Study on Acceleration of Real-Time Task Scheduling

Doan Duy

School of Information Science
Japan Advanced Institute of Science and Technology
Septemper, 2016

Master's Thesis

Study on Acceleration of Real-Time Task Scheduling

1410221 Doan Duy

Supervisor : Associate Professor Kiyofumi Tanaka
Main Examiner : Associate Professor Kiyofumi Tanaka
Examiners : Professor Mineo Kaneko
Professor Yasushi Inoguchi

School of Information Science
Japan Advanced Institute of Science and Technology

August, 2016

Acknowledgments

As the author of this research, I would like to express my gratefulness to the following people.

First, I would like to thank my parents. They always encourage me and are my spiritual pillar for everything, especially whenever I face difficulties. Being their hope and proudness is the motivation for me to adjust with the student life far from my home.

Next, I must thank my great supervisor Associate Professor Kiyofumi Tanaka. He is really a kindhearted, responsible, and professional instructor. During my studying period, he was very patient to explain for me once I was lacked of related knowledge. Whenever a difficulty arises, he timely gives me advice to solve it. Especially, under his supervising, I learn the lessons of how to set the goals and how to approach the solutions. Besides, Tanaka Sensei is so nice to support me to complete required official procedures related to the life including monthly reports, scholarchip applications, and even financial issues. I myself am very grateful for his advice and help. Without his best effort, I could never gain such achievement of this research.

Then I also need to thank my second supervisor, my advisor, and other members in the laboratory. They are always very nice to help me. As a foreigner, I respect their helps to adjust with the studying life at a new education environment, JAIST.

Finally, I need to thank my friends and many staffs at JAIST. The way that my friends share my time helps me enjoy a happy and warm life and reduces the homesick inside me. And the faculties are always willing to support me any time I need helps. Furthermore, I'm very thankful that they are always ready to help me, even on problems in daily life in Japan.

Last but not least, I would like to thank JAIST, and other related agents to give me such a chance to study at the School of Information Science and a chance to discover nature, culture, and food in Japan.

Thank you very much!

Doan Duy

Contents

This dissertation was prepared according to the curriculum for the Collaborative Education Program organized by Japan Advanced Institute of Science and Technology and Vietnam National University - Ho Chi Minh City.

List of Figures

List of Tables

Chapter 1

Introduction

In the era of modern technologies, real-time computing systems are quickly developed and play a substantial role in life and science. Real-time applications are employed in many areas such as industrial production lines, telecommunication, automotive, surgical operations, and even aerospace controls. As a result, real-time systems become more diverse and complicated with different types of criticalities of tasks. Therefore, these systems must be able to handle not only periodic tasks, but also aperiodic tasks, that is, tasks with irregular arrival times. The importance of real-time scheduling is hence growing.

Real-time systems are computational systems that must satisfy the requirements of value correctness and response time constraints. In applications above, time is the basic constraint to decide the quality of services provided by the computing systems. Under the requirement of time, the output values of the real-time systems must be not only correct in logic, but also in time when they are released. It means that the systems are actually meaningful only if they react to external events during their processing time. Any late reaction may become meaningless and even dangerous to the systems. That is also what the word *real* indicates.

In real-time embedded systems, there are many requirements to assess the effectiveness of a real-time scheduling algorithm. The most important requirements always considered by researchers are performance, response time, schedulability, and implementation complexity. It is, however, the fact that algorithms that are (semi-)optimal in terms of schedulability or response time usually have high complexity, which prevents actual systems from adopting them. Efficient real-time scheduling algorithms and corresponding hardware mechanisms are still aim and challenge to researchers working in the real-time embedded system area.

To meet an incredibly increasing number of applications, the time constraints of the real-time systems become more severe. In any cases, the overall goal of the real-time systems is to finish tasks correctly as soon as possible in time satisfaction. Thus, task scheduling occupies a key importance in the real-time systems. Task scheduling is now under the rules of timelines, load limitation, predictability, fault tolerance, and maintainability. In order to follow these rules, tasks are in fact classified into different types of hard real-time, soft real-time, aperiodic, and periodic ones. In the scope of this research, two aspects of task response time and runtime overhead are considered for aperiodic tasks.

Virtual release advancing [1][2] is one of the introduced techniques that can substantially improve the response time of the real-time task scheduling. The technique is developed based-on the original TBS (Total Bandwidth Server), which has a very good performance and a relatively simple implementation complexity [3], and EDF (Earliest Deadline First) algorithm. Although achieving a good response time, this technique generates not a small

scheduling overhead, which makes it difficult for these algorithms to be applied to future real-time systems that operate at high precision (or fine periods).

In this study, an enhanced virtual release advancing algorithm is proposed to alleviate the runtime overhead of the original virtual release advancing technique. The evaluations show that the time complexity of the proposed algorithm is low enough to be applied to precise systems. In addition, to accomplish the goal, a hardware mechanism is designed and implemented.

The evaluations are conducted based-on the ARM Cortex A9 instruction set [4][5] and the ITRON embedded operating system [6]. The Cortex-A9 processor is a high-performance processor which implements the ARMv7 architecture and runs 32-bit ARM instructions and 16-bit and 32-bit Thumb instructions [4]. Possessing highlight features, the Cortex-A9 processor is one of the most processors widely employed in actual embedded systems. This is the reason why the Cortex-A9 processor is chosen in the evaluation of the research where the practical applicability is necessary.

In the evaluations of the research, the Cortex-A9 processor is implemented on the Zynq7000 processor family on the FPGA ZedBoard, which is based on the Xilinx® All Programmable SoC (AP SoC) architecture [7]. This evaluation board is integrated with a dual-core ARM Cortex-A9 MPCore processing system and Xilinx programmable logic which are very flexible for both software and hardware evaluations of the proposed algorithm.

The evaluation environments are conducted on a real-time embedded operating system called the ITRON system. This is an industrial version of the TRON System, which is popularly used in millions of electronic devices and embedded systems [6]. Since this research takes place in the real-time system, an actual real-time operating system is the most important consideration when conducting the environment. Thanks to its actual use in many different fields and places, the ITRON System is a very suitable environment for this study. Furthermore, the ITRON System has open limitation of scheduling policy, illustrated in Figure 1.1, which allows users flexibly and conveniently to modify and replace by new strategies and algorithms. This is obviously the essential condition for evaluating the proposed algorithm of the research.

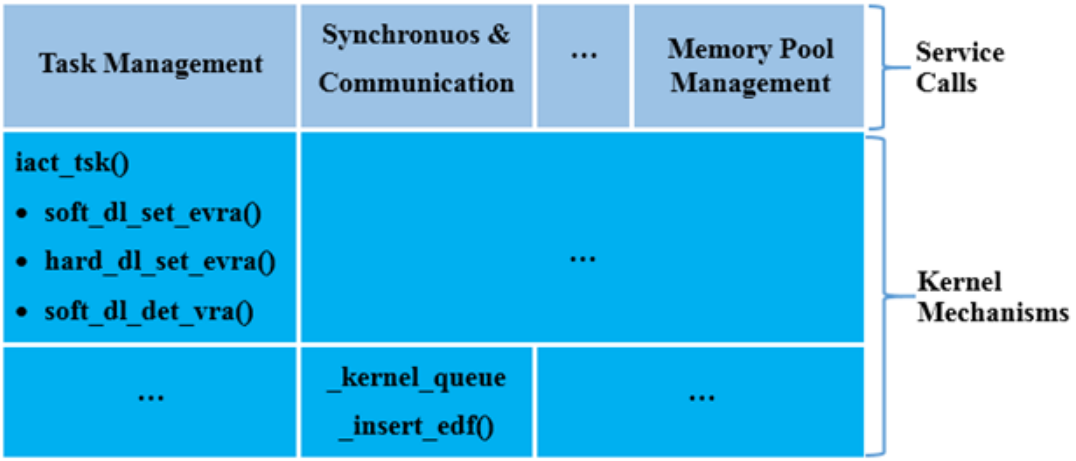

Figure 1.1: ITRON kernel structure

Chapter 2

Related works

As introduced in **Chapter 1**, this research is to mitigate the runtime overhead of the virtual release advancing technique. And similar to the original one, the proposed algorithm is also conducted based-on the TBS/EDF server. Therefore, it is necessary to first present the TBS/EDF server and virtual release advancing technique as related works before taking account of the proposed algorithm.

2.1 TBS/EDF Server

The TBS/EDF Server is introduced as one of the resource reservation methods and is also a scheduling algorithm under the EDF control [8]. This is a scheduling approach for both periodic and aperiodic tasks. In the TBS/EDF Server, the periodic tasks are assumed to be periodically active and have the relative deadline equal to their periods, while the aperiodic tasks are released irregularly to the system. When entering the system, aperiodic tasks are assigned absolute deadlines which are calculated by the following formula [8]:

$$d_k = max(r_k, d_{k-1}) + \frac{C_k}{U_s} \tag{2.1}$$

This is the original formula of the TBS/EDF Server, in which k means the k-th aperiodic task, d_k and r_k are the absolute deadline and the release (arrival) time of the target task, d_{k-1} is the absolute deadline of the k-1-th (previous) task, C_k is the worst-case execution time of the k-th task, and U_k is the bandwidth of the server. U_s indicates the total processor utilization that the server probably occupies to execute the task. Given U_p is the total processor utilization for periodic tasks. It is proved that a task set (includes both periodic and aperiodic tasks) is theoretically schedulable if and only if $U_s + U_p \leq 1$ [3]. Moreover, using the term $max(r_k, d_{k-1})$ in Formula 2.1 assures that there is no overlapping occurring between successive aperiodic tasks [2].

The original TBS/EDF Server is then improved to give a beneficial deadline for the subsequent aperiodic tasks by using a so-called resource reclaiming method. This is a kind of slack reclaiming method [9], where the absolute deadline is recalculated by using the actually elapsed execution time when the task finishes. Under the Formula 2.1, the recalculated deadline may lead an earlier deadline for the subsequent aperiodic task and then an earlier scheduling.

By using resource reclaiming method, the absolute deadline for the k-th aperiodic task is calculated as the Formula 2.2[9]:

$$d'_k = \bar{r_k} + \frac{C_k}{U_s} \tag{2.2}$$

Where $\bar{r_k}$ is determined by Formula 2.3 [9].

$$\bar{r_k} = max(r_k, \bar{d_{k-1}}, f_{k-1}) \tag{2.3}$$

Formula 2.3 shows that $\bar{r_k}$ is the maximum value among the arrival time, the recalculated deadline ($\bar{d_{k-1}}$), and the originating finishing time (f_{k-1}) of the previous aperiodic task. When the k-1-th aperiodic task finishes, the deadline is recalculated by Formula 2.4 that includes the actual elapsed execution time ($\bar{C_{k-1}}$) of the task [9].

$$\bar{d_{k-1}} = \bar{r_{k-1}} + \frac{\bar{C_{k-1}}}{U_s} \tag{2.4}$$

These basic formulas and methods will be implemented in the involved algorithms, the original virtual release advancing and the enhanced virtual release advancing, in this research.

2.2 Virtual release advancing

2.2.1 Algorithm of virtual release advancing

Virtual release advancing (VRA) is the original algorithm of the research. This is a technique based-on the TBS Server and EDF algorithm. The technique is aimed to obtain an earlier deadline by virtually and retroactively moving the release time backward to the past while not changing the past schedule [3]. Idea of VRA originates from Formula 2.1 that allows an earlier deadline from an earlier release time. The earlier deadline then leads to an earlier scheduling for the target task under the EDF algorithm.

Figure 2.1 shows an example of the technique with two periodic tasks. The first periodic task τ_1 has period $T_1 = 3$ and execution time $C_1 = 1$. Similarly, the second periodic task τ_2 has $T_2 = 6$ and $C_2 = 3$. The processor utilization by the periodic tasks is $U_p = 1/3 + 1/2 = 5/6$ and then the bandwidth of TBS server is $U_s = 1 - 5/6 = 1/6$. It is assumed that there is an aperiodic task entering the system at $t = 8$ with its execution time of 1. According to the original TBS scheduling, the aperiodic task is assigned an absolute deadline of $t = 14$, then finishes at $t = 12$ with response time of 4.

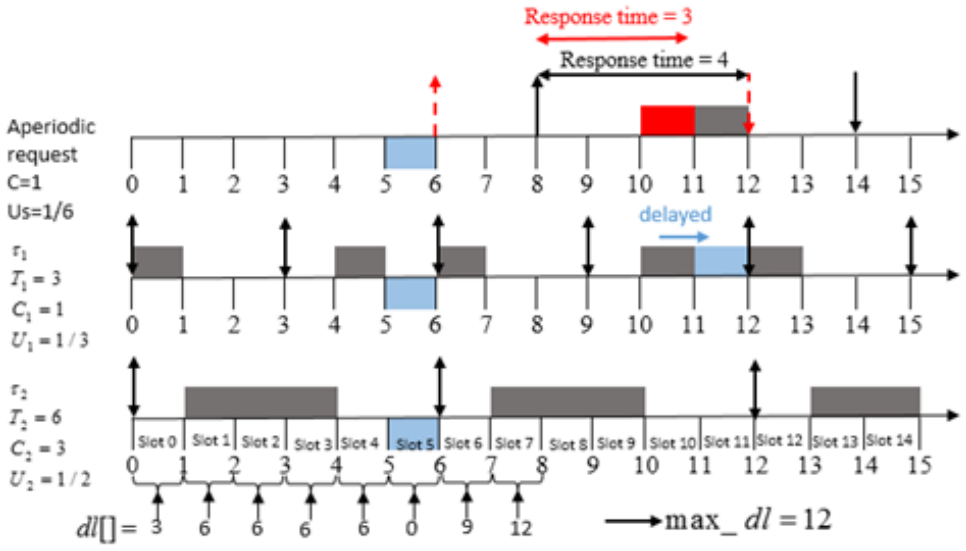

Figure 2.1: Example of virtual release advancing

In this case, by applying the virtual release advancing technique, a virtual release time, which is drawn by a red, upward dashed arrow, for the aperiodic task is set up at $t = 6$. With this new release time, the deadline calculation in TBS would give an absolute deadline at $t = 12$, two ticks earlier than the original deadline. This new deadline, which is highlighted by a red, downward dashed arrow, is the same with that of the third instance of τ_1. Since the aperiodic tasks are more preemptive than periodic tasks by assumption, the target task has a chance to be executed ahead of the periodic instance and finishes sooner at $t = 11$ with the response time of 3.

