Empirical Study of Adaptive EDF

By WU Qiong

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Kiyofumi Tanaka

March, 2015

Empirical Study of Adaptive EDF

By WU Qiong (1310021)

A thesis submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Associate Professor Kiyofumi Tanaka

And approved by
Associate Professor Kiyofumi Tanaka
Professor Mineo Kaneko
Professor Yasushi Inoguchi

February, 2015 (Submitted)

Acknowledgements

First of all, I would like to express my deepest thanks to my Supervisor, Associate Professor Tanaka Kiyofumi. He helped me a lot to finish my master study with his knowledge, enthusiasm and patience. I especially treasure the experience gained over the past year from him.

Also won't forget my family for their continuous support and sacrifice. My parents with their great love and full understanding all through those years, I was strongly motivated to go on in my study and pursue my life goals.

I would also like to extend warmest thanks to Ishikawa Prefecture for the generous scholarship I was lucky to receive.

I also want to show my gratitude for my lab-mates and friends for their help, encouragement that got me through many difficult periods.

Abstract

Due to the growing complexity of embedded systems, real-time task scheduling is becoming increasingly important for real-time systems. A real-time operating system (RTOS) should manage both periodic and aperiodic tasks. In order to handle different scenarios, RTOS manages tasks using well-defined and sophisticated scheduling algorithms. There exist various scheduling algorithms such as RM, EDF, TBS and adaptive EDF. Adaptive EDF is known to reduce response times and jitter for some particular tasks.

In this thesis, the basic EDF scheduler is extended with Adaptive EDF and the new retrospective releasing technique. The combination is implemented and compared to other methods. The schedulers are evaluated using a clock-cycle-based CPU simulator which simulates binary codes consisting of tasks' codes and ITRON kernel codes and reports response times and jitters of tasks. The evaluation is performed for 31 task sets.

EDF shows the worst results, while Adaptive EDF has around 50% faster response times and 30% less jitters. The retrospective releasing with Adaptive EDF is little worse than Adaptive EDF. Combining the retrospective releasing and EDF is better than EDF but a little worse than the latter two algorithms.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Real-time computing is an important part of our daily life. Real-time operating systems (RTOS) are designed to handle real time-applications [9]. RTOS manages the execution of programs with specified time constraints. That means RTOS should guarantee to finish tasks within a certain time.

The increasing complexity of embedded systems makes task scheduling more important for real-time operating systems. In this thesis, a basic EDF scheduler is extended with Adaptive EDF [1] and a new retrospective releasing technique [10]. This combination is implemented and tested. The four schedulers will be compared in the evaluations.

1.1 Research Background

Various scheduling algorithms were proposed to handle tasks with different time constraints.

Earliest Deadline First (EDF) [3] scheduling algorithm is widely used as a representative real-time task scheduler. EDF was first proposed to handle tasks with dynamic priorities and can operate when the system is utilized by up to 100%. EDF algorithm assigns dynamic priorities to tasks based on their absolute deadlines. The main drawback is that EDF cannot give some particular important tasks fixed priorities and faster responsiveness.

Rate Monotonic (RM) [3] scheduling algorithm was designed to prefer important tasks with faster responsiveness. RM algorithm is one of static-priority scheduling algorithms that assign fixed priorities to tasks. The demerit is that RM cannot guarantee schedulability when the processor

utilization reaches 100%.

Total Bandwidth Server (TBS) schedules both periodic tasks and aperiodic tasks. TBS is basically extended EDF by assuming that aperiodic tasks have some calculated deadlines. That makes TBS another dynamic priority algorithm that can exhibit fair responsiveness for aperiodic tasks.

Adaptive EDF [1] [2] is also based on EDF. It can decrease the response times of particular (important) tasks. Adaptive EDF is a dynamic priority assignment scheduler. It can make some target important tasks run before the other tasks. Therefore Adaptive EDF achieves shorter response times and smaller jitters for the target important tasks similar to RM while still being able operate at 100% processor utilization.

1.2 Objective and Contributions

The purpose of this research is to test the practicality of Adaptive Earliest Deadline First (Adaptive EDF) [1] [2]. In addition, it includes evaluating, through quantitative experiments, how much Adaptive EDF reduces response times of real-time periodic tasks, involving the runtime overheads of the operating system.

According to the research in [1] [2], the Adaptive EDF shortens response times more than the other existing algorithms. However, since probability distribution models were used for generating task sets (execution times, periods, etc.), the practicability was not assured.

The contribution of this thesis is:

- Using an actual real-time operating system (RTOS) and real programs running on a CPU simulator. Both EDF and Adaptive EDF have been implemented and evaluated.
- A new additional function “retrospective releasing” to EDF and Adaptive EDF. This newly proposed scheduling technique is

implemented, and the effectiveness is evaluated in the evaluation.

1.3 Thesis Outline

This chapter introduces the research background, objective and contributions. Chapter 2 explains some of the related works: real-time operating system, EDF, RM and Total Bandwidth Server. Chapter 3 explains Adaptive EDF algorithm in detail. Chapter 4 introduces a new technique, the retrospective releasing. Chapter 5 describes ITRON and how to implement algorithms considered. Chapter 6 shows the evaluation environment and task sets that used in the evaluation. Then, the scheduling results of the ITRON original baseline, EDF, Adaptive EDF, and combination of Adaptive EDF and retrospective releasing. Finally, Chapter 7 draws conclusions of the thesis and some of the future works.

Chapter 2

Related Works

2.1 Real-Time Operating Systems

A real-time system is a class of systems that when some events happen it processes data with specified time constraints. This kind of time-related requirement can be seen in various fields: automotive control, industrial control, manufacturing plants, etc.

Real-time systems are classified into two types; first is hard real-time computing systems (HRTCS) and the other is soft real-time computing systems (SRTCS). HRTCS demands that tasks must meet their deadlines in all working cases. If tasks cannot complete the work in time, the system will terminate with a failure. At the worst, it results in catastrophic consequences. For example, when traffic accident happens, airbags must be inflated instantaneously on impact. On the other hand, SRTCS allow tasks to occasionally miss their deadlines, which do not lead to serious damage to the system.

Operating systems manage hardware resources and applications. The difference between general OSs and real-time OSs is that the latter have the following additional feature. It is designed to run applications with precise timing constraints. There are two main goals for any RTOS. The first is meeting the tasks timing constraints, which requires timely execution of those tasks. The second is shortening response times and reducing jitters of important tasks compared to the other application tasks.

There are several basic features of RTOS to be introduced

(1) Timeliness

Operating systems provide time control (management) to handle tasks

with time constraints by using real-time scheduling algorithms.

(2) Predictability

It guarantees in advance deadline satisfaction. If not possible, it informs the result, which is not allowed in HRTCS.

(3) Fault tolerance

Soundness should be maintained even if hardware or software failures occur.

(4) Design for peak load

Overload situations must be considered and managed.

(5) Maintainability

In RTOS methodologies, to deal with different scenarios, there are various scheduling algorithms: ex.) algorithms only for periodic tasks, other algorithms for both periodic and aperiodic tasks. In the following sections, several representative scheduling algorithms are introduced.

2.2 EDF Algorithm

Earliest Deadline First (EDF) [3] is an optimal dynamic priority scheduling algorithm for periodic tasks. It assigns dynamic priorities to tasks based on their absolute deadlines. Tasks with earlier absolute deadlines have higher priorities. In other words, the EDF scheduler always schedules a task with the earliest absolute deadline.

Dynamic scheduling algorithms have higher schedulability bounds than fixed priority algorithms. Therefore, they can provide higher overall processor utilization factors. EDF has a merit that it can fully make processor utilization reach 100% with schedulability. However, it is impossible to give tasks fixed priorities and therefore it cannot keep response

times or jitters small for particular (important in the system) tasks.

Figure 2.1 is an example to explain the EDF rule. The horizontal axis shows processor time in ticks. In this example, there are two periodic tasks τ_1 *and* τ_2. As shown in Table 2.1, τ_1 has a period of 5 and execution time of 2. T_2 has a period of 7 and execution time of 4. In the beginning at tick 0, task τ_1 has the earliest absolute deadline, so it has the highest priority and is executed first. At tick 15, task τ_2 is preempted by the higher priority task τ_1 since the new instance of task τ_1 has an earlier absolute deadline of 20, earlier than the deadline of task τ_2's current instance.

