound,\gamma} - U_{bound,L}}{U_{bound,L}}.$$

We observe that we achieve savings in many scenarios. In general, our utilization bound fares better compared to Lopez as the number of tasks increases and as the maximum utilization decreases (though for very small utilizations we our savings decline). By contrast, the reduction in processors improves as γ increase and as u_{max} *increase*. For small values of u_{max} and γ, we see that the savings can be fairly eratic. This occurs because the required number of processors is small and the total utilization assigned to the last processor can have a big impact.

For example, when $u_{max} = 0.8$ and $\gamma = 0.6$, our method requires 2 processors and the utilization bound for those processors is 2 – i.e., the 2 processors can be 100% utilized without having to put any tasks on a 3^{rd} processor. The Lopez utilization bound for $u_{max} = 0.8$ and $m = 2$ is 1.5. When $u_{max} = 0.9$ and $u_{max} = 0.6$, our method needs 3 processors and the utilization bound is 2.25, so the system idles 25% of the time. This idle time diminishes the savings. The Lopez utilization bound for $u_{max} = 0.9$ and $m = 3$ is 2.

Figure 4.2(a) shows the percentage savings and Figure 4.2(b) calculated with fixed number of tasks (10, 25, 50, ∞). In Figure 4.2(a) and Figure 4.2(b), the results for 10 tasks and 25 tasks completely overlap and result for 50 tasks partially overlaps with results for 10 and 25 tasks.

Using a small value for γ means the task utilization depreciates quickly. This is desirable as the number of processors required to schedule tasks with small utilization is lesser than or equal to schedule the same number of tasks of relatively large utilization values. To be able to use a smaller γ value than what is needed, u_{max} can be slightly increased. Let α is the factor by which we want to decrease γ. We have also experimented with ${u_{max}}^{\gamma} = \alpha {u_{max}}^{L}$ where $\alpha = 1, 1.01, 1.10$. α is restricted to small values because bigger values of α may dramatically increase ${u_{max}}^{\gamma}$ or even make it greater than 1. Figure 4.3(a) shows the reduction in number of processors we get using ${u_{max}}^{\gamma}$ in our method compared to using ${u_{max}}^{L}$ in Lopez, et al., method. Figure 4.3(b) shows the percentage increase in utilization bound calculated in a similar way.

In Figure 4.3(a), though the values of α are different, the percentage savings for same γ overlap significantly with each other. In Figure 4.3(b), the values for $\alpha = 1.01$ & $\gamma = 0.9$ and $\alpha = 1$ & $\gamma = 0.9$ overlap. the rest of the lines in Figure 4.3(b) except $\alpha = 1.1$ & $\gamma = 0.99$ also overlap with each other.

All the graphs exhibit non-uniformity in growth. It curves up and down for different values of gamma depending on u_{max}. The irregularity in both the cases has been explained below. Erratic behavior is exhibited in scenarios requiring a small number of processors. Therefore, a small change in m_L that does not also occur in m_Γ will appear to be a very large change in percentage reduction. For instance consider 4.1(a), for the same value of $\gamma = 0.8$ and infinite number of tasks,

the savings is 57% when $\gamma = 0.6$ while savings reduce to 50% when $u_{max} = 0.7$. For $u_{max} = 0.6$ and $\gamma = 0.8$, the number of processors required required using the method suggested by Lopez, et al., is 7 but our method requires only 3. When u_{max} increases to 0.7, both the method requires an additional processor which makes Lopez, et al., method use 8 processors and ours to use 4 processors. This brings down the savings from 57% to 50%. Similar reasoning applies to erratic points in the improvement of the utilization bound.

The value of γ does not have to be small to benefit from our method. Conversely, when γ is high and close to 1, we get better savings. As the number of tasks increases, the number of processors required by both our method and Lopez, et al., method increases. This in turn increases the gap on each processor. As the measured gap increases, more tasks can be scheduled on the $gaps$. Since our method keeps an account of gap on each processor, the savings on the number of processors increases with the increase in γ. Which means, even when the γ value cannot be estimated correctly, using $\gamma = 0.99$ can result in lot of savings.