Algorithm 1 Virtual Release Advancing

```
1: max_dl ⇐ 0 /* Initialization */
2: vr_k ⇐ r_k
3: while TRUE do
4:    /* Reaching the limit of d_{k-1} */
5:    if vr_k ≤ d_{k-1} then
6:       vr_k ⇐ d_{k-1}
7:       d_k ⇐ vr_k + C_k/U_s
8:       break
9:    end if
10:   d_k ⇐ vr_k + C_k/U_s
11:   /* Reaching the limit of empty slot */
12:   if vr_k = last_empty + 1 then
13:      break
14:   end if
15:   if max_dl < dl[vr_k − 1] then
16:      max_dl ⇐ dl[vr_k − 1]
17:   end if
18:   /* Reaching the limit of prev-used max deadline */
19:   if d_k ≤ max_dl then
20:      break
21:   else
22:      /* Release advancing by one time slot */
23:      vr_k ⇐ vr_k − 1
24:   end if
25: end while
```

Figure 2.2: The algorithm of virtual release advancing

Figure 2.2 depicts the main algorithm of virtual release advancing. In the virtual release advancing technique, an earlier virtual release time for the target task is probably introduced based-on three factors: previous aperiodic task's deadline, last empty slot, and maximum previously-used deadline [3]. These factors are known as the limits of how long the release time can be advanced. It is stated that moving release time backward over any one of these limits is ineffective or leads to changing the past schedule.

The first limit is the previous aperiodic task's deadline. Since this technique follows the Formula 2.1 of the TBS server, the larger one between the previous aperiodic task's deadline and the target task's release time is used on determining the absolute deadline for the target task [3]. Moving release time backward over this limits, therefore, has no effects.

The second limit is the last empty slot. An empty slot is defined as the time slot which is not assisged to any task [3]. In the example in Figure 2.1, slot 5 is an empty slot. If the release time is advanced over an empty slot, the slot should be spent by the task. This action can cause the past schedule to be changed. Therefore, the advancing in this case is not eligible.

The final limit is the maximum previously-used deadline. Used slots are time slots

with an associated deadline of some task that is assigned to the slots [3]. If task's virtual deadline calculated with the advanced release time is earlier than the associated deadline of the slot, the slot must be spent for the task. This means the past schedule is changed. The advanced release time in this case is not allowable.

Following the algorithm, the output values are the virtual release time and virtual absolute deadline for the target task. The process of virtual release advancing starts with initiating the virtual release time, vr_k, at the task's actual request time. This process continues to an advancing loop of sequentially checking three above-mentioned factors. The previous aperiodic task's deadline is first dealt with from line 4 to line 9. If it is passed the previous task's deadline, the virtual absolute deadline is updated at line 10. Then the virtual absolute deadline is compared with the tick time just after the last empty slot. Continuously, the virtual absolute deadline is measured at line 19 with the maximum previously-used deadline that has been updated from line 15 to line 17. After all of limits are passed, the virtual release time is moved backward to the past by one slot and the loop repeats. Since these factors are limits of advancing, an advanced release time earlier or equal to one of factors can stop the advancing loop and then the output values are probably obtained.

2.2.2 The problem of runtime overhead in virtual release advancing

Although solidly improving the response time of the target task, the virtual release advancing technique has problems with its high time complexity. Due to the high time complexity, the algorithm is seemly not ready to be applied to precise systems. This restriction of the algorithm originates and motivates the research for an enhanced algorithm which can effectively reduce the runtime overhead.

As mentioned above, the goal of the virtual release advancing is to obtain as earlier virtual release time as possible than the actual one. To achieve this goal, the technique checks limit factors slot by slot from the target task's arrival time (r_k) backward to the past. A loop procedure is implemented for this checking.

The algorithm increases runtime overhead as follows. Firstly, the algorithm checks two factors, the previous deadline and the last empty slot, in every iteration. However, there are actually only one previous deadline and one last empty slot. Thus, it is desirable to check only one time for each factor. Secondly, consecutive slots spent by the same instance of tasks are checked one by one. It is obviously unnecessary since these slots have the same associated deadline. Once again, it is also desirable to check only the representative one of the slots.

Repeat of these unexpected checks is the main cause of the high time complexity of the original algorithm. The proposed algorithm of the research coming in the following chapter is concentratedly lessening the checking repeats so that the time complexity may be reduced.

Chapter 3

Enhanced virtual release advancing

We have analyzed the original virtual release advancing on the runtime overhead problem and then brought out the goal of the research. In order to reduce runtime overhead, the proposed algorithm, also known as an enhanced virtual release advancing (EVRA), introduces techniques to reject repeats while keeping the output values the same with those of the original algorithm.

The EVRA is coming in details in this chapter. The chapter is structured of three sections: first, introduction of definitions using in the algorithm; second, the analyses of the EVRA algorithm; and third, the enhanced points of the proposed algorithm.

3.1 Definitions of EVRA

3.1.1 Boundary deadlines

In the EVRA algorithm, boundary deadlines are known as the limits of virtual deadline advancing. There are three boundary deadlines first defined in this algorithm.

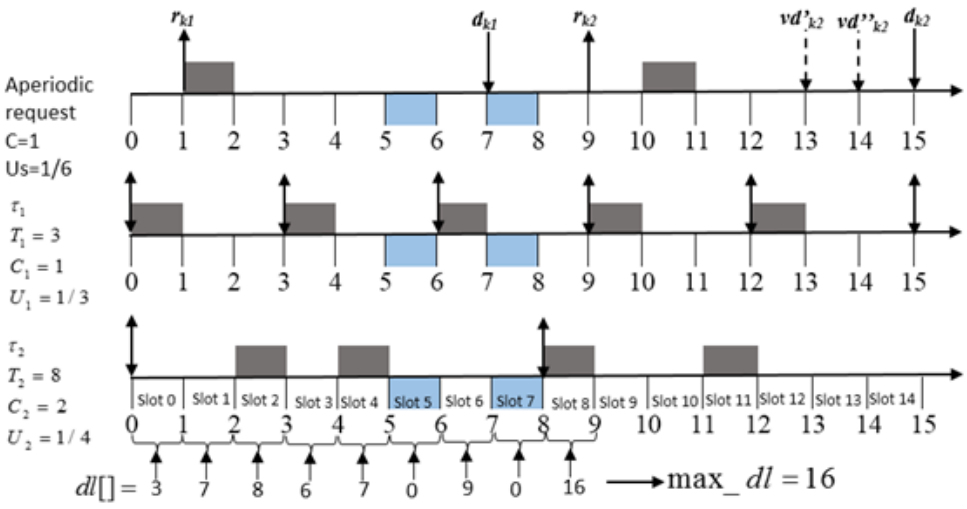

Figure 3.1: Example of boundary deadlines

As mentioned in Chapter 2, in the original virtual release advancing algorithm there are three limits consisting of previous aperiodic task's deadline, last empty slot, and maximum previously-used deadline. The first two limits are directly compared to the tentative virtual release time in each loop iteration. It is different from the proposed algorithm where these two factors are converted to the corresponding deadlines to be

dealt with outside of the advancing loop. For example in Figure 3.1, let's consider the second aperiodic task arriving at $t = 9$ with $C_{k2} = 1$. The previous aperiodic task's deadline is at $t = 7$ and the last empty slot is slot 7.

Using the previous deadline ($t = 7$) and the tick time ($t = 8$) just after the empty slot, two corresponding deadlines (vd'_{k2} and vd''_{k2}) for the target task are calculated to be at $t = 13$ and $t = 14$, respectively. These two deadlines are considered as two boundary deadlines of the algorithm. The last boundary deadline is related to the check-bounding slot. It is coming in the next section. In order to satisfy the limits of the algorithm, the expected virtual deadline for the target task hence has to be greater than or equal to all of the boundary deadlines.

3.1.2 Check-bounding slot

Given τ_{max} is the periodic task having the maximum period T_{max}, and ls_{max} is the starting time of the last instance of τ_{max}. Under the EDF algorithm, the associated deadline of ls_{max}-th slot is greater than or equal to the associated deadline of slots before it. That is, we have:

$$\forall x < ls_{max} \Rightarrow dl[x] \leq dl[ls_{max}] \tag{3.1}$$

Where $dl[x]$ and $dl[ls_{max}]$ are the associated deadlines of the x-th slot and the ls_{max}-th slot, respectively.

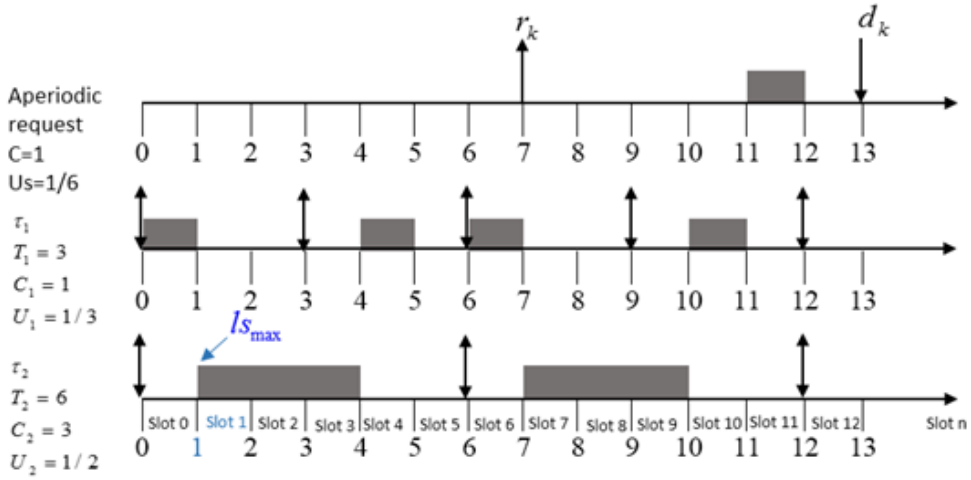

Figure 3.2: Example of check-bounding slot (ls_{max})

Figure 3.2 shows an illustration for the check-bounding slot. In this example, τ_2 corresponds to τ_{max} and ls_{max} is at $t = 1$. Reminding that the maximum previously-used deadline is the maximum associated deadline of slots from the r_k backward to the past. The Equation 3.1 therefore allows an inference that the maximum previously-used deadline can affirmatively be determined among slots from r_k backward to ls_{max}. Only the slots from r_k backward to ls_{max} are to be scanned ones when the algorithm is done under the limit of the maximum previously-used deadline. ls_{max} is beneficially chosen as the check-bounding slot and the associated deadline of this slot is considered as the last boundary deadline of the proposed algorithm.

3.1.3 Representative slots of instances

As discussed in Chapter 2, checking all of slots of the same instance one by one is unnecessary because these slots have the same associated deadline. Instead, in the proposed

algorithm the first slot of each instance is selected as the representative slot. In Figure 3.3, for instance, the original VRA algorithm checks all slots from $t = 4$ backward to $t = 0$, that is, slot 3, 2, 1, and 0. However, the enhanced algorithm by selecting the representatives will check only slot 1 and 0 for advancing. It is obviously more effective in terms of the check loop count and memory usage. In the context when tasks' execution may be separated into several portions by preemption, each portion is considered as an instance. By combining with the definition of the check-bounding slot above (only instances from r_k (target tasks arrival time) to ls_{max} are involved in the algorithm's execution) only the slot 1 is determinately checked for the first instance of τ_2 in Figure 3.3.

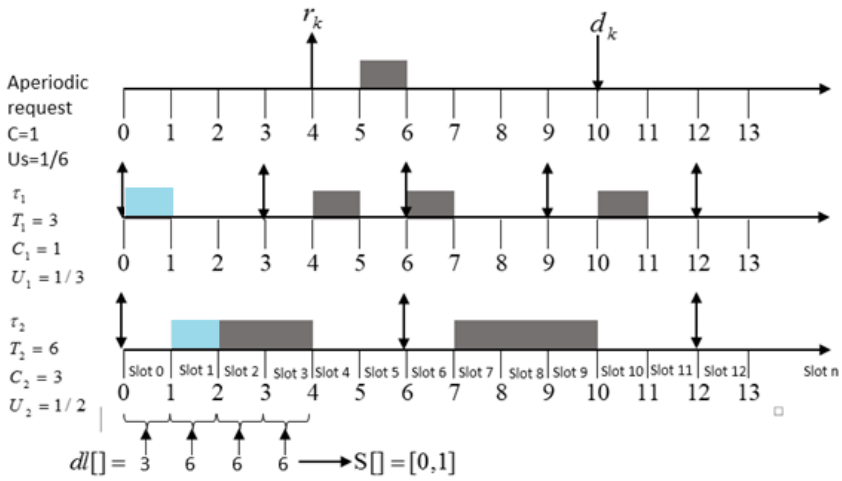

Figure 3.3: Example of representative slot of instances

3.2 EVRA algorithm

The whole EVRA algorithm is shown in Figure 3.4. In the algorithm, r_k and vr_k are the real and virtual release times of the target aperiodic task, respectively. vd_k is the expected virtual deadline. C_k and U_s are task's execution time and the bandwidth of the TBS server. d_{k-1} is the deadline of the $k-1$-th (previous) aperiodic task. For other variables, $last\_empty$ is the slot number of the last empty slot; n and ls_{max} are the number of released instances and the starting time of the last instance of τ_{max}, respectively. $dl[]$ and $S[]$ are the arrays saving used slots' associated deadline and released instances' starting time. Particularly, $d[m]$ is the associated deadlines of the m-th slot and $S[n]$ is the starting time of the n-th instance.