Task	Period (T)	Execution time (C)
τ_1	5	2
τ_2	7	4

Table 2.1: Periods and execution times for two periodic tasks respectively

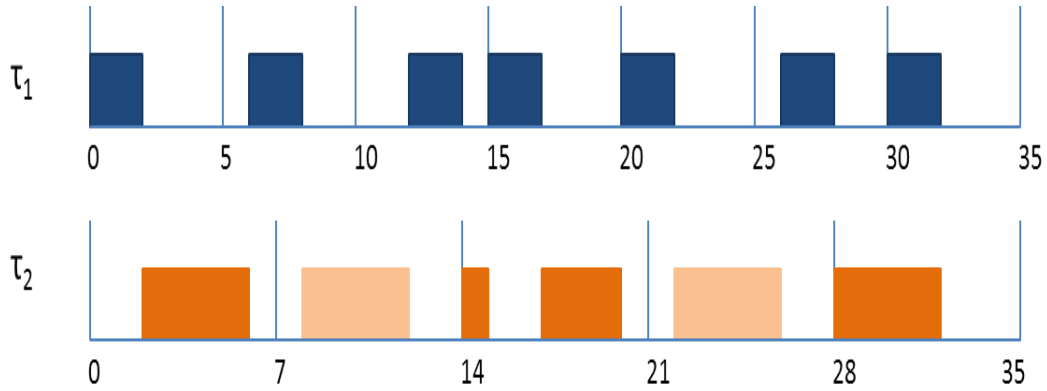

Figure 2.1: Example of Earliest Deadline First (EDF) Scheduling

2.3 RM Algorithm

Rate Monotonic (RM) is an optimal static-priority scheduling algorithm for periodic tasks [3]. It assigns fixed priorities based on task periods. That means tasks with shorter periods have higher priorities. In other words, RM

schedules tasks with shorter periods preferentially.

RM has a merit that it is possible to achieve short response times for some particular tasks by giving shorter periods to them. However, tasks with long periods cannot be given higher priority even if they are important in the system. In other words, importance must depend on periods. Unlike EDF, RM cannot utilize processor up to 100% while maintaining schedulability.

Figure 2.2 is an example of RM scheduling. Tasks τ_1 and τ_2 have periods 5 and 7, respectively. Their execution times are 2 and 4, respectively. Since task τ_1 has the shortest period, it always has the highest priority and is executed first. At tick 5, task τ_2 is preempted by the higher priority task τ_1. At tick 7, the first instance of task τ_2 has not finished yet and therefore it experiences time overflow, or deadline missing. This task set has the processor utilization of 5/7 + 2/4 = 34/35. This value is larger than the allowable utilization value for RM [3]. Therefore, a deadline miss necessarily happens.

Task	Period (T)	Execution time (C)
τ_1	5	2
τ_2	7	4

Table 2.2: Periods and execution times of two periodic tasks respectively

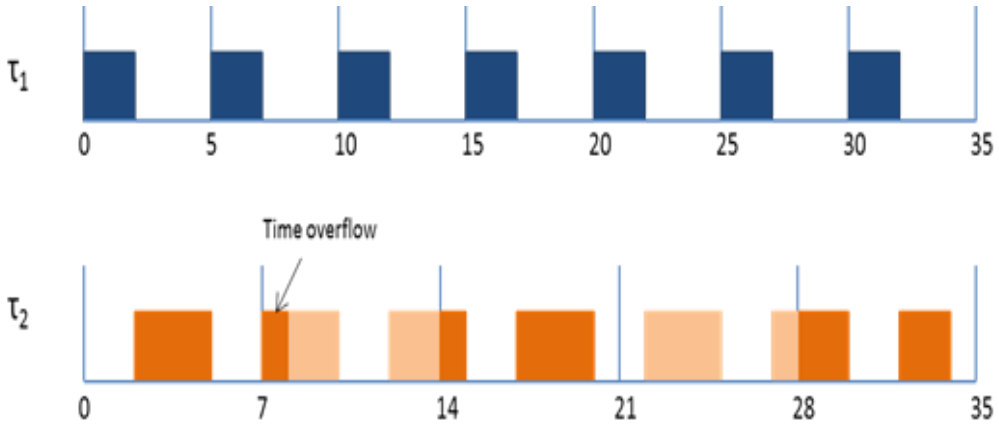

Figure 2.2: Example of Rate Monotonic (RM) algorithm scheduling

The comparison between RM and EDF is shown in Table 2.3. The comparison is carried out with respect to their priority, processor utilization, and jitter and response times.

Scheduling algorithm	Priority	Processor utilization	Jitter and response times for high-priority tasks
Rate Monotonic	Fixed priorities based on periods	Cannot reach 100%	Small (good)
Earliest Deadline First	Dynamic priorities based on absolute deadlines	Can reach 100% (good)	Not small

Table 2.3: The comparison of RM and EDF

2.4 Total Bandwidth Server Algorithm

Total Bandwidth Server (TBS) is a scheduling algorithm which can handle both periodic tasks and aperiodic tasks. TBS schedules tasks based on EDF. When an aperiodic task execution is requested, it assigns a deadline to it immediately by using equation 1.

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s} \quad (1)$$

r_k is the release (request/arrival/invocation) time of the k^{th} aperiodic request. C_k is the execution time of the request. Us is the server bandwidth which can be dedicated to aperiodic tasks' executions. In addition, d_{k-1} is the absolute deadline of the k-1th (previous) aperiodic execution. d_0 is defined as zero. When the k^{th} aperiodic task execution is requested, the calculated deadline is assigned to the job. After that, the job is inserted into the ready queue of the

system and all tasks are scheduled by the EDF algorithm.

Figure 2.3 shows an example of two periodic tasks τ_1 and τ_2 along with an aperiodic task which is served by TBS. τ_1 and τ_2 have periods $T_1 = 4$ and $T_2 = 6$ and execution times $C_1 = 1$ and $C_2 = 3$, respectively. TBS has the server utilization of $U_s = 1 - U_p = 1 - (1/4 + 3/6) = 0.25$. The first aperiodic request arrives at the releasing time $r_1 = 1$. The deadline is calculated by equation 1 which is $d_1 = r_1 + C_1/U_s = 1 + 1/0.25 = 5$. Since d_1 is the earliest deadline at the moment, this aperiodic task is executed immediately. Similarly, the second aperiodic request arrives at $r_2 = 4$ and the deadline is calculated as $d_2 = r_2 + C_2/U_s = 5 + 2/0.25 = 13$. Notice that $r_2 < d_1$ and then d_1 is used in the deadline calculation. This time the aperiodic task cannot be run immediately, since at time 4, other periodic tasks (τ_1 and τ_2) have earlier deadlines, 8 and 6, respectively. After τ_2's first instance finishes at time 5, τ_1's second instance is executed until time 6. Then τ_2's second instance starts to run since it has deadline of 12, earlier than the aperiodic job. Further, τ_1's third instance has priority over the aperiodic task. Finally, the second aperiodic task is scheduled at time 10 and it finishes at 12. Further instances of τ_1 and τ_2 follow this scenario.