CHAPTER 5

USING CUMULATIVE UTILIZATION FUNCTION

We now present another method of developing utilization bounds and determining a number of processors that guarantees a task set can be partitioned using FFD. In this chapter, we bound the *cumulative* utilization values rather than the individual task's utilization values. The tasks of τ are sorted in the non-decreasing order according to their utilization values. A task τ can be described by the cumulative utilization function $U_{cum}(.)$ if, for each $i = 1, 2, \ldots, n$, the value of $U_{cum}(i)$ is greater than or equal to the total utilization the i tasks in τ with the largest utilization values.

As in the previous chapter, our analysis of relies on determining the minimum number of tasks that can be assigned to each processor. Because EDF's utilization bound on a single processor is 1, when partitioning using FFD we will assign the first i tasks to processor ρ_0 as long as $U_{cum}(i) \leq 1$. Hence, we let

$$\beta_0 = \lfloor {U_{cum}}^{-1}(1) \rfloor.$$

Our first inclination is to determine the number of tasks assigned to ρ_1 by considering the values of $U_{cum}(.)$ after the first β_0 tasks. Specifically, we might want to approach this recursively, letting

$$U_{cum,1}(i) = U_{cum}(i) - U_{cum}(\beta_0),$$

and using $U_{cum,1}$ to find β_1 accordingly, viz.

$$\beta_1 = \lfloor U_{cum,1}^{-1}(1) \rfloor.$$

This approach is illustrated in Figure 5.1.

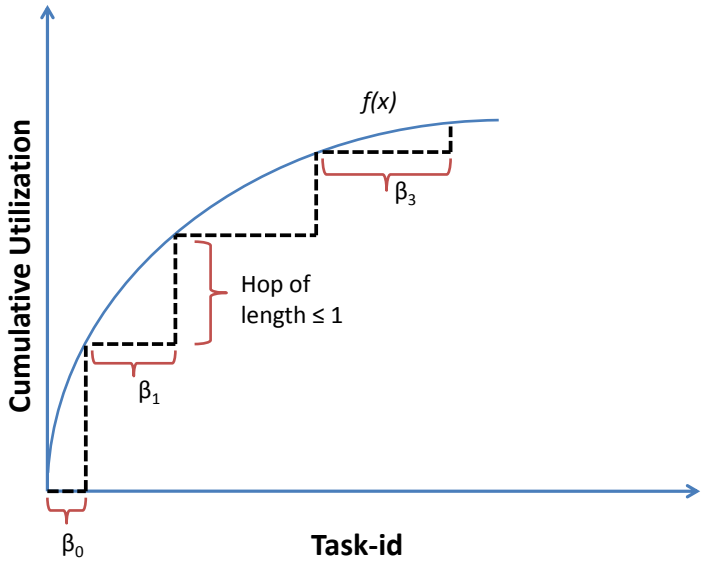


Figure 5.1: Hop over the curve to find the number of processors

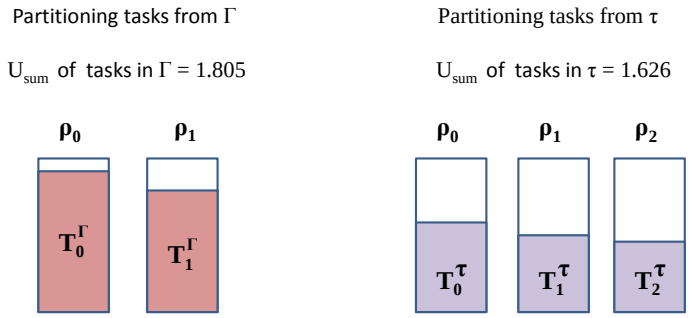
Unfortunately, our intuition leads us astray in developing this approach. As the following example illustrates, using the analysis described above may cause us to underestimate the number of processors a given task set may require.

Example 5. *Consider a function $U_{cum}(i)$ whose first 5 values are 0.5, 1.0, 1.4, 1.8 and 2.0. According to the method described above, we would conclude that a task described by $U_{cum}(i)$ would require only 2 processors.*

Now consider the task set containing 5 tasks, all of which have a utilization of 0.4. This task set is correctly described by $U_{cum}(i)$ because $U_{cum}(i) \geq 0.4 \times i$ for $1 \leq i \leq 5$. Even so, partitioning the task set requires 3 processors instead of just 2. This discrepancy occurs because $U_{cum}(i)$ bounds the cumulative utilization values of the tasks at every instance but not the individual utilization values of the tasks.