Then, three boundary deadlines (vd'_k, vd''_k, vd'''_k) defined in Section 3.1 are calculated corresponding to the previous deadline, last empty slot, and last starting time of τ_{max}. When advancing to the past, max_dl holds the maximum associated deadline of traced $S[n]$ elements. Under the mentioned condition of limits how long the release time can be advanced, the expected virtual deadline vd_k cannot be earlier than anyone among vd'_k, vd''_k, vd'''_k and max_dl.

The target output of the proposed algorithm is the virtual deadline for the target task. The proposed algorithm inherits the idea of advancing from the original algorithm. That is, the virtual deadline is compared with the limits and then moved backward to the past if the limits are satisfied. At the beginning of advancing, vd_k is initialized to the original deadline using the TBS's formula 2.1. The boundary deadline vd'_k corresponding to the previous deadline is calculated and checked first from line 4 to line 9. When the

Algorithm 1: The EVRA algorithm

1: /*Definition*/
2: $vd_k = r_k + C_k/U_s$
3: /*Limit of k-1-th deadline*/
4: $vr'_k = d_{k-1}$
5: $vd'_k = vr'_k + C_k/U_s$
6: if $r_k \leq vr'_k$ then
7: $vd_k = vd'_k$
8: Goto End
9: endif
10: /*Limit of last empty slot*/
11: $vr''_k = last\_empty + 1$
12: $vd''_k = vr''_k + C_k/U_s$
13: /*Limit of previously-used slots*/
14: $i = n - 1$
15: $max\_dl = 0$
16: $vd'''_k = dl[ls_{max}]$
17: $bound = max(vd'_k, vd''_k, vd'''_k)$
18: While $vd_k > bound$ do
19: $vr_k = S[i]$
20: if $max\_dl < dl[vr_k]$ then
21: $max\_dl = dl[vr_k]$
22: endif
23: if $vd_k \leq max\_dl$ then
24: *break*
25: endif
26: $vd_k = vr_k + C_k/U_s$
27: if $vd_k \leq max\_dl$ then
28: $vd_k = max\_dl$
29: break
30: else
31: $i = i - 1$
32: endif
33: endwhile
34: /*Calculating the advanced release time*/
35: Label: End

Figure 3.4: The enhanced virtual release advancing

virtual deadline is earlier than or equal to vd'_k, the virtual deadline is set to vd'_k, and the algorithm finishes. This check is done only one time. Then, the boundary deadlines of the last empty and ls_{max}-th slots (vd''_k and vd'''_k) are calculated at line 12 and line 16. All of the boundary deadlines are combined to form a variable *bound* at line 17. Variable *bound* plays a role as the overall check-bounding deadline of the algorithm.

Next, to satisfy the limit of previously-used maximum deadline, for each of the traced $S[n]$ elements, the corresponding vd_k is compared to max_dl from line 18 to line 33. The expected virtual deadline is obtained after the loop execution under the condition of *bound* finishes.

3.3 Enhanced points of the algorithm

In Chapter 2, main sources of considerable runtime overhead in the original algorithm are repeated checking of limits. Therefore, enhanced points of the proposed algorithm are aimed to decrease these sources.

The first improved point in the new algorithm is the way to deal with the previous task's deadline and the last empty slot. As shown in the algorithm in Figure 3.4, the effect of these two factors are processed through vd'_k and vd''_k outside of the loop. This helps to reduce the influence of repeated checking.

The second main point of the algorithm is that checking is done for representative slots. In the original algorithm, to achieve the limit of the maximum previously-used deadline, used slots have to be involved one by one. Therefore, the associated deadlines of used slots have to be recorded by array $dl[m]$ sized by m. In a different way, the new algorithm uses only the first slots of instances recorded by array $S[n]$ sized by n. Naturally, n (the number of released instances) is smaller than m (the number of past slots) unless all of instances are executed in exactly one slot. Using $S[n]$, instead of $dl[m]$, has two advantages. Firstly, the number of times that array $S[n]$ is checked is significantly less than that on array $dl[m]$. It means that the runtime overheads caused by the loop execution are lowered. Secondly, since array $S[n]$ is smaller, the time and space costs of saving it are more efficient.

The last enhanced point is introduction of check-bounding slot or the last starting time ls_{max} of the task having the maximum period. In the original algorithm, the advancing may theoretically recur backward to the past without any limitations. However, checking slots before ls_{max} is actually unmeaning since the associated deadline of ls_{max}-th slot is always greater than or equal to those of past slots under EDF-based scheduling. The definition of ls_{max}-th slot helps to determine how long past the advancing is performed. The worst (or longest) case of advancing is T_{max} where T_{max} is the period of τ_{max}, respectively. The worst case of advancing can be determined as following.

The worst cast of advancing can be determined under the statement that it exists at least one of the following things among T_{max} consecutive slots:

(1) An empty slot

(2) An instance of τ_{max}

Now, lets continue to prove this statement. Given U_p^n is the processor utilization of periodic tasks of n consecutive slots. U_p^n is defined by following equation:

$$U_p^n = \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_k}{T_k} \leq 1 \tag{3.2}$$

Where C, T, and k are the computation time, period, and the number of periodic tasks executed during these n slots. Note that each task is involved at most one time in Equation 3.2. Since the total processor utilization is actually less than 1, U_p^n in Equation 3.2 is always apparently less than 1. Then, the processor utilization of periodic tasks of T_{max} consecutive slots may calculated as:

In case of including τ_{max}:

$$U_p^{T_{max}} = \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_{k-1}}{T_{k-1}} + \frac{C_{max}}{T_{max}} \leq 1 \tag{3.3}$$

In case of being without τ_{max}:

$$U_p^{T_{max}} = \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_k}{T_k} \leq 1 \tag{3.4}$$

The statement can be here proved by contradiction. Assuming that there exists T_{max} consecutive slots in which there is no any empty slot or instance of τ_{max}. The processor utilization of periodic tasks is then calculated by Equation 3.4. On one hand, since $T_{max} \geq T_1, T_2, ..., T_k$, we have:

$$U_p^{T_{max}} = \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_k}{T_k} \geq \frac{C_1}{T_{max}} + \frac{C_2}{T_{max}} + ... + \frac{C_k}{T_{max}} = \frac{C_1 + C_2 + ... + C_k}{T_{max}} \quad (3.5)$$

On the other hand, due to no empty slot, all of T_{max} consecutive slots are assigned to periodic tasks excepting τ_{max}. So we have:

$$C_1 + C_2 + ... + C_k = T_{max} \quad (3.6)$$

By substituting Equation 3.6 into Equation 3.5, we have:

$$U_p^{T_{max}} = \frac{C_1}{T_1} + \frac{C_2}{T_2} + ... + \frac{C_k}{T_k} \geq 1 \quad (3.7)$$

This contradict with the definition of processor utilization. Therefore, it eventually states that there is at least one empty slot or instance of τ_{max} among T_{max} consecutive slots.

As the boundary checking, empty slot and last instance of τ_{max} may become the limit of advancing. In the other word, the advancing is limited in T_{max} slots from the release time backward to the past.

Chapter 4

Implementation of the proposed algorithm

4.1 Introduction of basic implementation

4.1.1 Runtime overhead in actual real-time system

There are critical problems related to the runtime overhead when the proposed algorithm is implemented into the actual real-time system, particularly on the ITRON System. In the algorithms presented in previous chapters, there are theoretical assumptions that are really different from the real systems. These differences actually affect to the runtime overhead calculation and needs to be considered.

The first problem relates to the system's time unit. That is, tick is exactly the smallest time unit observed in the system. In the description of the proposed algorithm, tasks is started and then completed exactly at system ticks. This is, however, different from the actual context of the system where tasks' starting and finishing can occur at any time. Figure 4.1 shows an example of tasks' execution off a ticks. In this example, the line labeled tick indicates the time ticks; τ_1 and τ_2 are two periodic tasks. These tasks can be executed in precise ticks. The empty rooms of time from a task's finishing time to another's starting time is spent for the system's scheduling. Therefore, overheads of task activation and task switching occur not only at a tick timing, but also at any timing. These overheads must be considered to achieve practical real-time task scheduling.

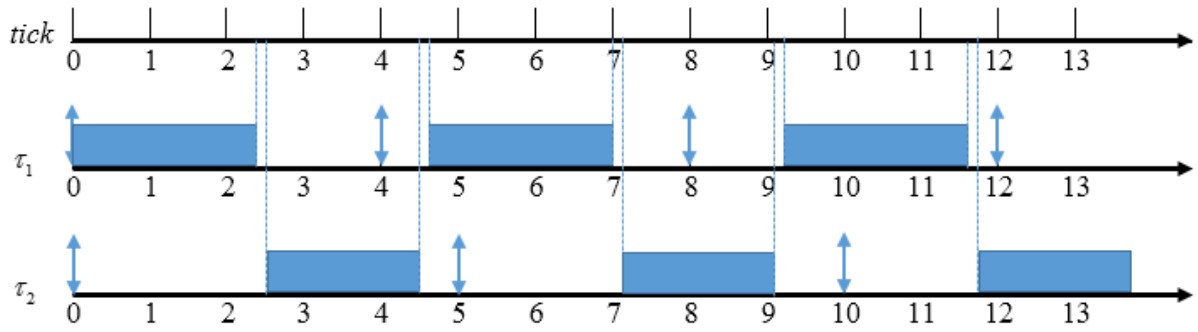

Figure 4.1: Tasks' off-tick execution time in actual real-time systems

The second problem is that in the theoretical explanation of the algorithms, all of the required system parameters are usually ready before executing scheduling algorithm. On the other words, the cost of storing and loading these parameters is temporarily ignored.

Nevertheless, this cost may obviously affect to the runtime overhead of the real system. Thus, it needs to be included in the overhead calculation.

4.1.2 System structures and organizations

Before introducing the algorithm implementations, there are some important system's structure and organization necessary to be explained at first.

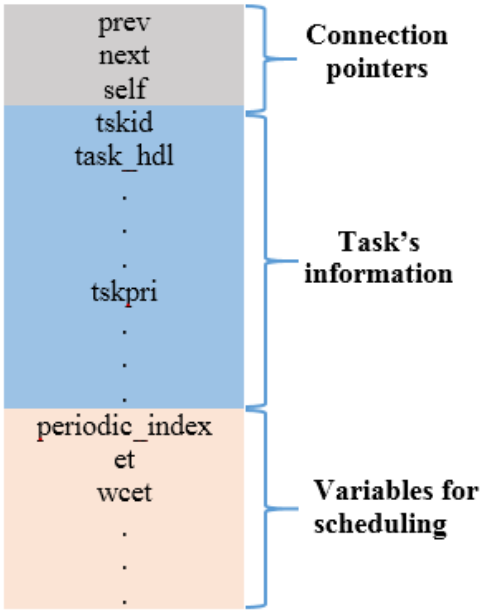

Figure 4.2: Structure of _KERNEL_TCB [10]

The first important structure is called _KERNEL_TCB (kernel task control block) which the operating system maintains to manage each task. A _KERNEL_TCB, as illustrated in Figure 4.2, has member variables including task's attribute, priority, and state. _KERNEL_TCBs are connected to each other by a structure of pointers: "prev", "next", and "self" [10]. The first and second pointers, "prev" and "next", always point to the previous and next tasks. In the case that anyone of the previous task and the next task is not available, the corresponding element of the queue structure will be refered instead. The other pointer, "self", is employed to make a reference to the other member variables of _KERNEL_TCB when the system wants to access them. In addition, in order to support the scheduling algorithm, added variables are used for task type, execution time, worst-case execution time, period, and so on.

The next structure of task management is the ready queue _KERNEL_READY_QUEUE [10] as depicted in the Figure 4.3. _KERNEL_READY_QUEUE is an array of double linked lists. Each linked list is a structure of _KERNEL_TCBs which are connected by above-mentioned pointers "prev" and "next". The elements or linked lists of ready queue are indexed correspondingly to their priorities from 0 to the maximum number of priority. The higher the index is, the lower the priority of the list becomes. In this way, priority 0 and priority 1 are preserved for system tasks, that is, tasks serve the operating system. The others priorities are for application tasks. As having higher priorities, the system tasks can preempt any other application tasks. In the ready queue, each element is the head of each list corresponding to the priorities. The elements' "next" points to the top address of the first _KERNEL_TCB task to be executed. If there is no task to be executed, next will point to the element's own top address. Similarly, the elements' "prev" points the last _KERNEL_TCB task to be executed. If there is no task to be executed, "prev"

will also point to the element's own top address. In the implementations of algorithm in this research, the priority 2 is assigned to all of application tasks including both periodic and aperiodic tasks.

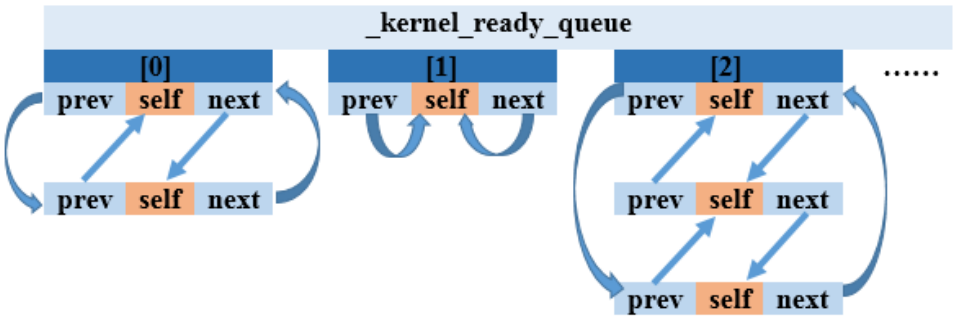

Figure 4.3: The structure of kernel ready queue

Another basic structure is called _KERNEL_ACT_CELL [10]. As shown in Figure 4.4, _KERNEL_ACT_CELL has two main parts: a pointer to another _KERNEL_ACT_CELL structure, "next", and a value of time, "a_dl". In the system, a task may be activated at multiple times. For each time of activating, the task is assigned with an absolute deadline. The _KERNEL_ACT_CELL structure is used to refer to these activations on the way that "a_dl" is the absolute deadline and that "next" points the absolute deadline of the next activation.