Task	Period (T)	Execution time (C)
τ_1	4	1
τ_2	6	3

Table 2.4: Periods and execution times for two periodic tasks respectively

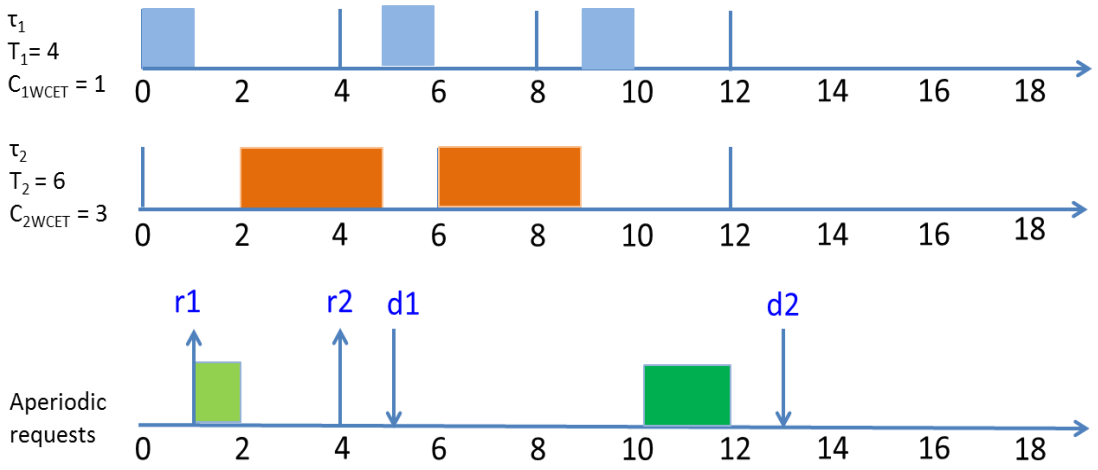

Figure 2.3 Example of Total Bandwidth Server (TBS) Scheduling

Chapter 3

Adaptive EDF

3.1 Introduction of Adaptive EDF

With the increase of complexity of embedded systems, the importance of task scheduling keeps increasing for real-time operating system. Adaptive EDF is a scheduling algorithm for periodic tasks which aims to reduce response times of certain important tasks as well as their jitter in real-time systems. Adaptive EDF was devised based on Earliest Deadline First (EDF). According to the evaluation in [1] [2], the decrease of response times of a particular (target or important) task is better compared to most of the existing algorithms. This research focuses on Adaptive EDF, implementing it on ITRON environment to evaluate its practicality.

Adaptive EDF is an improvement of the basic EDF algorithm. EDF is widely used as a representative real-time task scheduler. EDF is based on dynamic priority assignment. The earlier the task's deadline, the sooner it is scheduled to run. In addition, EDF is superior to Rate monotonic (RM) in terms of processor utilization, that is, it can maintain schedulability even when the processor utilization reaches 100%. However, since the priorities used in EDF change dynamically, it is difficult for the system to keep the response times and jitters short for target important periodic tasks. On the other hand, RM is a scheduling algorithm which fixes tasks' priorities and schedules the tasks with shorter periods preferentially. However, the processor utilization can never reach 100%.

RM can give the target tasks short response times only when the tasks have relatively short periods. Thus, it is impossible to decide the importance of tasks independent of their periods, as shown in the section 2.4. Hence, to reduce the response times and jitters of such particular periodic tasks,

Adaptive EDF [1] [2] was proposed.

Adaptive EDF can make target important tasks run before the other tasks even when the target tasks have longer periods or later deadlines. Therefore Adaptive EDF achieves shorter response time and smaller jitters for the target important tasks.

3.2 Adaptive EDF Algorithm

This section introduces Adaptive EDF algorithm. Adaptive EDF basically follows the EDF algorithm. In other words, the task with the earliest deadline is scheduled and executed first. However, the additional feature of Adaptive EDF focuses on the fluctuation of actual execution times of tasks. Adaptive EDF tries to divide execution of some tasks into two or more sub instances (or jobs). Based on predictive execution times (PET), Adaptive EDF gives tasks stepwise deadlines. That is, the deadlines are updated during the tasks' execution. Accordingly, short response times and small jitters can be obtained when the actual execution finishes earlier than supposed times.

Before explaining the Adaptive EDF algorithm, both predictive execution times (PET) and server utilization (U_s) are introduced. Adaptive EDF uses predictive execution times (PET), instead of worst case execution times (WCET), to shorten the deadlines, which leads to reduction of response times and jitters.

In real-time operating systems, WCET is usually regarded as the task execution time. Real-time systems, especially hard real-time ones, must guarantee that every task execution can completely meet its own deadline. However, sometimes, actually the execution time is not so much longer than WCET. Besides, the overhead exists in the operating system processing such as dispatching, task switching, scheduling process, and so on. From the safety point of view, WCET tends to be estimated much larger than the actual execution time (AET). Thus, there is a large gap between WCET and AET. Hence, Adaptive EDF algorithm tries to reduce this gap by using different predictive execution time (PET) in order to reduce the response time and jitter.

In Adaptive EDF, when an invocation request for the target task is made, PET is estimated to be one tick. Then the corresponding deadline is calculated according the PET. To account for a longer execution time, when the task execution exceeds one tick, PET is re-estimated to be one and then the deadline is updated.

The target task is virtually served by a server. The server utilization is U_s. Then Adaptive EDF uses the same concept as Total Bandwidth Server (TBS) algorithm [4]. That is, the deadline is calculated to allow the task execution to occupy the server utilization. The schedulability is satisfied when the total processor utilization, summation of utilization by each task, is equal to or less than one. In Adaptive EDF, U_s for the target task is shown in Equation 2. U_i is the utilization of the target periodic task (τi), which is calculated by using the task's WCET ($WCETi$) and period (Ti), that is $WCETi / Ti$. Up is the total utilization of all periodic tasks.

$$U_i = U_t + (1 - U_p) \tag{2}$$

The calculations of deadlines in Adaptive EDF are as follows.

$$d_k^1 = k \times T_i + \frac{1}{U_s} \tag{3}$$

$$d_k^j = d_k^{j-1} + \frac{1}{U_s} \quad (j > 1) \tag{4}$$

The deadline of the target periodic task can be divided into two or more sub instances. The first instance ($J1$) starts from the beginning to one tick later, and the following sub instances (Ji) starts from j-1 ticks to j ticks.

Equation 3 is the absolute deadline of the first sub instance of the k^{th} execution instance for the target periodic task. T_i is the period of the target task. In this task set model, each task has a relative deadline which is equal to its period. U_s is the server utilization. By using equation 3, the deadline of the first sub instance of the target task is computed. Equation 4 is the absolute deadline of the rest instances of the target task. If the target task can be finishes in the first sub instance, Equation 4 is not needed to be

calculated. If the target task finishes in *j* ticks, *j+1*-th and following sub instances are not needed to be executed.

The Adaptive EDF algorithm is divided into 3 parts. The following 3 code blocks are the cores of Adaptive EDF.

```
IF  (targettsk == current_task) {
       ADL = current_tim + 1/Us
       }
```

Figure 3.1. Task Deadline Initialization

As shown in Figure 3.1, the first code block, *Task deadline initialization,* is executed on every new task instance invocation (releasing). First, the code checks whether the running task is target task. If so, it initializes the absolute deadline by the current time plus the reciprocal of the server utilization U_s as shown in Equation 3.

```
IF (schtsk == targettsk) {
       // next is target
       target_start_time = current_tim
       }
```

Figure 3.2. Task Switching Setting

The second code block, task switching setting in Figure 3.2, is executed every time task switching happens. First, the code checks whether the next (scheduled) task is the target task. If so, it saves the current time as the target task's re-start time.

The third code block, task execution time and deadline update, is shown in Figure 3.3. This code is executed on every tick timing and task switching.

```
IF (current_task == targettsk){
       target_exec_time += current_tim - target_start_time;
       IF ( target_exec_time >= (80% of TICK) )  {

              targettsk -> ADL += 1/Us

              target_exec_time = 0
              }
}
```

Figure 3.3. Task Execution Time & Deadline Update

First, the code checks whether the running task is the target task, If so, it updates the target task's execution time (***target_exec_time***). The target task execution time will be incremented by the current time – start time. Then it checks whether the cumulative execution time of the target task is larger than 80% of the tick time. If so, it updates the deadline. The absolute deadline of the target task is incremented by the reciprocal of the server utilization U_s. Finally, it resets the target task execution time to 0. If execution time of the target task is found to be less, nothing is done.

3.3 Example of Adaptive EDF

Figure 3.4 is an exmple of scheduling by EDF and Adaptive EDF. The horizontal axis is time in ticks. In both algorihms, two tasks, τ_1 and τ_2, have periods, 4 and 6, respectively. *WCET* is 2 for both of them. Therefore, the processor utilizations for τ_1 and τ_2 are calculated as $U_1 = 2/4 = 0.5$ and $U_2 = 2/6 = 0.33$, respectively. The actual execution times for τ_1 and τ_2 are given as 2 and 1 respectively. In this example, task τ_2 is assumed to be the target (important) task favored by Adaptive EDF.