Example 6. *Let $u_{max} = 0.95$ and $\gamma = 0.9$, then the total utilization of $\Gamma_{u_{max},\gamma}$ on 2 processors would be 0.95 + 0.855 = 1.805. However, if τ tau has 3 tasks with utilization .6, .54, and .486, then τ cannot be partitioned onto 2 processors even though τ's total utilization is $1.626 < 1.805$ as shown in Figure 5.2.*

Utilization values of tasks in Γ	Utilization values of tasks in τ
0.95	0.6
0.855	0.54
	0.486



(a)

Figure 5.2: Number of processors required does not depend on the total utilization

This example indicates that our approach needs a slight change to accommodate such cases. Rather than starting our next "hop" from the point where the last hop ended, we must account for the *worst-case* behavior of any task set described by U_{cum} and start our hop from that point. By Theorem 3 in Chapter 4, we know that if β_j tasks are assigned to processor ρ_j when some task does not fit onto the processor, then the total utilization of tasks assigned to that processor must be at least $\beta_j/(\beta_j + 1)$. We use this expression to determine a bound on the total utilization of tasks assigned to a processor ρ_j.

Theorem 4. *Let U_{cum} be a cumulative utilization function. Assume that for any task set τ described by U_{cum}, when a task is assigned to processor ρ_j, there must be at least B_{j-1} tasks assigned to processors ρ_0 through ρ_{j-1} and the minimum total utilization of these tasks is at least $mintot$. Let $U_{cum,j}(i) = U_{cum}(i) - mintot$. Then at least*

$$B_j = \lfloor U_{cum,j}^{-1}(1) \rfloor$$

tasks of τ will be assigned to processors ρ_0 through ρ_j before the first task is assigned to processor ρ_{j+1}.

Now let $\beta_j = B_j - B_{j-1}$. When the first task of τ is assigned to processor ρ_{j+1}, there will be at least β_j assigned to ρ_j and the total utilization of tasks assigned to processors ρ_0 through ρ_j must be at least

$$mintot + \frac{\beta_j}{\beta_j + 1}$$

.

Proof: The proof is by induction. Because no tasks can have been assigned to any processor before the first task is assigned to ρ_0, we start with $U_{cum,0}(i) = U_{cum}(i)$ and $mintot = 0$. Let $\beta_0 = \lfloor {U_{cum}}^{-1}(1) \rfloor$. By definition of U_{cum}, for any task τ described by U_{cum} the β_0 tasks of τ with largest utilization cannot have a cumulative utilization larger than $U_{cum}(\beta_0)$, which is at most 1. Hence, when partitioning τ, there will be at least β_0 tasks assigned to processor ρ_0 before any task is assigned to processor ρ_1. Furthermore, by Theorem 3, when the first task is assigned to processor ρ_1, the total utilization of tasks assigned to processor ρ_0 must be at least $\beta_0/(\beta_0 + 1)$. Thus, the base case holds.

Now assume the theorem holds for processors ρ_0 through ρ_{k-1} for some $k \geq 1$ and consider processor ρ_k. By the induction hypothesis, when the first task of τ is assigned to processor ρ_k there are at least B_{k-1} task assigned to processors ρ_0 through ρ_{k-1} and their total utilization is at most $mintot$. Consider any $i > B_{k-1}$ such that $U_{cum}(i) \leq mintot + 1$. By the induction hypothesis, we can partition the first B_{k-1} tasks on processors ρ_0 through ρ_{k-1}. Moreover, the total utilization of those tasks is at least $mintot$. Therefore, the total utilization of the remaining β_k tasks is at most 1. Hence, after partitioning the first B_{k-1} tasks onto processors ρ_0 through ρ_{k-1}, we can assign all of the next β_k tasks to processor ρ_k.