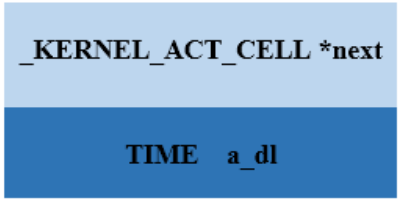

Figure 4.4: _KERNEL_ACT_CELL structure for task's absolute deadline

Since the system scheduler is under the EDF algorithm, that is, the task with the earlier deadline will be selected to execute. Therefore, at each list of the kernel ready queue _KERNEL_TCB tasks are inserted in the ascending order of deadlines so that the first task always has the earliest deadline. The flow of new task insertion is shown in the Figure 4.5, in which *entry* is the new _KERNEL_TCB task, *tskpri* is the task's priority, and *adl* is the task's absolute deadline.

Finally, for the purpose of conveniently managing the system and tasks, ITRON System supports groups of system function (or kernel system calls). These system calls are probably categorized into main groups of system configuration and initialization management, task management and synchronization, time management, and other communication managements. System calls are synthesized through system files. Table 4.1 shows an overview of main system files which would be involved in this research and their short description. Files including **savana.h**, **a9.c**, and **comp.m** contains main developed functions based-on the A9 Cortex processor in order to implement the design of the research. The developed functions are explained closely in the next section of software implementation. The system eventually runs follow the initialization procedure in the Figure 4.6.

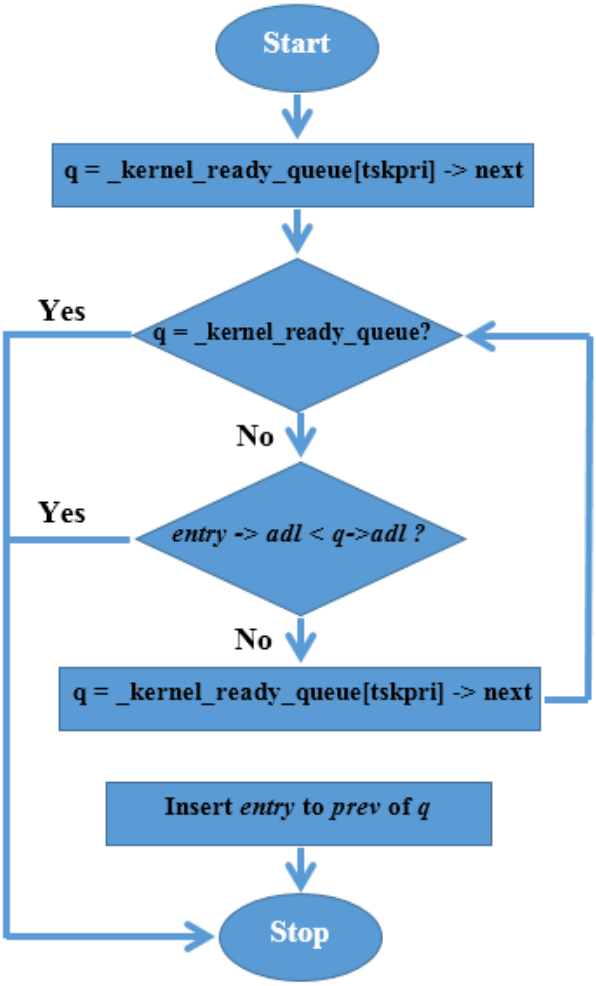

Figure 4.5: The flow of task insertion

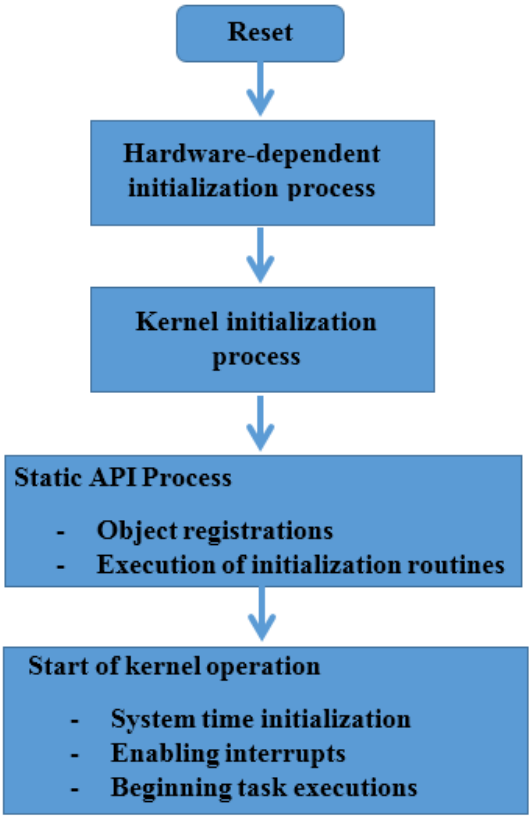

Figure 4.6: System initialization procedure

Table 4.1: Overview of main system files

Category	**System call function**	**Decription**
System configuration and initialization management	kernel.h	Define structures and constants on ITRON System.
	kernel_queue.c	Support procedure to initiate and manage system queues such as _kernel_ready_queue.
	kernel_sched.c	Support procedure to enable scheduler and related functions.
	kernel_globals.c	Define the global variables.
	kernel_init.c	Initiate the global variables.
	kernel_timer.c	Support procedure to initiate and manage kernel system timer.
Task management and synchronization	cre_tsk.c	Create tasks.
	iact_tsk.c	Support interrupts to activate tasks. Required calculations are done prior to activating tasks as well.
	ext_tsk	Support procedure of exiting a task.
Time management	cre_cyc.c	Support procedure to create and register cyclics for tasks.
	get_tim.c	Support procedure to access the system time.
Developed functions	savana.h	Define functions and constants to implement the design on A9 Cortex processor.
	a9.c	Support developed functions to implement the research based-on A9 Cortex processor's instruction set.
	comp.m4	Support Developed functions in assembly based-on A9 Cortex processors instruction set.

4.2 Software implementation

4.2.1 Task creation

Periodic tasks and aperiodic tasks are created to evaluate the implemented algorithms. So in this research, files consisting of **appl.c**, **appl.h**, and **system.cfg** are used to create periodic tasks and aperiodic tasks as the application tasks. With **appl.c** and **appl.h** files, tasks's handler are defined and implemented. Task's handler addresses to task's content which is executed when task is allocated processor. The **system.cfg** file, partially illustrated in Figure 4.7, includes task's important configuration information. The figure

shows several creation functions with 6 periodic tasks and 1 aperiodic task.

The structure of a creation function is presented at line 9 and line 10 in which CRE_TSK is the function name and the others are the arguments. This structure is compatible with ITRON System. First seven arguments including TID, attribute, exinfo, task_pointer, priority, stksiz, stk are task's information for ITRON System [6]. The other arguments are worst case execution time (wcet), relative deadline, period, and periodic_index, respectively. Value 1 at periodic_index mean this is a periodic task. Whereas, Value 0 indicates that it is an aperiodic task.

```
8  // Task Creation
9  /* CRE_TSK ( TID, { attribute, exinf, task_pointer, priority,
10                  stksiz, stk, wcet, relative_dline, period, periodic_index } ); */
11 /*-------------------------------------------------------------------------------*/
12 /*----------------------Creation of periodic tasks-------------------------------*/
13 CRE_TSK ( PTSK_1, { TA_HLNG, 1400000, ptask_hdl_1, 2, 4096, NULL, 20, 200, 200, 1 } );
14 CRE_TSK ( PTSK_2, { TA_HLNG, 150000, ptask_hdl_2, 2, 4096, NULL, 3, 30, 30, 1 } );
15 CRE_TSK ( PTSK_3, { TA_HLNG, 600000, ptask_hdl_3, 2, 4096, NULL, 7, 70, 70, 1 } );
16 CRE_TSK ( PTSK_4, { TA_HLNG, 300000, ptask_hdl_4, 2, 4096, NULL, 5, 40, 40, 1 } );
17 CRE_TSK ( PTSK_5, { TA_HLNG, 600000, ptask_hdl_5, 2, 4096, NULL, 8, 90, 90, 1 } );
18 CRE_TSK ( PTSK_6, { TA_HLNG, 700000, ptask_hdl_6, 2, 4096, NULL, 10, 50, 50, 1 } );
19
20 /*----------------------Creation of aperiodic tasks-----------------------------*/
21 CRE_TSK ( ATSK_1, { TA_HLNG, 100000, atask_hdl_1, 2, 4096, NULL, 5, 0, 0, 0 } );
```

Figure 4.7: Task creation

The period of periodic tasks are implemented easily by registering a cyclicity for each task. An API system function called CRE_CYC is employed for cyclicity registration as seen in Figure 4.8. The function CRE_CYC complies with the structure offered by the ITRON System [6], in which the fifth argument is corresponding to task's period. The periodic tasks then would be activated frequently at each period.

```
101 /*---------------Creation of Cyclics for Periodic Tasks---------------------*/
102 /* CRE_CYC ( CID, { attribute, exinf, cyc_pointer, cyc_time, cyc_phase } ); */
103 /*-------------------------------------------------------------------------------*/
104 /*--------------------Cyclics for Periodic Task-----------------------------*/
105 //CRE_CYC ( PCYC_1, { TA_HLNG | TA_STA, 1023, pcyc_hdl_1, 200, 0 } );
106 //CRE_CYC ( PCYC_2, { TA_HLNG | TA_STA, 1023, pcyc_hdl_2, 30, 0 } );
107 //CRE_CYC ( PCYC_3, { TA_HLNG | TA_STA, 1023, pcyc_hdl_3, 70, 0 } );
108 //CRE_CYC ( PCYC_4, { TA_HLNG | TA_STA, 1023, pcyc_hdl_4, 40, 0 } );
109 //CRE_CYC ( PCYC_5, { TA_HLNG | TA_STA, 1023, pcyc_hdl_5, 90, 0 } );
110 //CRE_CYC ( PCYC_6, { TA_HLNG | TA_STA, 1023, pcyc_hdl_6, 50, 0 } );
111 //periodic task set: e1, 19a, 1f0, 2d5, 3a2, 3fc, 585, 6c3, 729, 7da
112
113 /*--------------------Cyclics for Aperiodic Task----------------------------*/
114 CRE_CYC ( ACYC_1, { TA_HLNG | TA_STA, 1023, acyc_hdl_1, 1, 1 });
```

Figure 4.8: Tasks' cyclic creation

Since ITRON System do not support any mechanism to enable irregular tasks, it is not so easy to implement the aperiodic tasks as the periodic ones. In fact, interrupts may be used to satisfy noncyclical tasks. But, interrupts are seemly too complicated for this research. A simpler alternative mechanism should be used instead in this case. A cyclicity with the period of 1, as listed at the last line in Figure 4.8, is registered to the system. Under the period of 1, at each system tick time this cyclics will check a prepared list of time of which elements are corresponding to release time of aperiodic tasks. If the system

time and task's release time are matching together, an aperiodic task is indirectly enable. The list of release times is generated randomly and ascendant. It is quite efficient enough to deal with aperiodic tasks in this research.

4.2.2 Software design

The proposed algorithm introduced in Chapter 3 is implemented in details in this section. The section is explaining the software implementation of the proposed algorithm in the structure of 4 parts: procedure of algorithm, added variables, developed functions, and finally the code implementation. The code implementation is actually supported and developed based-on the real-time operating system for Zynq7000.

Procedure of algorithm

The scheduling function starts with creating and initiating local variables. Then following the proposed algorithm, the limit of the previous task's deadline is first calculated and compared to the task's release time, as illustrated by the condition C1 in Figure 4.9. If the release time is equal to or less than the release time, the scheduling process will stop. On the contrary, the process continues to calculate the boundary deadline.

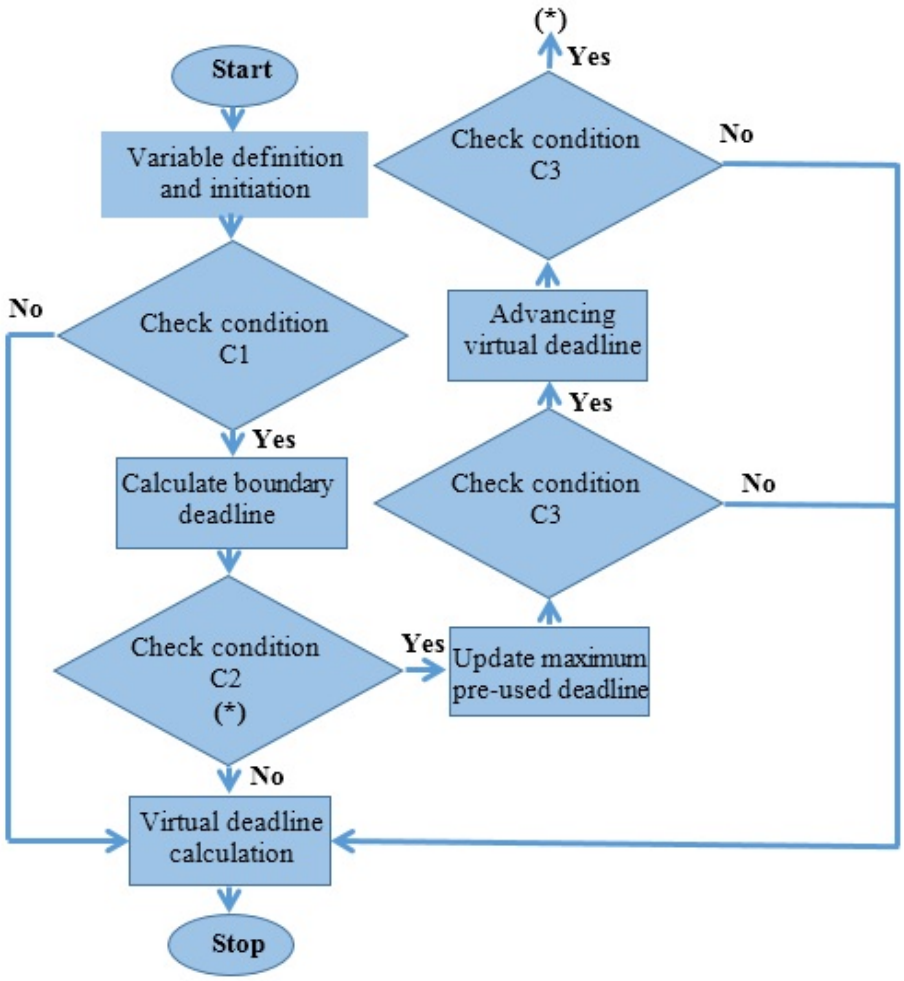

Figure 4.9: Procedure of the proposed algorithm

Then, a loop of virtual deadline advancing begins under the condition (C2 in Figure 4.9)