Using the formulas (2), the deadlines of τ_2 are calculated. In case of Adaptive EDF, the deadlines of τ_2 are calculated 3, 9, and 15 as illustrated in Figure 3.4. By Adaptive EDF, the average response time of the three instances of τ_2 is (1+1+1)/3 = 1. On the other hand, the average response time by EDF for the same instances of τ_2 is (3+1+3)/3 = 2.33. This means that

Adaptive EDF has shorter response time than EDF for the selected target task τ_2.

Task	Period (T)	Execution time (WCET)	Actual exec.time (AET)
τ_1	4	2	2
τ_2	6	2	1

Table 3.1 Task set for EDF and Adaptive EDF Example

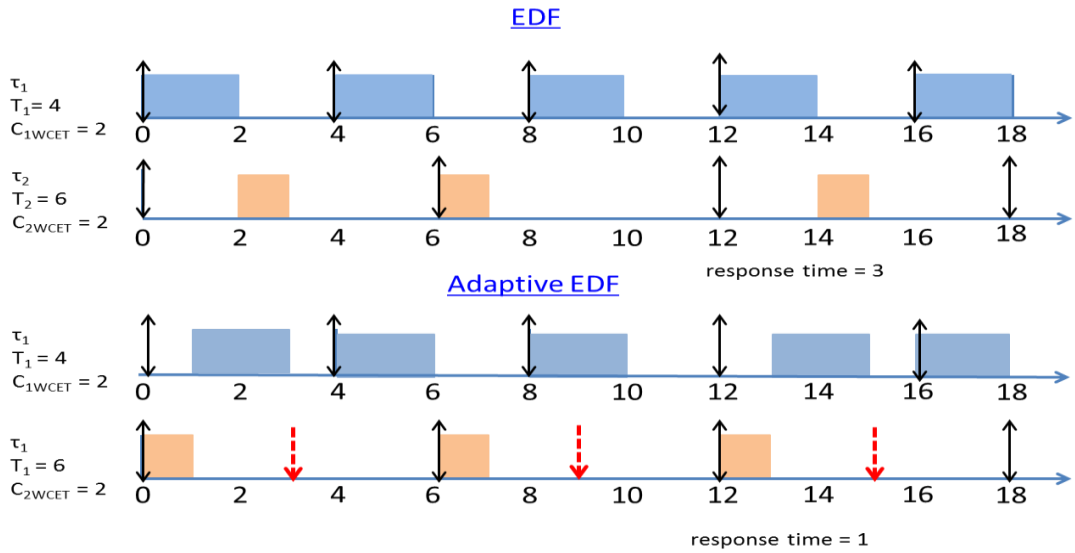

Figure 3.4. Example of the EDF and Adaptive EDF

Chapter 4

Retrospective Releasing

4.1 Introduction of Retrospective Releasing

This section introduces a new technique, Retrospective Releasing. It is basically a function added on top of both EDF and Adaptive EDF.

Retrospective releasing is used to further achieve shorter response times and small jitters for particular periodic tasks. This technique will achieve the goal by assuming an advanced release (invocation) time and obtaining corresponding earlier deadlines.

The main idea of retrospective releasing is an enhancement of Total Bandwidth Server (TBS) [3]. TBS is a scheduling algorithm which can schedule (handle) both periodic and aperiodic tasks. In TBS, actual arriving times of tasks are used as the releasing times to calculate the absolute deadline. Therefore, assuming an earlier release time for a particular task, an earlier deadline can be obtained. For periodic tasks, the releasing time is its own period. Accordingly, it is impossible to advance the releasing time, however, retrospective releasing technique, tries to assume an earlier release time to obtain an earlier deadline. Thus, it produces the shorter response times and small jitters.

The retrospective releasing technique advances release times but never influences the past schedules. This means that the retrospective releasing technique can be performed only when tasks' scheduling order never changes between the virtual release time and the actual release time. Therefore, in order to use this kind of technique, there are three conditions to be discussed. They are previous deadline, empty slot, and maximum used deadline which are explained below.

(1) Previous deadline

In the retrospective releasing technique, the virtually advanced release time of the target task must not exceed (be earlier than) the deadline of previous task, since it is not allowed in the basic rules of TBS [5]. To achieve this checking, the previous task' deadline should be recorded.

(2) Empty slot

When trying to advance the virtual release time, it must be checked whether the time slot to be passed over is an empty slot or not. This is because the past schedules must not be changed. Therefore, it is necessary to record the last empty slot number.

(3) Maximum used deadline

The retrospective releasing technique must guarantee that the virtual release time is not earlier than any deadlines which have been already used in the past scheduling. In order to check this condition, an array is prepared. Each element of the array records the deadline of the task that was executed in the corresponding time slot.

4.2 Retrospective Releasing Algorithm

In this section, the retrospective releasing algorithm is described. The pseudo code is shown as the basic retrospective releasing algorithm.

The presented pseudo code of the retrospective releasing algorithm focuses on applying only to the target task. First, several variables are introduced in this algorithm. "r_k" is the actual release time of the target task and "v_k" is the virtual release time of the task. "d_{k-1}" is the deadline of the previous task and "d_k" is the absolute deadline of the target task. "C" is the worst cast execution time or the predicted execution time in case of Adaptive EDF. "U_s" is the server utilization of this task. "*last_empty*" is used to record the last empty slot number and array "dl" records the used deadline of the executed tasks.

The retrospective releasing algorithm works as follows. In lines 1-2, *max_dl* is initialized to 0 and v_k is initialized to r_k. At line 3, the whole loop starts. At first, the absolute deadline of the target task, d_k, is computed by

using the current virtual release time (v_k). In lines 6-8, the first condition (previous deadline) is examined. If v_k is equal to the deadline of the previous task d_{k-1}, the algorithm finishes.

```
1.  max_dl ⇐ 0 /* initialize the maximum deadline */
2.  v_k ⇐ r_k      /* initialize the virtual release time */
3.  while TRUE do
4.      d_k ⇐ v_k + C/U_s
5.      /* the previous task deadline constraint ( d_k-1 ) */
6.      if v_k = d_k-1 then
7.          break
8.      end if
9.      /* empty slot condition */
10.     if v_k = last_empty +1 then
11.         break
12.     end if
13.     if max_dl < dl[v_k-1] then
14.         max_dl ⇐ dl[v_k-1]
15.     end if
16.     /* maximum deadline comparison */
17.     if d_k ≤ max_dl then
18.         break
19.     else
20.         /* virtual release advance one slot */
21.         v_k ⇐ v_k-1
22.     end if
23. end while
```

Figure 4.1 Pseudo code of Retrospective Releasing Algorithm

If v_k is not equal to d_{k-1}, in lines 10-12, the algorithm checks the second condition (empty slot). If the v_k is equal to the *last_empty* number pulse one, the algorithm finishes since it encounters an empty slot.

In lines 13-18, the third condition (maximum deadline) is checked. As preparation, comparing the *max_dl* and $dl[v_{k-1}]$, *max_dl* is updated if necessary. Then d_k is compared to *max_dl*. If the former is equal to or earlier than the latter, the algorithm finishes. When all the conditions above are passed, the virtual release time is decremented (or advanced) by one slot in

line 21. Then the loop is repeated.

In order for the retrospective releasing algorithm not to alter the past schedules, previous deadline, last empty slot and maximum used deadline must be checked before advancing the release time.

4.3 Example of Retrospective Releasing

There is an example to demonstrate the retrospective releasing technique. In Figure 4.2, there are three periodic tasks. Task τ_1 is assumed to be the target task. Task τ_1 has the original release time of 5, and the original deadline of 11. By applying the retrospective releasing technique, the release time of τ_1 is set to 2 and at the same time, its deadline is set to 8. Therefore, the shorter response time can be obtained. Note that the past schedule between 2 and 5 is not changed by this technique.