Let T_i be the first task assigned to processor ρ_k. When T_i is assigned to processor ρ_k, at least B_{k-1} tasks have been assigned to processors $\rho_0, \rho_1 \ldots \rho_{j-1}$ and the total utilization of tasks T_i through $T_{i+\beta_k-1}$ cannot exceed 1. Because tasks are sorted by utilization, any set of β_k tasks with index greater than or equal to i must have total utilization less than or equal to 1. Hence, there will be at least β_k tasks assigned to processor ρ_k before any task is assigned to processor ρ_{k+1}. By

Theorem 3, the total utilization of these tasks will be at least $\beta_k/(\beta_k + 1)$. Thus, the theorem is proved. □

Figure 5.3, illustrates the approach described in Theorem 4. In Figure 5.3(a), the minimum number of tasks assigned to ρ_0 is determined by finding the largest value β_0 such that $U_{cum}(\beta_0) \leq 1$. The minimum total utilization of the tasks assigned to ρ_0 is then determined. In Figure 5.3(b), we see that β_1 is determined by considering the vertical offset starting at ρ_0's minimum total utilization. In Figure 5.3(c), we see how β_j is determined after finding β_0 through β_{j-1}. As opposed to Figure 5.1, we see that the steps formed in this manner no longer touch the graph of U_{cum}.

We can actually improve our analysis by making two observations. First, the values of β_j cannot decrease as j increases. Second, when finding B_j, we are not taking advantage of the gap on processors ρ_0 through ρ_{j-1}. We now address each of these issues.

Lemma 3. *Let β_j be the number of tasks assigned to processor ρ_j when the first task is assigned to processor ρ_{j+1}. Then, for all $k > j$ there will be at least β_j tasks assigned to ρ_k before the first task is assigned to processor ρ_{k+1}.*

Proof: Because tasks are sorted by utilization, we know that any tasks that are assigned to a processor ρ_k where $k > j$ must have utilization small than or equal to the utilization of any of the β_j tasks assigned to ρ_j before when the first task was assigned to ρ_{j+1}. Therefore, the total utilization of any β_j tasks assigned to ρ_k cannot be larger than the total utilization of the first β_j tasks assigned to ρ_j. Because these β_j tasks fit onto ρ_j, their total utilization must be at most 1. Therefore, at least β_j tasks can fit onto ρ_k before any tasks are assigned to ρ_{k+1}. □

By Lemma 3, if $U_{cum}(B_{j-1} + \beta_{j-1}) > 1 + mintot$, we can let $\beta_j = \beta_{j-1}$ rather than using Theorem 4 to find β_j.

We now address our second observation. Because we are determining the bounds for FFD, we can assign tasks to a processor ρ_j even after tasks have been added to processor ρ_{j+1}. Because U_{cum} does not tell us the individual task utilization values, we cannot know the maximum total utilization assigned to any individual processor. Even so, after finding B_j, U_{cum} does tell us the maximum total utilization of the first B_j tasks. By construction, all those tasks must be assigned

to processors $\rho_0, \rho_1 \ldots \rho_j$. Hence, after assigning the first B_j tasks, weknow the total gap on those processors is at most $(j+1) - U_{cum}(B_j)$. Thus, while we cannot know the actual gap on any processor, we do know that the *average* gap is at least

$$\frac{j+1-U_{cum}(B_j)}{j+1} = 1 - \frac{U_{cum}(B_j)}{j+1}.$$

Clearly, the actual maximum gap cannot be smaller than this average. Hence, if we can guarantee the next tasks' utilization is smaller than the minimum average gap, then we know that FFD will assign that task to one of the first $(j+1)$ processors rather than assigning it to ρ_{j+1}.

Even though we cannot know a task's actual utilization, we can once again use avegrages to determine the largest utilization the next task can have. Specifically, the utilization of the $(B_j+1)^{th}$ task cannot be larger than the average utilization of the first (B_j+1) tasks. Nor can it be larger than the maximum average utilization of the tasks assigned to processor ρ_j. These observations lead us to our next lemma.

Lemma 4. *Assume the first $B_j = B_{j-1} + \beta_j$ tasks are assigned to processors ρ_0 through ρ_j and let $mintot$ be the minimum total utilization of the tasks assigned to processors ρ_0 through ρ_{j-1}. Then*

- *The average gap on processors ρ_0 through ρ_j after partitioning the first B_j tasks is at least $1 - \frac{U_{cum(B+j)}}{j+1}$.*
- *The utilization of the $(B_j+1)^{th}$ task is not larger than $U_{cum}(B_j+1)/(B_j+1)$.*
- *The utilization of the $(i+1)^{th}$ task is not larger than $\frac{U_{cum}(i)-mintot}{\beta_j}$.*

Thus, if

$$\min\left\{\frac{U_{cum}(B_j+1)}{B_j+1}, \frac{U_{cum}(B_j)-mintot}{\beta_j}\right\} \leq 1 - \frac{U_{cum}(B_j)}{j+1} \tag{5.1}$$

the $(B_j+1)^{th}$ task will be assigned to some processor ρ_k, where $k \leq j$.