Table 4.2: Main added global variables for the algorithm's execution

Name	**Size (bit)**	**Description**
p_util	32 x 2	Total processor utilization of periodic tasks
max_period	32	The maximum period of periodic tasks. It reflexes T_{max} as introduced in Chapter III.
ins_cnt	32	The number of released instances.
ins_start	32 x 5000	An array storing the starting time of released instances.
used_dl	32 x 5000	An array storing the associated deadline of the starting slot of released instances.
last_empty	32	The last empty slot
ls_max	32	The starting time of the last instance of τ_{max} as defined in Chapter III
ls_max_dl	32	The associate deadline of the ls_{max}-th slot.
di_1	32	The recalculated deadline of the previous aperiodic task.
fi_1	32	The finishing time of the previous aperiodic task.

that the virtual deadline is greater than the boundary deadline. In the advancing loop, the maximum previously-used deadline is updated first and then compared to the virtual deadline (C3 in Figure 4.9). In the case that C3 is not satisfied, the loop will break and then the advancing process will stop. After C3 is passed, the virtual deadline is advanced backward to the past. If the advanced virtual deadline is continuously greater than maximum previously-used deadline, the loop is repeated for an expected earlier deadline. It is necessary to note that the eventual virtual deadline is obviously updated with alternative values depended on which condition led the advancing process stop. These exact values is successively decided in the following code section.

Added variables

Added variables are created to necessarily update the system's appropriate factors for executing the algorithm. Table 4.2 describes in details some main added variables. These are all in type of unsigned 32-bit integer. Since the A9 Cortex processor actually does not directly support operations with decimal points, the periodic processor utilization is considered as a fraction of which dividend and divisor are separately stored in p_util[0] and p_util[1], respectively. In addition, the experiments for evaluating the designed system are done for such the first 5000 instances of tasks that variables ins_start and used_dl are initiatively created with size of 5000. This size is for shortening the evaluation time while still assuring the accuracy.

Developed functions

The main scheduling function is named dl_set_EVAR (the first one in Table 4.3) which directly implements the enhanced virtual release advancing algorithm. In Chapter III, we analyzed the algorithm and determined the virtual deadline as its target output. dl_set_EVAR function is designed as a system call with two arguments as pointers. One pointer directs to the structure of the current task under scheduling. The other refers

to the absolute deadline for the scheduled task. Whenever an aperiodic task is released, dl_set_EVAR is called to calculate the absolute deadline. After the call is done, the task is assigned a virtual deadline as earlier as possible than the original deadline. The function's procedure and code implementation are respectively showed in Figure 4.9 and Figure 4.10.

Table 4.3: Main developed functions of software design

Name	**Syntax**	**Description**
dl_set_EVRA	void dl_set_EVRA(_KERNEL_TCB*, _KERNEL_ACT_CELL*)	The main scheduling function
reducer	void reducer(UINT *, UINT *)	Reducing two integral numbers by their great common divisor
get_ins_cnt	UINT get_ins_cnt(void)	Returning instance count
system_record	void system_record(UINT)	System recorder
tsk_exe_cal	void tsk_exe_cal(UINT *)	Task execution calculation
max	UINT max(UINT, UINT)	Returning the maximum one between two values.
max3	UINT max3(UINT, UINT, UINT)	Returning the maximum one among three values.
get_used_dl	UINT get_used_dl(UINT)	Returning the associate deadline of the corresponding instance of the transferred argument.
get_ins_start	UINT get_ins_start(UINT)	Returning the starting time of the corresponding instance of the transferred argument.

However, before being ready for scheduling function to take place, the system needs to calculate and update required factors and variables as described in the previous section. It is hence needed to design further functions in Table 4.3. In the syntax of the functions, UINT is the type of unsigned 32-bit integer. The function named *reducer* is functional to reduce two integral numbers by their great common divisor. This function is targeted to simplify the fraction of periodic processor utilization as mentioned above in the section of added variables on way that *p_util[0]* and *p_util[1]* are transferred as input arguments.

Next, the function *get_ins_cnt* is designed to refer to the updated number of released instance. Different from the original algorithm where the advancing is done upon ticks, the enhanced algorithm works on the instances. As a result, a function accessing to the number of released instance is always helpful.

Next, function *system_record* works as a recorder that is in charge of record required elements (or system factors) for executing the scheduling algorithm. System factors updated by the recorder include instance count, current instance's starting time, current slots associate deadline, and the last instance of τ_{max}.

Finally, function *tsk_exe_cal* is functional to calculate task's associate execution time. A task's associate execution time, "a_et", is defined as Equation 4.1. In this equation, C_k and U_s are already known as k-th task's execution time and the utilization of the server. As seen in the pseudo code of the algorithm, the task's associate execution time much appears in the absolute deadline calculations. In addition, this factor of execution does not change during system's working. Therefore, it is clearly more efficient to statically design a system call to calculate this value.

$$a\_et = \frac{C_k}{U_s} \tag{4.1}$$

Code implementation

For the conciseness of the presentation, this section just focuses on explaining the code of the main scheduling function, *dl_set_EVRA*, in Table 4.3. Since the flow of scheduling algorithm is entirely presented in the algorithm procedure section, it is not imperative to be repeated here. The way that the virtual deadline is obtained is concentratedly explained instead.

The code implementation of the scheduling function is wholly shown in Figure 4.10. Reminding that the scheduling function has two arguments, a *_KERNEL_TCB* "tcb" and a *_KERNEL_ACT_CELL* "act_cell", which refer to the structure of task under scheduling and the absolute deadline. According to the structure of "tcb", factors including task's request time and associate execution time are accessed and used as *rq_time* and *a_et* in related calculations. Similarly, structure of *act_cell* allows to access to the absolute deadline of scheduled task as *a_dl*. In the process of absolute deadline calculation, global variables *di_1*, *fi_1*, *ls_max*, and *last_empty* are available as described in Table 4.2 and employed functions are as described in Table 4.3.

At the beginning of the execution, local variables are normally defined. Among these variables, *bound_dl* is the limit for the virtual deadline advancing. It is the maximum value over all of the boundary deadlines (Chapter III). Meanwhile, *bound_rl* is the limit for virtual release advancing virtual release. Since it is also one of output values of the algorithm and accompanies with the virtual deadline, virtual release time has to be limited. The other variables (*last_dl*, *max_used_dl*, and i) are created to refer to the last previous task's deadline, maximum previously-used deadline, and the number of instances, respectively. In this part, the virtual deadline is also initiated to the original one.

As mentioned above that the final virtual deadline, as absolute deadline for the task, is calculated differently depending on the stop condition of the advancing process. In general, the virtual deadline is defined by Equation 4.2 as the sum of task's virtual release time and task's associate execution time. However, it is interesting that the varied stop conditions characterize the varied virtual release times and then the varied virtual deadlines.

$$vd_k = vr_k + a\_et \tag{4.2}$$

Following the procedure of the scheduling algorithm, the previous task's deadline is the first limit of advancing and therefore is involved first. This limit is calculated as variable *last_dl*. If the limit is not passed for the previous task's deadline, the advancing stops. In this case, *last_dl* is considered as virtual release time and the final virtual deadline is set using *last_dl* and *a_et*.

After passing the first limit and calculating the boundary deadlines, the process continuously enters the advancing loop under the condition of the boundary deadlines (*bound_dl*). It is actual that when a loop interaction occurs successfully (it means no break occurs), the virtual deadline is advanced backward by the length of the current involved instance. In the case when the advancing stop due to the condition of the boundary deadlines, that

```
void    dl_set_EVRA(_KERNEL_TCB *tcb, _KERNEL_ACT_CELL *act_cell)
{
 /*Local variable definition*/
 UINT bound_dl = 0, bound_rl = 0;
 UINT last_dl = 0;
 UINT max_used_dl = 0;
 UINT i = 0;
 act_cell -> a_dl =  tcb -> rq_time + tcb -> a_et;

/*Limit of the previous task's deadline*/
 last_dl = max( di_1, fi_1 );
 if (tcb -> rq_time <= last_dl){
        act_cell -> a_dl = last_dl + tcb -> a_et;
        return;
 }

/*Boundary deadline calculation*/
 Bound_rl = max3(ls_max, last_empty + 1, last_dl);
 bound_dl = bound_rl + tcb -> a_et;
 i = get_ins_cnt();

/*Main advancing loop*/
 while (act_cell -> a_dl > bound_dl){
        /*Maximum pre-used deadline update*/
        if (max_used_dl < get_used_dl(i)){
                max_used_dl = get_used_dl(i);
        }
        /*Compare to the maximum pre-used deadline*/
        if (act_cell -> a_dl <= max_used_dl){
                break;
        }
        /*Virtual deadline advacing*/
        if (get_ins_start(i) > bound_rl){
                act_cell -> a_dl = get_ins_start(i)+ tcb -> a_et;
        }
        else{
                act_cell -> a_dl = bound_rl + tcb -> a_et;
        }
        /*After advancing, again compare to maximum pre-used deadline*/
        if(act_cell -> a_dl <= max_used_dl){
                act_cell -> a_dl = max_used_dl;
                break;
        }
        i -= 1;
 }
 return;
}
```

Figure 4.10: Source code of the software implementation

is, the virtual deadline is equal to or earlier than *bound_dl*, the starting time of the current involved instance is considered as the virtual release time and the final virtual deadline is set using this virtual release time and *a_et*.

The final stop condition of the advancing is the maximum previously-used deadline. Seeing the codes in Figure 4.10, checking this condition takes place two times in the advancing loop. One is when the maximum previously-used deadline is updated. The other follows the virtual deadline advancing. If the first checking is broken, no deadline calculation occurs and the current virtual deadline is kept. And if the second checking is broken, the virtual deadline is assigned as the current maximum previously-used deadline.

4.2.3 Recapitulation for software implementation

The software implementation of the enhanced virtual release advancing algorithm has presented in this section. The entire procedure of the algorithm gives an overview of processing flow following the proposed algorithm in Chapter 3. The software scheduling algorithm is made clear through explanations in details for its procedure, variables, developed functions, and the source code. The performance of this design is evaluated in accompaniment with the hardware implementation in the next chapter of evaluation.

4.3 Hardware implementation

In Section 4.2, a software design of the scheduling algorithm has been introduced. However, in real-time embedded system, a scheduling algorithm will be more effective in terms of system performance if it is able to be implemented on hardware. Moreover, a hardware implementation also confirms the applicability of the algorithm to the real system with an acceptable complexity. Thus, a hardware mechanism is needed to be designed in this research.

This section of hardware design is composed of two main parts. The first part is the basic processing system which is the implementation of the ARM Cortex-A9 processing core. The second one is the designed hardware which realizes the proposed algorithm. All of the designs are implemented on the Zynq70000 FPGA ZedBoard.

4.3.1 The basic processing system

The basis processing system is synthesized based-on the structure of the ARM Cortex-A9 processing core. As introduced in Chapter 1 as the introduction, the Zynq70000 FPGA Board supports to synthesize a dual-core ARM Cortex-A9 MPCore as the application processor units [7]. Besides the processing units, other components related to the hardware design include 2048 Mbit DDRAM and IO peripherals. UART and USB prototype as IO peripherals are employed to connect with the computer terminal in programming and evaluating the designs on FPGA. 32-bit GPIO Master-Slave connections is used to communicate with the added hardware as the implementation of the algorithm. The processing system runs at the frequency of 666MHz and the DDR RAM does at 533MHz.

In the connection with the added hardware, an AXI interconnection component instanced as **axi_interconnect_0** is employed as shown in the Figure 4.11. In this figure, the instanced **processing_system7_0** component is the implemetation of Zynq7000 core processor. A processor system reset **pro_sys_reset_0** is attached to the system for synchronization. The red rectangle in Figure 4.11 marks the connection area to the added hardware. Two AXI GPIO components are used for this connection area. For the convenience of the communication, one component **axi_gpio_0** serves as the IO data ports.

The other one does as the IO ACK port. There are here four IO ports (**GPIO**, **GPIO2**, **GPIO_1**, and **GPIO2_1**) preserved for the communication between the processing core (or CPU) and the added hardware as follows:

- **GPIO**: Serving as the ACK input signals. When CPU wants to confirm an ACK response from the added hardware, it needs to access this port through the port's mapped address.
- **GPIO2**: Serving as the ACK output signals. When CPU requests a service on the added hardware, it needs to send a request command to this port through the port's mapped address.
- **GPIO_1**: Serving as the data input signals. When CPUwants to load a data from the added hardware, it needs to access this port through the port's mapped address.
- **GPIO2_1**: Serving as the data output signals. When CPU wants to store a data to the added hardware, it needs to send the data to this port through the port's mapped address.