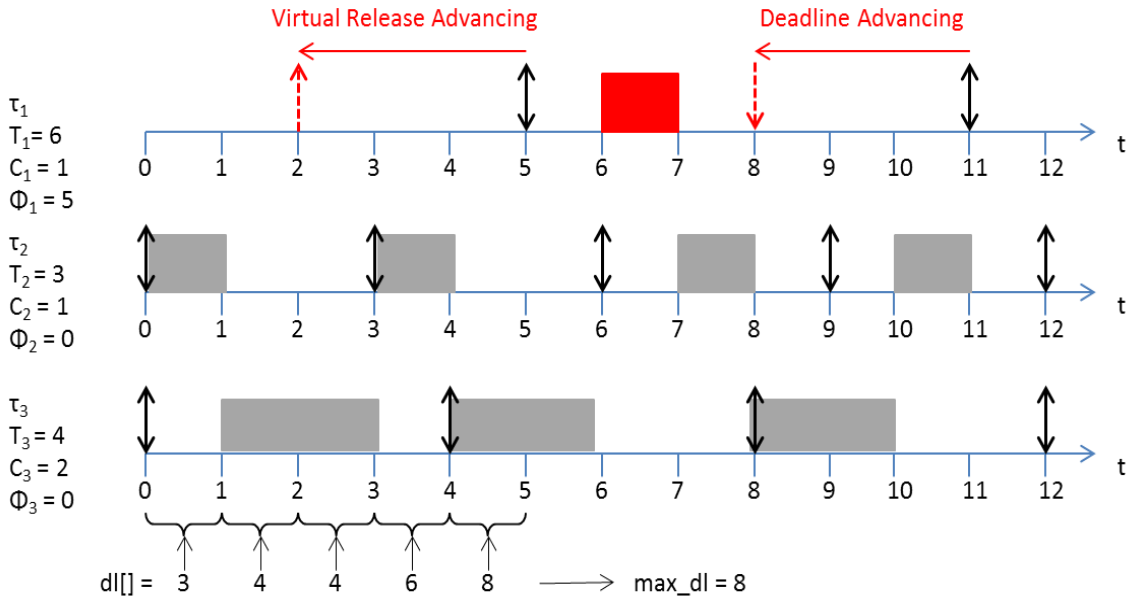

Figure 4.2. Example of Retrospective Releasing

Chapter 5

Implementation

5.1 ITRON

In conventional scheduling theories [3], it seems that the overheads of RTOS execution such as scheduling and dispatching are zero. Similarly in [1] [2], RTOS overhead is not taken into account. However, in practical systems, RTOS overheads exist.

Thus, in this research, baseline algorithm in ITRON, EDF, Adaptive EDF, EDF plus the retrospective releasing technique and Adaptive EDF plus the retrospective releasing technique will be evaluated quantitatively by executing not only tasks' codes but also RTOS codes on the CPU simulator.

In this research, the μITRON4.0 [6] operating system is used as the evaluation environment. Industrial The Real-time Operation system Nucleus (ITRON) introduces the standards for real-time operating systems. Its real-time kernel specifications can be applied to any small–scale embedded systems. μITRON4.0 is implemented for numerous 8-bit, 16-bit and 32-bit Microcontroller Units (MCUs).

There are several basic characteristic of ITRON.
(1) Supporting multi-tasking
(2) Task scheduling using fixed/static priority
(3) Communication and synchronization mechanisms
(4) Real-time clock control
(5) Fully pre-emptible kernel hard real-time response

In this evaluation, the μITRON4.0 kernel implementation in [7] is used as the baseline. This kernel provides all functions and system calls in the standard profile defined in [6].

5.2 EDF Implementation

In the used ITRON system, when a task is invoked (released), it is originally inserted as a new task at the end/tail of the ready queue with the corresponding fixed priority. The scheduler will always select the head task of the ready queue with the highest priority which has activated tasks. Therefore, in a ready queue, it schedules tasks based on first-in-first-out (FIFO) rule.

The original queue handling (insert/sort) functions are all in the "*kernel_queue.c*" file. The "*_kernel_queue_insert_prev*" function handles inserting new tasks in the ready queue. If a sorting algorithm different from the original ITRON specification is used during the task insertion, a different priority scheduling algorithm can be implemented, e.g. RM or EDF algorithm.

Figure 5.1 presents the original insert function "*_kernel_queue_insert_prev*". The function adds the new task "*entry*" before the task pointed to by the "*queue*" pointer. The ready queue is double circular linked. If a new task is inserted before the head of the ready queue pointer ("*queue*"), then the new task will be the tail of the queue. In that case, the new task will be considered to be of lowest priority in the corresponding priority. If the new task were inserted differently from the ITRON specification, different scheduling algorithms would be implemented.

In order to implement EDF, the process of inserting tasks in the ready queue should be modified by changing the order of tasks. EDF was easily implemented by sorting the queue according to the absolute deadlines of the tasks. Then the system would schedule the task with the earliest deadline first.

The change was made by adding a new function to insert the tasks in the right order. Then, the new function was called instead of the original function throughout the kernel codes. The new function was conveniently placed in the same file as the original function, "*kernel_queue.c*".

In Figure 5.2, the new EDF insert function, "*_kernel_queue_insert_ADL*" searches in the main loop for a task with priority lower (later deadline) than

that of the new task. The new task is then inserted before the task found using the original "*_kernel_queue_insert_prev*" function.

```
void _kernel_queue_insert_prev ( _KERNEL_QUEUE * queue,
_KERNEL_QUEUE *entry )
    {
    entry -> prev = queue -> prev;
    entry -> next = queue;
    queue -> prev -> next = entry;
    queue -> prev = entry;
    }
```

Figure 5.1.Original insert function (FIFO)

```
void _kernel_queue_insert_ADL ( _KERNEL_QUEUE *queue,
_KERNEL_QUEUE *entry )
    {
    _KERNEL_QUEUE *ptr;
    INT ADL = entry -> self -> act_fifo.head -> ADL;

    // Search by ADL to find the position to insert in order
    for ( ptr = queue -> next; ptr != queue; ptr = ptr -> next )
        if ( ADL < ptr -> self -> act_fifo.head -> ADL )
            break;

    // insert
    _kernel_queue_insert_prev ( ptr, entry );
    }
```

Figure 5.2. EDF insert function (ADL sorted)

In the kernel, every time the task is released, the function "*iact_tsk(ID)*" is called. To implement EDF, this function was modified to initialize the absolute deadline as shown in Figure 5.3. The absolute deadline of each task is initialized to be the sum of both *_kernel_systim* and *RDL*. ("*RDL*" means the task's relative deadline, or period.) "*_kernel_systim"* is the system

current time at which the task is invoked.

```
act_cell -> ADL = _kernel_systim + tcb -> RDL;
```

Fig 5.3 EDF Absolute Deadline Initialization

5.3 Adaptive EDF Implementation

Before introducing how to implement the Adaptive EDF algorithm, several global variables are described. All the variables below are declared in the "*kernel_globals.c*" file.

(1) ID _kernel_trgttskID
A certain particular target task will be preferred by Adaptive EDF. This variable indicates the target task. This variable is set in the "*init_system (VP_INT)*" function which is the system initialization function.

(2) _KERNEL_TCB *_kernel_prevtsk
This is a pointer in TCB (Task Control Block) structure. This pointer recodes the previously-scheduled task. This information is used for a timer handling routine, one of managers of Adaptive EDF, to find whether the previously-scheduled task is the target task or not.

(3) TIME _kernel_target_start_time
This variable records the target task's (re-)starting time.

(4) TIME _kernel_target_exec_time
This variable is used to accumulate the execution time of target task.

(5) SYSTIM Vs
This variable represents the reciprocal of the server utilization (1/Us). It is set in the "*init_system (VP_INT)*" function.

Adaptive EDF basically follows the EDF algorithm. Adaptive EDF first calculates the absolute deadline of the target task using the estimated

execution time. Then, the task is inserted into the ready queue and scheduled using EDF.