Proof. We show each case separately.

1. Because $B_j = B_{j-1} + \beta_j$, we know the first B_j tasks will be partitioned onto processors ρ_0 through ρ_j. After partitioning these tasks, the total gap on these processors is at least $(j+1) - U_{cum}(i)$, which means the average gap is at least $1 - \frac{U_{cum}(B_j)}{j+1}$.

2. The total utilization of the first (B_j+1) tasks is at most $U_{cum}(B_j+1)$. Therefore, the average utilization of these tasks is at most $\frac{U_{cum}(B_j+1)}{B_j+1}$. The $(i+1)^{th}$ task has the smallest utilization value of all of these tasks. Therefore, its utilization cannot exceed the average.

3. The total utilization of the β_j tasks of τ ending with the B_j^{th} task is at most $U_{cum}(B_j) - mintot$. Therefore, the average utilization of these tasks is at most $\frac{U_{cum}(B_j)-mintot}{\beta_j}$. The utilization of the $(B_j+1)^{th}$ task cannot exceed this average.

Therefore, if Equation 5.1 holds then the $(i+1)^{th}$ task will not need to be assigned to processor ρ_{j+1}. □

Algorithm 3 finds the required number of processors for a given any task set bounded by a function U_{cum} and its utilization bound for a fixed number of processors, m. The running time of this algorithm is O(m).

5.0.4 EVALUATION

In this section, we compare the bounds determined by our work using the cumulative utilization function, $U_{cum}(i)$ to the corresponding bounds developed by Lopez, et al., [1]. We used about 800 randomly generated task sets with maximum utilization ranging from 0.01 to 0.99 to form the cumulative utilization functions. Using these functions we found $m(U_{cum}, u_{max}, n)$ and $U_{bound}(U_{cum})$ using the method introduced in this chapter. The value of n i the size of the domain of U_{cum}. We also determine the utilization bound and the number of processors developed by Lopez, et al., [1]. Using maximum and total utilization values $U_{cum}(1)$ and $U_{cum}(n)$, respectively. The maximum utilization, u_{max}, for a given task set is the value denoted by $U_{cum}(o)$.

The graphs demonstrate that, for large values of u_{max} and larger values of cumulative utilization, our iterative method reduces the required number of processors by as much as 50% and increases the utilization bound by as much as 85%.

We tested for task sets containing n tasks for $n = 10, 25, 50, 100, \infty$ and report the average savings for each scenario. Figure 5.4 compares our results to the Lopez results for fixed values of

Algorithm 3 Find-bounds$(U_{cum}(i), m_{max})$

Require: $U_{cum}()$ is a valid cumulative utilization function of task-id. m_{max} (an optional argument) is the maximum processor id.

$B_{-1} \leftarrow 0$
$m \leftarrow 0$
if $m_{max} > 0$ **then**
  $mintot \leftarrow \frac{\beta_0}{\beta_0+1}$
else
  $mintot \leftarrow 1$
while $(mintot \leq U_{sum})$ and $(m < m_{max})$ and $(B_m < size(U_{cum}))$ **do**
  if $m = m_{max}$ **then**
    $mintot \leftarrow mintot + 1$
  else
    find $\beta_m \in \mathbb{Z}$ such that $U_{cum}(B_{m-1} + \beta_m) \leq 1 + mintot < U_{cum}(B_{m-1} + \beta_m + 1)$
    if $\beta_m < \beta_{m-1}$ **then**
      $\beta_m = \beta_{m-1}$
    $B_m \leftarrow B_{m-1} + \beta_m$
    $mintot \leftarrow mintot + \frac{\beta_m}{\beta_m+1}$
    $min\_util\_sup \leftarrow \min\left\{\frac{U_{cum}(B_m+1)}{B_m+1}, \frac{U_{cum}(B_m-mintot)}{\beta_m}\right\}$
    $avgGap \leftarrow \frac{U_{cum}(B_m)}{j+1}$
    if $min\_util\_sup \leq avgGap$ **then**
      $k \leftarrow \lfloor avgGap/min\_util\_sup \rfloor$
      $B_m \leftarrow B_m + k$
      $mintot = mintot + avgGap \cdot \frac{k}{k+1}$
    $m \leftarrow m + 1$
return $m, mintot$

n. Figure 5.4(a) shows the percent reduction in the number of processors and Figure 5.4(b) shows the percent increase in the utilization bound we get using our method. For each U_{cum}, we determine the savings in the number of processors as follows.