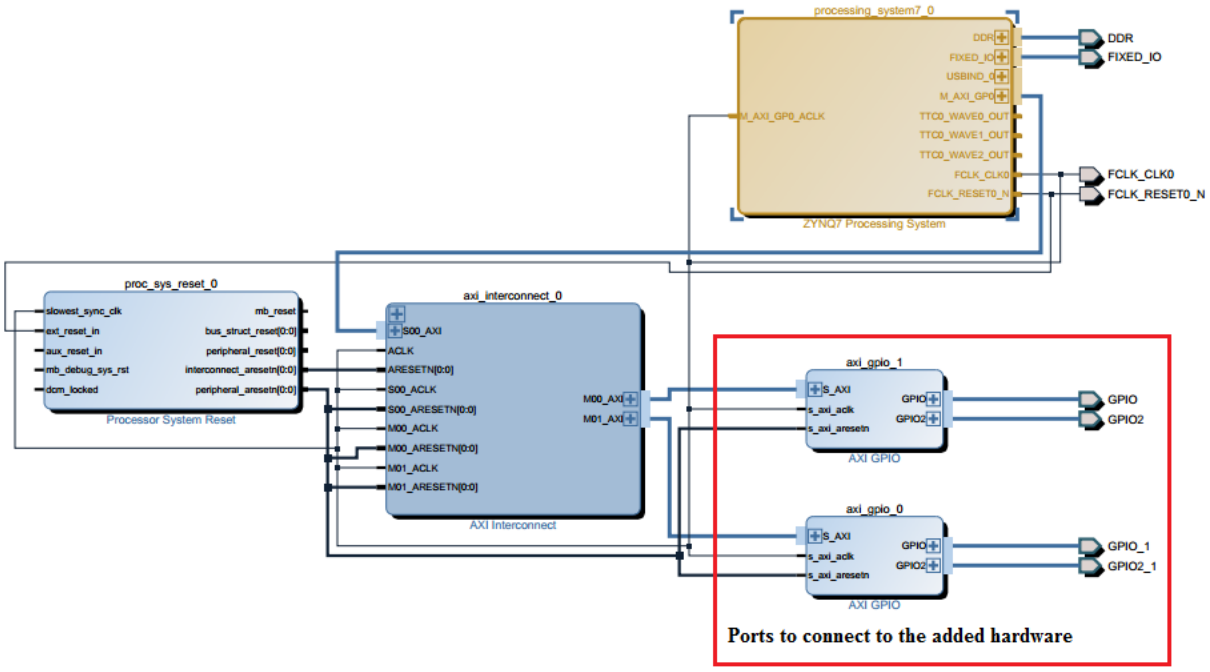

Figure 4.11: Design of basic processing system

The request commands and the ACK responses in the communication description would be designed in the following section.

These ports are described in details together with the other connections in Table 4.4. The output clock **FCLK_CLK0** is intended to be the clock for the added hardware. There are remarkable contents here in the Table 4.4, that is, the assigned address. These are the mapped addresses to the **GPIO** ports.

In summary, the basic processing system including the connection ports has been implemented. This hardware system is support to both the software and hardware designs of the proposed algorithm. In the context of software implementation, the added AXI GPIO components are not related since the software design is executed by the CPU core only. Meanwhile, these components server as the connection between the CPU and the hardware design. The detailed hardware design is coming in the next section.

Table 4.4: Port connection of processing system

Name	IO Type	Width	Assigned address	Description
GPIO	Input	16	41200000	Input ACK signals
GPIO2	Output	16	41200008	Output ACK signals
GPIO_1	Input	32	41300000	Input data
GPIO2_1	Output	32	41300008	Output data
FCLK_CLK0	Output	1		Clock to the target hardware
FCLK_RESET0_N	Output	1		System Reset
DDR	Inout	71		Connections to DDR memory
FIXED_IO	Inout	61		Connections to multiple IOs

4.3.2 Block Diagram of Hardware Implementation

A hardware design is implemented to show the actual applicability of the proposed algorithm. It is known as an added component to the basic system introduced in the previous section. The system block diagram and connections of this added component are depicted in Figure 4.12. In this figure, the CPU is the basic processing system which connects to the added hardware by the GPIO ports and the FCLK clock.

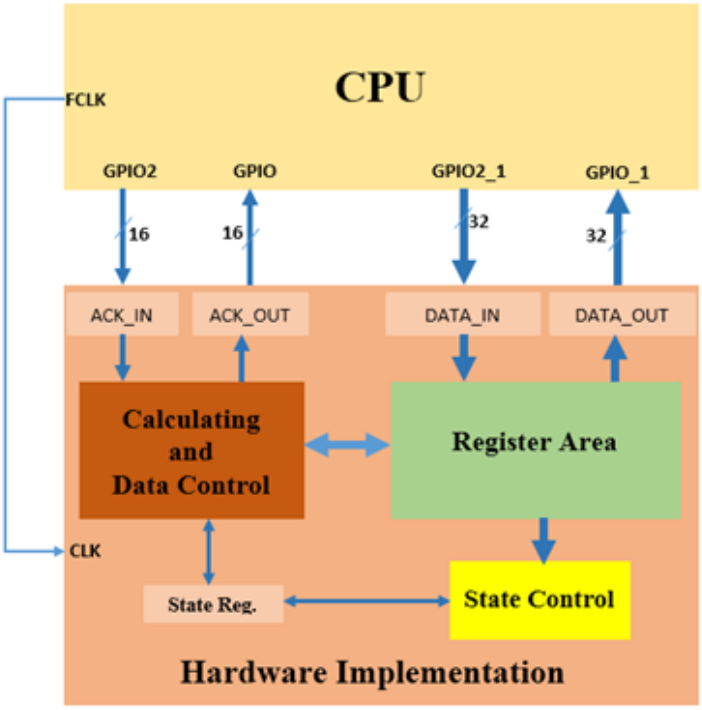

Figure 4.12: Block diagram of hardware implementation

In the block diagram of the hardware, there are four significant parts. These parts include the IO registers, the register area as memory, the state control block, and the main calculating and data control block.

IO registers

There are four IO registers connecting correspondingly to the above-decided GPIO ports of the CPU. These registers temporarily store the IO data and command for communicating with the CPU. Particularly, they are as follows:

- **ACK_IN**: connecting to the GPIO2 of the CPU. This stores the request command which has been sent by the CPU.
- **ACK_OUT**: connecting to the GPIO of the CPU. This stores the ACK response which the hardware needs to send to the CPU.
- **DATA_IN**: connecting to the GPIO2_1 of the CPU. This stores the input data which has been sent by the CPU.
- **DATA_OUT**: connecting to the GPIO_1 of the CPU. This stores the output data which the hardware needs to send to the CPU.

Register area

Register area plays a role as a memory of the implemented hardware. This area may be separated into two parts agreeing with local data and global data. Local part is composed of the registers where local variables serving the algorithm execution are stored. This local registers are accessed only by the hardware itself. The global part consists of the registers where the global variables recording the system factors are stored. The global registers, meanwhile, allow both internal and external accesses. For their purpose in communication with the CPU, the global register are assigned ID indexes. Through the ID indexes, data can be stored to or loaded from a specific global register by the CPU.

Table 4.5: Global registers of the hardware

Order No.	**Register name**	**Size (bit)**	**ID index**	**Description**
1	last_empty	32	8'h02	The last empty slot
2	ls_max_adl	32	8'h04	The last instance of τ_{max}
3	di_1	32	8'h10	The previous task's deadline
4	fi_1	32	8'h20	The previous task's finishing time
5	tcb_request_time	32	8'h40	The task's request time
6	tcb_aet	32	8'h80	The task's associated execution time
7	act_cell_dl	32	8'h00	The virtual deadline for task
8	tcb_request_adv	32	8'h11	The virtual release time for task
9	ins_start	32 x 5000	8'h08	Starting time of instances
10	used_dl	32 x 5000	8'h01	Associated deadline of instances

Table 4.5 shows the list of the global registers attached ID indexes. There are totally eight single 32-bit registers and two arrays of 32-bit registers. The way how these registers are accessed will be presented below in accompaniment with the structure of the request command.

State control block

The state control block manages a state register (Figure 4.12) where the system state is updated. Under the system state, the algorithm execution is controlled. The state control block and the state value are decided by a state machine. Figure 4.13 is the diagram of the state machine of the state control. The state machine is designed with 6 states of the system consisting of STATE_INI, STATE_10, STATE_20, STATE_11, STATE_21, and STATE_31. Particularly, they are defined as:

- **STATE_INI**: initial state of the system.
- **STATE_10**, **STATE_20**: processing states. In these states, the hardware follows the process of virtual deadline advancing .
- **STATE_11**, **STATE_21**, **STATE_31**: stop states. The scheduling is stopping at one of these states and the virtual deadline is available at corresponding registers as described in Table 4.5. As discussed in the section of software design, according to the different stop conditions of the advancing, the virtual deadline is set to the different values. That is the reason why there are three different stop states appearing here. Each stop state therefore represents for a state of a specific stop condition.

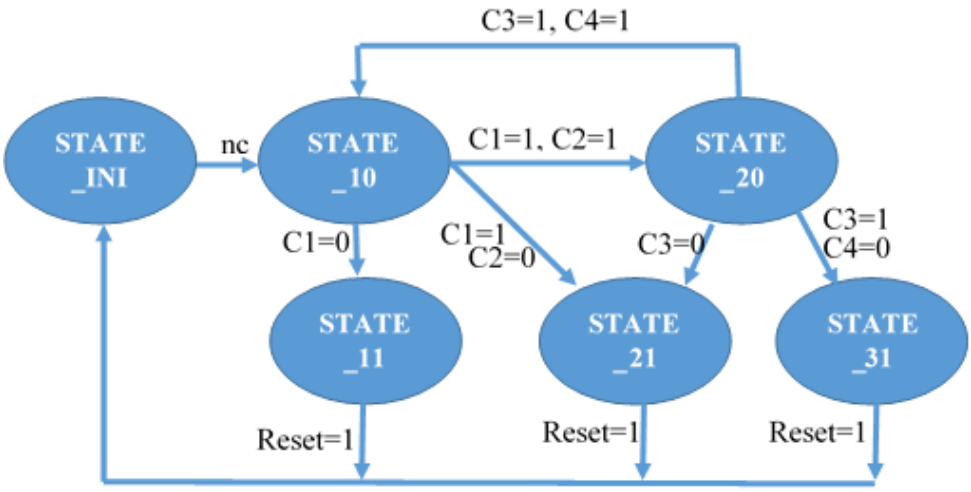

Figure 4.13: State changing diagram

In the state changing diagram in Figure 4.13, C1, C2, C3, and C4 correspond to the checking condition points of the algorithm at line 6, line 18, line 23, and line 27 in Figure 3.4 on page 15. The system states would change under the control of checking conditions. First, at the initial state, when clock enables, the state will change directly to the STATE_10 without any condition. "nc" here means no condition. Then, when the system is one of the processing states, it will be change to the next suitable state under checking conditions. Together with each state change, the system completes data calculations necessary for the next state. If anyone of checking conditions is unsatisfied, the system will appear in one of the stop states. A stop state is remainding until the system is reset to return the initial state with an active clock of the reset signal.

In the hardware implementation, there is an enhancement compared to the software version. That is, condition checking is done in parallel. It is natural in software implementation that the conditions C1, C2, C3, and C4 are checked in sequence. However, in hardware implementation, conditions C1 and C2 are dealt with consecutively at the same state. It is similar to conditions C3 and C4. Consequently, it needs less cycles to check all of conditions in the hardware design than in the software version. The performance of the system is therefore improved.

Main calculating and data control block

This block is the main hardware mechanism of the algorithm. Calculations of the algorithm are done at this block under the state of the system. This part actually works as a decoder that is charge of deciding which operation would be done. When receiving a request command from the CPU, the decoder will decode the operation code to determine the requested operation.

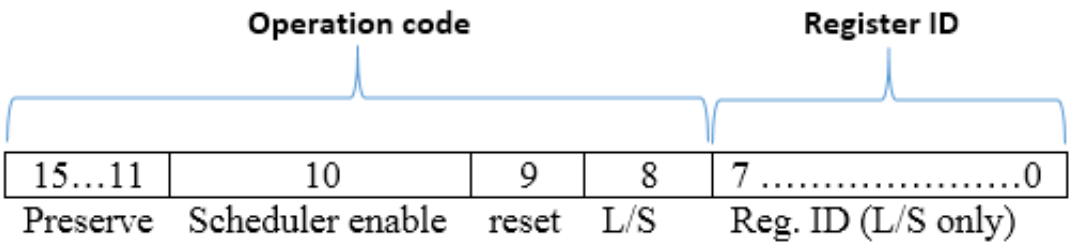

Figure 4.14: Structure of a request command

There are four operations designed for this hardware: reset, scheduling, storing data, and loading data. The request command in the communication between the CPU and the hardware has 16 bits in length which is structured in two parts as shown in Figure 4.14. The first part, 8 least significant bits, is the register ID. These bits are used to decode the target register in the storing and loading requests. The target register is one of the global registers listed in Table 4.5. The second part, 8 most significant bits, is the operation code (hereafter called *op-code*). These bits will be decoded to determine exactly the operation requested. In the structure of *op-code*, the 8-th bit is for loading and storing operation. If this bit is set to 1 (and other bits of *op-code* are 0s), the corresponding operation is loading data from a specific register realized by the 8-bit register ID. On the contrary, if this bit is 0, the operation is storing data to a specific register. Similarly, the 9-th bit and 10-th bit are preserved for reset and scheduling operations, respectively. These operation are both active at the high level of the assigned bit. *op-code* corresponding to four operations are defined in Table 4.6. It is necessary to note that for the reset and scheduling operations the 8 least significant bits are uncared.

Table 4.6: List of operation codes

Order No.	**Operation type**	**Operation code**
1	Reset	8'h02
2	Scheduling	8'h04
3	Store	8'h00
4	Load	8'h01

The following examples illustrate for the entire request commands of 16-bit length sent by the CPU.

- Reset hardware: 16'h0200
- Enable hardware: 16'h0400
- Store data to register ID 01: 16'h0001 (8th bit is 0 for store data)
- Load data from register ID 04: 16'h0104 (8th bit is 1 for load data)

4.3.3 Procedures of Communication

In this part, we will take account of procedures of communication between the operating system and the added hardware. There are three main procedures consisting of storing data, loading data, and scheduling.