The main code fragments of Adaptive EDF are shown in Figures 5.4 to 5.7. When the "*iact_tsk*" function is called on every releasing of the target task, the deadline of the target task is initialized as in Figure 5. The initial deadline is set to the sum of *_kernel_systim* (the current time of the system) and *Vs* (*1/Us*) as in equation 3. For other (not target) tasks, the deadline is initialized using the basic EDF.

```
if ( _kernel_trgttskID == tskid )
        act_cell -> ADL = _kernel_systim + Vs;
  else
     act_cell -> ADL = _kernel_systim + tcb -> RDL;
     act_cell -> next = (_KERNEL_ACT_CELL *) NULL;
```

Figure 5.4 Deadline initialization in Adaptive EDF algorithm

The variable for the execution time of the target task is initialized as in Figure 5.5.

```
if (_kernel_trgttskID == tskid) {
                    _kernel_target_exec_time     = 0;
                                          }
```

Figure 5.5 Initialization of the target task's execution time in Adaptive EDF

The scheduler function, *_kernel_sched (int)*, is called every time the ready queues are updated. One of the situations is on tick/timer handling. In this function, every tick timing, the execution time information and absolute deadline for the target task are managed if necessary as in Figure 5.6. If the previously running task is the target task, the cumulative execution time is incremented. At the same time, if the cumulative execution time reaches the tick period length (approximately 80% of the tick period), the corresponding absolute deadline should be updated by adding *Vs* according to the Adaptive EDF definition. Then the cumulative execution time information is reset to zero.

```
_kernel_iget_tim ( &tim );
if (_kernel_prevtsk -> tskid == _kernel_trgttskID){
        _kernel_target_exec_time += tim - _kernel_target_start_time;
                if ( _kernel_target_exec_time >= (TIME) (.8 *
(UINT)_KERNEL_TICK) ) {
                        _kernel_prevtsk -> act_fifo.head -> ADL += Vs;
                        _kernel_target_exec_time = 0;
                }
  }
```

Figure 5.6 Adaptive EDF execution time management.

In the last Adaptive EDF code fragment shown in Figure 5.7, on every task switching or tick timing, the (re)start time of the target task is set to the current cycle time (*tim*).

```
if (_kernel_schtsk -> tskid == _kernel_trgttskID) ) {
        _kernel_target_start_time = tim;
      }
```

Figure 5.7 Start time setting in Adaptive EDF

5.4 Retrospective Releasing Implementation

Retrospective releasing technique can be implemented by adding it to EDF or Adaptive EDF. The algorithm is shown in Figure 5.8.

```
if ( _kernel_trgttskID == tskid ) {
        max_dl = 0;
        vr = _kernel_systim;            // release time
    while ( 1 ) {
                dk = vr + Vs;

                if ( vr == dk_1 )
                        break;

                if ( vr == last_empty + 1 )
                        break;

         if ( max_dl < dl[vr-1] )
                max_dl = dl[vr-1];

         if ( dk <= max_dl )
                break;
         else
                vr = vr - 1;
                }
        act_cell -> ADL = dk;
        dk_1 =dk;
        }
```

Figure 5.8 Main code of Retrospective Releasing Technique

The variables used in this technique:

(1) INT dl [4096]

This array records the deadlines used in past tick slots. This is used to guarantee that the retrospective releasing technique never changes the past schedules, as described in the previous chapter.

(2) TIME last_empty

This variable records the tick number of the last empty slot encountered.

The code of Figure 5.8 is added in the "*iact_tsk.c*" file as part of the "*iact_tsk*" function. It has already explained in detail in chapter 4. The actual implementation is not different from this algorithm code except it includes few additional data structures and pointers.

Figure 5.9 shows code fragments for the deadline and empty slot recording. This part was introduced in the function "*_kernel_sched (int)*" of the "*kernel_sched.c*" file to run every task switching or tick. If the CPU is idle (*_kernel_schtsk=0*), the last empty is set as the current tick (*_kernel_systim*). Otherwise, it checks if the scheduled task is the same as the previously running task and if the currently running task is the operating system (*_kernel_runtsk=1*) (This corresponds that the "*_kernel_sched"* function is called on tick timing.). In that case it records the deadline of the newly scheduled task in the corresponding element of *dl[]*. On the other hand, in case that the newly scheduled task is different from the previously running task, it replaces the corresponding *dl[]* element with the new task's deadline value if the latter is later.

```
if ( _kernel_schtsk -> tskid == 0 )   //   this tick has an empty slot.
                last_empty = _kernel_systim; // tick time;

        else if ( (_kernel_schtsk -> tskid == _kernel_prevtsk -> tskid) &&
(_kernel_runtsk ->tskid == 1) )    // this case same task continue running
through this tick
        dl[_kernel_systim] = _kernel_schtsk -> act_fifo.head -> ADL;

    else if ( (_kernel_schtsk -> tskid != _kernel_prevtsk -> tskid) &&
(_kernel_schtsk -> act_fifo.head -> ADL > dl[_kernel_systim]) )   // this case
is task switch
                dl[_kernel_systim] = _kernel_schtsk -> act_fifo.head -> ADL;
```

Figure 5.9 The recording empty slots and previous deadlines

This implementation can be readily added to either EDF or Adaptive EDF without any additional modifications in their code fragments. (Precisely,

when it is added to the EDF version, two variables, *_kernel_trgttskID* and *Vs*, must be added.)

Chapter 6

Evaluation

6.1 Evaluation Environment

In this research, the existing CPU simulator is used for evaluating the scheduling algorithms. The CPU simulator executes binary executable codes in a cycle-based fashion. The target instruction set is SPARC version 8 [8]. Basically, the execution of one instruction takes one clock cycle. (Multiplication and division take several cycles.) The modeled microprocessor includes primary instruction/data caches of one cycle for hit and ten cycles for miss hit. In this evaluation, binary codes consisting of tasks' codes and OS codes are input to the simulator. Therefore, this evaluation enables accurate quantitative assessment of the whole system.

Five different methods are compared in this evaluation.

1. The ITRON original scheduling using a single priority, FIFO ready queue. This is called "baseline."
2. EDF model implementation
3. Adaptive EDF model implementation on top of the EDF model.
4. Retrospective Releasing model implementation on top of the EDF model.
5. Retrospective Releasing model implementation on top of the Adaptive EDF model.

The evaluation exposes improvement in both response time and jitter caused by the different scheduling models. The simulator runs each of the task sets (described in the next section) one time on every model. The simulator keeps running for 13 periods of the target periodic task.

The platform used in those experiments is a PC having Intel i3 Core, 1.33 GHz and 2GB main memory, which is running CYGWIN_NT-6.1-WOW64 1. 7.15 (0.260/5/3) i686 Cygwin, on top of Windows 7, Service Pack 1, 64 bits.

6.2 Task Sets

In this evaluation 31 task sets are used, each set consists of a number of periodic tasks. The name of the particular (target) task is always a task "C" in all task sets. Task sets have a different number of tasks to test different scenarios. The task sets are grouped by the total processor utilization (Up) of all the tasks in that set: 60%, 70%, 80% and 90%. Each group has 7 or 8 task sets.

Up is based on WCETs of all tasks in the following tables. The first task set group having 60% total utilization (U_p) is shown in tables 6.1 - 6.8. Each task has its WCET and period (T) in ticks. U_i is the task's utilization, which is shown only in Table 6.1 and can be calculated by dividing the WCET of the task by its period. This information is omitted for the other tables but can be calculated similarly.

Tables 6.9 - 6.16 show the tasks sets of processor utilization group with 70%. Tables 6.17 - 6.23 show the tasks sets of processor utilization group with 80%. Finally, tables 6.24 - 6.31 show the tasks sets of processor utilization group with 90%.

Task	WCET	T	Ui
A	9	72	0.125
B	8	80	0.1
C	3	24	0.125
D	2	10	0.2
E	5	100	0.05

Table 6.1 Up=0.6 NO.1

Task	WCET	T
A	9	72
B	2	16
C	5	40
D	3	24
E	8	80

Table 6.2 Up=0.6 NO.2

Task	WCET	T
A	9	72
B	2	16
C	3	12
D	8	160
E	5	100

Table 6.3 Up=0.6 NO.3

Task	WCET	T
A	9	72
B	6	48
C	9	45
D	3	20

Table 6.4 Up=0.6 NO.4

Task	WCET	T
A	9	90
B	5	100
C	6	120
D	4	10

Table 6.5 Up=0.6 NO.5

Task NO.	WCET	T
A	8	40
B	9	45
C	3	15
D	9	45

Table 6.6 Up=0.6 NO.6

Task NO.	WCET	T
A	3	10
B	8	80
C	9	90
D	9	90

Table 6.7 Up=0.6 NO.7

Task NO.	WCET	T
A	9	90
B	3	20
C	3	20
D	9	45

Table 6.8 Up=0.6 NO.8

.