$$\text{Percent savings} = \frac{m_L - m_{U_{cum}}}{m_L},$$

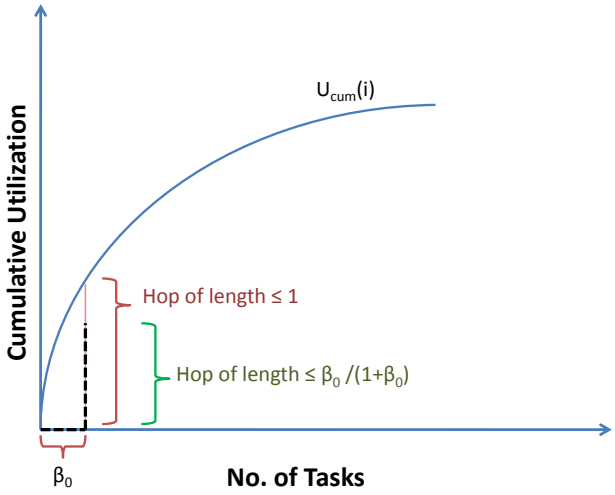
where m_L is the number of processors indicated by Lopez, et al., and $m_{U_{cum}}$ is the average number of processors derived by our algorithm. For each U_{cum}, we determine the savings in the number of processors as follows.

$$\text{Percent savings} = \frac{U_{bound,U_{cum}} - U_{bound,L}}{U_{bound,L}}$$

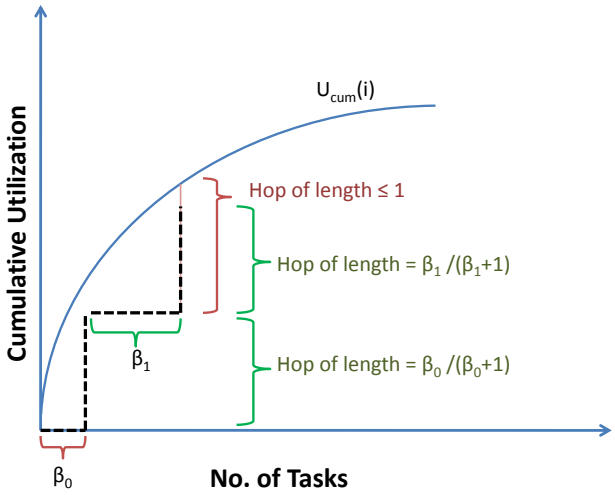
where $U_{bound,L}$ is utilization bound indicated by Lopez, et al., and $U_{bound,U_{cum}}$ is the utilization bound derived by our algorithm.

We tested for different ranges of total utilization values for $U_{sum} < 5$, $5 \leq U_{sum} < 50$, $50 \leq U_{sum} < 150$, $150 \leq U_{sum} < 500$, $U_{sum} \geq 500$ and report the average savings for each scenario. Figure 5.5 compares our results to the Lopez resultsfor different ranges of U_{sum}. Figure 5.5(a) shows the percentage reduction of the number of processors required and Figure 5.5(b) shows the percentage increase of the utilization bound.

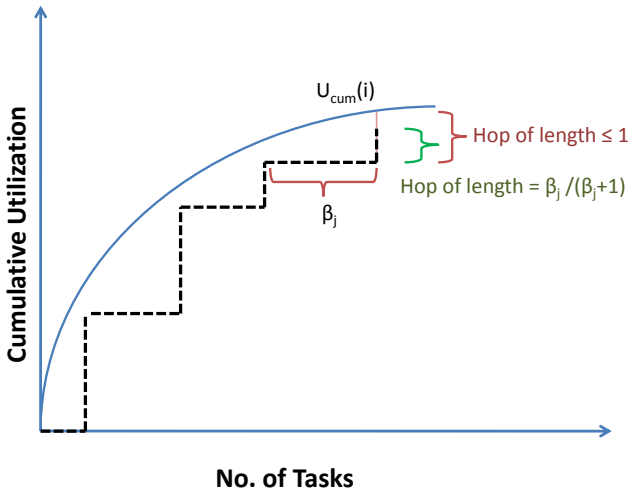
In all the graphs in this section, we aggregated the results based on u_{max} i.e., the percentage savings corresponding to $u_{max} = 0.5$ is actually the average of percentage savings for task sets with $0.4 < u_{max} \leq 0.5$. It is evident from the graphs that savings are high when $0.5 < u_{max} \leq 0.6$. This is because, when $u_{max} > 0.5$, Lopez, et al., allocate only one task to each processor. The gap on each processor increases when maximum utilization of tasks decreases. Thus the gap on each processor is largest when Lopez, et al., allocate a single task on each processor and the utilization value of the task is only a little over 0.5. Unlike Lopez, our method makes use of the average gap available on the previous processors. Thus, we get very high savings of up to 80% on the processor utilization and up to 50% on the required number of processors.