Storing data procedure

The storing data procedure follows the steps in Figure 4.15. At the beginning of storing, data needs to be put into the data bus at the GPIO2_1 port through its mapped address. Then the related storing command is sent to the hardware through the GPIO2 port. Next, the operating system needs to wait for the successfully-store response by reading the ACK response at the GPIO port. A storing procedure completes when the correct successfully-store response is released by the hardware.

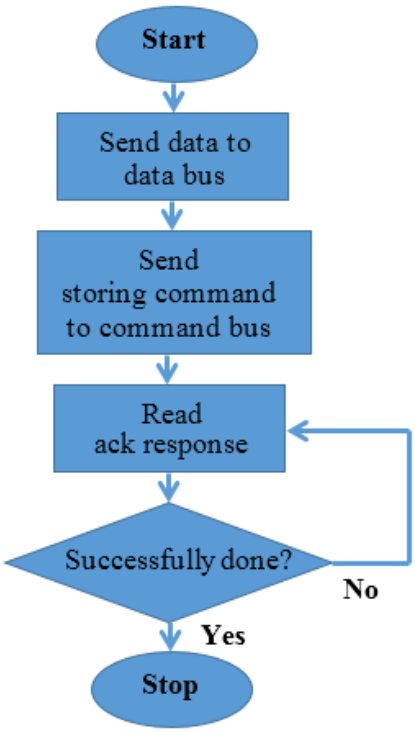

Figure 4.15: Storing data procedure

For instance, the following assembly code is to store #data to a register with ID 01 (the request command is 16'h0001):

```
mov r1, # 16'h0001      % storing command
mov r2, #data
str r2, GPIO2_1         % putting data to data bus
str r1, GPIO2           % sending the storing command
```

```
        ldr r1, GPIO          % reading the ack response
check:  cmp r1, #0100         % check whether storing is done
        bne check             % successfully?
```

Loading data procedure

For loading data, the procedure is stepped in Figure 4.16. To start the loading data operation, the operating system needs first to send the loading command to the command port at GPIO2. Then, it is waiting for the response of data validation by repeatedly reading the ACK response on the GPIO port. After the correct loading response is released by the hardware, the operating system can load the data which is already available at the GPIO_1 port.

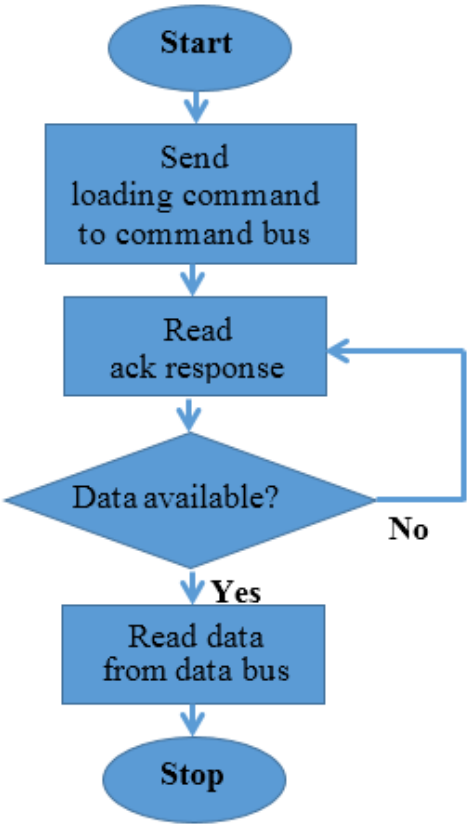

Figure 4.16: Loading data procedure

For example, the following assembly code is to load a data stored at register with ID 04 (the request command is 16'h0104):

```
        mov r1, #16'h0104     % loading command
        str r1, GPIO2         % sending loading command
        ldr r1, GPIO          % check whether data is available?
check:  cmp r1, #0100
        bne check
        ldr r1, GPIO_1        % loading the available data at port
```

Enable scheduling procedure

To enable the scheduling, a procedure is done as described in Figure 4.17. To be ready to execute the scheduling request from the operating system, the hardware first needs to be reset. Then, the hardware is enabled by sending a scheduling command to the GPIO2 port. The operating is waiting until the scheduling (or the advancing) is done successfully. During the waiting time, the ACK response is read recurrently at the GPIO port. After

a scheduling done response is released, the operating system can loading the related data including virtual deadline and virtual release time.

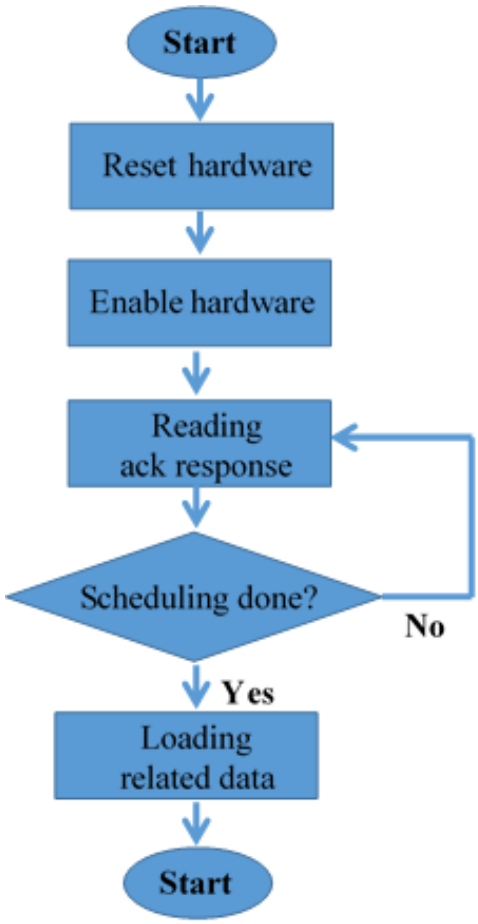

Figure 4.17: Enable scheduling procedure

The typical procedure of enabling scheduling in assembly code is shown as follows with the scheduling command of 16'h0400.

```
        mov r1,# 16'h0400   % enabling code
        mov r2,#16'h0200    % reset code
        str r2, GPIO2       % for reset hardware
        str r1, GPIO2       % for enabling scheduling
        ldr r1, GPIO
check: cmp r1, #0400        % check whether scheduling is done?
        bne check
        /*For next, loading required data after scheduling*/
```

Response codes

In every operation procedure, the operating system needs to check the response from the hardware in order to confirm the status of the request execution. These response codes for the hardware are defined in Table 4.7. There are five response codes:

- NO-OP: There is no operation done.
- Scheduling done: The scheduling is done successfully.
- Storing done: The storing operation is done successfully.

- Loading done: The loading operation is done successfully.
- ERROR: There is error occurring in the process and the data is not valid.

Table 4.7: List of response codes

Order No.	**Response type**	**Response code**
1	NO-OP	16'h0000
2	Scheduling done	16'h0400
3	Storing done	16'h0100
4	Loading done	16'h0100
5	ERROR	16'h0200

4.4 Implementation parameters on FPGA

The hardware implementation is designed in Verilog which is one of the popular hardware description languages. The design is then synthesized by the Vivado2014 Synthesizer and programed into the FPGA ZedBoard by the TB-7Z-020-EMC Debug Monitor Boot Program Revision 001, 2015 [11]. Implementation parameters of the whole design including the basic processing on FPGA are summarized in Table 4.8.

Table 4.8: Summary of hardware implementation

Order No.	**Parameter**	**Value**
1	Total number of cells	13949
2	Total of transformed instances	6
3	CARRY4 instances	16'h0100
4	FDRE instances	384
5	RAM64M instances	3160
6	RAM64X1D instances	632
7	SRL16E instances	1

Chapter 5

Evaluation

5.1 Methods of evaluation

5.1.1 Objectives

Evaluations were done to evaluate the runtime overhead of the proposed algorithm compared to that of the original algorithm while guaranteeing the response time of tasks. The runtime overheads were calculated by the total number of instruction execution cycles per tick for each algorithm on each task set.

Task sets used in the evaluation consist of both aperiodic and periodic tasks. The performance in terms of runtime overhead of two algorithms is shown in many different steps as described below.

5.1.2 Steps of evaluation

The enhancement of the proposed algorithm is evaluated through three steps: simulation, software implementation, and hardware implementation. The environments are established based-on the ARM processor structure and the ITRON System. Tables 5.1 describes the overview of steps of evaluation including the environment, involved algorithms, and objective results. The evaluation consits of three steps:

- The first step, simulation, would do calculations in estimation so as to confirm the theoritical effectiveness of the proposed algorithm compared to the original one. Therefore, both algorithms are involved in the simulation.
- The second step, software implementation, would do calculations in an real system so as to confirm the actual effectiveness of the proposed algorithm compared to the original one. Therefore, both algorithms are involved in the simulation. The real system is establised based-on the ITRON System and the ARM Cortex-A9 processor core in FPGA. In the software implementation, algorithms are processed by software programs.
- The third step, hardware implementation, is to improve the executating performance by hardware which is implemented in the FPGA together with the ARM Cortex-A9 processor core. The proposed algorithm is implemented only in this step.

Table 5.1: Overview of steps of evaluation

Step of evaluation	**Environment**	**Involved algorithms**	**Objective results**
Simulation	Software environment and ARM instructions in estimation	The original algorithm and the proposed algorithm	To compare theoretically the effectiveness of the proposed algorithm
Software implementation	ITRON System and ARM Cortex-A9 processor.	The original algorithm and the proposed algorithm	To compare the actual effectiveness of the proposed algorithm
Hardware implementation	Hardware implematation and ARM Cortex-A9 processor.	The proposed algorithm	Improve the executing performance by hardware

5.2 Evaluation on simulation

5.2.1 The simulation environment

The evaluation is done for the requirements of runtime overhead. The software simulation environment based-on the instruction set of the Cortex-A9 processor [4][5] is installed. The additional runtime overhead is estimated according to the number of instructions of the algorithms execution. Arithmetic, logical, and control operations in the algorithm are involved in runtime overhead calculation. To estimate the runtime overhead, instructions are assigned the approximate number of cycles, as shown in Table 5.2, in which the instructions can be completed.

Table 5.2: Instruction estimation for simulation

Order No.	Instruction	Assigned cycles	Description
1	FADD	4	Float number addition
2	FMUL	5	Float number multiplication
3	FDIV	15	Float number division
4	IADD	1	Integer number addition
5	ILOG	1	Integer Logic
6	IMUL	2	Integer number multiplication
7	COMP	1	Comparator
8	ASSIGN	1	Assignment
9	FLOOR	1	Number Flooring
10	CEIL	1	Number Ceiling
11	MEM	1	Memory access

5.2.2 The simulation results

The simulations are intented to show the estimated results of the additional execution runtime overheads. It is assumed that a processor's clock frequency is 100MHz and the tick length is 0.1ms (the tick length in simulation of the original algorithm is 1ms [3]). The simulation environment is built according to the TBS server and preemptive EDF algorithm similarly to that in [2]. The simulation is set to executed both periodic and aperiodic task sets. The periodic task sets would occupy from 60% to 95% of the total processor utilization with interval step of 5%. Meanwhile the aperiodic task sets would take about 2% of the total processor utilization. The simulations are observed in100,000 ticks. Both orignal and enhanced algorithms are under the simulations to executed totally 80 periodic and 10 aperiodic task sets.

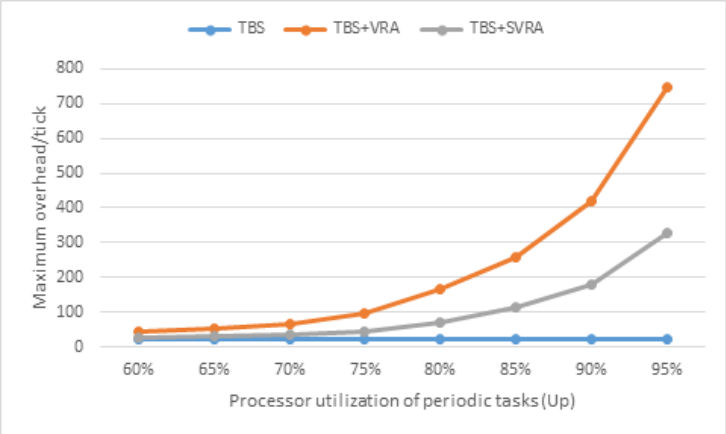

Figure 5.1: The maximum runtime overhead per tick

Figure 5.1 shows results of the maximum runtime overhead per tick. TBS+EVRA denotes the overhead of EVRA algorithm. TBS and TBS+VRA are the results of the

original TBS and the original virtual release advancing algorithm. As shown, the maximum overhead of EVRA is lower than that of TBS+VRA and greater than original TBS. At $U_p = 95\%$, the maximum total overhead per tick of the EVRA is around 327 whereas that of TBS+VRA is around 747. The EVRA is effective to reduce approximately 56% runtime overhead compared to TBS+VRA. This improved result allows to apply this technique to actual real time system. Along with evaluating the runtime overhead, there is another considerable requirement, that is, response time. The proposed algorithm is considered more effective than the original one if and only if the same responsiveness is guaranteed. Results of response time are shown in Table 5.3. The response time for EVRA is equal to VRA and clearly better than original TBS.

Table 5.3: Simulation results for responsiveness

	TBS	**TBS+VRA**	**TBS+EVRA**
60%	4.89	4.86	4.86
65%	5.31	5.27	5.27
70%	6.47	6.42	6.42
75%	7.51	7.25	7.25
80%	11.80	10.87	10.87
85%	18.00	15.91	15.91
90%	40.17	34.90	34.90
95%	116.40	109.23	109.23

5.3 Evaluations on software and hardware implementations

5.3.1 The environment for software and hardware evaluations

The evaluations are done on the Zynq7000 processing system integrated in the FPGA ZedBoard as introduced in Chapter IV. The processing system for the evaluations is set at the frequency of 666MHz. The software and hardware implementations are evaluated separately for the same periodic and aperiodic task sets. A software design for the original algorithm is also developed to compare with the performance of the enhanced algorithm.