Task	WCET	T
A	5	40
B	6	60
C	3	24
D	8	40
E	3	20

Table 6.9 Up=0.7 NO.1

Task	WCET	T
A	3	20
B	3	20
C	5	50
D	8	40
E	10	100

Table 6.10 Up=0.7 NO.2

Task	WCET	T
A	3	20
B	5	50
C	3	10
D	6	120
E	9	90

Table6.11 Up=0.7 NO.3

Task NO.	WCET	T
A	3	20
B	9	45
C	3	20
D	6	30

Table 6.12 Up=0.7 NO.4

Task NO.	WCET	T
A	5	40
B	9	72
C	8	40
D	9	36

Table 6.13 Up=0.7 NO.5

Task NO.	WCET	T
A	9	36
B	6	24
C	8	40

Table 6.14 Up=0.7 NO.6

Task NO.	WCET	T
A	9	36
B	3	20
C	9	30

Table 6.15 Up=0.7 NO.7

Task NO.	WCET	T
A	9	45
B	3	20
C	3	20
D	9	45

Table 6.16 Up=0.7 NO.8

.

Task	WCET	T
A	8	64
B	2	20
C	3	24
D	5	25
E	7	28

Table 6.17 Up=0.8 NO.1

Task	WCET	T
A	7	56
B	6	60
C	3	24
D	3	20
E	3	10

Table 6.18 Up=0.8 NO.2

Task	WCET	T
A	3	20
B	8	40
C	3	20
D	6	30
E	9	90

Table6.19 Up=0.8 NO.3

Task NO.	WCET	T
A	5	20
B	4	16
C	7	35
D	9	90

Table 6.20 Up=0.8 NO.4

Task NO.	WCET	T
A	5	20
B	3	20
C	3	10
D	9	90

Table 6.21 Up=0.8 NO.5

Task NO.	WCET	T
A	5	25
B	3	15
C	3	15
D	8	40

Table 6.22 Up=0.8 NO.6

Task NO.	WCET	T
A	5	20
B	3	20
C	3	20
D	9	36

Table 6.23 Up=0.8 NO.7

.

Task	WCET	T
A	10	50
B	3	15
C	3	15
D	5	25
E	6	60

Table 6.24 Up=0.9 NO.1

Task	WCET	T
A	10	40
B	3	20
C	3	10
D	4	40
E	5	50

Table 6.25 Up=0.9 NO.2

Task	WCET	T
A	5	40
B	3	24
C	3	10
D	6	60
E	4	16

Table 6.26 Up=0.9 NO.3

Task NO.	WCET	T
A	6	24
B	3	20
C	3	20
D	5	20
E	4	40

Table 6.27 Up=0.9 NO.4

Task NO.	WCET	T
A	5	50
B	3	29
C	3	20
D	6	30
E	3	10

Table 6.28 Up=0.9 NO.5

Task NO.	WCET	T
A	8	80
B	3	20
C	3	20
D	6	60
E	4	10

Table 6.29 Up=0.9 NO.6

Task NO.	WCET	T
A	9	90
B	3	20
C	3	20
D	8	32
E	6	24

Table 6.30 Up=0.9 NO.7

Task NO.	WCET	T
A	6	24
B	3	15
C	3	15
D	8	32

Table 6.31 Up=0.9 NO.8

6.3 Results

In this section, the result of evaluated algorithms will be considered.

Figure 6.2 shows the normalized average response time of all the task sets. The horizontal axis shows the 4 different groups of task sets, 60% utilization group first from the left. The vertical axis shows the obtained average normalized response time.

When running the task sets, the target periodic task has different response times in every period, which is averaged. The average response time is normalized, dividing it by the baseline model for the same task set. The normalized response times from all the task sets in the same processor utilization group is finally averaged to represent the group. Only the target task response times are considered in this evaluation.

The top green line represents the baseline scheduling model as a reference, which is always at 1. Among the other four scheduling methods, the EDF scheduling model always exhibits the longest average response time, not much better than the baseline model, corresponding to 95%-85% of the baseline response time. The two combinations of retrospective releasing (EDF+R and AEDF+R) have almost the same results as the Adaptive EDF (AEDF). AEDF clearly shows the fastest response time, on average half the response time of the original baseline model. At smallest, for 60% processor utilization group, AEDF is still 62% faster than the baseline model. Totally, its average response times are 38% - 62% shorter than the original baseline model.

The combination of retrospective releasing with EDF (EDF+R) shows similar performance to the best performance of AEDF. Only a little longer response time is observed in all processor utilization groups, especially in processor utilization group 60% and 90%, but not much. Totally, it exhibits

The combination of retrospective releasing with Adaptive EDF (AEDF+R) lies in between both AEDF and EDF+R. The results of AEDF+R seem to approach that of AEDF with very small difference. While its difference from EDF+R is a little more in all processor utilization groups. The reason why AEDF is a little worse than AEDF is that the execution overhead of the retrospective releasing algorithm influenced the overall performance.

Figure 6.3 shows the normalized worst response time of all the task sets. The same as figure 6.2 the top green line as a reference regards the baseline scheduling. Among the other four scheduling methods, the red line which is the EDF scheduling model always has the longest worst response time. To improve EDF by adding the new technique Retrospective Releasing (EDF +R), the worst response time is much shorter than before (purple line). The

blue line and the yellow line are respectively AEDF and AEDF + R scheduling models, they are same and exhibit the shortest worst response time.