(a) Step-1 to find β_0

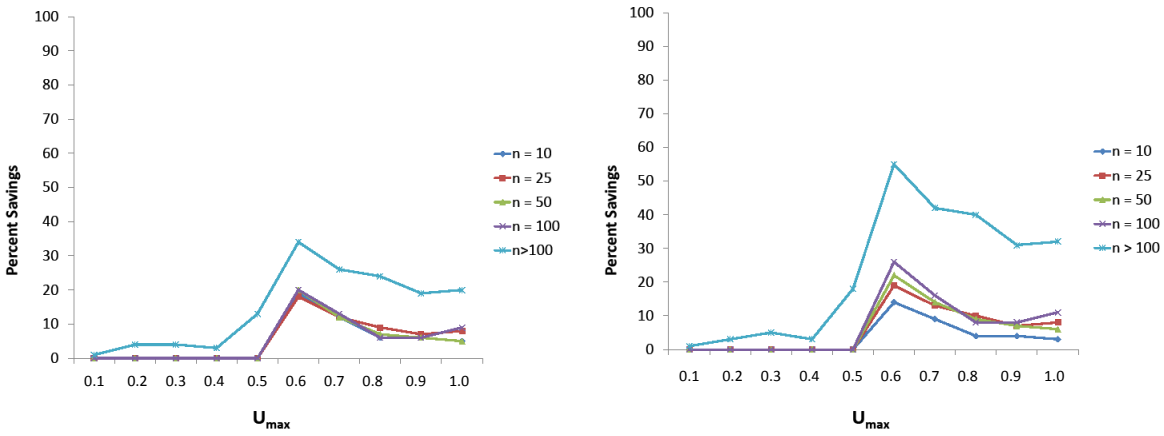


(b) Step-1 to find β_1



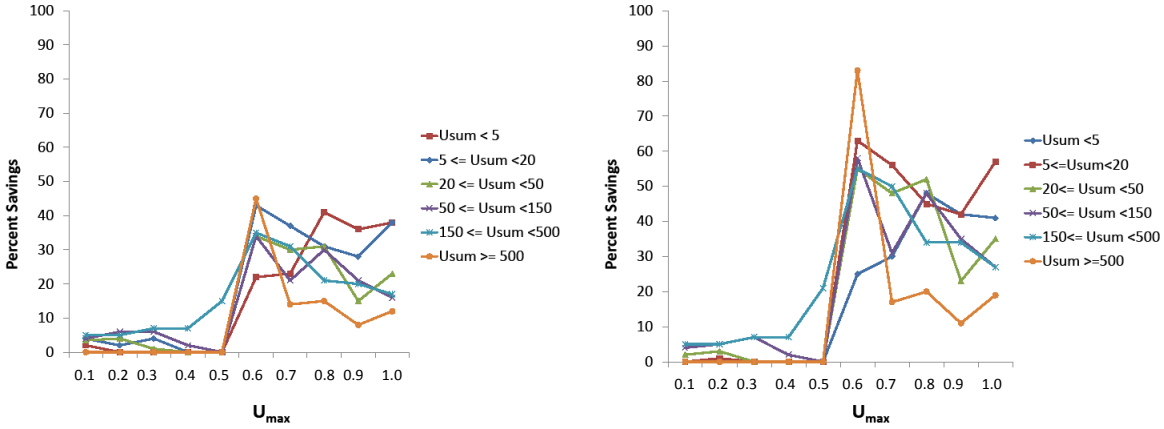
(c) Step-(i+1) to find β_i

Figure 5.3: Hop over the curve to find the number of processors



(a) Reduction in number of Processors. (b) Increase in U_{bound}.