There are five periodic task sets are under this evaluation. Since the aperiodic task sets are created manually, there are only three aperiodic task sets of ten activating points are used in this evaluation. The periodic task sets and associated aperiodic one are listed in Table 5.4 to Table 5.8. Therefore, there are five scenarios of evaluation. Each scenario includes six periodic tasks and one aperiodic task entering the system at ten different points of time. In the scenarios, the total processor utilization does not include the kernel utilization that is preserved for kernel tasks such as context switching, function calls, and so on.

Table 5.4: Tasks in Scenario 1

Periodic tasks (71%)			**Aperiodic task**
Task name	Wect	Period	Entering times
PTSK_1	20	200	225, 410, 496, 725, 930, 1020, 1413, 1731, 1833, 2010
PTSK_2	3	30	
PTSK_3	7	70	
PTSK_4	5	40	
PTSK_5	8	90	
PTSK_6	10	50	

Table 5.5: Tasks in Scenario 2

Periodic tasks (73%)			**Aperiodic task**
Task name	Wect	Period	Entering times
PTSK_1	3	21	90, 167, 271, 385, 497, 603, 724, 847, 965, 1116
PTSK_2	3	30	
PTSK_3	4	28	
PTSK_4	5	40	
PTSK_5	4	44	
PTSK_6	5	40	

Table 5.6: Tasks in Scenario 3

Periodic tasks (67.5%)			**Aperiodic task**
Task name	Wect	Period	Entering times
PTSK_1	3	27	90, 167, 271, 385, 497, 603, 724, 847, 965, 1116
PTSK_2	3	30	
PTSK_3	4	28	
PTSK_4	5	40	
PTSK_5	4	56	
PTSK_6	5	40	

Table 5.7: Tasks in Scenario 4

Periodic tasks (71%)			**Aperiodic task**
Task name	Wect	Period	Entering times
PTSK_1	3	24	90, 167, 237, 320, 392, 457, 541, 612, 698, 760
PTSK_2	3	30	
PTSK_3	4	28	
PTSK_4	5	45	
PTSK_5	4	44	
PTSK_6	5	35	

Table 5.8: Tasks in Scenario 5

Periodic tasks (73%)			**Aperiodic task**
Task name	Wect	Period	Entering times
PTSK_1	3	21	225, 410, 496, 725, 930, 1020, 1413, 1731, 1833, 2010
PTSK_2	3	30	
PTSK_3	4	28	
PTSK_4	5	40	
PTSK_5	4	44	
PTSK_6	5	40	

5.3.2 Results of software and hardware implementations

To evaluate the performance of algorithm implementations, there are two target factors that should be taken into account: the maximum added overhead per tick and the total execution overhead. These overheads are calculated as follows:

- *The maximum added overhead per tick*: Given $O_{sr}[i]$ and $O_{ae}[i]$ are the added overhead by system recodes, required for the algorithm's execution and the added overhead of the algorithm's execution at the i-th tick; the added overhead at i-th tick ($OVH[i]$) is $OVH[i] = O_{sr}[i]+O_{ae}[i]$. If there are ticks without task's activation, the corresponding O_{ae} is clearly 0. The maximum added overhead per tick ($maxOVH$) is then calculated as:
$$maxOVH = max(OVH[i]) \tag{5.1}$$

- *The total execution overhead*: Given O_{ae_k} is the added overhead of the algorithm's execution for the k-th task's activation. Since there are ten of release times of

aperiodic task, the total execution overhead ($totOVH$) is calculated as:

$$totOVH = \sum_{k=0}^{9} O_{ae\_k} \tag{5.2}$$

The number of cycles is obtained by access the cycle count register (PMCCNTR) of the processing system [10]. PMCCNTR, as shown in Figure 5.2, is a count-down register with the initial value of 166400. The initial value of PMCCNTR is exactly the total cycles per tick of the system. In other words, each tick time spends 166400 cycles. Given x is an immediate value of PMCCNTR, the number of used cycles of the tick (or cycle offset, C_{offset}) can be obtained by Formula 5.3. The cycle offset to calculate the exact value in cycle of the evaluations.

$$C_{offset} = 166400 - x \tag{5.3}$$

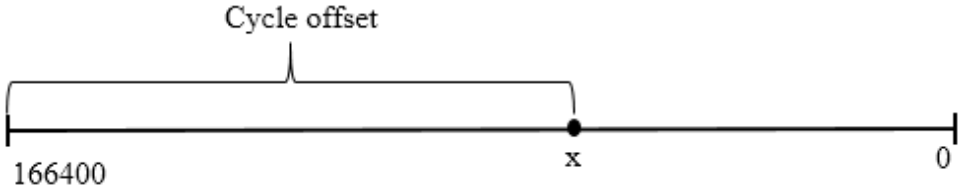

Figure 5.2: PMCCNTR register and cycle offset in a tick time

The evaluation results are shown in Table 5.9 to Table 5.13 for five scenarios. Column 1 in tables is the original release time of aperiodic tasks. Two following columns are the absolute deadlines. The original deadlines (D_ori) are calculated based-on the original TBS server while the virtual deadlines (D_vir) are introduced by the original virtual release time (VRA) and enhanced virtual release time (EVRA) algorithms. Two next columns are the response time corresponding arrival times of the target task. R_ori and R_vir are the response times corresponding to the original deadline and the virtual deadline, respectively. Two target factors of the evaluation are examined for the VRA software design, EVRA software design, and EVRA hardware design, as denoted as VRA, soft-EVRA, and hard-EVRA, respectively.

Table 5.9: Results on Scenario 1

Task's entering times	*D_ori*	*D_vir*	*R_ori* **(tick)**	*R_vir* **(tick)**	**Maximum overhead per tick (cycle)**			**Total execution overhead (cycle)**		
					VRA	Soft-EVRA	Hard-EVRA	VRA	Soft-EVRA	Hard-EVRA
225	289	277	1	1	178	135 (-24%)	126 (-6.7%)	593	432 (-27%)	406 (-6%)
410	474	464	5	5						
496	560	560	16	16						
725	789	789	6	6						
930	994	994	7	7						
1020	1084	1084	3	3						
1413	1477	1465	14	6						
1731	1795	1795	1	1						
1833	1897	1890	13	13						
2010	2074	2064	7	7						

Table 5.10: Results on Scenario 2

Task's entering times	*D_ori*	*D_vir*	*R_ori* **(tick)**	*R_vir* **(tick)**	**Maximum overhead per tick (cycle)**			**Total execution overhead (cycle)**		
					VRA	Soft-EVRA	Hard-EVRA	VRA	Soft-EVRA	Hard-EVRA
90	166	156	26	26	177	146 (-17%)	114 (-22%)	711	459 (-35%)	409 (-11%)
167	243	200	27	21						
271	347	308	27	23						
385	461	461	1	1						
497	573	557	21	21						
603	679	676	25	25						
724	800	796	18	18						
847	923	912	32	29						
965	1041	1036	31	31						
1116	1192	1192	2	2						

Table 5.11: Results on Scenario 3

Task's entering times	*D_ori*	*D_vir*	*R_ori* **(tick)**	*R_vir* **(tick)**	**Maximum overhead per tick (cycle)**			**Total execution overhead (cycle)**		
					VRA	Soft-EVRA	Hard-EVRA	VRA	Soft-EVRA	Hard-EVRA
90	133	123	14	14	183	150 (-18%)	124 (-17%)	554	420 (-23%)	389 (-7.4%)
167	210	203	13	13						
271	314	313	7	7						
385	428	428	1	1						
497	540	524	2	2						
603	646	643	16	13						
724	767	763	22	22						
847	890	887	17	17						
965	1008	1008	18	18						
1116	1159	1159	1	1						

Table 5.12: Results on Scenario 4

Task's entering times	*D_ori*	*D_vir*	*R_ori* **(tick)**	*R_vir* **(tick)**	**Maximum overhead per tick (cycle)**			**Total execution overhead (cycle)**		
					VRA	Soft-EVRA	Hard-EVRA	VRA	Soft-EVRA	Hard-EVRA
90	153	151	27	27	154	122 (-21%)	105 (-14%)	523	440 (-16%)	381 (-13%)
167	230	230	7	7						
267	300	274	26	23						
320	383	371	14	14						
392	455	447	23	19						
457	520	511	15	15						
541	604	588	16	16						
612	675	675	1	1						
698	761	761	1	1						
760	823	823	4	4						

Table 5.13: Results on Scenario 5

Task's entering times	*D_ori*	*D_vir*	*R_ori* **(tick)**	*R_vir* **(tick)**	**Maximum overhead per tick (cycle)**			**Total execution overhead (cycle)**		
					VRA	Soft-EVRA	Hard-EVRA	VRA	Soft-EVRA	Hard-EVRA
225	301	296	5	5	149	124 (-17%)	113 (-9%)	658	483 (-26%)	439 (-9%)
410	486	472	9	9						
496	572	557	22	18						
725	801	796	17	17						
930	1006	996	26	26						
1020	1096	1040	13	8						
1413	1489	1476	23	23						
1731	1807	1792	8	8						
1833	1909	1909	3	3						
2010	2086	2076	21	21						

The evaluation results first show that there is improvement in tasks' absolute deadline. By the virtual release advancing, the virtual deadlines are calculated earlier than the original ones at many points of tasks's entering time. The earlier deadlines then may allow some shorter response times compared with those of the original algorithm as denoted at rows in red. The virtual response times corresponding to the virtual deadlines are the same in tick for all VRA, soft-EVRA, and hard-EVRA examinees.

For the runtime overhead, the percentages of overhead reduction are here calculated for the soft-EVRA compared to the original VRA and for the hard-EVRA compared to the soft-EVRA. A negative percentage indicates the overhead reduction. Overall, all of scenarios of evaluations show that the proposed algorithm in this research can actually alleviate the runtime overhead compared to the original algorithm and that the runtime overhead of the hard-EVRA is lower than that of the soft-EVRA.

In details, the results exhibits that the maximum overhead per tick of the soft-EVRA is from 16.7% to 24% lower than that of the original VRA while the hard-EVRA is 13.7% in average lower than the soft-EVRA. Furthermore, the output of evaluations on total execution overhead also clearly confirms the positive improvements on the EVRA algorithm. That is, the total execution overhead averagely decreases by 25.4% compared to the original VRA. In the same trend, the hard-EVRA reduces the total overhead by around 9.3% compared to the soft-EVRA. Especially, although the evaluations are done with a limited number of tasks, results somewhat show that the maximum overhead per tick is such a very small part of a tick. In the Scenario 3 where the maximum overhead per tick are the most, it takes about 0.1% of a tick (totally 166400 cycles). This is an important result which allows the algorithm to be applied to precise systems.

However, it is necessary to note that these equal response times are obtained above as a consequence of that the additional overhead takes a very small percentage of a tick. Actually, the response times are different if calculated exactly in cycle under the effect of runtime overhead. Because response time is not an objective requirement of the

evaluation, it is not displayed in full in results tables. An example, instead, is shown in Table 5.14 as a further illustration for this concern. The table displays the difference of response times in cycle for the 2-th entering time of aperiodic task in Scenario 3. In this example, the target task is activated at tick 167 and finishes at tick 180 with response time of 13 for all of the original TBS, original VRA, soft-EVRA, and dard-EVRA. The cycle offset, defined in Figure 5.2, at row 5 presents the response time exactly in cycle at tick 180. As shown, task finishes earliest in the original TBS with the lowest overhead. The order of finishing time is followed by that of Hard-EVRA, soft-EVRA, and original VRA, respectively.

Table 5.14: A example of different response times in cycle under the effect of time overhead

	Original TBS	**Original VRA**	**Soft-EVRA**	**Hard-EVRA**
Release time	167			
Finishing time	180			
Response time (tick)	13			
Cycle offset (cycle) at the 180-th tick	49221	49508	49426	49358

Chapter 6

Conclusion

In this research, an enhanced virtual release advancing algorithm is proposed. This is an enhancement of the virtual release advancing in terms of runtime overhead. The runtime overhead is significantly reduced while the schedulability and responsiveness are guaranteed. With the lower time complexity, the new technique is more adaptive to precise real-time systems. In addition, this technique reduces the implementation complexity of the previous algorithm by determining the bound of the advancing.

In addition, the proposed algorithm is proven its effectiveness and applicability with the hardware implementation on FPGA. With designed mechanism for parallel processing, the hardware design generates a higher performance than the software one. Such achievements of this research show that this algorithm is worthy for a continuous research to become closer to the real systems. Applying the algorithm on multiprocessor systems is also an alternative approach.

Bibliography

[1] K. Tanaka, “Real-Time Scheduling for Reducing Jitters of Periodic Tasks”, Journal of Information Processing, Vol23, No.5, pp.542552, 2015.

[2] K. Tanaka, “Virtual Release Advancing for Earlier Deadlines”, SIGBED Review, Vol.12, No.3, June, 2015.

[3] Marco Spuri and Giorgio C. Buttazzo, “Efficient Aperiodic Service under Earliest Deadline Scheduling”, 15th IEEE Real-time System Symposium, San Juan, Puerto Rico, 1994.

[4] “Cortex-A9 Technical Reference Manual”, ARM.

[5] “Cortex-A9 Floating-Point Unit Technical Reference Manual”, Revision: r2p0, Copyright 2008-2009 ARM.

[6] ITRON Committee, TRON ASSOCIATION. ITRON4.0 Speciation Ver.4.00.00.

[7] Zynq-7000 All Programmable SoC Technical Reference Manual, v1.10, February 23, 2015, Xilinx.

[8] C. L. Liu and J. W. Layland, “Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment”, Journal of the Association for Computing Machinery, Vol. 20, No. 1, pp. 4661, January 1973.

[9] M. Spuri, G. Buttazzo, and F. Sensini, “Robust Aperiodic Scheduling under Dynamic Priority Systems”, Proc. of Real-Time Systems Sympo-sium, pp.210219 (1995).

[10] Savanna RTOS for Zynq version 0.100, Developed by Kiyofumi Tanaka, Japan Advanced Institute of Science and Technology.

[11] The TB-7Z-020-EMC Debug Monitor Boot Program Revision 001, 2015.