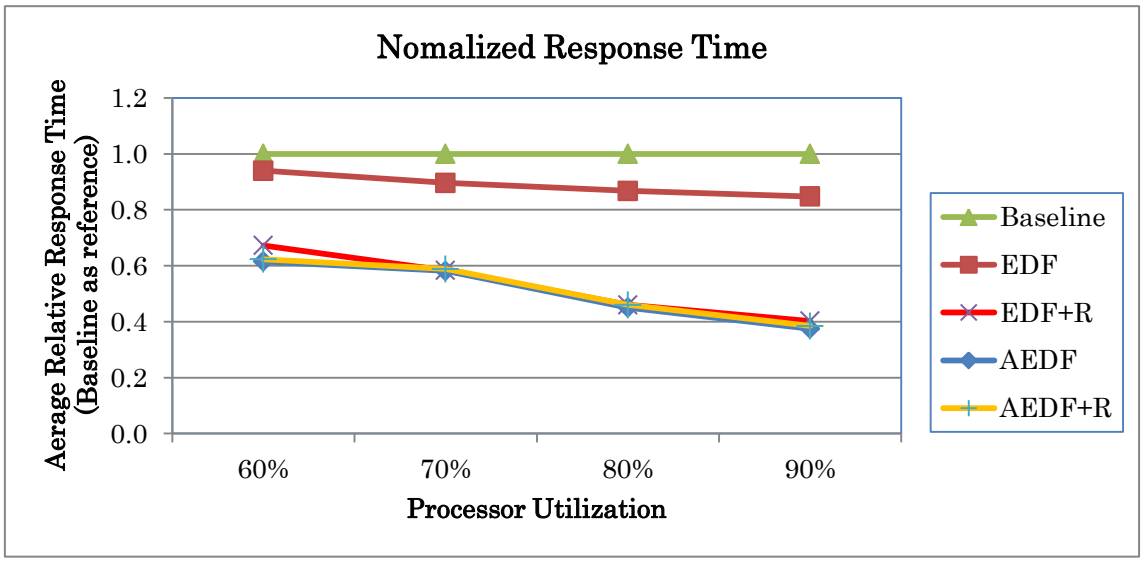

Figure 6.2 Normalized Average Response Time

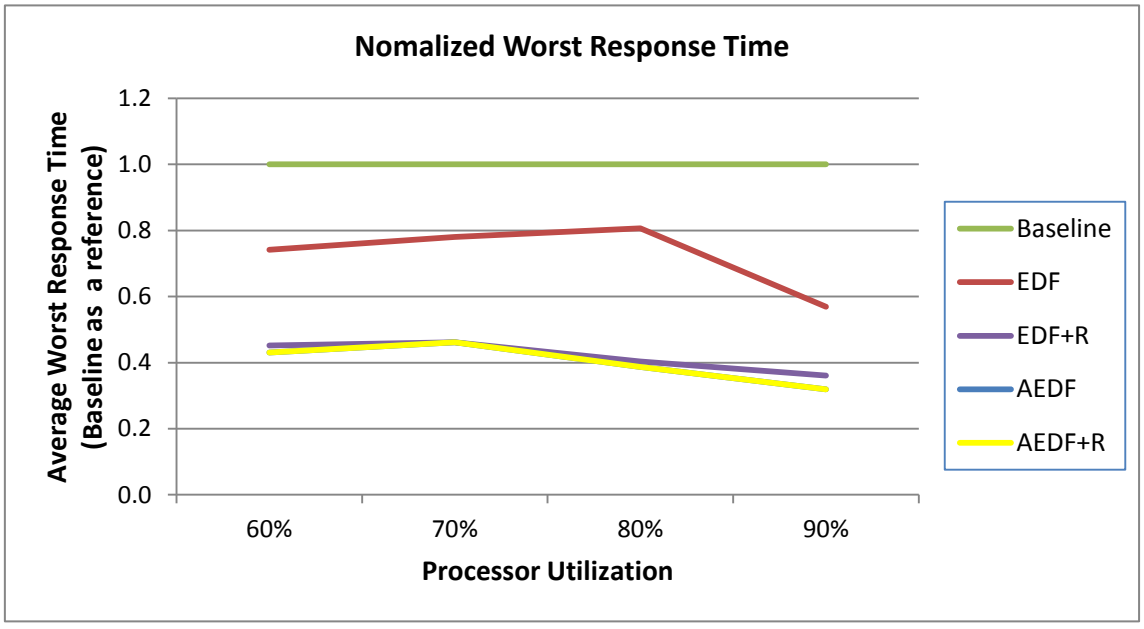

Figure 6.3 Normalized Average Response Time

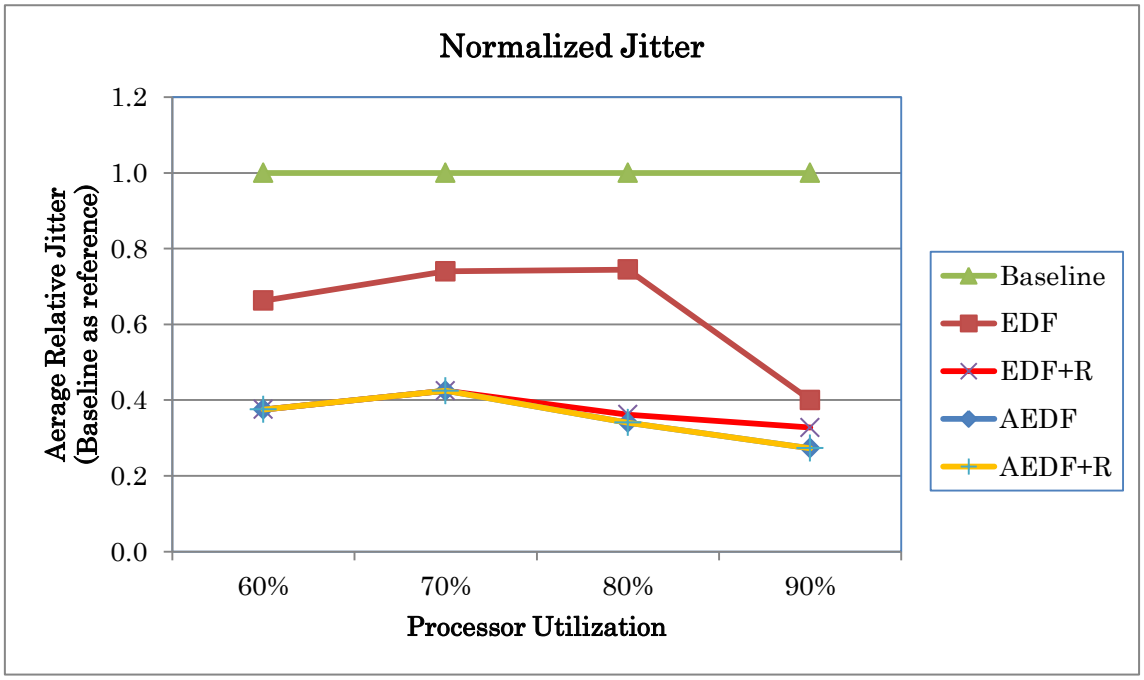

Figure 6.4 Normalized Average Absolute Jitter

Figure 6.4 shows the normalized absolute jitter of all the task sets. When executing each task set, the target task's behavior is observed to record its response times in all the 13 periods of the target task. The absolute jitter of the target task is the difference between its maximum and minimum response times. The jitter is normalized, dividing it by the baseline model for the same task set. The normalized jitter from all the task sets in the same processor utilization group is averaged to represent the group. Only the target task's jitters are considered in this evaluation.

The average absolute jitters of EDF are the worst compared to the other models, which correspond to 40% - 75% of the baseline jitters.

EDF+R has a similar tendency to AEDF. AEDF and AEDF+R are exactly identical. Both show minimum jitter, 27% - 43% of the original baseline model. EDF+R is only a little worse in the 80% and 90% processor utilization groups, which are 32% - 43% of the baseline.

Chapter 7

Conclusion and Future Work

In this research, the basic EDF scheduler was first implemented then Adaptive EDF was added. The new retrospective releasing technique was then implemented and added to both the EDF scheduler and the Adaptive EDF scheduler. The four scheduling methods were evaluated using the CPU simulator with the actual tasks' and OS's codes to test response times and jitters of the target important task.

Adaptive EDF was found to achieve the fastest response times and smallest jitters for the all the task sets used. Basic EDF had only a little improvement over the baseline and it was the worst results when compared with the other three advanced schedulers. Retrospective releasing combined with Adaptive EDF gave almost similar results to but a little worse results than Adaptive EDF. The results of combining the retrospective releasing and EDF were found worse than Adaptive EDF (whether with or without the retrospective releasing). The combination of the retrospective releasing with EDF is not much different than both Adaptive EDF schedulers.

Adaptive EDF was clearly more complicated than EDF or even the combination of EDF and retrospective releasing. The complexity of Adaptive EDF was further increased by adding retrospective releasing.

Obviously, the performances of combination of EDF and the retrospective releasing and both Adaptive EDF schedulers are not much different from each other. Thus, the scheduler with less complexity could be preferred for less scheduling overhead.

From the results, it can be concluded that the Adaptive EDF is effective and the retrospective releasing is effective only with EDF, not Adaptive EDF. As mentioned in the previous section, this is because the execution overhead of the retrospective releasing becomes an obstacle to further improvement.

Therefore, implementation of this technique with more light-weight complexity should be explored. In addition, more task sets from various benchmarks should be used to obtain widely acceptable results.

References

[1] K.Tanaka, “Adaptive EDF: Using predictive Execution Time,” ACM SIGBED Review, Vol.10, No.4, pp.41-44, 2013.

[2] K.Tanaka, “Improvement of Adaptive EDF,” ACM SIGBED Review, Vol.11, No.3, pp.40-43, 2014.

[3] G.C. Buttazzo, “Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications,” 3rd edition, Springer, 2011.

[4] C.L.Liu and J.W.Layland, “Scheduling Algorithms for Multiprogramming in Hard-Real-Time Environment,” Journal of the Association for Computing Machinery, Vol. 20, No. 1, pp. 46-61, 1973.

[5] M.Spuri and G.C.Buttazzo, “Efficient Aperiodic Service under Earliest Deadlne First Scheduling,” Proc. of Real-Time Systems Symposium, pp.2-11, 1994.

[6] ITRON Committee, TRON ASSOCIATION, “μITRON4.0 Specification Ver.4.00.00.”

[7] K.Tanaka, “Real-Time Operating System Kernel for Multithreaded Processor,” Proc. of Intl. Workshop on Innovative Architecture for Future Generation High-Performance Processors and Systems, pp.91-99, IEEE Computer Society, 2006.

[8] The SPARC Architecture Manual Version 8. SPARC International Inc., Prentice Hall, 1992.

[9] G.C.Buttazzo and F.Sensini, “Optimal Deadline Assignment for Scheduling Soft Aperiodic Tasks in Hard Real-Time Environments,” IEEE Trans. Computers, Vol.48, No.10, 1999.

[10] K.Tanaka, “Real-Time Scheduling for Reducing Jitters of Periodic Tasks,” Proc. of ISPJ Embedded Systems Symposium, pp.36-45, 2004.