Figure 5.4: Comparison of bounds using U_{cum} vs. Lopez, et al., bounds for different sizes of task sets



(a) Reduction in number of Processors. (b) Increase in U_{bound}.

Figure 5.5: Comparison of bounds using U_{cum} vs. Lopez, et al., bounds for different values of total utilization

CHAPTER 6

CONCLUSION

We have presented a new method of developing utilization bounds and determining a number of processors that guarantees a task set can be partitioned using FFD when each processor schedules tasks using EDF. Using the parameter γ instead of U_{sum}, we are able to reduce the number of processors and increase the utilization bound.The utilization binder reduces the number of processors by as much as 65% for values of u_{max} near $\frac{1}{2}$ and up to 50% for larger values of u_{max}. These savings continue to hold when the utilization binder γ, is close to 1. When $\gamma = 1$, the utilization binder produces same results as the current state of art method and they improve as γ decreases. While the method using the utilization binder produces excellent results for larger values of u_{max}, the cumulative utilization function produces very good results for u_{max} ranging between 0.5 and 0.6. Our experimental results found that when u_{max} ranges from 0.5 to 0.6, the savings on utilization bound is as large as 80% and the savings on required number of processors goes up to 50%.

Decisions made during the design phase are critical and being judicial is as important as being cautious in determining the quantity of resources. Making judgements based on restricted information – i.e., only u_{max} and U_{sum} can cost us a lot. Mohan, et al., in [8] and Ferdinand, et al., in [9] give us methods to find more detailed information about tasks than just u_{max} and U_{sum}. In this thesis we show that taking advantage of this extra knowledge can lead to much less wasteful (and hence less expensive) systems. We conclude by saying that we should take advantage of information whenever it is available, so we can determine the amount of resources required to achieve an efficient system.

BIBLIOGRAPHY

[1] J. M. López, M. Garcia, J. L. Diaz, and D. F. Garcia, "Worst-case utilization bound for EDF scheduling on real-time multiprocessor systems," in *Proceedings of the EuroMicro Conference on Real-Time Systems*. Stockholm, Sweden: IEEE Computer Society Press, June 2000, pp. 25–34.

[2] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[3] T. Baker, "Multiprocessor edf and deadline monotonic schedulability analysis," in *24th Real-Time Systems Symposium*, 2003.

[4] ——, "An analysis of edf schedulability on a multiprocessor," vol. 16, 2005.

[5] C. A. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, El Paso, Texas, 4–6 May 1997, pp. 140–149.

[6] D. Johnson, "Fast algorithms for bin packing," *Journal of Computer and Systems Science*, vol. 8, no. 3, pp. 272–314, 1974.

[7] D. S. Johnson, A. Demers, J. D. Ullman, M. R. Garey, and R. L. Graham, "Worst-case performance bounds for simple one-dimensional packing algorithms," *SIAM Journal of Computing*, vol. 3, no. 4, pp. 299–325, 1974.

[8] S. Mohan, M.-Y. Nam, R. Pellizoni, L. Sha1, R. Bradford, and S. Fliginger, "Rapid early-phase virtual integration," in *The 30th IEEE Real-Time Systems Symposium (RTSS 2009)*, Washington DC, USA, December 2009, pp. 33–44.

[9] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm, "Reliable and precise WCET determination for a real-life processor," in *Proceedings of the EuroMicro Conference on Real-Time Systems*, vol. 2211. Springer Berlin / Heidelberg, 2001, pp. 469–485.

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

[11] M. Dertouzos and A. K. Mok, "Multiprocessor scheduling in a hard real-time environment," *IEEE Transactions on Software Engineering*, vol. 15, no. 12, pp. 1497–1506, 1989.

[12] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Department of Mathematics, Massachusetts Institute of Technology, 1973.

[13] Mathworks, *Simulink simulation and model-based design*. [Online]. Available: http://www.mathworks.com/products/simulink/

[14] ——, *Real-time workshop*. [Online]. Available: http://www.mathworks.com/products/rtw/

[15] ——, *Real-time workshop embedded coder*. [Online]. Available: http://www.mathworks.com/products/rtwembedded/